

Design of Memetic Algorithms for Permutation Problems via Hybridization with Sorting

Rodolfo Torres-Velázquez

(BEng Chemical Petroleum Engineering, IPN México)

MSc Computer Science, UAM México)

A thesis submitted in fulfillment of the requirements for the degree of
Doctor of Philosophy



The UNIVERSITY
of NEWCASTLE
AUSTRALIA

School of Electrical Engineering and Computer Science

The University of Newcastle

Callaghan NSW 2308

Australia

January 2003

Certificate of Originality

I hereby certify that the work embodied in this thesis is the result of original research and has not been submitted for a higher degree at any other University or Institution.

(Signed) _____

Rodolfo Torres-Velázquez

Acknowledgements

I would like to thank my supervisor, Associate Professor Vladimir Estivill-Castro, for giving me invaluable guidance and encouragement during the preparation of this thesis and throughout my research. His knowledge, experience, motivation, enthusiasm and help make my research worthwhile.

Also, I would like to thank the National Council of Science and Technology (CONACYT), México for the support provided during the preparation of this thesis.

To my wife **Ana Maria** and
my children: **Lorena, Arturo and Rudy.**

List of publications arising from this thesis

1. Escalada-Imaz, G., Torres-Velázquez, R., Algoritmos Genéticos Genéricos y Basados en Orden Conceptos Fundamentales y Mecanismo de Base, Revista Iberoamericana de Inteligencia Artificial (ISSN 1137-3601), N3 pp 10-29, Spain, Autumn 1997, In Spanish.
2. Estivill-Castro V., Torres-Velázquez R., Hybrid genetic algorithm for solving the p-median problem, Proceedings of the Second Asia Pacific Conference on Simulated Evolution and Learning SEAL-98 (ISBN 3-540-65907-2), Lecture Notes in Artificial Intelligence Vol. 1585, pp 18-25, Springer-Verlag, Canberra, Australia, November 24-27 1998.
3. Estivill-Castro V., Torres-Velázquez R., Classical-Sorting Embedded in Genetic Algorithms for Improved Permutation Search, Congress on Evolutionary Computation CEC2001 (ISBN 0-7803-6658-1), pp 941-948, IEEE Press, Seoul Korea, May 27-30 2001.
4. Estivill-Castro V., Torres-Velázquez R., How should feasibility be handled by Genetic Algorithms on Constraint Combinatorial Optimization Problems? The case of the valued n-queens problem, Second Workshop on Memetic Algorithms,

Genetic and Evolutionary Computation Conference GECCO 2001, pp 146-151, San Francisco CA, July 7 2001.

5. Torres-Velázquez R., Estivill-Castro V., A Memetic Algorithm Guided by Quicksort for the Error-Correcting Graph Isomorphism Problem, Applications of Evolutionary Computing (ISBN 3-540-43432-1, ISSN 0302-9743), Lecture Notes in Computer Science LNCS 2279, pp 173-182, Springer-Verlag, 2002.
6. Torres-Velázquez R., Estivill-Castro V., Local Search for Hamiltonian Path with Applications to Clustering Visitation Paths, Local Search ,Two Day Workshop, OR Society (UK): Local Search Study Group, City University, London UK 16-17 April 2002.
7. Torres-Velázquez R., Estivill-Castro V., A Memetic Algorithm Instantiated with Selection Sort Consistently Finds Global Optima for the Error-Correcting Graph Isomorphism, Congress on Evolutionary Computation CEC2002 (ISBN 0-7803-7278-6, ISSN 1098-7576), held as part of the 2002 World Congress on Computational Intelligence, pp 1958-1963, IEEE Press, Honolulu, Hawaii, May 12-17 2002.

Publications produced towards PhD candidature

1. Torres-Velázquez R., GeneClas. Implementación de un Algoritmo Genético Basado en Orden, Instituto de Investigación en Inteligencia Artificial CSIC, Research Report IIIA 96/26, 47 pp, Barcelona, Spain, 1996, In Spanish.
2. Escalada-Imaz G., Torres Velázquez R., Complexity Issues in the Davis and Putnam Scheme, 9th International Conference, Artificial Intelligence: Methodology, Systems, and Applications AIMSA 2000 (ISBN 3-540-41044-9), Lecture Notes in Artificial Intelligence Vol. 1904, pp 261-271, Springer-Verlag, Varna Bulgaria, September 20-23 2000.
3. Torres-Velázquez R., Escalada-Imaz G., CSPs: Preprocessing by Factorizing Domains, Instituto de Investigación en Inteligencia Artificial CSIC, Research Report IIIA 2001-01, 7 pp, Barcelona, Spain, 2001.

Contents

List of Figures	xi
List of Tables	xv
1 Introduction	1
1.1 Motivation	2
1.2 Our Contribution	5
1.3 Thesis Layout	9
2 Basic concepts	11
2.1 Combinatorial Optimization	11
2.2 Solving Techniques	15
2.3 Genetic Algorithms	16
2.3.1 Selection Methods	19
2.3.2 Crossover	21
2.3.3 Mutation	22
2.4 Order-Based Genetic Algorithms	23
2.4.1 The PMX Crossover Operator	24
2.4.2 An Order-Based Mutation Operator	25
2.4.3 Constraint Optimization Problems: a challenging assignment for GAs and MAs	26

3	Local Search	30
3.1	Local Search for Combinatorial Optimization	30
3.2	LS for the <i>Longest Hamiltonian Path Problem</i> in Clustering Visitation Paths of Web-Pages	33
3.2.1	A Measure of Effectiveness	35
3.2.2	The Optimization of $e(A)$ can be Modeled as a <i>Longest Hamil-</i> <i>tonian Path Problem</i>	36
3.2.3	Different Implementations of <i>BEA</i>	38
3.2.4	Our Approach	50
3.2.5	A Case Study	55
3.2.6	The Experiments	55
3.3	Discussion	63
4	Memetic Algorithms	65
4.1	Introduction	65
4.2	The Lamarckian Approach and the Baldwin Effect	69
4.3	The p -Median Problem	71
4.3.1	The Solution Methods	74
4.3.2	The Structure of the Memetic Algorithm	77
4.4	Discussion	81
5	Classical Sorting Directs Mutation	82
5.1	Introduction	82
5.2	Classical Sorting as a Local-Search Mechanism	83
5.3	Classical Sorting as Director of Mutation	90
5.4	Attributes of Sorting that Influence the Neighborhood Structure	93
5.4.1	Swap Constraint	94
5.4.2	Record of the State	96
5.4.3	Focus	99
5.4.4	Dependency on Input	99

5.4.5	Presortedness Measure	100
5.5	Implementation	108
5.5.1	Order of Nominating Neighbors	108
5.5.2	Amount of Progress	111
6	Experiments with <i>MA-sorting</i>	117
6.1	Introduction	117
6.2	Harder Problems than Sorting	118
6.3	A Benchmark of Graphs	119
6.3.1	Fitness Function	121
6.3.2	Algorithms for ECGI	121
6.3.3	Our implementation of <i>MA-sorting</i>	123
6.4	Experimental Description	125
6.5	Impact of Sorting into Mutation	128
6.5.1	Sorting Method	131
6.5.2	Amount of Progress	157
6.5.3	Order of Nominating Permutations	176
6.6	A Comparison between <i>MA-sorting</i> and GBSA	201
6.6.1	Increasing the Number of Explorations	205
6.6.2	Providing a Better Initial Solution	210
6.7	Discussion	213
7	A Baldwinian Approach for Constraint Combinatorial Optimiza- tion	216
7.1	Introduction	216
7.2	Our <i>Baldwinian</i> Approach	218
7.2.1	The Constructor Stage	221
7.2.2	The Learner Stage	225
7.3	A case study: the <i>valued n-queens problem</i>	226
7.3.1	Genotype Representation	228

7.3.2	Fitness Evaluation	229
7.3.3	Genetic Operators	231
7.3.4	Experimental Results	232
7.4	Discussion	238
8	Conclusion	242
8.1	A Summary of the Main Conclusions	242
8.2	A Window to the Future	254
	Bibliography	256

List of Figures

2.1	<i>An example of CCOP search space S. Points a, b, c are in the feasible search space F and points d, e in the infeasible search space $S \setminus F$.</i>	12
3.1	<i>Pseudo-code for Algorithm Local Search (LS).</i>	31
3.2	<i>The iteration of the BEAm algorithm of McCormick et al. tests each column k in the remaining $n - i$ columns for placement in each of the $i + 1$ positions amongst the i already placed columns.</i>	41
3.3	<i>The iteration of the BEAo algorithm tests the $(i + 1)$-th column for placement in each of the $i + 1$ positions amongst the i placed columns.</i>	45
3.4	<i>The body of the BEAx algorithm during the column optimization phase as implemented by Xiao et al.</i>	48
3.5	<i>The body of the 2-opt algorithm during the column optimization phase.</i>	53
3.6	<i>Quality of solutions provided by the algorithms BEAm, BEAo and 2-opt; BEAx is the baseline.</i>	58
3.7	<i>Quality of solutions provided by the algorithms BEAm + 2-opt, BEAo + 2-opt, BEAx + 2-opt and 2-opt; BEAx is the baseline.</i>	59
4.1	<i>Lamarckian and Baldwinian. Two approaches to handle the results of genotype improving.</i>	70
4.2	<i>MA and GA optimization with respect to objective function evaluations. Final average for MA is 3.54 with a standard deviation of 0.03 while the final average for the GA is 3.60 with a standard deviation of 0.07.</i>	81

5.1	Insertion Sort as an algorithm for searching the permutation π such that $\pi^{-1}((x_1, x_2, x_3, x_4))$ is sorted. The array a encodes the permutation $\pi^{-1}(X)$	86
5.2	The search space of $4!$ permutations arranged into a graph by swaps of adjacent items.	88
5.3	The search tree for $4!$ permutations built by the Insertion Sort mechanism.	89
5.4	Pseudo-code for Sorting as Local Search (LS) in a Search Space of Permutations.	90
5.5	The search space of $4!$ permutations arranged into a graph by arbitrary swaps of items.	97
5.6	The computation of Eng. The sequence $[\diamond, \heartsuit, \clubsuit, \spadesuit]$ is ordered and $[\spadesuit, \clubsuit, \heartsuit, \diamond]$ is the current sequence.	103
5.7	Sub-case 1 where $\text{rank}(x_i) > i$ and $\text{rank}(x_j) < j$. Eng before swap is greater than Eng after swap.	105
5.8	Sub-case 2 where $\text{rank}(x_i) > i$ and $\text{rank}(x_j) \geq j$. Eng before and after the swap is the same.	106
5.9	Case 2 where $\text{rank}(x_i) \leq i$ and $\text{rank}(x_j) < j$. Eng before and after the swap is the same.	107
6.1	Minimal progress mode. State randomly generated.	134
6.2	Minimal progress mode. State deterministically generated, single-threading.	135
6.3	Minimal progress mode. State deterministically generated, multi-threading.	136
6.4	Maximal progress mode. State randomly generated.	139
6.5	Maximal progress mode. State deterministically generated, single-threading.	140
6.6	Maximal progress mode. State deterministically generated, multi-threading.	141
6.7	First improvement. State randomly generated.	146
6.8	First improvement. State deterministically generated, single-threading.	147
6.9	First improvement. State deterministically generated, multi-threading.	148

6.10	<i>Best improvement. State randomly generated.</i>	152
6.11	<i>Best improvement. State deterministically generated, single-threading.</i>	153
6.12	<i>Best improvement. State deterministically generated, multi-threading.</i>	154
6.13	<i>The Eng-values for MA-sorting instantiated with Insertion Sort, Partition and Selection Sort. Lower values of Eng means better results.</i>	158
6.14	<i>Insertion Sort. State randomly generated.</i>	160
6.15	<i>Insertion Sort. State deterministically generated, single-threading.</i>	161
6.16	<i>Insertion Sort. State deterministically generated, multi-threading.</i>	162
6.17	<i>Partition. State randomly generated.</i>	166
6.18	<i>Partition. State deterministically generated, single-threading.</i>	167
6.19	<i>Partition. State deterministically generated, multi-threading.</i>	168
6.20	<i>Selection Sort. State randomly generated.</i>	172
6.21	<i>Selection Sort. State deterministically generated, single-threading.</i>	173
6.22	<i>Selection Sort. State deterministically generated, multi-threading.</i>	174
6.23	<i>Insertion Sort, minimal progress mode.</i>	180
6.24	<i>Partition, minimal progress mode.</i>	181
6.25	<i>Selection Sort, minimal progress mode.</i>	182
6.26	<i>Insertion Sort, Maximal progress mode.</i>	185
6.27	<i>Partition, Maximal progress mode.</i>	186
6.28	<i>Selection Sort, Maximal progress mode.</i>	187
6.29	<i>Insertion Sort, First improvement.</i>	191
6.30	<i>Partition, First improvement.</i>	192
6.31	<i>Selection Sort, First improvement.</i>	193
6.32	<i>Insertion Sort, Best improvement.</i>	196
6.33	<i>Partition, Best improvement.</i>	197
6.34	<i>Selection Sort, Best improvement.</i>	198
6.35	<i>Comparison of quality of solutions between MA-sorting and GBSA for different graph sizes.</i>	204

6.36	<i>Comparison of quality of solutions between MA-sorting and GBSA for different graph sizes. MA-sorting orders a subsequence of the input sequence.</i>	209
6.37	<i>Comparison of quality of solutions between MA-sorting and GAB for different graph sizes. MA-sorting orders a subsequence of the input sequence, and a better initial solution is given to MA-Isort.</i>	212
7.1	<i>Our Baldwinian approach comprises two stages</i>	219
7.2	<i>The constructor stage</i>	224
7.3	<i>An example of a valued 8-queens problem</i>	227
7.4	<i>Positive and negative arrays</i>	230
7.5	<i>Baldwin-GA produces good quality solutions</i>	234
7.6	<i>Proportion of feasible solutions</i>	237
7.7	<i>SORT mutation provides benefits statistically significant.</i>	239
7.8	<i>An instance of the valued 4-queens problem</i>	241

List of Tables

3.1	<i>Execution time (in seconds) of BEAm, BEAo, BEAx and 2-opt.</i>	60
3.2	<i>Execution time (in seconds) of BEAm+2-opt, BEAo+2-opt, BEAx+2-opt and 2-opt.</i>	61
3.3	<i>Comparison of BEAm with a BEAo+2-opt, in terms of execution time (in seconds) and quality of solutions.</i>	62
4.1	<i>TAB and MA optimization with respect to objective function evaluations.</i>	79
5.1	<i>Attributes of Insertion Sort, Quicksort and Selection Sort.</i>	108
5.2	<i>Variants using Insertion Sort.</i>	114
5.3	<i>Variants using Quicksort.</i>	115
5.4	<i>Variants using Selection Sort.</i>	116
6.1	<i>GA Parameters.</i>	123
6.2	<i>Translation for labels in Figures and Tables.</i>	132
6.3	<i>Number of evaluations of the objective function for Minimal progress mode with Single-threading (where the information about the state is deterministically generated), and for different graph sizes.</i>	137
6.4	<i>Number of evaluations of the objective function for Minimal progress mode with Single-threading (where the information about the state is randomly generated), and for different graph sizes.</i>	138

6.5	<i>Number of evaluations of the objective function for Minimal progress mode with Multi-threading (where the information about the state is deterministically generated), and for different graph sizes.</i>	138
6.6	<i>Number of evaluations of the objective function for Maximal progress mode with Single-threading (where the information about the state is deterministically generated), and for different graph sizes.</i>	142
6.7	<i>Number of evaluations of the objective function for Maximal progress mode with Single-threading (where the information about the state is randomly generated), and for different graph sizes.</i>	143
6.8	<i>Number of evaluations of the objective function for Maximal progress mode with Multi-threading (where the information about the state is deterministically generated), and for different graph sizes.</i>	143
6.9	<i>Number of evaluations of the objective function for First improvement with Single-threading (where the information about the state is deterministically generated), and for different graph sizes.</i>	144
6.10	<i>Number of evaluations of the objective function for First improvement with Single-threading (where the information about the state is randomly generated), and for different graph sizes.</i>	145
6.11	<i>Number of evaluations of the objective function for First improvement with Multi-threading (where the information about the state is deterministically generated), and for different graph sizes.</i>	149
6.12	<i>Number of evaluations of the objective function for Best improvement with Single-threading (where the information about the state is deterministically generated), and for different graph sizes.</i>	150
6.13	<i>Number of evaluations of the objective function for Best improvement with Single-threading (where the information about the state is randomly generated), and for different graph sizes.</i>	151
6.14	<i>Number of evaluations of the objective function for Best improvement with Multi-threading (where the information about the state is deterministically generated), and for different graph sizes.</i>	155
6.15	<i>Impact of the sorting algorithm into MA-sorting.</i>	156
6.16	<i>Number of evaluations of the objective function for Insertion Sort with Single-threading where the information about the state is deterministically generated, and for different graph sizes.</i>	163

6.17	<i>Number of evaluations of the objective function for Insertion Sort with Single-threading where the information about the state is randomly generated, and for different graph sizes.</i>	164
6.18	<i>Number of evaluations of the objective function for Insertion Sort with Multi-threading where the information about the state is deterministically generated, and for different graph sizes.</i>	165
6.19	<i>Number of evaluations of the objective function for Partition with Single-threading where the information about the state is deterministically generated, and for different graph sizes.</i>	169
6.20	<i>Number of evaluations of the objective function for Partition with Single-threading where the information about the state is randomly generated, and for different graph sizes.</i>	170
6.21	<i>Number of evaluations of the objective function for Partition with Multi-threading where the information about the state is deterministically generated, and for different graph sizes.</i>	171
6.22	<i>Number of evaluations of the objective function for Selection Sort with Single-threading where the information about the state is deterministically generated, and for different graph sizes.</i>	175
6.23	<i>Number of evaluations of the objective function for Selection Sort with Single-threading where the information about the state is randomly generated, and for different graph sizes.</i>	176
6.24	<i>Number of evaluations of the objective function for Selection Sort with Multi-threading where the information about the state is deterministically generated, and for different graph sizes.</i>	177
6.25	<i>Impact of the amount of progress into MA-sorting.</i>	177
6.26	<i>Number of evaluations of the objective function for Insertion Sort with Minimal progress mode, and for different graph sizes.</i>	179
6.27	<i>Number of evaluations of the objective function for Partition with Minimal progress mode, and for different graph sizes.</i>	179
6.28	<i>Number of evaluations of the objective function for Selection Sort with Minimal progress mode, and for different graph sizes.</i>	179
6.29	<i>Number of evaluations of the objective function for Insertion Sort with Maximal progress mode, and for different graph sizes.</i>	184

6.30	<i>Number of evaluations of the objective function for Partition with Maximal progress mode, and for different graph sizes.</i>	184
6.31	<i>Number of evaluations of the objective function for Selection Sort with Maximal progress mode, and for different graph sizes.</i>	184
6.32	<i>Number of evaluations of the objective function for Insertion Sort with First improvement, and for different graph sizes.</i>	189
6.33	<i>Number of evaluations of the objective function for Partition with First improvement, and for different graph sizes.</i>	189
6.34	<i>Number of evaluations of the objective function for Selection Sort with First improvement, and for different graph sizes.</i>	190
6.35	<i>Number of evaluations of the objective function for Insertion Sort with Best improvement, and for different graph sizes.</i>	195
6.36	<i>Number of evaluations of the objective function for Partition with Best improvement, and for different graph sizes.</i>	199
6.37	<i>Number of evaluations of the objective function for Selection Sort with Best improvement, and for different graph sizes.</i>	200
6.38	<i>Impact of the method of generating the state of the sorting algorithm into MA-sorting.</i>	201
6.39	<i>Comparison of the number of evaluations of the objective function between GBSA and some variants of MA-sorting for different graph sizes.</i>	203
6.40	<i>Comparison of the number of evaluations of the objective function between GBSA and some variants of MA-sorting for different graph sizes. MA-sorting orders a subsequence of the input sequence.</i>	208
6.41	<i>Comparison of the number of evaluations of the objective function between GBSA and some variants of MA-sorting for different graph sizes. MA-sorting orders a subsequence of the input sequence, and a better initial solution is the input to MA-Isort.</i>	211
7.1	<i>Baldwin-GA parameters.</i>	232
7.2	<i>Comparison of a B&B algorithm with a Baldwin-GA.</i>	236

Abstract

Memetic algorithms have been shown to be effective in dealing with hard combinatorial optimization problems. Analytical and experimental attempts have been made to explain the behavior of memetic algorithms. While those approaches provide some insight into the mechanism of memetic algorithms, they provide few clues towards the effective design of these algorithms. Our research argues in favor of combining memetic algorithms with classical sorting, a combination that we call *MA-sorting*, mainly for two reasons: our approach facilitates the design of memetic algorithms, and it improves permutation search. For the design of our *MA-sorting* we follow a three-step process: 1) We identify a measure of disorder and algorithms that minimize this measure effectively; 2) We establish that the optimization of the measure of disorder is in close association with optimizing the objective function; 3) We merge the heuristic to optimize the measure of disorder as a local search into a memetic algorithm. In our approach classical sorting works as a map for the search space S_n of $n!$ permutations. We propose to guide memetic algorithms with a new family of mutation operators. Our mutation operators perform local search following the path traced by classical sorting mechanisms. For constraint combinatorial optimization, in particular for the *valued n -queens problem* as case study, our research utilizes the *Baldwinian* approach to handle the concept of feasibility. We provide experimental evidence that shows that for the Error-Correcting Graph Isomorphism and the *valued n -queens problem*, the combination of memetic algorithms with classical sorting improves permutation search. For these case studies our *MA-sorting* achieves better quality-versus-effort trade-off than previous proposals.

Notation

A, B, M, SM	denote matrices
$a[\]$	denotes an array
C	denotes a set of representatives
D	denotes a vector space
d	denotes distance
E	denotes a set of edges
F	denotes a feasible search space
$e()$	denotes a measure of effectiveness
$f()$	denotes an objective function
G, H	denote graphs
h	denotes the Hamming distance
$I()$	denotes an inversion operator
L	denotes a family of reversible transformations
$N \subseteq S$	denotes a neighborhood
P	denotes a permutation matrix
n, m	denote sizes
p	denotes facilities
p_m	denotes mutation probability
p_c	denotes crossover probability
Q	denotes a sorting algorithm
R	denotes a set of constraints
$r()$	denotes the effort of repairing the genotype
S	denotes a search space of solutions

$s()$	denotes a swap operator
T	denotes a comparison tree
U	denotes a set of points
u	denotes a point
V	denotes a set of vertices
w	denotes a weight
X	denotes a sequence
x	denotes an element in a sequence
Y	denotes a set of variables
W	denotes the sum of fitness
Z	denotes a (domain) set of values
a, b, c, d	denote items
y, z	denote nodes of a graph
i, j, k, l, q, r, t, z	denote indices
$\alpha()$	denotes a vertex interpreter
$\beta()$	denotes an edge interpreter
ε	denotes noise
ζ	denotes a chromosome
ϑ	denotes a pivot
ι	denotes a pass
κ	denotes a non-primary key attribute
$\nu()$	denotes a neighborhood operator
π, σ	denote permutations
$\varphi(\kappa)$	denotes the κ^{th} Bell number

ρ	denotes a penalty function
ϕ	denotes a feasibility condition
ψ	denotes a fitness (Evaluation) function
ω	denotes a generic operator
Δ	denotes a gradient
Ξ	denotes a state
Υ	denotes a bijection
Ψ	denotes a minimization problem
$< ()$	denotes a comparison predicate

Chapter 1

Introduction

Many problems of practical and theoretical interest can be modeled as searching in a space of states (or approximate solutions). In this context, the search engine constructs a set of paths (transitions) between states. These paths are traveled by the application of a set of operators (actions). Each state may represent: an approximate total solution or a partial solution. In the first case, the operators transform a given solution into a presumably better solution. In the second case, the operators extend a given partial solution. In both cases, alternative paths, from one state to another, are evaluated typically in polynomial time. The aim may be either to optimize (minimize or maximize) the number of paths walked from the initial to the final state, or, if the paths are weighted, to optimize the sum of weights on the traveled paths.

In order to search in a space of states we must define: 1) a suitable representation of solutions; 2) an initial solution; 3) the set of valid operators; 4) a procedure

to choose a path amongst several alternatives; 5) a method to identify the search goal (the final solution). In summary, we must formulate the problem as a set of states (approximate solutions), identifying amongst them the initial and the final state, a set of permitted actions (operators), and a criterion to make a distinction between solutions. Unfortunately, for many relevant problems it is not possible to explore exhaustively their space of approximate solutions, mainly because their space is huge.

Both Genetic Algorithms (GAs) [19, 44, 52, 89, 90, 139] and local-search algorithms (LS) [2] can be conceptualized as mechanisms that search in a space of approximate solutions. Also, both can be applied to problems where the full exploration of their space of approximate solutions is impractical.

1.1 Motivation

In combinatorial optimization, simple GAs are not usually considered to be as good, in terms of efficiency and quality of solutions, as domain-specific methods [19, 48, 137]. However, plenty of experimental evidence has shown that combining genetic algorithms with other mechanisms, mainly with local-search methods (this particular combination is called memetic algorithms), results in a more effective optimization when dealing with hard combinatorial optimization problems [19, 31, 32, 49, 82, 85, 93, 94, 115, 131–133].

Memetic Algorithms (MAs) have been applied to several combinatorial optimization problems, amongst them: Traveling Salesman Problem (TSP) [8, 40, 45, 48, 49, 53, 60,

64, 76, 83, 94, 98, 128, 133, 137], Quadratic Assignment problem (QAP) [82, 85, 93], Job Shop Scheduling problem (JSS) [18, 76, 121, 151], p -median problem [25, 30], Error-Correcting Graph Isomorphism problem (ECGI) [31, 131, 132, 141], and the *valued n -queens problem* [32].

Analytical and experimental attempts have been made to explain the behavior of genetic algorithms when a local-search mechanism is included on it [46, 109, 144]. For example: Radcliffe and Surry [109] propose a formalization of MAs with the purpose of devising representation-independent operators; Whitley [144] models a memetic algorithm in the context of the existing models for the simple genetic algorithm; and Goldberg and Voessner [46] propose a framework oriented to calculate an optimal balance of local and global search. While those approaches provide some insight into the mechanism of memetic algorithms, they provide few clues towards the effective design of these algorithms. Our research provides information about the behavior of genetic algorithms which integrates local search; this information is useful for the design of memetic algorithms.

Jones and Forrest [63] introduce the concept of “fitness landscape” mainly to analyze the difficulty of problems for genetic algorithms. This concept can be explained as follows. Observe that, in order to choose the path to be traveled during the search in a space of states, we must assign to each approximate solution a figure of merit. This value usually comes from the application of an objective function to the problem at hand. In the GAs arena this value is called “fitness”. It has been useful [63], particularly in GAs, to consider that as soon as we compute the fitness of approximate solutions a landscape emerges from the space of states. On it, the hills

correspond to high fitness values (if we are maximizing) and the valleys correspond to low fitness values. More formally [86], a fitness landscape is a triplet (S, f, d) where S is the set of approximate solutions (the space of states), f is an objective function, and d is a measure of distance. In particular, Jones and Forrest [63] propose, in the context of fitness landscape, that the relationship between fitness and distance to goal has a great influence on search difficulty for GAs. The measure of distance is relevant because different landscapes will emerge by using different definitions of distance. Moreover, Jones and Forrest [63] point out a crucial issue: the analysis of fitness landscape will be more accurate if it is based on the distance between points in the space of states according to the operator in use by the algorithm.

Yamada and Reeves [152], and Merz and Freisleben [85, 86] have used the concept of fitness landscape to characterize the difficulty of some instances of hard combinatorial problems for memetic algorithms. Their analysis is more informative for the design of memetic algorithms, mainly because they provide experimental evidence that shows that for some instances of those hard combinatorial optimization problems their variants of MAs are successful. But they left open the relevant issue about the link between the local-search operator, which is embedded in the the memetic algorithms, and the measure of distance. Our research addresses this question.

Yao [154], Yamada and Reeves [152], and Merz and Freisleben [85, 86] recognize that the operator, which converts one approximate solution into another, defines a distance on the search space given by the minimum number of applications of this operator. In particular, Yamada and Reeves [152] admit that this distance is difficult to compute and they apply generic measures of distance not necessarily

related to their local-search operators. Merz and Freisleben [85, 86], for the graph bipartitioning and the quadratic assignment problem, follow the same approach and also apply generic measures of distance. Yao [154] describes a Simulated Annealing (SA) model where two approximate solutions that have exactly μ elements different between them are μ -steps reachable from each other. Yao [154] uses such a distance to develop an SA model with dynamic generation and acceptance probabilities, which can gradually reduce its neighborhood with the decrease of temperature.

1.2 Our Contribution

Including local search in GAs raises an important question: Where to insert the local-search mechanism?

The answers can be grouped in three categories. The first category proposes to extend the capabilities of the crossover operator by means of local searching [6, 8, 13, 48, 49, 60, 76, 84, 86, 98, 115, 121, 128, 133, 151, 152]. The second category applies local searching to some, or all, of the chromosomes in the population, as a separated and independent step of the GA mechanism [8, 10, 40, 49, 53, 59, 60, 65, 74, 75, 82–86, 93, 94, 137, 144, 151, 153]. The last category uses mutation as a local-search operator [6, 13, 19, 30, 49, 115, 133, 145, 150]. These categories are not necessarily disjoint. That is, in many cases more than one category is applied.

For those problems with a computationally expensive evaluation of their objective function, the application of local search to a high proportion of individuals in the population, either using a separated stage or by crossover, is impractical. Moreover,

Hart [51] illustrates that applying local search to a small fraction of the population can improve the efficiency of memetic algorithms. Thus, for problems with an expensive objective function, local search can be applied to a small fraction of individuals in the population either using a separated step or by the mutation operator. We argue in favor of using local search in the mutation operator, mainly to allow the evolutionary search to regulate the effect of this operator. Our research provides empirical evidence which shows that for several problems, the use of local search in the mutation operator provides good results [30–32, 131, 132]. Thus, our techniques are particularly well suited for problems with an expensive objective function.

Our research argues in favor of combining memetic algorithms with classical sorting to improve permutation search [31, 32, 131, 132] for several reasons. First, sorting algorithms can be conceptualized as local-search mechanisms which define paths in the search space of $n!$ permutations based on the information provided by a comparison predicate. Second, sorting algorithms are implicitly guided by measures of presortedness. In the context of fitness landscape, these measures of presortedness correspond to the concept of distance, and sorting algorithms are the operators. Thus, in contrast with previous proposals [85, 86, 152] in our techniques the local-search operators and the measures of distance d are highly coupled. Finally, using our approach, which we call *MA-sorting*, we improve previous results for a benchmark of experiments of the Error-Correcting Graph Isomorphism and for a benchmark of the *valued n -queens problem*. For these case studies our *MA-sorting* achieves a better quality-versus-effort trade-off. Our approach is specific to problems involving permutations, but in any other sense it is domain-independent. Therefore, it is applicable to the Error-Correcting Graph Isomorphism and the *valued n -queens problem*,

but also to other problems involving permutations (provided that for the problem in question a suitable pair: measure of presortedness and sorting algorithm, is defined).

Thus, we propose to guide a memetic algorithm with a new family of mutation operators. Our mutation operators perform local search following the path traced by classical sorting mechanisms. The comparison predicate and the evaluation function are made to correspond and guide the evolutionary search. Finally, for the practitioner, our research provides plenty of information about the design of our proposed *MA-sorting*.

Two techniques are commonly applied to improve the performance of local-search algorithms [1]: 1) Increasing the number of approximate solutions explored in the search space (the improvements are more evident when the operators generate a fruitful structure of the search space); and 2) Using a “good” approximate solution to start the exploration of the search space. Our research shows that *MA-sorting* improves its performance when we apply these techniques to the local searcher embedded in our *MA-sorting*.

Note that visiting more points of the search space will increase the quality of an approximation at the expense of more computational cost (in the extreme, exhaustive search explores all solutions and guarantees optimality, but is infeasible for all except trivially small search spaces). Thus, the challenge is to find the fruitful structure to the search space that achieves the most effective trade-offs in terms of quality of solution versus points explored (measured also as objective function evaluations). This is achieved in our case by establishing a method involving three processes. First, we identify a measure of disorder and algorithms that minimize this measure effectively.

Second, we establish that the optimization of the measure of disorder is in close association with optimizing the objective function. Finally, the third process merges the heuristic to optimize the measure of disorder as a local search into a Memetic Algorithm. We emphasize that classical sorting algorithms optimize measures of disorder. We identify three sorting algorithms that monotonically optimize a new measure of disorder. We also show that the measure of disorder has close association with our case studies. Thus, the effectiveness of *MA-sorting*. This connection to classical sorting opens new avenues for its application to other problems involving permutations as long as the corresponding disorder measure and sorting algorithm are identified.

Constraint combinatorial optimization problems pose an important question to the field of genetic and memetic algorithms: How should the concept of feasibility be handled? For memetic algorithms there are two general approaches for dealing with this issue: *Baldwinian* and *Lamarckian*. The former does not bring back genotypes improved by local search to the evolutionary cycle. As a result, without altering the genotype, the *Baldwinian* approach changes the fitness of the individual and modifies the fitness landscape [146]. If genotypes are modified by hybridization with local search, then the approach is *Lamarckian*. Thus, *Lamarckian* approaches encode the resulting improvements onto the genotype processed by the genetic algorithm. For constraint combinatorial optimization, in particular for the *valued n -queens problem* as case study, our research introduces the *Baldwinian* approach to handle the concept of feasibility [32]. Our approach, for a benchmark of experiments, is superior to the most recent proposal [120].

1.3 Thesis Layout

In order to offer a self-contained presentation of our research, Chapter 2 provides basic concepts about combinatorial optimization. In this chapter we also describe the basic mechanism of a genetic algorithm and we explain the most commonly applied genetic operators.

Chapter 3 describes in detail our local-search model and illustrates its application to the problem of clustering visitation paths of web-pages. In this chapter we demonstrate that the problem of clustering a matrix (a fundamental step in clustering visitation paths of web-pages) can be reduced to the *Longest Hamiltonian Path Problem*. For this problem we propose a novel approach that, for a benchmark problem suggested in the literature, is superior to the most recent approach.

Chapter 4 describes in detail the *Lamarckian* and *Baldwinian* approaches in memetic algorithms. There, we show that for the p -median problem, an NP-complete problem, our memetic algorithm is robust and effective and balances effort/quality of solution better than the plain GA. The p -median problem is a central facilities location problem that recently has been identified as a robust method for spatial classification, clustering and knowledge discovery.

Chapter 5 sets up the theory that supports our *MA-sorting*. There, we explain in detail how classical sorting, as a local-search mechanism, directs mutation. We also identify attributes and parameters which are relevant in the performance of *MA-sorting*. Finally, we describe in detail important issues for the design and implementation of our *MA-sorting*.

Chapter 6 provides the practitioner with more specific information about the impact, into the performance of *MA-sorting*, of some of the attributes and parameters described in Chapter 5. There, we provide substantial information that shows that, for our case study, our *MA-sorting* improves its performance: by increasing the number of approximate solutions explored in the search space; and by the use of a “good” approximate solution to start the exploration of the search space. We expect all this information will be helpful for the application of our *MA-sorting* to other permutation problems.

Chapter 7 deals with the problem of feasibility in constraint combinatorial optimization. There we show that a *Baldwinian* approach, in the context of the *valued n -queens problem*, provides much better results than the most recent approach, for a benchmark of experiments proposed in the literature.

Chapter 8 summarizes our conclusions and provides new avenues that deserve future research.

Chapter 2

Basic concepts

2.1 Combinatorial Optimization

Many problems of both practical and theoretical importance concern themselves with the choice of a “best” configuration amongst a set of parameters. In Combinatorial Optimization we are looking for an object from a finite, or possibly countably infinite, set (typically an integer, set, permutation or graph) [103].

More formally [1]:

Definition 1 *A combinatorial optimization problem is specified by a set of problem instances and is either a minimization problem or a maximization problem.*

If the search space consists only of feasible solutions we are dealing with an unconstrained combinatorial optimization problem. If the search space consists of both feasible and infeasible solutions we are dealing with a constraint combinatorial optimization problem (CCOP).

Figure 2.1 displays an example of the search space explored when solving a constraint combinatorial optimization problem.

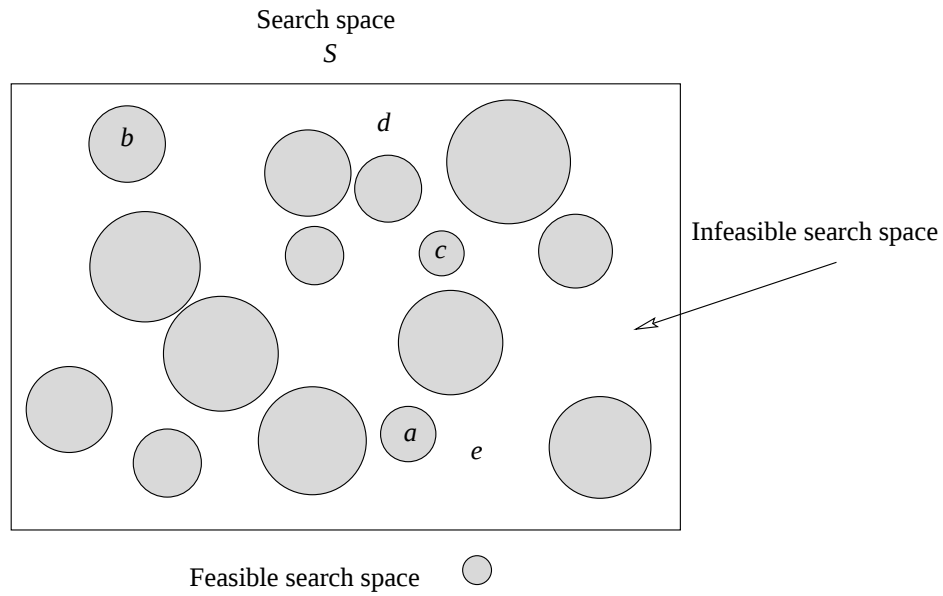


Figure 2.1: An example of CCOP search space S . Points a, b, c are in the feasible search space F and points d, e in the infeasible search space $S \setminus F$.

On one side, unconstrained combinatorial optimization can be defined as follows [1]:

Definition 2 An instance of an unconstrained combinatorial optimization problem is a pair (F, f) , where the solution set F is the set of feasible solutions and the objective function f is a mapping $f : F \rightarrow \mathbb{R}$. For a minimization problem, the purpose is to

find a globally optimal solution, i.e., an $s^ \in F$ such that $f(s^*) \leq f(s)$ for all $s \in F$. Furthermore, $f^* = f(s^*)$ denotes the optimal cost, and $F^* = \{s \in F \mid f(s) = f^*\}$ denotes the set of optimal solutions.*

On the other side, constraint optimization problems can be defined with the help of the concept of Constraint Satisfaction Problems.

Constraint Satisfaction Problems (CSPs) [135] have long been studied, mainly because many problems arising in Operations Research can be represented as CSPs. The general CSP is known to be NP-complete [23]. Additionally, when the search seeks an optimum feasible solution, the CSP is named a Constraint Optimization Problem [104].

In a Constraint Satisfaction Problem (CSP), the aim is to find values for a (finite) set of variables satisfying a (finite) set of constraints [135]. More formally [23], we consider a set Y of n variables y_i , where each variable y_i may assume values z_j from a given domain

$$Z_i = \{z_j \mid 1 \leq j \leq k_i\}; |Z_i| = k_i.$$

Definition 3 (FSS) *A Free Search Space is the Cartesian Product $S = Z_1 \times \cdots \times Z_i \times \cdots \times Z_n$.*

Definition 4 (CSP) *A Constraint Satisfaction Problem is a pair $\langle S, \phi \rangle$ where S is an FSS and ϕ is a formula (Boolean functions on S). A solution of a CSP is an $s \in S$ with $\phi(s) = \text{true}$.*

Definition 5 (COP) *A Constraint Optimization Problem is a triple $\langle S, f, \phi \rangle$, where S is an FSS, f is a (real valued) objective function on S and ϕ is a formula (Boolean function on S). A solution of a COP is an $s \in S$ with $\phi(s) = \text{true}$ and an optimal f -value.*

Definition 6 (CCOP) *A Constraint Combinatorial Optimization Problem is a COP where S is finite (typically because each Z_i is finite).*

Note that we have added, to the original definition, the adjective *Combinatorial* to emphasize the fact that we are not dealing with constraint numerical optimization problems [88].

Definition 7 (FeSS) *For CSPs and CCOPs we call ϕ the feasibility condition and the set $F = \{s \in S \mid \phi(s) = \text{true}\}$ will be called the Feasible Search Space.*

The subclass of CCOPs (and CSPs) which can be formulated as permutation problems require solutions that can be thought of as assigning a permutation of the values to the variables [124]. More formally:

Definition 8 (PP) *A Constraint Problem (CSP or CCOP) admits formulation as a Permutation Problem, provided that: $Z_1 = \dots = Z_i = \dots = Z_n$, and to each variable y_i a different value z_j is assigned. In this case, the search space S consists of integer permutations $\pi = \langle \sigma_1, \dots, \sigma_n \rangle$.*

2.2 Solving Techniques

Many combinatorial optimization problems (unconstraint and constraint) are *NP-hard* [38]. It is generally believed that *NP-hard* problems cannot be solved to optimality within polynomially bounded computation times.

Garey and Johnson [38] propose to divide the algorithms that deal with those hard problems into two groups. Those algorithms which perform a systematic search (or implicit enumeration) of the search space belong to the first group. Examples of algorithms that perform a systematic strategy are the methods based on search trees. For example in CSPs, at each step they assign a value to the current variable, and make sure that no constraint is violated. If this is not possible, they backtrack to revise previously assigned values. This process ends when a valid value has been assigned to every variable (or an exhaustive proof concludes no feasible assignment exists). Note, however, that if the CSP has n variables each with m possible values, the depth in the search tree is n , and up to m branches are created from each node. Hence, the tree has up to m^n leaf nodes. In consequence these methods are by nature exponential in time (even if the number of constraints is bounded by n and we assume evaluating a constraint or the objective function takes constant time). While acknowledging the apparent inevitability of exponential time complexity, some algorithms seek to obtain as much improvement over plain exhaustive search as possible. Amongst them are those based on branch-and-bound, dynamic programming, cutting plane and Lagrangian techniques.

Algorithms which belong to the second group are indistinctly called approximation algorithms [1], or heuristics [38]. These methods try to find a “good” solution, maybe not the optimal, within an acceptable length of time. They search among a collection of approximate solutions for a desired solution. While the most widely applied techniques are based on local search [2], they can be trapped in a poor local optimum. Simulated Annealing [68,96], Tabu search [42,43,116], Neural Networks [55,105] and Genetic Algorithms [52,92,93] try to remedy this drawback by broadening the scope of local search.

2.3 Genetic Algorithms

Genetic Algorithms [19, 44, 52, 89, 90, 139] (GAs) are general-purpose search methods inspired by biological evolution. They adopt from this discipline three basic principles:

1. Evolution operates on chromosomes with a measure of fitness attached to them.
2. There is a population of chromosomes which evolves using the genetic operators of crossover and mutation.
3. In accordance with the principle of Natural Selection, chromosomes with higher fitness will reproduce more frequently.

From the perspective of GAs, evolution can be conceived as a method of searching among a vast space of approximate solutions. In biology, this search space is the set of possible genetic sequences, and the desired solutions are those highly fit.

Genetic Algorithms, which were invented in the 1960s [44], have been used in two ways: as techniques for solving technological problems, and as simplified scientific models that can answer questions about nature. Our research focuses on the former use.

GAs can work well when the space to be searched is large, it is known not be perfectly smooth and unimodal, and it is not well understood; the fitness function is noisy; and the task does not require a global optimum to be found (i.e., if quickly finding a sufficiently good solution is enough).

Together, genetic algorithms, evolution strategies, evolutionary programming, genetic programming and memetic algorithms form the backbone of the field of evolutionary computation.

GAs provide a method for moving from one population of chromosomes to a new population by using a kind of Natural Selection together with the genetic-inspired operators of crossover and mutation. Each chromosome consists of genes, and one can think of a gene as encoding a trait. The different possible settings for a trait are called alleles. In GAs, the term *chromosome* refers to an approximate solution to the problem in question. If such an approximate solution can be expressed as a string of bits, the genes are single bits or short blocks of adjacent bits, and an allele in this “bit string” is either 0 or 1. Each chromosome can be thought of as a point in the search space of approximate solutions. Natural Selection is driven by the fitness of the chromosomes. In the context of Genetic Algorithms, fitness measures the quality of chromosomes as an approximate solution to the problem at hand. Thus, fitness influences the number of offspring the organism has.

The simplest form of genetic algorithm involves three operators: selection, crossover, and mutation.

The selection operator chooses those chromosomes in the population that will be allowed to reproduce. On average the fitter chromosomes produce more offspring than the less fit ones. Crossover exchanges subparts of two chromosomes, roughly mimicking biological recombination. Mutation randomly changes the allele values of some locations in the chromosome.

A simple GA works as follows:

1. Start with a randomly generated population of n l -bit chromosomes (approximate solutions).
2. Compute the fitness $\psi(x)$ of each chromosome ζ in the population.
3. Repeat the following steps until n offspring have been created:
 - (a) Select a pair of chromosomes from the current population. The probability of selection is an increasing function of fitness. Selection is done with replacement, meaning that the same chromosome can be selected more than once to become a parent.
 - (b) With probability p_c (the crossover probability), cross over the pair at a randomly chosen point (chosen with uniform probability) to produce two offspring. If no crossover takes place, form two offspring that are exact copies of their respective parents.

- (c) Mutate the two offspring at each locus with probability p_m (the mutation probability), and place the resulting chromosomes in the new population.

If n is odd, one new population member can be discarded at random.

4. Replace the current population with the new population.
5. Go to step 2.

Each iteration of this process is called a generation. A GA is typically iterated for 50 to 500 generations, or more. The entire set of generations is called a run. At the end of a run there are often one or more highly fit chromosomes in the population.

We describe now more in detail some of the most typical GA operators.

2.3.1 Selection Methods

The purpose of parent selection in a GA is to give more reproductive chances, on the whole, to those members of the population that are more fit. A simple method of implementing selection is a roulette-wheel sampling [44]. It consists in giving each individual a slice of a circular roulette wheel equal in area to the individual's fitness. Each time the roulette wheel is spun the corresponding individual is selected.

The method can be implemented as follows.

1. Sum the total expected value of individuals in the population. Call this sum W .
2. Repeat N times

- (a) Choose a random integer q between 0 and W .
- (b) Loop through the individuals in the population, summing the expected values, until the sum is greater than or equal to q . The individual whose expected value puts the sum over this limit is the one selected.

Although this selection procedure is random, for each parent the chance of being selected is directly proportional to its fitness.

A common problem in GAs, the so called premature convergence problem, is a premature loss of diversity in the population which increases the risk of converging to a sub-optimal solution. The purpose of Ranking Scaling is to prevent premature convergence. In the version proposed by Baker [4], the individuals in the population are ranked according to fitness, and the expected value of each individual depends on its rank rather than on its absolute fitness. Ranking avoids giving the far largest share of offspring to a small group of highly fit individuals, and thus reduces the selection pressure when the fitness variance is high. It also keeps up selection pressure when the fitness variance is low: the ratio of expected values of individuals ranked i and $i + 1$ will be the same whether their absolute fitness differences are high or low.

The linear ranking method proposed by Baker operates as follows. Each individual in the population is ranked in increasing order of fitness, from 1 to N . The user chooses the expected value Max of the individual with rank N , with $Max \geq 0$. The expected value of each individual i in the population at time t is given by

$$ExpVal(i, t) = Min + (Max - Min) \frac{ranking(i, t) - 1}{N - 1}$$

where Min is the expected value of the individual with rank 1. Given the constraints

$Max \geq 0$ and $\sum_i ExpVal(i, t) = N$ (since population size stays constant from generation to generation), it is required that $1 \leq Max \leq 2$ and $Min = 2 - Max$.

At each generation the individuals in the population are ranked and assigned expected values as explained. Baker recommends the use of a value of $Max = 1.1$ ($Min = 0.9$).

2.3.2 Crossover

Genetic Algorithms assume that high-quality parent-approximate solutions from different regions in the search space can be combined via crossover to, occasionally, produce high-quality offspring-approximate solutions.

In GA, crossover recombines the genetic material in two parents to make two children. Crossover is an important component of GAs. The so called One Point crossover is the simplest version of this operator.

The One Point crossover operator proceeds in three steps. In the first step, two individuals (parents) are chosen uniformly at random from the current population. In the second step, an integer position k along the string is selected, also uniformly at random between 1 and the string length l less one. In the third step, two new strings are created by swapping all the elements between positions $k + 1$ and l inclusive. For example, consider the following strings chosen from the current population, and position $k = 5$:

$$\begin{aligned}\zeta_1 &= 0 \ 1 \ 1 \ 1 \ 0 | 0 \ 0 \ 1 \ 0 \ 1 \\ \zeta_2 &= 1 \ 1 \ 0 \ 0 \ 1 | 1 \ 0 \ 0 \ 1 \ 1.\end{aligned}$$

The offspring that results from the One Point crossover is:

$$\begin{aligned}\zeta_1^o &= 0 \ 1 \ 1 \ 1 \ 0 | 1 \ 0 \ 0 \ 1 \ 1 \\ \zeta_2^o &= 1 \ 1 \ 0 \ 0 \ 1 | 0 \ 0 \ 1 \ 0 \ 1.\end{aligned}$$

2.3.3 Mutation

Bit mutation is the simplest form of mutation operator. Bit mutation flips each bit on the bit string if a probability test is passed. This probability parameter p_m is typically quite low. Observe, this mutation operator is a random walk through the string space.

This operator samples all the search space, but with much higher probability near the chromosome that is being mutated. For example, if we consider a binary chromosome of length l , with probability

$$\binom{l}{h} p_m^h (1 - p_m)^{l-h}$$

we obtain a chromosome at Hamming distance h from the original chromosome. This mutation operator is uniform anywhere in the search space and it is blind to what was the result of previous mutations. It is fine as an operator to preserve diversity, and in order not to radically disrupt the current solutions and upset convergence, it has a bias to produce chromosomes near the originator (specially since p_m is much closer to 0 than to 1, and $1 - p_m$ is much closer to 1).

2.4 Order-Based Genetic Algorithms

As for any search and learning method, the way in which approximate solutions are encoded is a central factor in the success of a genetic algorithm. Lawrence Davis [19] strongly advocates using whatever encoding is the most natural for the problem in question, and then devising a GA that can use that encoding.

In GAs, the idea of searching for permutations has been present since 1985 [45]. Since naively crossing or mutating the chromosomes of permutations can produce invalid representations, compatible *order-based* operators are designed to assure that the offspring is a valid representation of a permutation. There are several proposals for these order-based operators [127]. In the case of crossover, variants try to preserve ordering properties (adjacency, absolute position, or relative order of elements) of the parents to be passed to the offspring. While for some problems it is possible to observe differences by using different crossover operators [127], for other problems there are no noticeable differences. For example, for the Error-Correcting Graph Isomorphism Problem, which is described in detail in Chapter 6, Wang *et al.* [141] performed several experiments and they could not detect relevant differences by using three different types of crossover operators: Partially-Mapped crossover (PMX), Order crossover, and Cycle crossover. For this problem they adopt the PMX operator.

Compatible order-based operators are those that preserve a valid representation of a permutation. We briefly describe now two of the most commonly used order-based operators.

2.4.1 The PMX Crossover Operator

The Partially-Mapped Crossover (*PMX*) operator proposed by Goldberg and Lin-
gle [45] exploits similarities in ordering among different strings. This crossover oper-
ator for chromosomes that encode permutations is widely used.

PMX proceeds as follows. First, two positions are chosen along the string uniformly
at random. The substrings in between the positions chosen are called the Mapping
Sections. Next, the Mapping Sections of parents are exchanged. Because the permu-
tations that result from this operation are not valid, we swap the rest of the elements
as prescribed by the Mapping Sections.

For example, given two parent permutations:

$$\begin{aligned}\zeta_1 &= 2 \ 4 \ 1 \ 3 \ 5 \ 10 \ 8 \ 7 \ 9 \ 6 \\ \zeta_2 &= 1 \ 6 \ 8 \ 2 \ 4 \ 3 \ 9 \ 10 \ 5 \ 7\end{aligned}$$

We generate two random positions, for example 5 and 7.

$$\begin{aligned}\zeta_1 &= 2 \ 4 \ 1 \ 3 \ |5 \ 10 \ 8| \ 7 \ 9 \ 6 \\ \zeta_2 &= 1 \ 6 \ 8 \ 2 \ |4 \ 3 \ 9| \ 10 \ 5 \ 7\end{aligned}$$

We exchange the Mapping Sections between parents.

$$\begin{aligned}\zeta_1^o &= 2 \ 4 \ 1 \ 3 \ |4 \ 3 \ 9| \ 7 \ 9 \ 6 \\ \zeta_2^o &= 1 \ 6 \ 8 \ 2 \ |5 \ 10 \ 8| \ 10 \ 5 \ 7\end{aligned}$$

To produce a valid offspring we perform swaps as prescribed by the mapping section.
In the example, to produce the first child the Mapping Section indicates that: $4 \rightarrow 5$,

$3 \rightarrow 10$ and $9 \rightarrow 8$. To produce the second child we use the inverse of that mapping:
 $5 \rightarrow 4$, $10 \rightarrow 3$ and $8 \rightarrow 9$.

$$\begin{aligned}\zeta_1^o &= 2 \quad 5 \quad 1 \quad 10 \quad 4 \quad 3 \quad 9 \quad 7 \quad 8 \quad 6 \\ \zeta_2^o &= 1 \quad 6 \quad 9 \quad 2 \quad 5 \quad 10 \quad 8 \quad 3 \quad 4 \quad 7\end{aligned}$$

2.4.2 An Order-Based Mutation Operator

For mutation, several compatible order-based operators have been proposed [19]. One of the simplest mutation operators for permutations is the swap mutation operator [19]. This operator chooses a uniform randomly pair of positions in the chromosomes and swaps their contents. For example, consider the following permutation and positions 4 and 8

$$\zeta = 2 \quad 4 \quad 1 \quad \mathbf{3} \quad 5 \quad 10 \quad 8 \quad \mathbf{7} \quad 9 \quad 6.$$

After performing swap mutation, we obtain,

$$\zeta = 2 \quad 4 \quad 1 \quad \mathbf{7} \quad 5 \quad 10 \quad 8 \quad \mathbf{3} \quad 9 \quad 6.$$

However, this operation often produces a chromosome with worse performance. This is not considered a problem because traditionally the mutation operator has been conceived as a diversity generator.

2.4.3 Constraint Optimization Problems: a challenging assignment for GAs and MAs

Simple genetic algorithms are not usually considered as good as either local-search mechanisms or domain-specific methods. However, the integration of genetic algorithms with local search [49, 92, 93, 137] (integration that is often known as memetic algorithms) provides much better results. It has been empirically shown that memetic algorithms (MAs) can provide a competitive performance when dealing with hard combinatorial optimization problems [85, 86, 130]. According to the way of handling the improvement that results from its local search component, memetic algorithms are classified as *Lamarckian* or *Baldwinian*.

Since for constraint combinatorial optimization it is typically difficult to define the local search around an approximate solution, these problems pose additional difficulties to GAs and MAs. We introduce some of those difficulties in the next section.

CSPs and CCOPs have the common reputation of being GA-hard [23]. Most of them have a feasible search space F that it is not easy to characterize. It is also typically difficult to define an operator ω_1 such that $\omega_1(a)$ produces a feasible assignment, given an element a from the feasible search space F (refer to Figure 2.1). It then becomes more difficult to define $\omega_2(a, b)$ to produce a feasible solution for $a, b \in F$. Operators ω_1 where the Markov chain $\omega_1^t(a) = \omega_1^{t-1}(\omega_1(a))$ samples F result in the application of Simulated Annealing, while operators $\omega_2(a, b)$ may constitute themselves into the crossover operators and operators $\omega_1(a)$ into mutation operators for Genetic Algorithms.

The first line of attack is naturally a genotype representation that enforces a large number of the constraints (minimizes the size of the set $S \setminus F$ of points of the search space that do not satisfy the constraints). In fact, if the encoding could represent exactly and uniformly the search space of feasible solutions F , the problem could become GA-easy (there would be no issue about genetic operators producing infeasible offspring). Thus, a more computational definition of genotype is the structure that is the input and output of genetic operators and is maintained as representation of individuals in the simulated evolution cycle of the GA.

Since $S \setminus F$ is typically non-empty, the usual next step is the transformation of feasibility from a qualitative concept into a quantitative concept. In the quantitative approach a function $\rho : S \rightarrow \mathbb{R}^+$ is applied to obtain the feasibility value of a given genotype. In the context of GAs, this function is called a penalty function [114] and attempts to measure how far $d \in S \setminus F$ is from F (typically, a feasible genotype has no penalty, i.e. $\rho(a) = 0$ if $a \in F$). Approaches based on penalty functions differ in how the penalty function is designed and applied to infeasible genotypes [88]. The evaluation function ψ , used by the GA to compute the fitness of individuals in the population, is a combination of the penalty function ρ and the objective function f [88].

Once the problem has been transformed (to handle feasibility as a quantitative concept), the basic GA mechanism can be enhanced in two ways to encourage feasible solutions:

1. Using operators that try to create children that are “less infeasible” than the parents [88, 104, 120]; that is, find $\omega(\cdot, \cdot)$ such that $\rho(\omega(a, b)) \leq \rho(a)$ and $\rho(\omega(a, b)) \leq \rho(b)$.
2. Fixing individuals if they are not feasible [22]; that is find $\omega(\cdot)$ such that $\rho(\omega(a)) = 0$.

However, these two approaches can be considered very similar. Namely, if $\times(\cdot, \cdot)$ is the genetic operator in the GA plan that produces infeasible genotype, and we have a fixing operator $\omega(\cdot)$ from 2 above, then $\omega(\times(\cdot, \cdot))$ is the genetic operator that creates children that are less infeasible, as required by 1.

Also, a drawback of these approaches is that either the attempt to land close to F (although in $S \setminus F$), or map $S \setminus F$ to F , could affect necessary properties of genetic operators (like assortment [107]).

Other problems of the first case, are that general operators [104, 120] do not guarantee to produce only feasible solutions (the output of their operators may contain infeasible assignments, even if the input is feasible). Also, specialized operators are conceivable only for severely restricted contexts [88].

In the second case, the challenge is the design of a fixing mechanism. In fact, the approach is applied as a *Lamarckian* approach that modifies genotypes. *Lamarckianism* enjoys particular popularity in the evolutionary computation community because it has been relatively easy to fix an infeasible genotype for some classical combinatorial optimization problems (e.g., traveling salesman problem, knapsack problem, set covering problem) [88]. Unfortunately, for most CSPs and CCOPs such design is

a really hard task. However, it is viable to build mechanisms that create partially feasible solutions [22]. Our approach for constraint optimization problems, which is detailed in Chapter 7, belongs to this second kind of enhancement, but in *Baldwinian* style, so we avoid the assortment problem. With a *Baldwinian* approach we do not need to change the original crossover and mutation operators.

Chapter 3

Local Search

3.1 Local Search for Combinatorial Optimization

Local Search (LS) can deal successfully with hard combinatorial optimization problems [1]. In such problems, the search space is the set S of approximate solutions and the purpose is to find a solution $i^* \in S$ such that, given an objective function $f : S \rightarrow \mathbb{R}$, the value $f(i^*)$ is globally optimal. LS algorithms iteratively search for a better solution in the local neighborhood of a current solution i . The neighborhood $N \subseteq S$ of a LS algorithm is the set of solutions that can be reached from the current solution i by the application of a neighborhood operator $\nu : S \rightarrow 2^S$. LS mechanisms are approximation algorithms with the common feature of an underlying neighborhood operator $\nu(i)$ which is used to guide the search.

LS comprises four basic steps: *Initialization*, *Nomination*, *Comparison* and *Termination*. An outline of the basic LS mechanism is displayed in Fig. 3.1. A well known disadvantage of LS is that it can be trapped in local optima potentially far away from the global optimum.

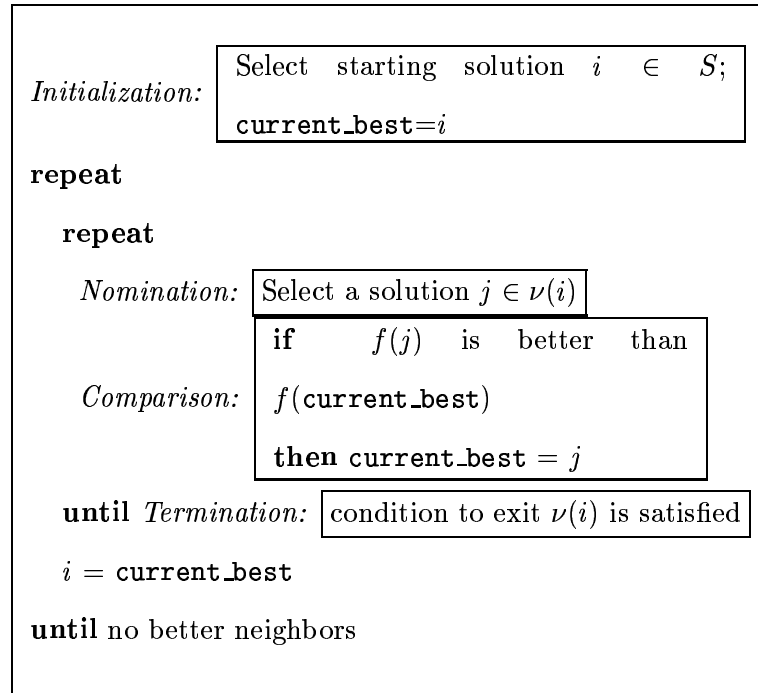


Figure 3.1: Pseudo-code for Algorithm Local Search (LS).

The starting solution selected during the *Initialization* step may impact on the quality of the solution obtained. For that reason, it is common to execute LS several times using different starting solutions. When the starting solution is randomly generated, this approach is called *multi-start local search* [1]. If the starting solution is the result of the perturbation of a previously obtained local optimum, the approach is

called *iterated local search* [1]. The latter has been recognized as the most effective approximate approach to the Traveling Salesman Problem [1].

During the *Nomination* step, the selection of a point from the neighborhood $\nu(i)$ may be complex, particularly if we are performing a systematic nomination of neighbors. In this case, we maintain either a record of some of the previous nominations, as for example tabu search does, or the state of the last nomination, if the itinerary of nominations is deterministic (nomination becomes an enumeration). Therefore, ν determines a neighborhood structure of the search space S and we consider that nomination may establish an order for nominating solutions. Finding efficient and effective neighborhood operators and nomination schedules which lead to high-quality solutions is one of the challenges of LS [1].

In what follows we consider $\nu(i)$ not only as a set of neighbors for i , but as a nomination schedule.

In some cases the criterion used in the *Comparison* step is more elaborated. For example, simulated annealing uses randomized thresholds as the comparison criterion. This allows for a solution that is not the current best to be adopted as the current solution to continue the search.

The *Termination* condition may offer several options: stop as soon as a better neighbor solution is found (*first improvement*); stop after all the neighbors of the current solution i have been evaluated (*best improvement*); or stop after performing a previously defined number of nominations (budget of neighbors). Observe that the order of nominations prescribed by ν is relevant, for example, when *first improvement* has been selected as the *Termination* condition.

As an illustration of a LS mechanism, we describe in detail our method of dealing with the *Longest Hamiltonian Path Problem* in the context of clustering visitation paths of web-pages.

3.2 LS for the *Longest Hamiltonian Path Problem* in Clustering Visitation Paths of Web-Pages

The clustering of a matrix [81] has been applied to deal with two important practical problems: the design of distributed database systems [102] and the design of web sites [149]. A distinctive characteristic of these applications is the large amount of data they handle under highly dynamic environments. A key issue in the performance of these applications is the use of fast algorithms.

On the one hand, the design of distributed database systems requires the decomposition of a relation into fragments of attributes (vertical fragmentation). This decomposition permits us to execute concurrently a single query, by dividing it into a set of sub-queries that operate on fragments. Each fragment can potentially be allocated to different sites. The “optimal” fragmentation minimizes the execution time of user applications that run on these fragments. However, if a relation has κ non-primary key attributes the number of possible fragments is equal to $\varphi(\kappa)$, which is the κ^{th} Bell number. For large values of κ , $\varphi(\kappa) \approx \kappa^\kappa$. Because of that, approximate algorithms are recommended [102]. A typical method for the decomposition of a relation into fragments is based on clustering a matrix of attributes. The purpose of the clustering is to identify how closely related the attributes are, based on the

access frequency of applications. The preferred method for clustering such a matrix is the Bond Energy Algorithm (*BEA*), proposed by McCormick *et al.* [81].

On the other hand, the identification of similarities between web-users is relevant for the design of web-sites. For example, based on visitation paths, the generation of dynamic hypertext links could benefit from clustering web-users [149]. Different criteria of similarity can be used for clustering web-users. These may be based on the order of visits, on the frequency of visits or on the visit of similar web-pages. Such measures of similarity can be represented in matrix form. Xiao *et al.* [149] propose to cluster a matrix which represents similarities between web-users using a variant of the algorithm proposed by McCormick *et al.*

The purpose of *BEA* is to identify and display clusters that occur in matrices [81]. This method has been shown to be successful in many applications [81, 102, 149] and several implementations have been proposed in the literature [81, 102, 149] (we offer a detailed description of them). In particular, we observe that differences in implementation are precisely reflected in the underlying neighborhood operator ν . Also, the trade-off between quality of solutions and computational effort is not the same.

We show that a critical step in clustering a matrix, based on the measure of effectiveness proposed by McCormick *et al.*, can be modeled as a *Longest Hamiltonian Path Problem* [38]. Based on this model, we show that complementing *BEA* with the 2-opt algorithm [16] provides better results than the most recent approach [149].

3.2.1 A Measure of Effectiveness

BEA is a *constructive* approach which works on a nonnegative $m \times n$ matrix A . Each entry in A stores a value of similarity between elements. The mechanism is guided by a measure of effectiveness e which reaches larger values when the input matrix possesses dense clumps of numerically large elements. The effectiveness is low when permutations of rows and columns distribute large elements more uniformly throughout the matrix. McCormick *et al.* recommend as a measure of effectiveness

$$e(A) = \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^n a_{i,j} (a_{i,j+1} + a_{i,j-1} + a_{i+1,j} + a_{i-1,j}), \quad (3.2.1)$$

where $a_{0,j} = a_{m+1,j} = a_{i,0} = a_{i,n+1} = 0$.

The problem is finding a permutation of rows and columns such that $e(A)$ is maximized.

While in this formulation the *bond* strength between two nearest-neighbors is defined as their product, other alternatives exist (like the product of some power q of them, such as $(a_{i,j})^q$). However, other alternatives besides $e(A)$ to formalize finding groups in a matrix of values have now been historically discarded because the measure of effectiveness $e(A)$ as defined by McCormick *et al.* has some important theoretical and computational advantages [81]:

- It is applicable to matrices of any size or shape.
- Since the vertical *bonds* are unaffected by interchanging the columns (the same reasoning applies for horizontal *bonds* and rows), the objective function $e(A)$

decomposes into two parts: one that depends only on row permutations; and the other that depends only on column permutations. For that reason the problem can be reduced to two separate and similar optimizations problems (both are solved by the same procedure).

- Since the contribution to $e(A)$ from any column is only affected by the two adjacent columns (and likewise, the contribution to $e(A)$ from any row is only affected by the two adjacent rows), the optimization of $e(A)$ is suitable for local search where the exploration neighborhood is defined by swaps of adjacent columns or swaps of adjacent rows.

3.2.2 The Optimization of $e(A)$ can be Modeled as a *Longest Hamiltonian Path Problem*

As established in Section 3.2.1, the problem can be reduced to two separate and similar optimizations, one for the columns and the other for the rows. For that reason, and without loss of generality, we simplify our exposition by considering a square symmetric matrix A .

In this case, and following Özsu and Valduriez [102], the *bond* strength (when defined as a product of two nearest-neighbors) is $bond(i, j) = \sum_{z=1}^n (a_{i,z})(a_{z,j})$. We observe that $bond(i, j)$ corresponds to the $b_{i,j}$ element of a matrix B that results from multiplying A by itself. That is $B = A \times A$. Therefore, the problem of maximizing Equation (3.2.1) can be rewritten as maximizing

$$e(B) = \frac{1}{2} \sum_{i=1}^n b_{i,i-1} + b_{i,i+1}. \quad (3.2.2)$$

Recall that the problem is to find a permutation of rows and columns which maximizes e (in a symmetric matrix the same permutation is applied to rows and columns). Given a permutation $\pi = \langle \sigma(1), \sigma(2), \dots, \sigma(n) \rangle$ of rows and columns of B we have

$$e(B, \pi) = \frac{1}{2} \sum_{i=1}^n b_{\sigma(i), \sigma(i-1)} + b_{\sigma(i), \sigma(i+1)}. \quad (3.2.3)$$

The elements immediately down and above the main diagonal of B (what Croes [16] called the contra-slant and the slant of a matrix, respectively) are the terms in the sum of Equation (3.2.3). But for symmetric matrices, the sum of the slant is equal to the sum of the contra-slant. Thus we have $\sum_{i=2}^n b_{\sigma(i), \sigma(i-1)} = \sum_{i=1}^{n-1} b_{\sigma(i), \sigma(i+1)}$, and the problem of maximizing Equation (3.2.3) is the problem of maximizing

$$e(B, \pi) = \sum_{i=1}^{n-1} b_{\sigma(i), \sigma(i+1)}. \quad (3.2.4)$$

Note that B can be conceptualized as the adjacency matrix of a weighted graph $G = (V, E)$. In such a graph, a path which includes a sequence of vertices $b_1, \dots, b_n$ is called a *Hamiltonian Path* if each b_i is different and $V(G) = \{b_1, \dots, b_n\}$. This is a well-known *NP*-complete problem [38]. If the edges $E(G)$ are weighted, we may look for the shortest or the longest path in the graph G . According to Equation (3.2.4) the final solution must include all vertices of G and, because we are looking for a permutation of vertices, each vertex in the final solution is different. In conclusion, the maximization of $e(A)$ reduces to the *Longest Hamiltonian Path Problem*.

3.2.3 Different Implementations of *BEA*

The heuristic usually referred as *BEA* has been implemented in different ways by several authors [81,102,149]. The results obtained, in terms of quality of solutions and computational effort, are dissimilar; they depend on the particular implementation. For that reason we consider it relevant to examine each one in more detail.

The implementations of McCormick *et al.* and Özsu and Valduriez can be conceived as *constructive* mechanisms that iteratively attempt to complete a partially defined permutation. Let $\langle \sigma_1, \sigma_2, \dots, \sigma_i \rangle$ be a partially defined permutation where $i < n$. The body of the iteration attempts to extend the current partial assignment by determining values for the $(i + 1)$ -th element and beyond. The resulting (complete) permutation is used in Equation (3.2.4) to compute the length of the *Hamiltonian Path* (or equivalently the value of $e(B, \pi)$).

Note that in a maximization problem, the *Comparison* step of Local Search tests if $f(j) > f(\text{current_best})$. This is also checked if $f(j) - f(\text{current_best}) > 0$. In many optimization problems it is computationally easier to evaluate the gradient

$$\Delta(j, \text{current_best}) = f(j) - f(\text{current_best})$$

and test if it is larger than 0. This is because j and current_best are neighbors, so they are not very different, and many terms in $f(j)$ and $f(\text{current_best})$ are the same. These common terms cancel in $\Delta(j, \text{current_best})$ making calculation of $\Delta(j, \text{current_best})$ faster than two evaluations of the objective function.

In implementing the proposals of McCormick *et al.* and Özsu and Valduriez we follow the gradient approach developed by the latter authors. The current solution is $\langle \sigma_1, \dots, \sigma_i \rangle$ ($i < n$) and its neighbors are all the permutations that place the k -th column between the j -th column and the $(j + 1)$ -th column ($j = 0, \dots, i$ and $i < k \leq n$, the case $j = 0$ places the k -th column as the first column). The value of the objective function in the current solution is $e(B, \langle \sigma_1, \dots, \sigma_i \rangle)$ while the value of a neighbor is given by $e(B, \langle \sigma_1, \dots, \sigma_j, k, \sigma_{j+1}, \dots, \sigma_i \rangle)$. However, in general only the columns σ_j, σ_{j+1} stop interacting while we have interactions between the j -th and the new arriving k -th column as well as the k -th column and the $(j + 1)$ -th column. Thus the gradient for $j \geq 1$ and $j < i$ is

$$\Delta(\pi, j, k) = b_{\sigma(j),k} + b_{k,\sigma(j+1)} - b_{\sigma(j),\sigma(j+1)}.$$

Because we can have $j = 0$ or $j = i$, these boundary cases must be handled differently as illustrated in Procedure 1.

Procedure 1 *Gradient by inserting column k between column j and column $j + 1$:*

$\Delta(\pi, j, k)$

if $j < 1$ **then** $\{ /* j \text{ must be } 0 */ \}$

$\Delta = b_{k,\sigma(j+1)} \{ /* b_{k,\sigma(1)} */ \}$

else if $(j + 1) > \text{the length of the partially defined permutation } \sigma$ **then** $\{ /* j = i */ \}$

$\Delta = b_{\sigma(j),k} \{ /* b_{\sigma(i),k} */ \}$

else

$\Delta = b_{\sigma(j),k} + b_{k,\sigma(j+1)} - b_{\sigma(j),\sigma(j+1)}$

end if

return Δ

The Description of McCormick *et al.*

We reproduce here the description given by McCormick *et al.* In this description, ME is their notation for $e(A)$.

“The algorithm is as follows.

1. Place one of the columns arbitrarily. Set $i = 1$.
2. Try placing individually each of the remaining $n - i$ columns in each of the $i + 1$ possible positions (to the left and right of the i columns already placed), and compute each column’s contribution to ME. Place the column that gives the largest incremental contribution to ME in its best location. Increment i by 1 and repeat until $i = n$.
3. When all the columns have been placed, repeat the procedure on the rows. (The row placement is unnecessary, however, if the input array is symmetric, since the final row and column ordering will be identical).”

Figure 3.2 displays the tests (gradients) evaluated by this version of the *BEA* algorithm, that we call *BEAm*, when iterating (incrementing i to $i + 1$ during the column optimization phase). This figure shows the process of testing the insertion of column k (which is taken as one in the list of $n - i$ columns not already placed) to the left and right of the i columns already placed. We compute the value of $\Delta(\pi, j, k)$ each time an insertion is tested. After testing all $n - i$ columns, the insertion that produces the highest value of $\Delta(\pi, j, k)$ ($j = 0, \dots, i$) is carried out and the column that has been just inserted is deleted from the list of columns not already placed. At this point, the number of columns already placed has been increased by one and the number of columns not already placed has been decreased by one. We repeat this process with a new column k taken from the list of columns not already placed. The algorithm finishes when the list of columns not already placed is empty.

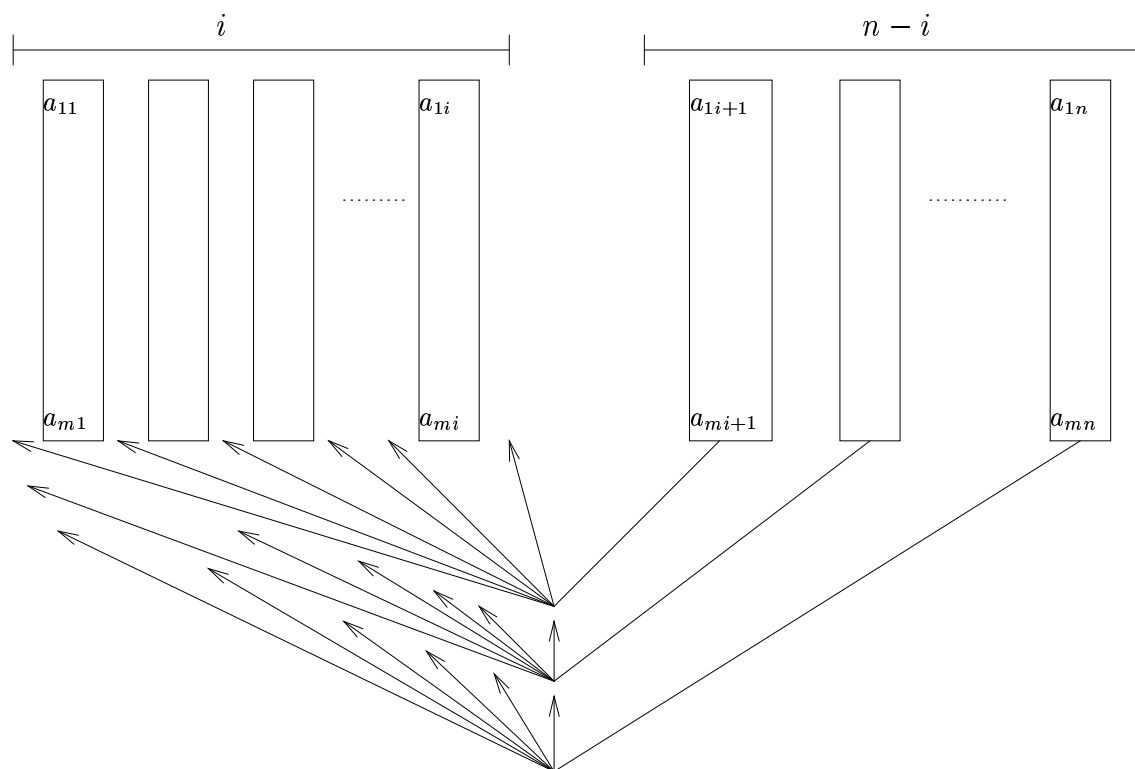


Figure 3.2: *The iteration of the BEAm algorithm of McCormick et al. tests each column k in the remaining $n - i$ columns for placement in each of the $i + 1$ positions amongst the i already placed columns.*

An important characteristic of this approach is that the final groupings are insensitive to the order in which items are presented to the algorithm [81, 102].

Our implementation of *BEAm*, as described by McCormick *et al.* [81], is delineated in Algorithm 2.

Algorithm 2 *Implementation of BEAm (McCormick et al.)*

```

1: input:  $B$ 
2: output:  $\pi = \langle \sigma_1, \dots, \sigma_n \rangle$  which maximizes  $e(A)$ 
3: for  $i$  from 1 to  $n$  by 1 do
4:   for all columns  $k$  not already included in  $\pi$  do
5:     for  $j$  from 1 to  $i + 1$  by 1 do
6:       compute  $\Delta(\pi, j - 1, k)$ 
7:     end for
8:   end for
9:    $loc \leftarrow$  placement  $j$  that gives the maximum gradient value
10:   $col \leftarrow$  column  $k$  that gives the maximum gradient value
11:  for  $l$  from  $i$  to  $loc$  by  $-1$  do
12:     $\sigma_l \leftarrow \sigma_{l-1}$ 
13:  end for
14:   $\sigma_{loc} \leftarrow col$ 
15: end for

```

The Description of Özsu and Valduriez

Özsu and Valduriez describe a slightly modified version of *BEA* (that we call *BEAo*). We reproduce here their description [102]. In Özsu and Valduriez’s description, global affinity measure is their notation for $e(A)$, and AA is their notation for A . While CA stands for the matrix with permuted columns.

“Generation of the clustered affinity matrix (CA) is done in three steps:

1. *Initialization.* Place and fix one of the columns of AA arbitrarily into CA . Column 1 was chosen in the algorithm.
2. *Iteration.* Pick each of the remaining $n - i$ columns (where i is the number of columns already placed in CA) and try to place them in the remaining $i + 1$ positions in the CA matrix. Choose the placement that makes the greatest contribution to the global affinity measure described above. Continue this step until no more columns remain to be placed.
3. *Row ordering.* Once the column ordering is determined, the placement of the rows should also be changed so that their relative positions match the relative positions of the columns.”

Özsu and Valduriez [102, pp136-137] give pseudo-code that details this procedure. This detailed pseudo-code describes the second step as a test for placing the current ($i + 1$ -th) column in its best place. In contrast, the algorithm of McCormick *et al.*, as described previously, asks for placing the best column in its best place. This means that the description of Özsu and Valduriez utilizes a different approach for selecting the column to be inserted. While McCormick *et al.* select the best amongst the $n - i$ elements, Özsu and Valduriez only test columns incrementally. This accounts for the lower time complexity of this version of the algorithm.

Figure 3.3 displays the iteration step of the *BEAo* algorithm during the column optimization phase as detailed by Özsu and Valduriez. It shows the process of testing the insertion of the $(i + 1)$ -th column (the first in the list of $n - i$ columns not already placed), to the left and right of the i columns already placed. Observe that in this case, in contrast with the implementation of McCormick *et al.*, they do not test all $n - i$ columns per iteration. While in the implementation of McCormick *et al.* the purpose is to find the best pair column-location, the implementation of Özsu and Valduriez asks only for the best location for the $(i + 1)$ -th column. They compute the value of $\Delta(\pi, j, i + 1)$ each time an insertion is tested. The insertion that produces the highest value of $\Delta(\pi, j, i + 1)$ ($j = 0, \dots, i$) is carried out and the column that has just been inserted is deleted from the list of columns not already placed. The number of columns already placed has been increased by one and the number of columns not already placed has been decreased by one. This process is repeated with the next column from the list of columns not already placed. The algorithm finishes when the list of columns not already placed is empty.

The differences between the approaches of McCormick *et al.* and Özsu and Valduriez are more evident if we compare Algorithm 2 and Algorithm 3.

The Description of Xiao *et al.*

A variant of *BEA* [148] (that we call *BEAx*) is applied by Xiao *et al.* [149] to group web-users based on visitation paths. We reproduce here their description. In the description of Xiao *et al.*, GA (global affinity) is their notation for $e(A)$.

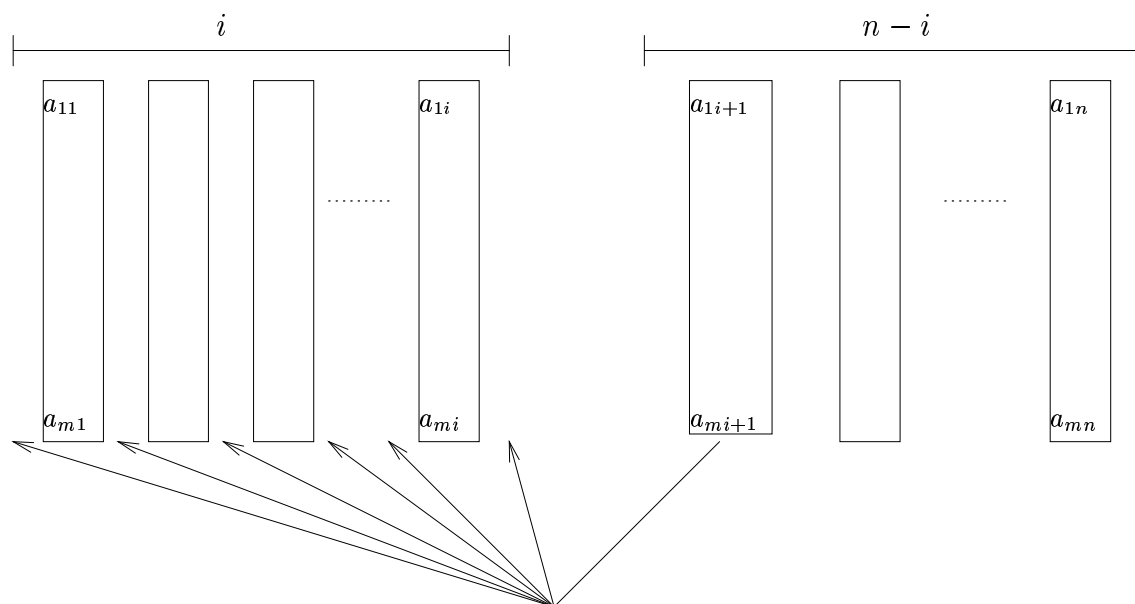


Figure 3.3: *The iteration of the BEAo algorithm tests the $(i + 1)$ -th column for placement in each of the $i + 1$ positions amongst the i placed columns.*

Algorithm 3 BEAo (*Özsu and Valduriiez*)

```
1: input:  $B$ 
2: output:  $\pi = \langle \sigma_1, \dots, \sigma_n \rangle$  which maximizes  $e(A)$ 
3: for  $i$  from 1 to  $n$  by 1 do
4:   for  $j$  from 1 to  $i + 1$  by 1 do
5:     compute  $\Delta(\pi, j - 1, i)$ 
6:   end for
7:    $loc \leftarrow$  placement  $j$  that gives the maximum  $\Delta$  value
8:   for  $l$  from  $i$  to  $loc$  by  $-1$  do
9:      $\sigma_l \leftarrow \sigma_{l-1}$ 
10:  end for
11:   $\sigma_{loc} \leftarrow i$ 
12: end for
```

“In order to get a higher global affinity, we permute the rows and columns of the matrix. The permutation procedure starts from the first column. It compares the GAs by swapping the positions of every pair of columns. If the GA becomes higher after swapping, the two columns swap their positions (the rows should be swapped accordingly to maintain the same relative positions as columns); otherwise the positions of the two columns remain unchanged.

When the permutation is complete, the global affinity of the matrix is maximized. That is, those closely related users are located closely to each other in the matrix”

Note that the implementation of *BEA* of Xiao *et al.*, in contrast with the implementations of McCormick *et al.* and Özsu and Valduriez, does not iteratively attempt to complete a partially defined permutation.

Figure 3.4 displays the body of the *BEAx* algorithm during the column optimization phase as implemented by Xiao *et al.* It shows the process of swapping. Starting with column $i = 1$, the algorithm tests the swap of columns i and $i + 1$, then the swap of columns $i + 1$ and $i + 2$, and so on until testing the swap of columns $n - 1$ and n . If any of those swaps improves the current solution the swap is confirmed. A second pass over the columns begins with column $i = 2$ following the same pattern of swappings. Each pass ι sets the index i of the starting column to ι . The operation of passing over stops when the index i of the starting column reaches the value n . The whole process is repeated n times.

Our implementation of this mechanism is described in Algorithm 4.

It is important to mention that, in order to validate the correctness of our interpretation, we executed *BEAx* with the same input data as Xiao *et al.* and we obtained the same results than those reported by the authors [149, page 111].

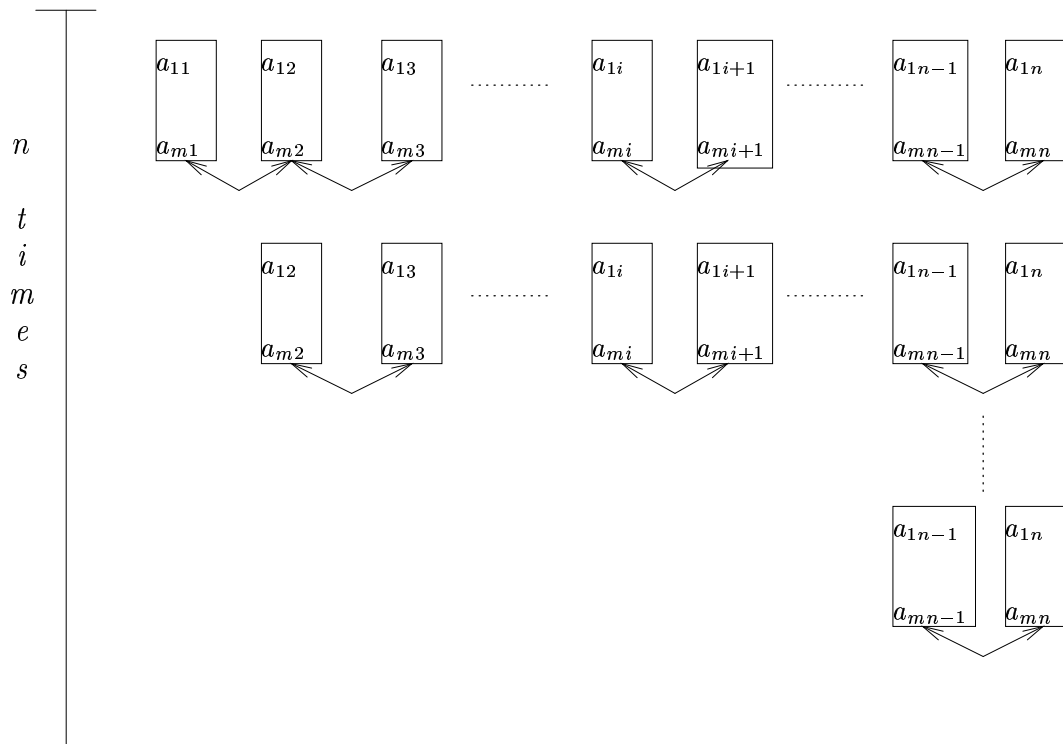


Figure 3.4: *The body of the BEAx algorithm during the column optimization phase as implemented by Xiao et al.*

Algorithm 4 *Our implementation of BEAx (Xiao et al.)*

```

1: input:  $B, \pi$ 

2: output:  $\pi = \langle \sigma_1, \dots, \sigma_n \rangle$  which maximizes  $e(A)$ 

3: copy  $\pi$  to a temporary  $\pi_{old}$ 

4: for  $i$  from 1 to  $n$  by 1 do

5:   for  $j$  from 1 to  $n$  by 1 do

6:     for  $k$  from  $j$  to  $n - 1$  by 1 do

7:        $swap(\sigma_k, \sigma_{k+1})$  of  $\pi_{old}$ 

8:       compute  $\Delta(\pi_{old}, \pi)$ 

9:       if  $\Delta > 0$  then

10:        confirm swapping of  $\pi_{old}$ 

11:       else

12:        discard swapping

13:       end if

14:     end for

15:   end for

16: copy  $\pi_{old}$  to  $\pi$ 

17: end for

```

Even though *BEAo* is cheaper than *BEAm* in terms of time complexity, which is confirmed by our experimental results (see Section 3.2.6), we found that the quality of solutions provided by *BEAm* are better than those provided by *BEAo* and *BEAx*. The point here is: How could we increase the quality of the solutions without sacrificing speed?

3.2.4 Our Approach

McCormick *et al.* indicate that *BEA* can be rewritten as a *quadratic assignment problem* (QAP) [81]. However, they found that specialized algorithms for QAP do not exploit the nature of $e(A)$. In particular, we have seen that calculating $\Delta(\pi, j, k)$ is an order of magnitude faster than evaluating $e(A)$ (or $e(B, \pi)$). Moreover, *BEA* can be expressed as a *traveling salesman problem* [39, 71, 72], but so far, historically, the method proposed by McCormick *et al.* has been preferred when dealing with clustering a matrix [102, 149]. We consider *BEA* as a good starting point when looking for better results, but we also integrate local search operators used for searching permutations in problems like TSP and *Longest Hamiltonian Path Problem*.

Our approach is based on two well-known results. On one hand, as explained in Section 3.1, the starting solution selected during the *Initialization* step may impact on the quality of the solution obtained. On the other hand, as Fredman *et al.* [37] argue, the best approximation algorithms for the TSP underlie a two-step procedure. In the first step, a starting solution is obtained by a constructive heuristic. The second step improves this solution (we use the term *composite* algorithm for this sequential application of both steps).

Recall that in Section 3.2.2 we showed that clustering a matrix using Equation (3.2.1) can be modeled as a *Longest Hamiltonian Path Problem*. We enhance *BEA* with methods suitable for the *Longest Hamiltonian Path Problem*. In particular, we propose to apply *BEA* to build the starting solution, then a complementary local-search mechanism improves this preliminary solution. Our purpose is to achieve a better trade-off between quality of clustering and CPU time.

We choose the 2-opt algorithm [16] as the complementary local-search mechanism for several reasons:

- The 2-opt algorithm, as an *improving* mechanism, is a natural complement to a *constructive* procedure, like *BEA*, to build a *composite* algorithm.
- The 2-opt algorithm gives rise [37] to the famous Lin-Kernighan algorithm [77] for solving the Traveling Salesman Problem.
- The 2-opt algorithm has been shown to be useful in a broad range of combinatorial optimization problems [61, 67].
- The 2-opt algorithm is the basic building block of many approaches (genetic algorithms and simulated annealing, amongst others) [61] that deal with combinatorial optimization problems.
- The theoretical properties of 2-opt have been well investigated [12, 61].
- Well tuned data structures have been developed [37, 61] for improving the efficiency of 2-opt.

In the 2-opt algorithm, the basic move consists of inverting a sequence of values in a part of the encoding of the trial solution. For example, for a given trial solution:

$$\sigma_1, \dots, \sigma_{i-1}, \sigma_i, \sigma_{i+1}, \dots, \sigma_{j-1}, \sigma_j, \sigma_{j+1}, \dots, \sigma_n,$$

the application of the inversion operator $I(i, j)$ is

$$\sigma_1, \dots, \sigma_{i-1}, \sigma_j, \sigma_{j-1}, \dots, \sigma_{i+1}, \sigma_i, \sigma_{j+1}, \dots, \sigma_n.$$

In order to improve the efficiency in computing e , each time a call to the inversion operator is made, Croes [16] also proposes a gradient approach. Using our notation (see Section 3.2.2) and considering (i, j) as the indices of the matrix B , such a gradient function is given by:

$$\Delta_e e(\pi, i, j) = b_{\sigma(i-1), \sigma(j)} + b_{\sigma(i), \sigma(j+1)} - b_{\sigma(i-1), \sigma(i)} - b_{\sigma(j), \sigma(j+1)}.$$

This is efficient because it does not depend on the separation of the indices i and j . So it works as a reasonable operator to increase the neighborhood of BEA in local search. In particular, the cost of evaluating $\Delta_e e(\pi, i, j)$ is the same as the gradient $\Delta_e e(\pi, j, k)$ in $BEAm$ and $BEAo$.

The algorithm performs a sequence of applications of the inversion operator $I(i, j)$ at each iteration t .

In particular, at each iteration t , it applies $I(i, j)$ for all pairs i, j with $1 \leq i < j \leq n$. It adopts as the new current solution the best from all these applications of $I(i, j)$ and increments the iteration from t to $t + 1$ until no further improvement results

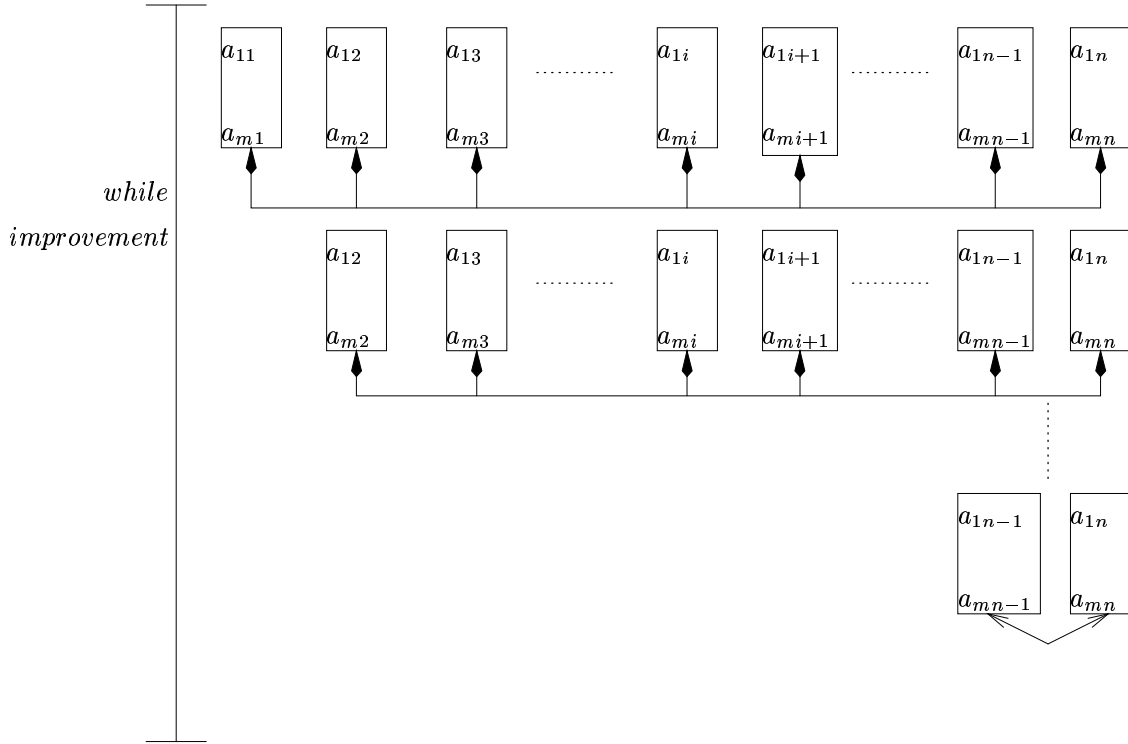


Figure 3.5: *The body of the 2-opt algorithm during the column optimization phase.*

from any application of $I(i, j)$. The algorithm starts with $i = 1$, and tests all $I(i, j)$ for $j = i, \dots, n$ before continuing with the next value of i (until $i = n - 1$).

Our implementation of this algorithm inspired by the work of Croes [16] is displayed in Figure 3.5 and detailed in Algorithm 5.

Because the starting solution can be built by the algorithms $BEAm$, $BEAo$, or $BEAx$, three variants of the composite algorithm can be produced: $BEAm+2\text{-opt}$, $BEAo+2\text{-opt}$ and $BEAx+2\text{-opt}$.

Algorithm 5 *Our implementation of 2-opt (Croes)*

```

1: input:  $B, \pi$ 
2: output:  $\pi = \langle \sigma_1, \dots, \sigma_n \rangle$  which maximizes  $e(A)$ 
3: while improvement do
4:    $\Delta_c \leftarrow 0.0$ 
5:   optimum  $\leftarrow 0.0$ 
6:   improvement  $\leftarrow False$ 
7:   for  $i$  from 1 to  $n - 1$  by 1 do
8:     for  $j$  from  $i + 1$  to  $n - 1$  by 1 do
9:       compute  $\Delta_c e(\pi, i, j)$ 
10:      if  $\Delta_c > \textit{optimum}$  then
11:        optimum  $\leftarrow \Delta_c$ 
12:         $x \leftarrow i$ 
13:         $y \leftarrow j$ 
14:        improvement  $\leftarrow True$ 
15:      end if
16:    end for
17:  end for
18:  if improvement then
19:     $I(x, y)$ 
20:  end if
21: end while

```

We carried out several experiments to evaluate the plausibility of these techniques in the context of clustering visitation paths of web-users.

3.2.5 A Case Study

Clustering visitation paths of web-pages has been shown to be useful in clustering web-users [149]. The discovery of similar interests between web-users can be applied to improve the design of web-sites. A similarity function may be based on visiting similar pages, or the frequency of access to a page, or the visiting order of links. In any case this definition depends on the particular application at hand.

After a suitable definition of similarity has been found, we can build a similarity matrix SM of web-users. The value stored on each cell $sm_{i,j}$ represents the similarity between users i and j . Observe that this matrix is square and symmetric.

3.2.6 The Experiments

The benchmark of Xiao *et al.* consists of synthetic data. This procedure assigns real values in the interval $[0, 1]$ to each cell $sm_{i,j}$ of the matrix SM , independently and uniformly. The size of the matrix SM is $n = 100$. Xiao *et al.* apply a parameter named *similarity threshold* to regulate similarity amongst users. Higher *threshold* values emphasize dissimilarities between web-users. This parameter varies from 0.0 to 0.9. The matrix SM is transformed into a matrix A by zeroing those cells with values of similarity lower than the defined *similarity threshold*. The matrix A is also square and symmetric. Xiao *et al.* propose to execute the algorithm ten times

and report the average at each value of the *similarity threshold* parameter. In the construction of our test cases we apply this method.

Note that each value of similarity threshold represents a new input matrix and a new optimization problem, although it is not very different from another with slightly different threshold.

Quality of Solutions

During the experiments we compare the quality of the solutions provided by seven algorithms, the algorithms from the literature *BEAm*, *BEAo*, *BEAx*, and our proposals for this problem, *2-opt*, *BEAm+2-opt*, *BEAo+2-opt* and *BEAx+2-opt* (these algorithms are implemented as described in Sections 3.2.3 and 3.2.4). In order to obtain the *quality* at each value of *threshold*, we divide the average length obtained from the current algorithm by the average shortest length, obtained through the execution of the seven algorithms,

$$quality = \frac{current_{soln}}{worst_{soln}} - 1$$

(larger values mean better quality). Thus, this measure evaluates relative optimization quality amongst the algorithms in the experiment. It is important to report that in this set of experiments the shortest length (*worst_{soln}*) was obtained when the *BEAx* method was executed.

Plots of the results are displayed in Figures 3.6 and 3.7. The *x*-axis shows the range of *threshold* values from 0.0 to 0.9. The *y*-axis is a quality scale, from, 0.0 to 0.13. The plotted lines are the average *quality*, taken over 10 problems.

Figure 3.6 illustrates that the quality of the solutions provided by the variant *BEAm* (as implemented by McCormick *et al.* [81]) is superior to that provided by *BEAo* and *BEAx*, but not better than the quality provided by *2-opt*. In other words, *2-opt* is better than all tested variants of *BEA*.

In a second set of experiments we compare the algorithms *BEAm+2-opt*, *BEAo+2-opt* and *BEAx+2-opt* (as a reference, we also include the quality provided by the *2-opt* algorithm). Figure 3.7 shows that the algorithm *BEAm+2-opt* is superior to the others in terms of quality of solutions. Observe that it means that the composition of *BEAm+2-opt* is better than the algorithms *BEAm* and *2-opt* when taken separately. This figure illustrates that the quality of the solutions provided by the algorithms separately complemented with *2-opt* is superior to both the quality of the solutions provided by the algorithms not complemented with *2-opt*, and the quality of the solutions provided by the *2-opt* method.

Computational Effort

We compare the computational effort of the six algorithms. Our results are reported in Tables 3.1 and 3.2. At each *threshold* value the *time* (given in seconds) is the average taken over 10 problems.

Table 3.1 shows that, along the entire *threshold* interval, *BEAo* is the fastest of the four algorithms. Also, that the algorithm *BEAm* is faster than the algorithm *2-opt*.

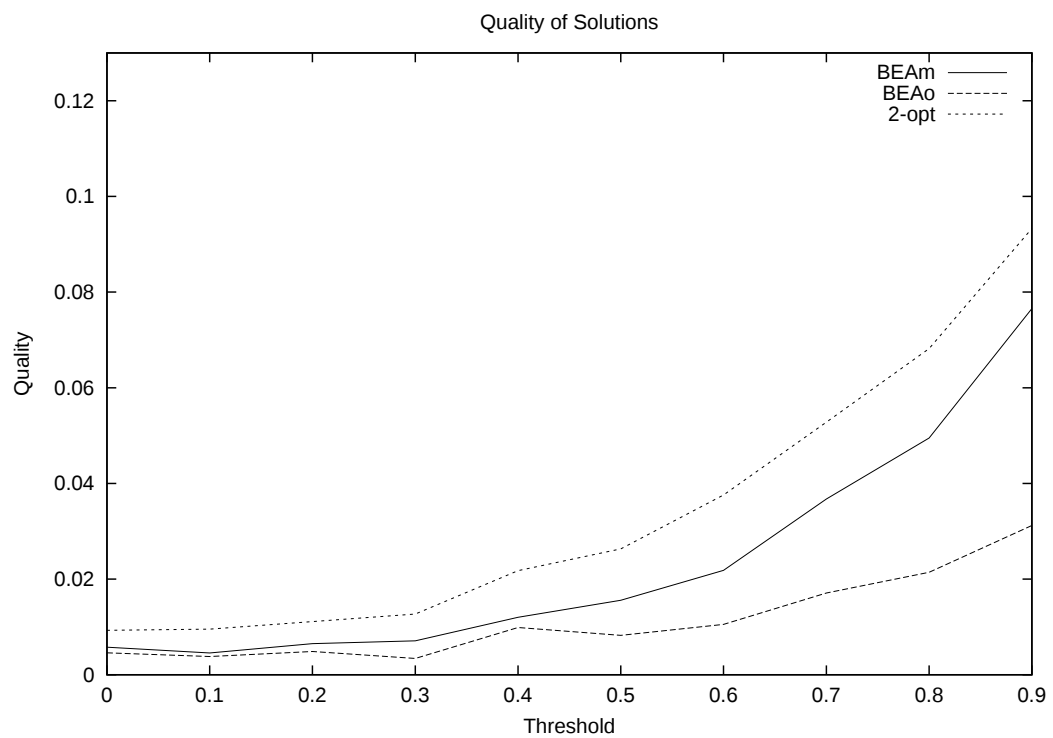


Figure 3.6: *Quality of solutions provided by the algorithms BEAm, BEAo and 2-opt; BEAx is the baseline.*

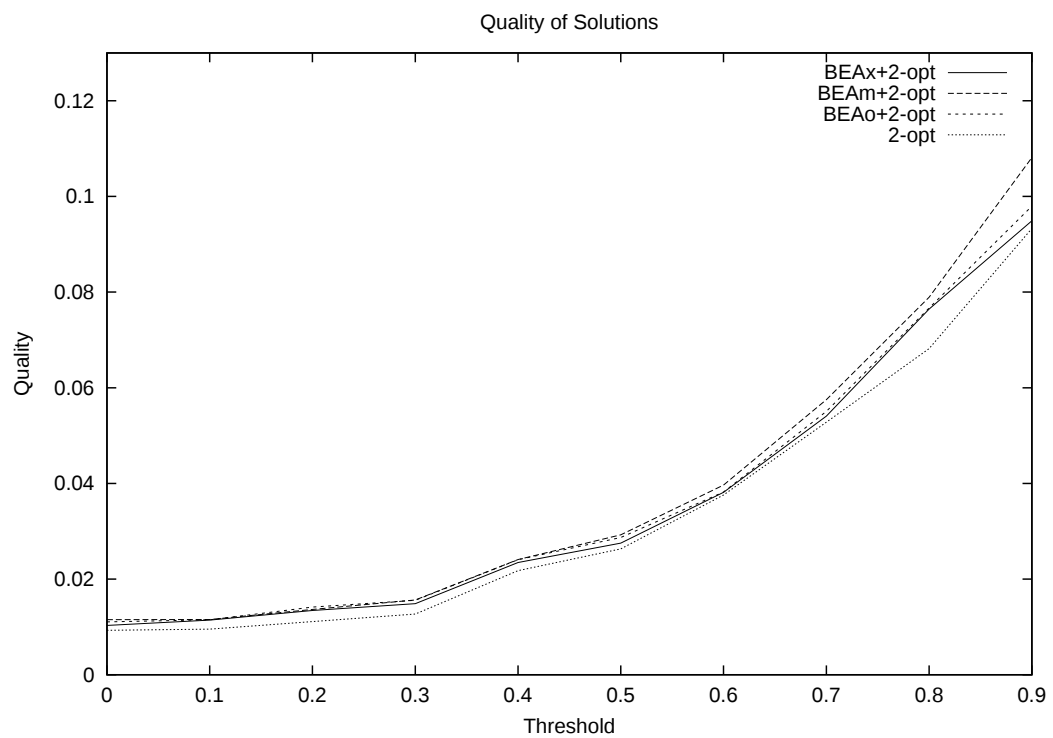


Figure 3.7: *Quality of solutions provided by the algorithms BEAm + 2-opt, BEAo + 2-opt, BEAx + 2-opt and 2-opt; BEAx is the baseline.*

<i>Threshold</i>	<i>BEAm</i>	<i>BEAo</i>	<i>BEAx</i>	<i>2-opt</i>
0.0	0.021	0.000	1.283	0.46
0.1	0.022	0.002	1.280	0.48
0.2	0.021	0.000	1.273	0.46
0.3	0.023	0.001	1.270	0.45
0.4	0.021	0.000	1.270	0.48
0.5	0.023	0.003	1.273	0.47
0.6	0.022	0.000	1.283	0.46
0.7	0.022	0.000	1.283	0.48
0.8	0.021	0.001	1.289	0.47
0.9	0.024	0.001	1.308	0.47

Table 3.1: *Execution time (in seconds) of BEAm, BEAo, BEAx and 2-opt.*

<i>Threshold</i>	<i>BEAm+</i> <i>2-opt</i>	<i>BEAo+ 2-opt</i>	<i>BEAx+ 2-opt</i>	<i>2-opt</i>
0.0	0.037	0.018	1.314	0.46
0.1	0.044	0.019	1.318	0.48
0.2	0.039	0.019	1.296	0.46
0.3	0.039	0.020	1.312	0.45
0.4	0.042	0.018	1.299	0.48
0.5	0.036	0.021	1.296	0.47
0.6	0.035	0.020	1.292	0.46
0.7	0.032	0.019	1.301	0.48
0.8	0.031	0.019	1.296	0.47
0.9	0.036	0.016	1.292	0.47

Table 3.2: *Execution time (in seconds) of BEAm+2-opt, BEAo+2-opt, BEAx+2-opt and 2-opt.*

Table 3.2 shows that along the entire *threshold* interval *BEAo+2-opt* is the fastest algorithm amongst those that include *2-opt* as the complementary local-search mechanism. Moreover, this table shows the algorithm *BEAm+2-opt* is faster than the *2-opt* algorithm.

Table 3.3 shows that *BEAo+2-opt* is faster than *BEAm* and provides the best trade-off between quality of solutions and computational effort for the tested problem size.

<i>Threshold</i>	<i>BEAm</i> time	<i>BEAm</i> quality	<i>BEAo+ 2-opt</i> time	<i>BEAo+ 2-opt</i> quality
0.0	0.021	0.0057	0.018	0.0110
0.1	0.022	0.0045	0.019	0.0114
0.2	0.021	0.0065	0.019	0.0141
0.3	0.023	0.0070	0.020	0.0155
0.4	0.021	0.0120	0.018	0.0240
0.5	0.023	0.0155	0.021	0.0286
0.6	0.022	0.0218	0.020	0.0381
0.7	0.022	0.0367	0.019	0.0550
0.8	0.021	0.0494	0.019	0.0766
0.9	0.024	0.0765	0.016	0.0979

Table 3.3: *Comparison of BEAm with a BEAo+2-opt, in terms of execution time (in seconds) and quality of solutions.*

3.3 Discussion

We have illustrated that LS is a powerful model to deal with hard combinatorial optimization problems. In particular, and after showing that clustering a matrix making use of the measure of effectiveness $e(A)$ proposed by McCormick *et al.* is equivalent to solving the *Longest Hamiltonian Path Problem*, we have applied a LS mechanism to cope with the latter problem.

Also, we have provided empirical evidence about the impact of the starting solution on the performance of the LS mechanism. In particular, we have built the starting solution using three variants of the Bond Energy Algorithm, and the results are dissimilar.

The neighborhood structure of LS is crucial to the trade-off of quality and cost. In our case study we use the algorithm 2-opt as the local search operator. We have found that the combination of the algorithm that we call *BEAm* with the algorithm 2-opt provides the best results in terms of quality. Moreover, we show that this combination provides better results than those obtained by the same algorithms when they are executed separately.

For the case study we have shown that the algorithm 2-opt, usually applied in the solution of the Traveling Salesman Problem (TSP), is superior to the variants of the Bond Energy Algorithm. This is remarkably because the Bond Energy Algorithm has been historically applied in clustering a matrix using the measure of effectiveness $e(A)$.

Finally, for the case study, we have found that the LS mechanism based on the algorithms *BEAo+2-opt* provides the best trade-off between quality and computational effort amongst the tested algorithms.

Chapter 4

Memetic Algorithms

4.1 Introduction

Memetic Algorithms [92] can be seen as population based heuristics that combine Genetic Algorithms (GAs) and Local Search (LS). In combinatorial optimization simple Genetic Algorithms are not usually considered as good as domain-specific methods, in terms of efficiency and quality of solutions [19, 48, 137]. However, plenty of empirical evidence has shown [19, 31, 32, 49, 82, 85, 93, 94, 115, 133] that such combination successfully attacks hard combinatorial optimization problems.

Memetic Algorithms (MAs) are inspired by the notion of a *meme*, which is defined as a unit of information that reproduces itself as people exchange ideas [109]. Thus, MAs can be thought of as mimicking cultural evolution instead of biological evolution.

Analytical and experimental attempts have been made to explain the success of MAs [109, 144, 152]. For example Yamada and Reeves [152] maintain, basing their argument on small benchmark problems, that local optima occur in clusters on the fitness landscape. Thus, a Hybrid GA with local search, which explores around local optima, is able to find better optima. Nevertheless, in general terms, there are no guarantees, and the exact nature of the fitness landscape is highly dependent, in a dynamic way, on every factor involved in the mechanism applied to solve the problem [17]. Several theoretical models are actually extensions of *simple* GA models and they share with those their virtues and limitations. It is widely recognized that, in this respect, fundamental questions remain open [90, p 117].

In order to put in perspective our proposal, we briefly describe some of the most recent strategies, based on MAs, that successfully deal with combinatorial optimization problems.

Baraglia *et al.*, Folino *et al.*, Burke & Newall, and Schnecke & Vornberger [5, 11, 36, 122], propose the integration of domain-specific local search algorithms into Genetic Algorithms. In particular, for the Traveling Salesman Problem (TSP), Baraglia *et al.* [5] apply the Lin-Kernighan (LK) algorithm as a local searcher. The algorithm of Lin and Kernighan is the basis of most successful approaches proposed over the years for solving the TSP. Baraglia *et al.* propose to solve the TSP by producing the initial population of the evolutionary search with the LK algorithm, then a compact genetic algorithm (the term compact derives from the fact that their algorithm does not manage a population of solutions but only mimics its existence) generates high-quality tours, which are then refined again by means of the LK algorithm.

For the Satisfiability Problem (SAT), Folino *et al.* [36] apply the algorithm WSAT as a local searcher. The algorithm WSAT, which is an extension of GSAT, is one of the fastest (though incomplete) implemented procedures for the (SAT) problem: it can solve hard problems with thousands of variables in a few seconds.

Burke and Newall [11] have designed, for the timetabling problem (which consists of allocating a number of events to a finite number of time slots such that the necessary constraints are satisfied), a problem-specific hill-climber. Additionally, the hybrid proposed by Burke and Newall improves on its components by utilizing knowledge of the problem to order events. They decompose a large problem into small components, each one of a size that a MA can effectively handle. In particular, they decompose larger problems by using three heuristics that are generally accepted to be suitable for the exam timetabling problem.

To solve Constrained Placement Problems, that deal with the computation of an optimal arrangement of items on a planar site, Schnecke & Vornberger [122] integrate several problem-specific heuristics to construct the individuals in the population. The reason for including problem-specific knowledge during creation of individuals is to make possible the identification of unfavorable or redundant partial solutions and consider only the most promising ones. Thus, under the approach of Schnecke & Vornberger, each individual encodes a high-quality solution.

Unfortunately, for many combinatorial optimization problems, it is not easy to design effective local search algorithms. Moreover, there is no guarantee that incorporating an effective local algorithm into a genetic algorithm will automatically produce a successful MA which improves on its components. Our thesis addresses this issue by

proposing a family of problem-independent local search algorithms based on sorting algorithms.

Other authors [5,78,138] include problem-specific knowledge in the genetic operators, mainly in the crossover operator.

In particular, Baraglia *et al.* [5] design a crossover operator, which is based on a greedy algorithm, to generate feasible tours for the TSP.

Valenzuela and Smith [138] proposed what they call a window crossover operator, which transfers more genes from the parent with the better fitness value, when solving the Unit Commitment Problem (UCP). UCP consist of determining a minimal cost turn-on and turn-off schedule of a set of electrical power generating units to meet a load demand while satisfying a set of operational constraints.

For the three-matching problem (3MP) (where the purpose is to partition a set of points into disjoint subsets, each consisting of three elements, while minimizing the total cost of the resulting partition), Magyar *et al.* [78] design specific crossover operators that perform local search.

In contrast with those approaches our MA keeps the probabilistic nature of the crossover operator. Thus, we apply the Partially-Mapped Crossover (PMX) operator, a well-known crossover operator for ordering problems.

4.2 The Lamarckian Approach and the Baldwin Effect

There are two general approaches for dealing with the resulting improvements: *Baldwinian* and *Lamarckian*.

The characteristic of the *Baldwinian* approach is not to bring back genotypes improved by local search to the evolutionary cycle. As a result, without altering the genotype, the *Baldwinian* approach changes the fitness of the individual and modifies the fitness landscape [146]. If genotypes are modified by hybridization with local search, then the approach is *Lamarckian*. Thus, *Lamarckian* approaches encode the resulting improvements onto the genotype processed by the genetic algorithm. Figure 4.1 describes both approaches.

It becomes important to clarify the meaning of the terms genotype and phenotype because the distinction between *Baldwinian* and *Lamarckian* is concerned with the inclusion or exclusion of structural change to the genotype [146], while both assess fitness (or use local search) on the phenotype. Genotype means the internally coded inheritable information of a living organism. Phenotype means anything that is part of the observable structure, function or behavior of a living organism. While the spirit of these definitions imported from Biological Theory assumes that to every genotype there is one phenotype, the application of Genetic Algorithms to Constraint Problems faces the challenge that genotypes may be infeasible, and therefore, they may lack a phenotype. We make the observation that this implies two subtypes of (*Baldwinian* or *Lamarckian*) approaches: one that repairs genotypes for feasibility and one that explores a neighborhood of feasible phenotypes. The latter is as originally applied [136, 146].

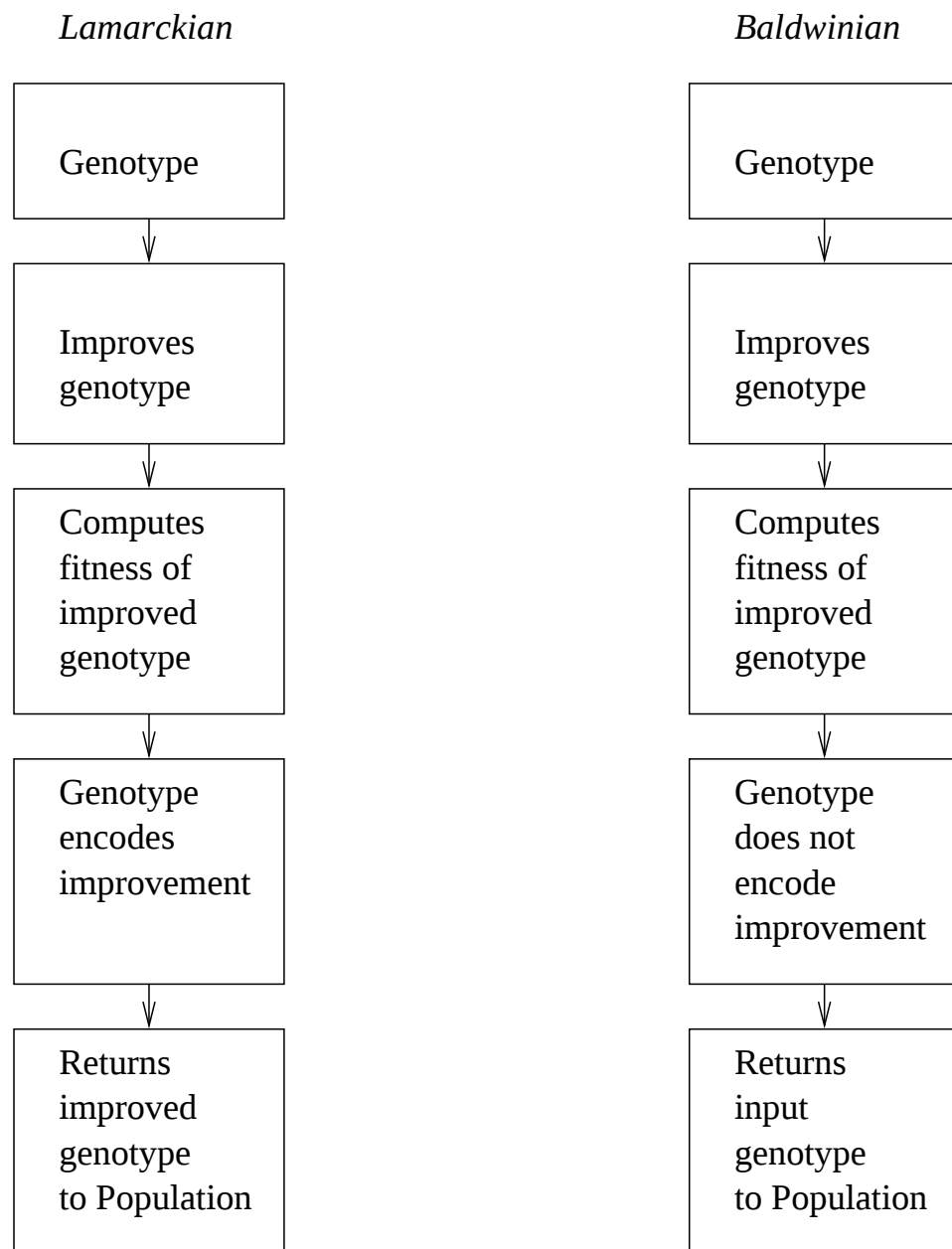


Figure 4.1: Lamarckian and Baldwinian. Two approaches to handle the results of genotype improving.

Whitley *et al.* used a common form of local search, and a set of test functions, in experiments that showed that *Baldwinian* algorithms find better solutions than *Lamarckian* algorithms [146]. However, in the experiments, the former are slower than the latter. Thus, this result left open the need to evaluate the effectiveness of *Baldwinian* algorithms in more complex domains (we address this issue in Chapter 7). The application of the *Baldwinian* approach to problems arising in Machine Learning [136] shows that *Baldwinian* strategies are good for such a class of problems. Recently, experiments with a dynamic environment, in which each individual's fitness is decided by interactions with others and there is no explicit solution through generations, have shown that the *Baldwinian* approach provides outstanding performance [3].

We explore the use of a *Lamarckian* approach for Unconstraint Combinatorial Optimization (Chapters 5 and 6) and the use of a *Baldwinian* approach for Constraint Combinatorial Optimization (Chapter 7).

As an illustration, we describe our MA to deal with the p -median problem.

4.3 The p -Median Problem

The p -median problem is a central facilities location problem that seeks the location of p facilities on a network of n points minimizing a weighted distance objective function [50]. The problem is NP-complete and has a zero-one integer programming formulation [113] with n^2 variables and $n^2 + 1$ constraints, and many techniques have been developed to heuristically solve instances of the problem [20, 116–119, 129]. For

finding high quality approximate solutions, hill-climbing variations of an interchange heuristic [20, 47, 96, 129] are considered very effective, but they risk being trapped in local optima. Other alternatives have been also explored: amongst them, Lagrangian relaxation [14, 99, 142], Tabu search [116] and Simulated Annealing [96]. Recently the p -median problem has been identified as a robust method for spatial classification, clustering and knowledge discovery [28, 97, 101]. While facility location problems may involve perhaps hundreds of points, knowledge discovery applications will face thousands of points.

Progress towards solving the p -median problem with GAs displays a chronology analogous to the efforts to use GAs for solving other combinatorial optimization problems. On one side, we have the recent records on optimally solving instances of the Traveling Salesman Problem (TSP) with linear programming and local cuts [62, 100] that dim the efforts to solve TSP with GAs [95, 137]. On the other side, we see GAs providing very good solutions for the bin packing problem [35, 111].

For the p -median problem, early attempts with GAs used direct binary encoding and the results were discouraging [56]. It was then accepted that GAs could not compete with the efficient and well-designed hill-climbing approaches used for heuristically solving the p -median problem. More recently, the use of integer encoding [7, 9] and some theory of set recombination [29] has shown that genetic algorithms could become competitive. Although it is now clear that integer encoding is better than binary encoding for the p -median problem, none of the at least 5 crossover operators has been identified as the most appropriate. Also, the adequate balance between quality of the approximate solution (proximity to optimality) and computational effort has not been established.

We argue that although GAs for the p -median problem have improved, they hardly outperform hill-climbers. The trade-off of effort versus quality slightly favors hill-climbers. However, occasionally, GAs happen to identify solutions which are closer to optimality. In order to incorporate the efficiency of hill-climbers and the potential higher quality solutions of GA an approach to hybridization is proposed. We will discuss why this hybridization is challenging and present results that illustrate the benefits of this MA approach.

In real D -dimensional space, the p -median problem is concerned with selecting p stations out of $U = \{u_1, u_2, \dots, u_n\}$ points, so that the sum of the distances of all u_i in $\mathbb{R}^D$ to their nearest station is minimum. The problem is motivated by the 2D scenarios where the points are sites that must be serviced by p stations selected from the sites. Naturally, the distance from every site to its nearest station should be as small as possible. In fact, the p -median problem has a formulation where the assignment of stations minimizes the expected distance for servicing a site from its station. As an example, say that U indicates positions in the plane of possible fires and we are to place p fire-fighting stations. Let w_i be the probability that site u_i has a fire. Then, we seek to minimize

$$\begin{aligned} E[d(u_i, \text{station for } u_i)] &= \sum_{i=1}^n \text{Prob}(u_i \text{ has a fire}) d(u_i, \text{station for } u_i) \\ &= \sum_{i=1}^n w_i d(u_i, \text{station for } u_i). \end{aligned}$$

The distance d could be the Euclidean metric or any other metric.

Note that, for any $C \subset U$ of size p , the station for u_i is the site in C nearest to u_i , and we will denote it as $rep[u_i, C]$. That is, $rep[u_i, C]$ is the *representative* for u_i in C , and it satisfies that $\min_{u_j \in C} d(u_i, u_j) = d(u_i, rep[u_i, C])$. In this context, the p -median problem is a problem of finding the best set C of representatives, so every site is as similar as possible to its representative. This is how the p -median translates into a clustering formulation where the sites are partitioned into p groups. Each group is a set of sites with a common representative. The expected dissimilarity between a site and its representative is minimum. We write the p -median problem in this context as

$$\text{Minimize}_{C \subset U \& |C|=p} M(C) = \sum_{i=1}^n w_i d(u_i, rep[u_i, C]). \quad (4.3.1)$$

4.3.1 The Solution Methods

There have been several formulations of the p -median problem as a 0/1-integer programming problem. A trade-off between the type of constraints in the formulation and the frequency by which integer solutions occur when solving the relaxed linear programming problem allows the problem to be solved to optimality when n is small (a few hundred sites) [119]. Since the p -median problem is NP-complete, we expect that exact methods will not be effective for large problems. Hence, heuristics are necessary to obtain approximate solutions.

In what follows we discuss the most effective approaches from the literature, emphasizing similarities rather than differences, since we want to combine their features to obtain methods that integrate their advantages. The most popular type of heuristics are hill-climbers rediscovered in many contexts and best known as interchange heuristics [96, 129]. The hill-climbing nature of these heuristics is clearly revealed if we structure the search space of the p -median problem as a graph with $\binom{n}{p}$ nodes. The nodes of this graph are all $C \subset U$ with $|C| = p$. The edges of the graph are defined as follows: two nodes C and C' are adjacent if and only if $|C \cap C'| = p - 1$; that is, if they differ in exactly one representative. So, since every node in the graph is a feasible solution, we seek to find the node that minimizes $M(C)$ in Equation (4.3.1). The hill-climber interchange heuristics start on a random solution C_0 (a random node in the graph). Iteratively, the heuristic explores a set $N(C_t)$ of adjacent nodes and moves to the best alternative in this neighborhood if the alternative is an improvement (i.e. $M(C_{t+1}) < M(C_t)$). Thus, the new node C_{t+1} is such that $M(C_{t+1}) = \min_{C \in N(C_t)} M(C)$. The search halts when no better solution is found in the neighborhood $N(C_t)$. The interchange hill-climbers offer different variants in how they define the set $N(C_t)$ of adjacent nodes to explore. Complete hill-climbers (and other hill-climbers as well [47, 66, 80, 125]) have been shown not to be as efficient [116] in finding a local optimum of high quality as an original heuristic proposed in 1968 by Teitz and Bart [129]. We will refer to this heuristic as TAB.

In an amortized sense, in the TAB search only a constant number of neighbors of C_t are examined for the next interchange [28]. When searching for a profitable interchange, it considers the points in turn, according to a fixed circular ordering

$(u_1, u_2, \dots, u_n)$ of the points. Whenever the turn belonging to a point u_i comes up, if u_i is currently a representative, it is ignored, and the turn passes to the next point in the circular list, u_{i+1} (or u_1 if $i = n$). If u_i is not a representative point, then it is considered for inclusion in the set of representatives. The most advantageous interchange C^j of non-representative u_i and representative u_j is determined, over all possible choices of $u_j \in C_t$. If $C^j = \{u_j\} \cup C_t \setminus \{u_i\}$ is better than C_t , then C^j becomes the new current solution C_{t+1} ; otherwise, $C_{t+1} = C_t$. In either case, the turn then passes to the next point in the circular list, u_{i+1} (or u_1 if $i = n$). If a full cycle through the set of points yields no improvement, a local optimum has been reached, and the search halts. The TAB heuristic forbids the reconsideration of u_i for inclusion until all other non-representatives points have been considered as well. The heuristic can be therefore be regarded as a local variant of *Tabu search* [41]. TAB's careful design balances the need to explore a variety of possible interchanges against the 'greedy' desire to improve the solution as quickly as possible.

The time required to compute $M(C')$ on an adjacent node C' of C is $O(n)$ time — $O(n)$ steps to find $rep[u, C']$ for all $u \in U$ ($rep[u, C']$ is either unchanged or the new representative u_i), and $O(n)$ to compute $M(C')$ as defined in Equation (4.3.1). Therefore, the time required to test points of C_t for replacement by u_i is $O(pn)$ time. In most situations, p can be viewed as a small constant, and thus the test can be considered to take linear time.

Simulated Annealing can be considered a hill-climber that may accept solutions C_t with $M(C_{t+1}) > M(C_t)$. It also starts with a random C_0 and iteratively redefines a current solution C_t . Not all $p(n - p)$ neighbors of C_t are explored, but all are

sampled. A temperature T works as a tolerance parameter for accepting a $\Delta_t = M(C_{t+1}) - M(C_t)$. When $T > \Delta_t$ a worse solution may be probabilistically accepted as the current solution. The value of T decreases as more solutions are explored ($t \rightarrow \infty$). Although Simulated Annealing opens the possibility to better approximation because it escapes local optima, its computation time is much larger than that of hill-climbers [96] and it demands tuning of more parameters.

4.3.2 The Structure of the Memetic Algorithm

The MA proposed here seeks to find a set C of representatives that optimizes Equation (4.3.1). Thus, Equation (4.3.1) defines the objective function. Genetic operators and the encoding of feasible solutions are tightly related. The literature [7, 9, 29] has reached consensus that because feasible solutions are subsets $C \subset \{u_1, \dots, u_n\}$ with p elements and $p \ll n$, the chromosomes are strings of p different positive integers less than n . This encoding has some redundancy since the same integer values in different order represent the same feasible solution. However, empirical evidence [7, 9, 29] shows that no significant improvement in performance is obtained by choosing some canonical form (for example, keeping the integer values sorted by ascending order within the chromosome), but the extra bookkeeping slows down the search. Moreover, some genetic operators depend on this redundancy for preserving diversity in the population. For example, the *Template Operator* [9] would produce offspring identical to their parents if a canonical form is enforced, but when representing subsets C as strings of p integers, the operator can produce offspring that encode a different phenotype from their parents despite the parents having identi-

cal phenotype. Simple crossover on integer strings [7] does not guarantee different integer values in the offspring (thus a penalty is applied when evaluating infeasible solutions). B. Bozkaya et al [9] define 4 crossover operators, none of which is a definite winner. They are progressively more complex versions of simple crossover on integer strings that ensure that values are not replicated within a chromosome. As the operator's complexity increases, there is a small improvement in the search but at more computational effort per crossover. Since crossover occurs in the innermost loops of the GA's program, slightly more complex crossover operators rapidly raise the demand on computational effort. Also, all these crossover operators mentioned earlier have a strong physical bias. That is, the possible offspring of two parents do not have uniform probability and the distribution is closely related to the encoding, rather than to the semantics of the chromosome in the problem. Operators that use the theory of Random Assorting Recombination [106, 108] have also been proposed [29]. These operators balance two desirable properties of crossover in GAs: assortment and respect.

The goal here is not to argue in favor of one or another crossover operator, because besides using chromosomes that are integer strings it is not clear that the added complexity of more sophisticated crossing is worth it. Moreover, typically, more complex operators imply more parameters that the end user must properly set at the start of the optimization (a challenging task in itself). We argue that a more effective search is obtained by a MA with a rather simple and fast crossover than by a GA whose genetic operators are complex, heavily parameterized and difficult to use. The operator provided by the TAB hill-climber takes a feasible solution C_t and improves it to a new solution C_{t+1} . We consider this a mutation operator and incorporate it as

Table 4.1: TAB and MA optimization with respect to objective function evaluations.

Method	Values of solution found	Average	Evaluations
TAB	11.75 (3 times) 11.77 11.78 (4 times) 11.86 12.08	11.89 ± 0.23	804 ± 14
MA	11.67 (6 times) 11.68 11.71 (3 times)	11.68 ± 0.01	$5,897 \pm 35$

such in the MA. This mutation operator may appear computationally costly at first sight, but as we discussed in the analysis of TAB, it typically requires $O(n)$ time (or $O(n^2)$ time if a local optima is reached). However, evaluating the objective function requires $O(n)$ time, thus $O(n)$ time is required every time an unseen individual comes into play. Since the rate of application of this TAB mutation assures that it occurs sparingly per generation, its cost is well amortized in the genetic program.

The second consideration is that the hybridization must be a tight integration that must preserve those aspects of TAB that make it the most effective hill-climber. Thus, a non-representative u_i that fails to become a representative must be banned until all other non-representatives have also been attempted. Our mutation operator remembers the index i where the last promotion of a u_i to a representative took place. Moreover, it can detect local optima, and in this case, they are typically of high quality. By adjusting the mutation rate (and the population size) the MA can be configured to resemble TAB or to be more independent (TAB is the case population size 1 and mutation rate 1).

Table 4.1 shows the performance of TAB and an MA on a p -median problem with 100 data points and $p = 5$. The computational effort is measured as the number of evalu-

ations of the objective function, a uniform measure of resources required. Since TAB and the MA are randomized, we show results over 10 executions. TAB is typically much faster than the optimization with MAs, but risks getting stuck on local optima. The best solution with TAB is worse than the poorest solution with the MA. The MA has a population of 25 chromosomes. For the test problem, smaller population sizes result in poor performance of the MA. As we have already mentioned, hybridization is complicated because TAB's operation as a mutation that does a hill-climbing step needs to remember how it operated in its previous invocation. However, even if we remember the last point whose promotion was attempted, this may be for a very different chromosome than the one which is now being mutated. Remembering the context of mutation per chromosome is not a successful alternative because it makes the GA work as many concurrent TAB searches, each disrupted (rather than helped) by crossover. We have found that this results in much computational effort and no better search.

Another problem that complicates hybridization is that the chromosome mutated by TAB may exhibit an above average fitness with respect to the rest of the population, moving sharply in the direction of a local optimum. This has the effect that the chromosome dominates the population and the MA converges early to local optima. This is a problem of diversity loss. Thus, we found that the population size of our MA cannot be very small. In the results of Fig. 4.2, the MA improves the performance GA, the MA's population size is 15, while the GA's is 50. Smaller population sizes in the MA result in poor optimization performance of the hybrid, and larger population sizes result in as many function evaluations as the simple GA, or more. We also found that although the mutation rate of the TAB mutation must be small, it must be of

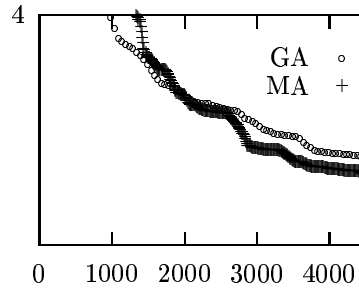


Figure 4.2: *MA and GA optimization with respect to objective function evaluations. Final average for MA is 3.54 with a standard deviation of 0.03 while the final average for the GA is 3.60 with a standard deviation of 0.07.*

some impact when it occurs, otherwise it just looks like a costly random mutation. For this, when a TAB mutation occurs, two hill-climbing steps on the chromosome are performed (and not just one). More hill-climbing steps over-fit one individual with respect to the population. Thus, our results are consistent with those obtained by W. Hart and M. W. S. Land [51, 70].

4.4 Discussion

We have presented a Memetic Algorithm that incorporates exploitation characteristics of a hill-climber into the GA program. LS is efficiently incorporated in the mutation operator. This approach applies with all crossover operators but it demands delicate balancing of the impact of hill-climbing so that the MA avoids early convergence. As a result, the MA optimization is robust and effective and balances effort/quality of solution better than the plain GA.

Chapter 5

Classical Sorting Directs Mutation

5.1 Introduction

We obtain more effective genetic algorithms for unconstrained optimization problems involving permutations. We achieve this through hybridization, in a *Lamarckian* fashion, with classical sorting algorithms. The search for the permutation that optimizes some criteria is a genetic search. However, we dramatically improve this search with a hill-climber in the mutation operator. Sorting algorithms direct the hill-climber (rather than only a discrete gradient). We obtain improved results for a benchmark of experiments of the *Error-Correcting Graph Isomorphism*. This problem seems so hard that in this case, genetic algorithms do perform better than several

other optimization strategies [141]. We implemented our proposal to solve this problem and the results of our experiments show a marked improvement over previous optimization with GAs. Since our proposal is more generic than the case study, we describe it thoroughly and present a discussion of the rationale behind it.

5.2 Classical Sorting as a Local-Search Mechanism

Sorting algorithms are well understood mechanisms that cope with a class of permutation problems [69]. The sorting problem deals with sequences of items from a total order, and usually consists of re-arranging any given sequence X so that the items are in ascending or descending order.

Sorting can be conceived of as determining the permutation $\pi(X)$ on $\{1, \dots, |X|\}$ that re-arranges the sequence X [69]. When X is sorted, $\pi(X)[i]$ is the final position of x_i .

A relation $< \subseteq X \times X$ on the items of a sequence defines a total order if the following conditions are satisfied for any item values $\clubsuit, \diamond, \heartsuit$ [69].

1. Exactly one of the possibilities $\clubsuit < \diamond$, $\clubsuit = \diamond$, $\diamond < \clubsuit$ is true.
2. If $\clubsuit < \diamond$ and $\diamond < \heartsuit$, then $\clubsuit < \heartsuit$ (law of transitivity).

The fundamental operation performed by any sorting algorithm is the comparison of a pair of items from a sequence X . The result of the comparison is enough to decide if the elements are in the right or the wrong order. For example consider the

predicate $<: X \times X \rightarrow \{True, False\}$ defined by

$$<(x_i, x_j) = \begin{cases} True & \text{if } x_i < x_j \\ False & \text{otherwise.} \end{cases}$$

All sorting algorithms define a decision tree. This is a binary tree where nodes are labeled by the two indices of the items to be compared. In a sorting algorithm, after a comparison, two possible outcomes define edges where the computation continues. The edges may involve some operation, like a swap, where the algorithm encodes the permutation. The essence of a sorting algorithms is the decision tree. It specifies the comparisons that follow as a result of previous comparisons in order to determine the permutation of the original input. In fact, this is how the $\Omega(n \log n)$ lower bounds on the number of comparisons required for sorting, in the worst case and in the expected case, are proved.

We now illustrate these ideas in the context of *Insertion Sort*.

Note that if a sorting algorithm were to operate in parallel on an array a where initially $a[i] = i$ (for $i = 1, \dots, n$), then, on completion of the sorting, the array a encodes the permutation $\pi^{-1}(X)$ of the input. For example, to rearrange the input sequence $\langle 2, 4, 3, 1 \rangle$ in ascending order, *Insertion Sort* will first swap the third and the second items. The new sequence is $\langle 2, 3, 4, 1 \rangle$, and array a is $[1, 3, 2, 4]$. *Insertion Sort* continues the sorting by swapping the fourth with the third item. The new sequence is $\langle 2, 3, 1, 4 \rangle$ and the array $a = [1, 3, 4, 2]$. *Insertion Sort* continues the sorting, swapping the third with the second item. The new sequence is $\langle 2, 1, 3, 4 \rangle$ and the array $a = [1, 4, 3, 2]$. Finally, *Insertion Sort* swaps the second with the first item. The final sequence is $\langle 1, 2, 3, 4 \rangle$ and the array $a = [4, 1, 3, 2]$. This encodes the

inverse $\pi^{-1}(X)$ of the original permutation $\pi(X)$. Where $a[i] = j$ means the i -th smallest is in the j -th position of the input.

Figure 5.1 shows a comparison tree for *Insertion Sort* on sequences of length 4. Figure 5.1 also shows the actions (note that $s(i, j)$ swaps items in positions i and j), but formally the decision tree is just the scheme of comparisons. Leaves indicate the array a which encodes the permutation $\pi^{-1}(X)$ that re-arranges the original sequence.

Using measures of presortedness it is possible to formalize sorting as an optimization problem. An *inversion* is any pair of elements that are in the wrong order. More formally, an inversion is a pair $i < j$ where $<(x_i, x_j)$ is *False*. The total number of inversions in a sequence X is the measure *Inv* that counts the total number of pairs in the wrong order (that is, the number of *inversions*) [27, 69]. Then, for a given sequence $\langle x_1, \dots, x_n \rangle$ of n items in X , classical sorting corresponds to the following problem [**sorting inversions**]:

$$\text{minimize } \text{Inv}(\pi) = \text{Inv}(\langle x_{\pi(1)}, \dots, x_{\pi(n)} \rangle).$$

This measure has an operational alternative as the smallest number of adjacent swaps necessary to bring the sequence to sorted order.

It is now well known that this inversion-minimization problem can be solved by comparison-based sorting algorithms in $\Theta(n \log n)$ comparisons in the worst case by *Heapsort* and *Merge sort* and in $\Theta(n \log n)$ comparisons in the expected case by *Quicksort*.

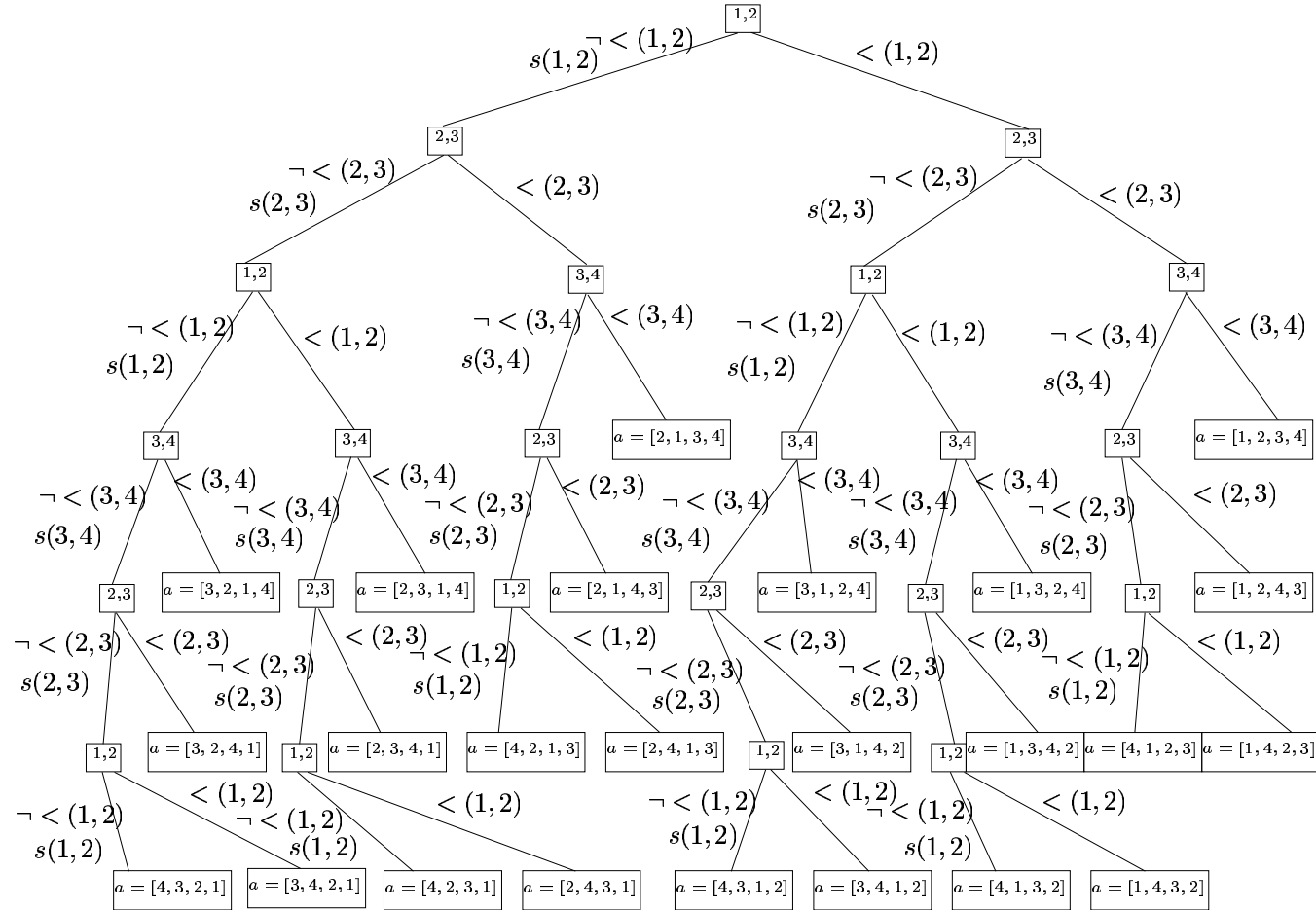


Figure 5.1: Insertion Sort as an algorithm for searching the permutation π such that $\pi^{-1}(\langle x_1, x_2, x_3, x_4 \rangle)$ is sorted. The array a encodes the permutation $\pi^{-1}(X)$.

However if we were only to perform swaps of adjacent items in the input array, then the search space $S = S_n$ of $n!$ permutations would have a structure of a graph amenable for search by LS. In fact, each permutation π is a feasible solution with a value $Inv(\pi)$, and two permutations π_i and π_j are neighbors if π_i and π_j differ in swapping an adjacent pair.

A naive design for local search to solve the inversion-minimization problem on S_n would make $\nu(\pi)$ the schedule $\langle (1, 2)\pi, (2, 3)\pi, \dots, (n-1, n)\pi \rangle$ where the permutation $(i, i+1)\pi$ is the result of swapping the i -th and $i+1$ -th items in π . Note that each current solution π_i has $n-1$ neighbors.

Clearly, such Local Search would not match the $\Theta(n \log n)$ bound of sorting algorithms. Sorting algorithms do not explore the $n-1$ neighbors of each current solution (they explore much less). In its state of execution, a sorting algorithm remembers the results of previous comparisons, and can reduce the number of neighbors to explore in the next iteration of its very efficient local search. In some cases the exploration is reduced to a path.

For example, starting with $\langle 3, 4, 2, 1 \rangle$, *Insertion Sort* would first test $< (x_1, x_2)$. Then, typically it would record in its state that the prefix $\langle 3, 4 \rangle$ of the input is already sorted. The next test $< (x_2, x_3)$ would return *False* and a swap would be performed. In each swap, the measure of disorder Inv is reduced by one. However, although $\langle 3, 4, 1, 2 \rangle$ is a neighbor of the original input, this is never explored by *Insertion Sort*. The inversion-minimization for $\langle 3, 4, 2, 1 \rangle$ always follows the path $\langle 3, 4, 2, 1 \rangle \rightarrow \langle 3, 2, 4, 1 \rangle \rightarrow \langle 2, 3, 4, 1 \rangle \rightarrow \langle 2, 3, 1, 4 \rangle \rightarrow \langle 2, 1, 3, 4 \rangle \rightarrow \langle 1, 2, 3, 4 \rangle$.

Figure 5.2 displays the neighborhood structure of the search space of $n!$ permutations in the case of $n = 4$ and where swaps of adjacent items in the sequence are the actions.

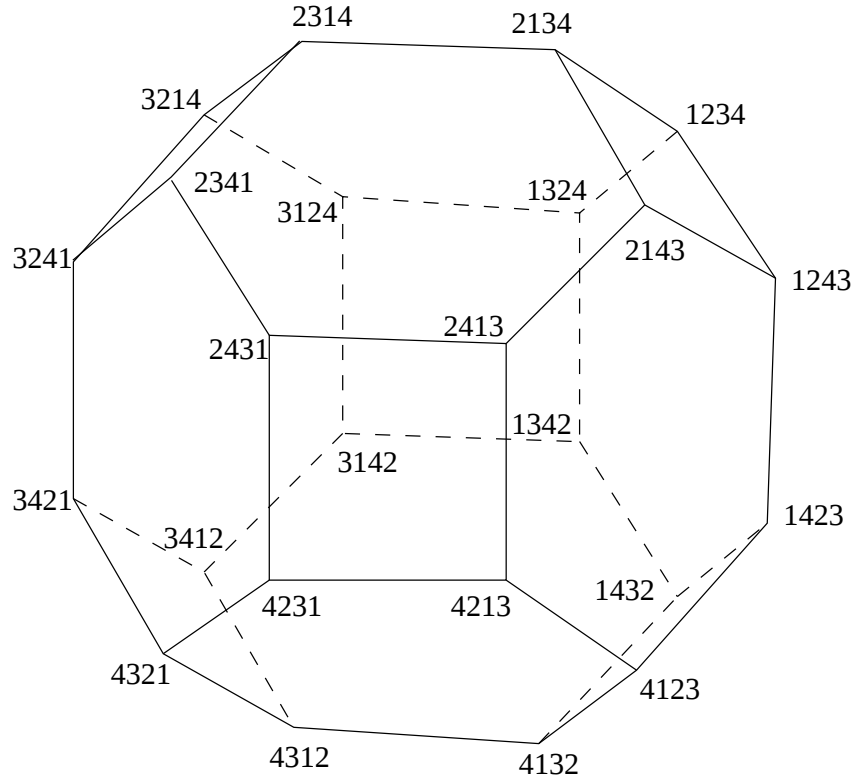


Figure 5.2: The search space of $4!$ permutations arranged into a graph by swaps of adjacent items.

For example, Figure 5.3 shows the search tree (the set of possible paths) for $4!$ permutations, built by the *Insertion Sort* mechanism. Testing the value of the predicate $< (x_i, x_j)$ defines the path. For any permutation π , the *Insertion Sort* algorithm visits permutations in the left branch from the node for π , all the way down to a leaf.

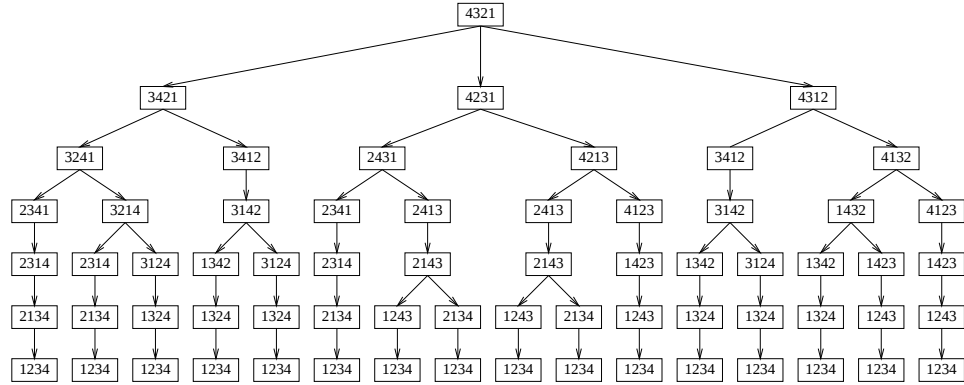


Figure 5.3: The search tree for $4!$ permutations built by the Insertion Sort mechanism.

Thus, sorting algorithms define paths in the search space of $n!$ permutations, from any permutation to the goal permutation.

Although it is unusual, sorting algorithms can be conceptualized as LS mechanisms. We make use of our model of LS (explained in Section 3.1) to describe such correspondence. Figure 5.4 displays sorting as a LS in a search space S_n of permutations.

In the *Initialization* step, a permutation π_i , which encodes a sequence X , is the input to the LS mechanism.

The sorting algorithm maintains a record of its state of execution.

In the *Comparison step* we compare the current permutation π_i with the permutation π_j obtained from the *Nomination* step. In the case of *Insertion Sort* this matches evaluating the objective function *Inv*. If the arrangement represented by π_j is better than the arrangement represented by π_i , the former becomes the **current.best**.

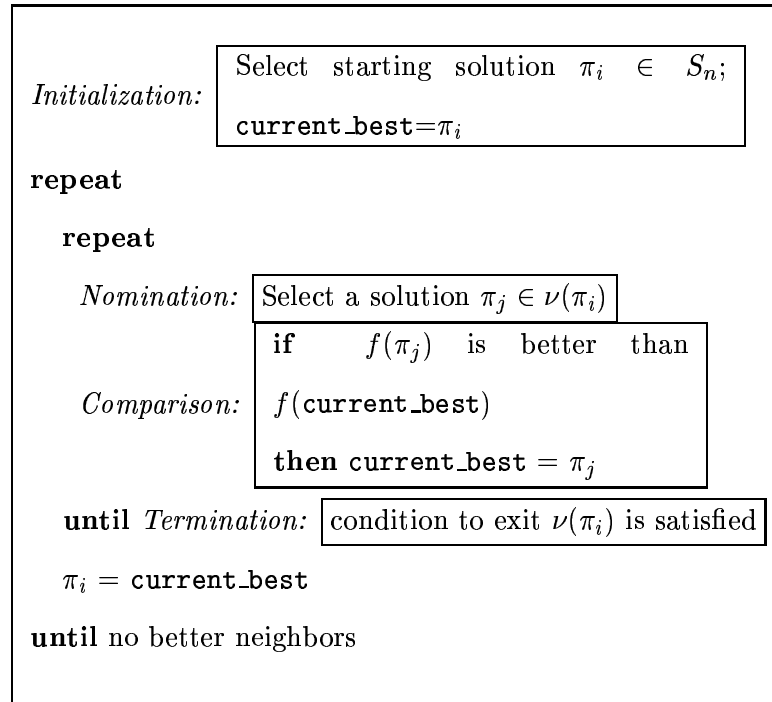


Figure 5.4: *Pseudo-code for Sorting as Local Search (LS) in a Search Space of Permutations.*

The *Termination* condition is always *First improvement* because any swap of adjacent inverted elements improves the objective function by exactly one.

5.3 Classical Sorting as Director of Mutation

For genetic algorithms working on permutations, the most simple swap mutation operator [19] chooses uniformly a random pair of positions in the chromosome and swaps the contents of the selected positions. This operation often produces a chromosome with worse performance. This is not considered a problem when a mutation

operator is conceived as a diversity generator. This is the spirit of a memoryless mutation which works oblivious to what has been learned in the past (it does not require any record of state). By contrast, we apply the results of the mutation operator only if we achieve better performance. Thus, some record of state will influence a schedule of swaps very different from random choices. In this sense, our mutation operator is a hill-climber. Other mutation operators which perform LS have been proposed in the literature. Many of them [19, 49, 115, 133] are variants of the most simple (random) swap mutation operator. Our approach is to add exploitation capability to the mutation operator.

Memetic Algorithms can be defined as the integration of LS and GA. In our approach [31, 32], classical sorting works as a map for the search space S_n of $n!$ permutations.

For the moment, assume that our Memetic Algorithm consists of a sorting algorithm optimizing a measure of disorder and working in parallel with the GA. When the GA is executing, it has a given mutation probability p_m . After the generation of offspring the GA randomly selects these chromosomes for mutation with the probability p_m . If a chromosome is chosen for mutation, the sorting algorithm will perform the mutation. That is, the sorting algorithm will indicate how to swap items to create a new permutation.

Our mutation operator requests direction from a sorting algorithm. That is, our mutation operator does not randomly swap items of a chromosome encoding a solution. Our mutation actually requests from the sorting algorithm which permutation to construct from the current permutation. The neighborhood operator ν suggested

by a sorting algorithm is applied to mutate the current permutation π_i into π_j . This operator ν will be defined in the next section where it actually has the structure of a schedule.

Our integration is called *MA-sorting*, and a mutation is the result of a sequence of swaps suggested by our mutation operator referred to as *SORT*. The acceptance or not of the swap is performed by evaluating the objective function in the entire permutation. It is important to notice that we do not compare the individual elements that are being swapped, but the fitness of the entire permutation as a whole, before and after the swap. Note that the evaluation function, which guides the evolutionary search, and the *Comparison* function f in LS, which guides our mutation operator, are made to correspond. However, if the resulting permutation represents an improvement (with respect to the objective function), the message relayed to the concurrent *SORT* is that the comparison has returned *False*, as if the two elements were swapped and the disorder reduced (from the perspective of the sorting mechanism, we have compared the individual elements). Symmetrically, if the resulting permutation does not correspond to an improvement with respect to the objective function, the message relayed back to *SORT* is that the comparison has resulted in *True* (as if the individual items were not swapped, and the permutation did not change).

The only explicit knowledge about the problem in our *MA-sorting* algorithm is that the problem involves permutations. Our *SORT* mutation operator determines facts such as that item k should be before item l . That is, it adopts how sorting algorithms learn the permutation $\pi(X)$ to sort the input X .

5.4 Attributes of Sorting that Influence the Neighborhood Structure

Our claim is that the following five attributes of sorting algorithms influence the neighborhood structure that is used for a LS, and can be used to increase the degree of coupling between the sorting algorithm and the GA. In particular, they control what is to be understood as the neighbor operator ν in the LS part and play a significant role in the performance of our *MA-sorting* algorithm.

The five attributes of sorting algorithms on the neighborhood structure are as follows:

1. *Swap constraint.*
2. *Record of the state.*
3. *Focus.*
4. *Dependency on input.*
5. *Presortedness measure.*

In order to explain these attributes we briefly introduce *Selection Sort* and *Quicksort*. We skip details of *Insertion Sort* here because it has been introduced already.

Quicksort is a divide-and-conquer strategy for sorting invented in 1960 by C. A. R. Hoare. In *Quicksort*, we divide the sequence of items to be sorted into two subsequences, a left and a right part, and then call the *Quicksort* procedure recursively to sort the two parts. There are several ways to achieve such a partitioning. Our

implementation adheres to the description of Sedgewick [123] because it is considered as the best scheme [69]. During partitioning, we choose a pivot ϑ and arrange that all the items in the left part are less than the pivot and all those in the right part are greater. This process keeps a left pointer l and a right pointer r . Pointer l scans from the left and pointer r scans from the right, moving inward until the left pointer finds an element greater than the pivot and the right pointer finds an element less than the pivot. Every time two such elements are found, they are swapped (and the two pointers restart scanning inward). If they cross, we exchange the pivot with the element pointed to by the left pointer, completing the partition. Sedgewick [123, p 118] describes pseudo-code for *Quicksort*.

One of the simplest sorting algorithms is *Selection Sort*. It works as follows: first find the smallest element in the array and exchange it with the element in the first position, then find the second smallest element and exchange it with the element in the second position. Continue in this way until the entire array of elements is sorted [123, p 96].

5.4.1 Swap Constraint

The most common action performed by sorting algorithms when the comparison predicate returns *False* is the swap of the elements involved in the comparison. Depending on the type of sorting method, the comparison, and in consequence the swap, may be restricted to certain positions on the sequence. Therefore, a first distinction amongst sorting methods emerges: 1) Swaps are only allowed between specific positions on the sequence; and 2) Swaps are not restricted to specific positions on the

sequence. A notorious example of a method that restricts the position of the elements being part of a swap operation is *Insertion Sort*. It only permits swaps of adjacent items. The map of comparisons defined by *Insertion Sort* is displayed in Figure 5.2. By contrast, the swaps performed by *Quicksort* and *Selection Sort* are not restricted to adjacent positions on the sequence. This attribute has an impact on the neighborhood structure of the search space of permutations and we name it *swap constraint*. In particular, from the perspective of *Insertion Sort*, a permutation π has at most $n - 1$ neighbors and $\|\nu(\pi_i)\| \leq n - 1$. However, for algorithms like *Quicksort* and *Selection Sort*, an arbitrary swap implies that $\|\nu(\pi_i)\|$ could have as many as $n(n - 1)/2$ neighbors.

Let $G_{as}(n)$ be the graph that has all permutations of n items as nodes. Two nodes π_i and π_j are adjacent if π_i can be transformed to π_j by an adjacent swap. Also, let $G_s(n)$ be the graph that has all permutations of n items as nodes. Two nodes π_i and π_j are adjacent if π_i can be transformed to π_j by an arbitrary swap. If a given sorting algorithm Q only compares adjacent items, its search space is $G_{as}(n)$, otherwise its search space is $G_s(n)$. Examples of $G_{as}(n)$ and $G_s(n)$ are displayed in Figure 5.2 and Figure 5.5, respectively.

Let T be the comparison tree of a sorting algorithm Q . An execution state of Q is any piece Ξ of information that uniquely identifies a branch of the comparison tree T . Note that with *Insertion Sort*, the state can be succinctly encoded as a pair $\Xi = (j, i)$ with $1 \leq j < i < n$. This state means that $x_1, \dots, x_i$ are sorted and x_{i+1} is less than x_j . In other words, the $(i + 1)$ -th item is to be inserted amongst a prefix of i -sorted items, and we know it goes before x_j (it has already compared with $x_j, x_{j+1}, \dots, x_i$).

Thus $x_j, x_{j+1}, \dots, x_i$ is being shifted to $x_{j+1}, x_{j+2}, \dots, x_{i+1}$, and x_{i+1} is being placed in x_j . So the next comparison in the schedule is $< (j - 1, j)$ (unless $j = 1$ in which case i is incremented and the comparison is $< (i, i + 1)$). As we indicated before *Insertion Sort* uses a schedule with adjacent items and its search space is $G_{as}(n)$. However, *Selection Sort* uses arbitrary swaps and so the search space is $G_s(n)$. In particular *Selection Sort* uses the following schedule of comparisons

$$\begin{array}{ccccccc}
 (1, 2) & (1, 3) & (1, 4) & \cdots & (1, n) \\
 & (2, 3) & (2, 4) & \cdots & (2, n) \\
 & & (3, 4) & \cdots & (3, n) \\
 & & & \ddots & \vdots \\
 & & & & (n - 1, n).
 \end{array}$$

In *Selection Sort*, a state could be encoded as a pair $\Xi = (k, j)$ with $1 \leq k < j < n$. Such a state means that: 1) the prefix $x_1, \dots, x_k$ is sorted; 2) these are the k -smallest items in the input; and 3) the item x_{k+1} is the smallest in $x_{k+1}, \dots, x_j$. Note that in our definition of state, the comparisons and the swaps are atomic.

Figure 5.5 displays the graph $G_s(n)$ which has all permutations of n items as nodes, and two nodes π_i and π_j are adjacent if π_i can be transformed to π_j by an arbitrary swap.

5.4.2 Record of the State

While the schedule of comparisons in *Selection Sort* does not depend on its outcome, it does depend on it in the case of *Insertion Sort*. Our definition of state serves to identify where the schedule of comparisons can be restarted.

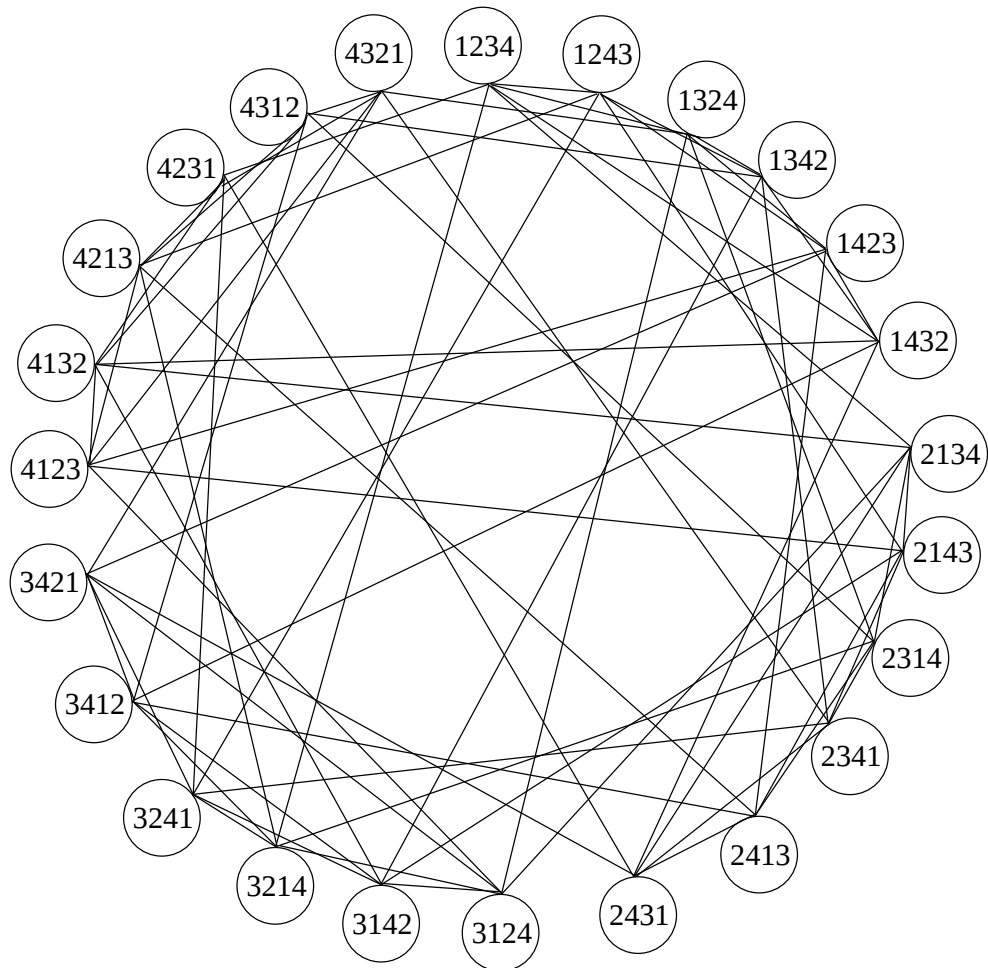


Figure 5.5: *The search space of $4!$ permutations arranged into a graph by arbitrary swaps of items.*

Definition 9 (Neighborhood $\nu(\pi)$) Given π , a sorting algorithm Q and a state Ξ for Q , the neighborhood $\nu(\pi)$ consists of all the neighbors in the search space of Q that are reached by the schedule of Q when restarted with Ξ .

Because the schedule of Q when restarted with Ξ traverses $\nu(\pi)$ in a given order, we can also consider $\nu(\pi)$ a schedule for Local Search.

For example, consider the array $a = [\clubsuit, \diamond, \heartsuit, \spadesuit]$. We determine the neighborhood of this array by *Insertion Sort* on state $\Xi = (2, 4)$. This means that *Insertion Sort* believes $\clubsuit < \heartsuit < \spadesuit$, and $\diamond$ is the i -th item of the arrangement. It has already performed $< (\spadesuit, \diamond)$ and $< (\heartsuit, \diamond)$ in that order, and both have returned *False*. So the next comparison is $< (\clubsuit, \diamond)$ (testing the neighbor $a = [\diamond, \clubsuit, \heartsuit, \spadesuit]$). If the comparison returns *False*, *Insertion Sort* would move to $a = [\diamond, \clubsuit, \heartsuit, \spadesuit]$ so note that *Insertion Sort* uses the *First improvement* format.

In particular the neighbor $\nu(\pi)$ by *Insertion Sort* has size the number of comparisons until a comparison returns *False*. It is 1 in the best case and $n - 1$ in the worst case. *Insertion Sort* always performs a swap after it finds a pair in the wrong order. This is the *Termination* condition in its LS, and it has moved to a new current permutation.

Now consider again the array $a = [\clubsuit, \diamond, \heartsuit, \spadesuit]$ but let us use *Selection Sort* with the state $\Xi = (2, 3)$. So the next comparison is $(2, 4)$ which is $< (\diamond, \spadesuit)$ (because the state $\Xi = (2, 3)$ means that $\clubsuit$ is the smallest item and the last comparison ensures that $\diamond < \heartsuit$). The neighbor is therefore $a = [\clubsuit, \spadesuit, \heartsuit, \diamond]$. Once again the neighborhood $\nu(\pi)$ is exited when *Selection Sort* attaches a swap to a comparison. In the best case, $\nu(\pi)$ has size 1 but in the worst case it has size $\frac{(n-1)(n)}{2}$.

5.4.3 Focus

A relevant characteristic of a sorting method is the number of times a particular element is repeatedly involved in the comparison predicate. We call this attribute the *focus*. This attribute is relevant because it influences the schedule structure of the neighborhood and its orientation in the search space. Some sorting algorithms induce a local neighborhood that is highly focused on some element (low spread of indexes). This is the case of *Selection Sort*. While in state $\Xi = (k, j)$ under this method, the item in position k on the current permutation π_i is part of a sequence of comparisons performed to generate the neighbors of the current permutation. On the other hand, *Quicksort* (implemented recursively) generates a schedule (until a swap is performed after the comparison returns *False*) with a higher spread than *Selection Sort*. Thus, *Quicksort* distributes more evenly the chances for a particular element of being part of a comparison (and swap) operation. Moreover, if the implementation of *Quicksort* is not recursive, but instead of using a stack it uses a queue, the algorithm achieves even higher spread.

That is, the indices nominated by *Quicksort* for a comparison are spread more evenly among the range $1, \dots, n$. *Selection Sort* focuses for a rather long section on an index k .

5.4.4 Dependency on Input

The fourth attribute is related to the order of nominations. Either the sorting mechanism predefines the order of comparisons, independently of the value returned

by the comparison predicate, or the order of comparisons depends on the previous result of the comparison predicate. An example of the first type is *Selection Sort*. The schedule of comparisons predefined by *Selection Sort* is as follows: $(1, 2), (1, 3), \dots, (1, n), (2, 3), \dots, (2, n), \dots, (n-1, n)$. Examples of the second type are *Insertion Sort* and *Quicksort*. For instance, *Insertion Sort* decides which pair of elements are going to be compared next depending on the value returned by the comparison predicate. For example, if the current state of comparison involves the elements in positions $\Xi = (3, 4)$ and *Insertion Sort* receives a *False* signal, a swap is performed and the next pair of elements to be compared is $(2, 3)$. If the opposite situation occurs (the comparison predicate returns a *True* signal), the swap is not performed and the next pair compared is $(4, 5)$. We call this attribute *dependency on input*.

5.4.5 Presortedness Measure

For a neighborhood operator ν and minimization problem Ψ (with objective function f), another function g will be called a monotonic function for Ψ if and only if, for any given permutation π , $g(\pi) > g(\nu(\pi))$ and $\nu(\pi)$ is a better solution than π (ie $f(\pi) > f(\nu(\pi))$).

Thus, $Inv(\pi)$ is a function that monotonically reflects the disorder (the distance from the optimum) of a given permutation with respect to swaps of adjacent elements. Of course this works when we can apply simple operations to permutations and we have a function that can assess if that local change is a step in the right direction.

Finding such monotonic functions is usually not an easy task, specially for other harder combinatorial optimization problems that search for an optimal permutation among the $n!$ possible alternatives.

We illustrate this once more with our running example in which we are using sorting as an optimization problem. Consider the following version [**sorting runs**]:

$$\text{minimize } Runs(\pi) = 1 + \|\{(i, i+1) \mid 1 \leq i < n \wedge \neg <(x_{\pi(i)}, x_{\pi(i+1)})\}\|.$$

In this version, sorting corresponds to minimizing the number of ascending runs (the optimal value is $Runs = 1$ when the sequence is sorted). It is not hard to see that Inv is not a monotonic function for swaps of adjacent items in the sorting runs version. For example, consider the permutation $\pi = \langle 1, 4, 5, 2, 3 \rangle$, and let the operator ν return the first swap performed by *Insertion Sort*. Then, the next permutation in the search space visited by *Insertion Sort* is $\nu(\pi) = \langle 1, 4, 2, 5, 3 \rangle$. We see that this swap has increased the number of runs. However, consider the neighborhood structure induced by the operator ν of Natural Merge Sort [69]. This algorithm iterates a pass over the input sequence where it merges the first and second run, then the third and fourth run, and so on. Each pass reduces the number of runs by half (almost). When a pass produces only one run, Natural Merge Sort terminates. Clearly, the guiding function of Natural Merge Sort is $Runs$. More interestingly, for each pass by Natural Merge Sort over the sequence, $Runs$ is monotonic with respect to sorting inversions. That is, monotonically reducing $Runs$ with Natural Merge Sort monotonically reduces Inv .

Relaxing the requirement of adjacency in swaps results in the measure Exc . Exc is the minimum number of exchanges (arbitrary swaps) required to sort the sequence.

More than 150 years ago, Cayley realized that $Exc(X) = |X| -$ the number of cycles in $\pi(X)$. Thus, the identity permutation has X cycles, and Exc is zero, the minimum.

A specific measure of presortedness may be more appropriate than others for the sorting algorithm at hand. Then, we may distinguish sorting methods depending on the suitability of the measure of presortedness. We name this attribute *presortedness measure*.

For example, if sorting operations are restricted to swaps of adjacent elements (i.e. the sequence of items is stored in a double-linked list), then the best algorithm is *Insertion Sort*, requiring $n + Inv(\langle x_1, \dots, x_n \rangle)$ comparison and swaps. *Insertion Sort* has the property that every swap suggested along its way to sorting a sequence monotonically reduces Inv . Therefore, Inv is a suitable *presortedness measure* for *Insertion Sort*.

This is not true of *Quicksort*. A single swap, or an entire partitioning step (that sends all elements less than the pivot to the left and those larger to the right) may also increase the value of Inv or Exc temporarily. However, Lemma 3.2 in [34] proves that *Quicksort* monotonically reduces Exc on the average. So, Exc is an appropriate *presortedness measure* for *Quicksort*.

Estivill-Castro [26] proposes a new (to our best knowledge) measure of presortedness (which we call Eng) [33, 79, 91]. This presortedness measure is based on the total difference between the position occupied by the element x_i after sorting X (that we call the $rank(x_i)$), and the current value of i . Formally, $Eng = \sum_i^n |rank(x_i) - i|$. When the sequence is ordered, $Eng = 0$, the minimum. Figure 5.6 displays an example for computing Eng .

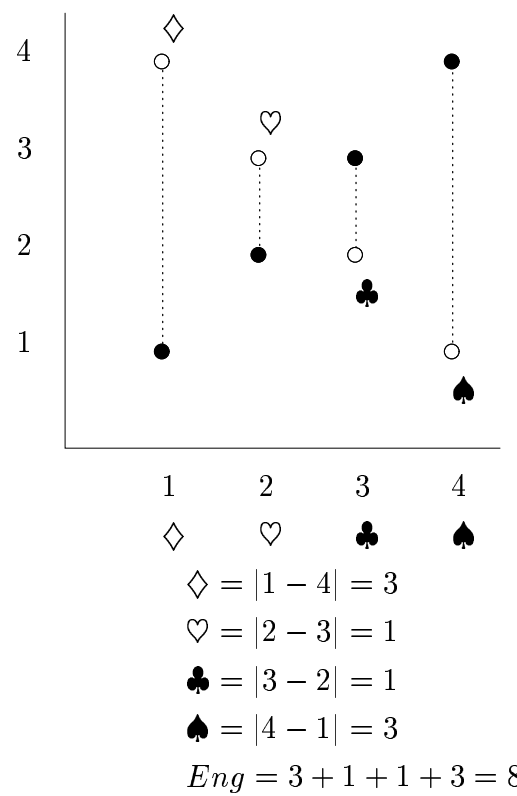


Figure 5.6: *The computation of Eng. The sequence $[\diamond, \heartsuit, \clubsuit, \spadesuit]$ is ordered and $[\spadesuit, \clubsuit, \heartsuit, \diamond]$ is the current sequence.*

We show here that a relevant characteristic of this measure of presortedness is that *Eng* does not increase its value when a swap is performed after a comparison predicate returns *False*. Because our demonstration does not make any other assumption about a particular sorting mechanism, our claim is valid for *Selection Sort*, *Quicksort* and *Insertion Sort*. That is, those 3 algorithms are well behaved as local searchers for the *Eng* measure of presortedness.

Proposition 1 *Let Q be a sorting algorithm that sorts any given sequence X in ascending order, such that swaps $s(x_i, x_j)$ are performed only if, for $i < j$ the comparison predicate $<(x_i, x_j)$ returns *False*. Then the measure *Eng* is never increased by the algorithm Q as it sorts X with such swaps.*

We demonstrate now that the value of Eng_f after a swap operation is performed is not greater than the value Eng_i before the swap operation.

First, note that any difference (if any) in the value of *Eng* can only be produced by the exchange of the items involved in the swap operation (that is, the elements x_i and x_j). Therefore, it is enough to compute the gradient ΔEng , instead of calculating all the differences $|rank(x_i) - i|$, for $i = 1, \dots, n$.

We have $i < j$ and $<(x_i, x_j)$ returned *False*. Thus $rank(x_i) > rank(x_j)$.

Case 1 The $rank(x_i) > i$.

Subcase 1 The $rank(x_j) < j$.

Figure 5.7 displays that in this sub-case $Eng_i > Eng_f$.

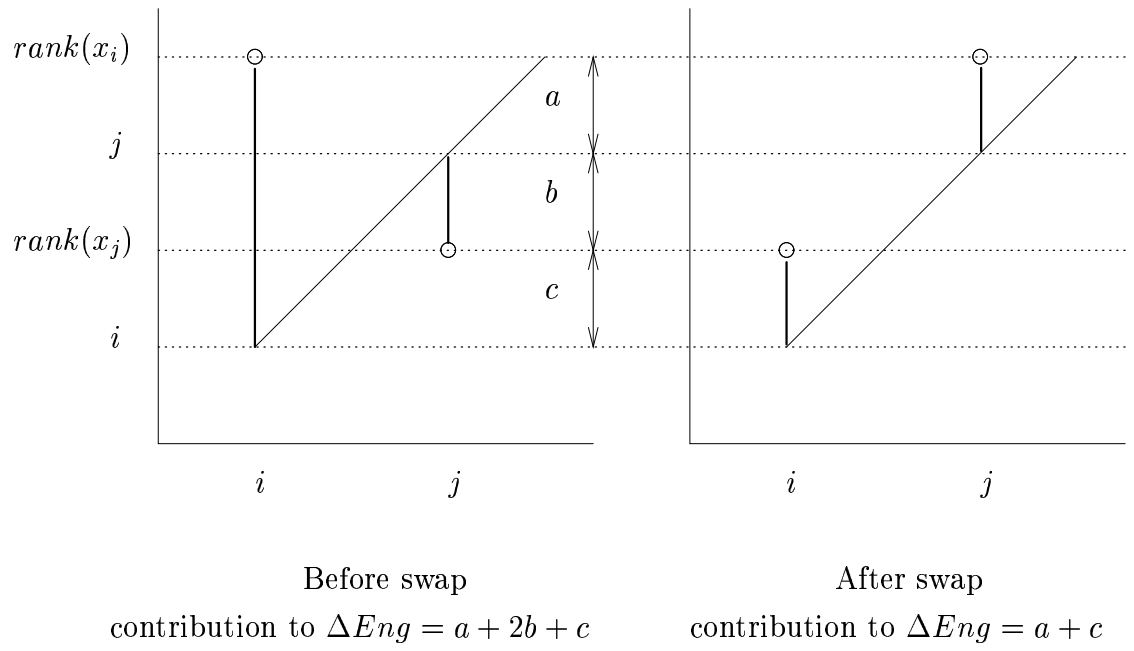


Figure 5.7: *Sub-case 1 where $rank(x_i) > i$ and $rank(x_j) < j$. Eng before swap is greater than Eng after swap.*

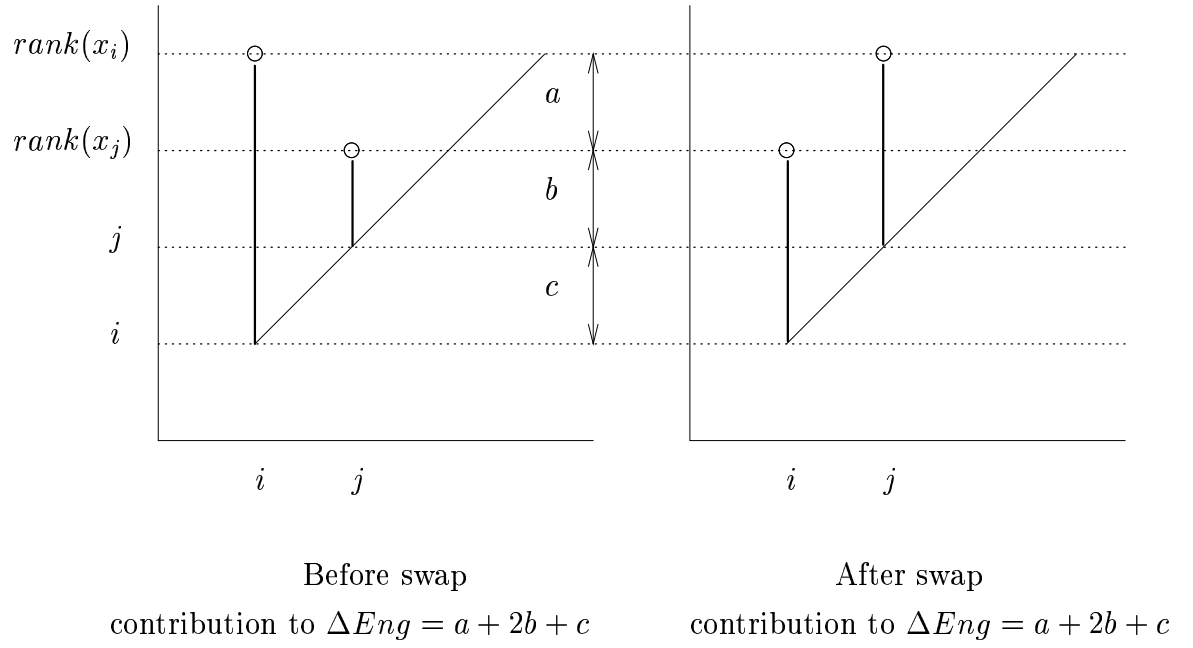


Figure 5.8: *Sub-case 2 where $rank(x_i) > i$ and $rank(x_j) \geq j$. Eng before and after the swap is the same.*

Subcase 2 The $rank(x_j) \geq j$.

Figure 5.8 displays that in this sub-case $Eng_i = Eng_f$.

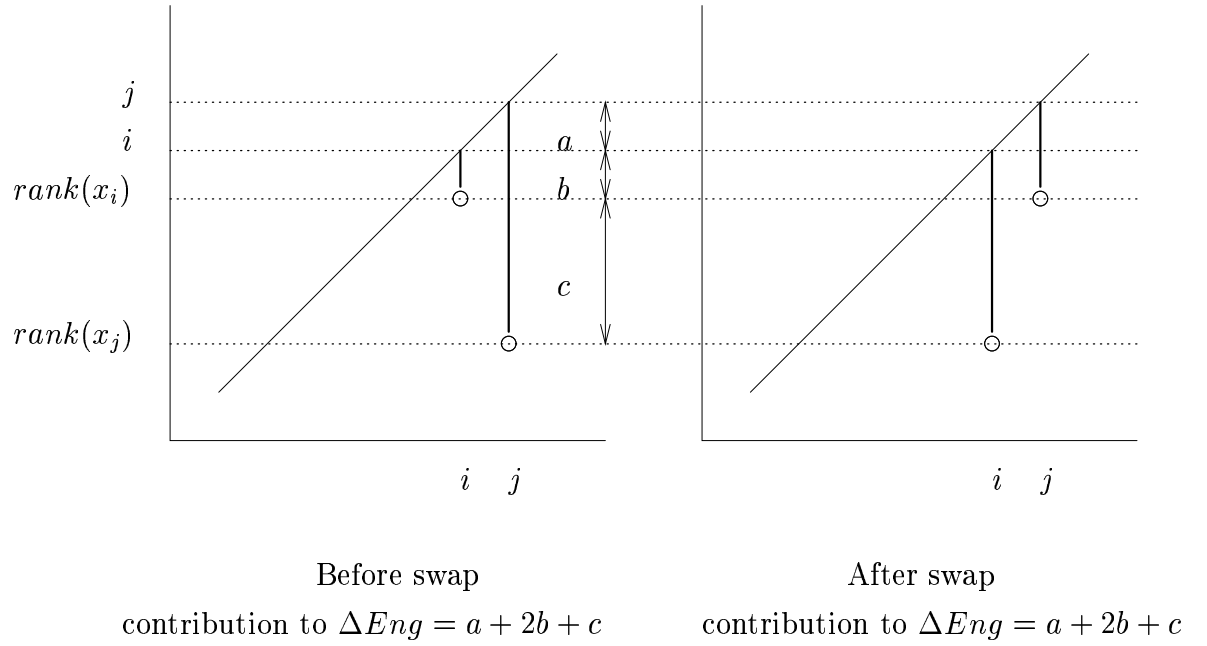


Figure 5.9: *Case 2 where $rank(x_i) \leq i$ and $rank(x_j) < j$. Eng before and after the swap is the same.*

Case 2 The $rank(x_i) \leq i$. In this case $j > i \geq rank(x_i) > rank(x_j)$. Thus, the $rank(x_j) < j$.

Figure 5.9 displays that in this case $Eng_i = Eng_j$.

□

Table 5.1 displays the values of the attributes previously described for *Insertion Sort*, *Quicksort* and *Selection Sort*.

Attribute	<i>Insertion Sort</i>	<i>Quicksort</i>	<i>Selection Sort</i>
<i>swap constraint</i>	adjacent items	none	none
<i>focus</i>	medium	high	low
<i>dependency on input</i>	yes	yes	no
<i>presortedness measure</i>	<i>Inv, Eng</i>	<i>Exc, Eng</i>	<i>Exc, Eng</i>

Table 5.1: *Attributes of Insertion Sort, Quicksort and Selection Sort.*

5.5 Implementation

Additionally, the impact of the sorting algorithm into the mutation, and thus into *MA-sorting*, can be regulated in two dimensions. The first dimension is concerned with the order of nominating neighbors. This dimension corresponds to the loop structure in the sorting algorithm. As a consequence, a second aspect comes into play. This second dimension is the amount of progress in the sorting process at each invocation of mutation; this corresponds to letting the LS structure of the sorting algorithm progress beyond the *Termination* step in our model of LS. We now explain these two dimensions.

5.5.1 Order of Nominating Neighbors

Recall from our LS model that the *Nomination* step may maintain a record of previous nominations. Also, note that a sorting algorithm maintains its state, namely, in which node of its decision tree it is currently operating. For *Insertion Sort* and

Selection Sort this state can be defined by two indices. However, for the purposes of implementation, we maintain a sequence of nominations that mimics the loop structure of these sorting algorithms.

Thus, information that determines a state or a set of states is used to reactivate the schedule of comparisons. When the information about a state determines uniquely one state of the sorting algorithm, the schedule of comparisons is resumed unambiguously from that state of the sorting algorithm. When the information defines not one state but a family of states, some rule completes the information to determine a state and then the schedule of comparisons is resumed from them.

For example, consider $\pi_i = \langle 3, 2, 4, 1 \rangle$ as the current permutation, and $k = 1$ as the first index of a state partially determined for *Selection Sort*. Because *Selection Sort* requires another index to uniquely determine its state, we can have a deterministic rule that always sets the second index l to $k + 1$ (in this case $l = 2$). Then, *Selection Sort* can resume its schedule of comparisons and suggest a new neighbor as a result of mutation, namely the new neighbor nominated is $\langle 2, 3, 4, 1 \rangle$. If now we request the next swap, we proceed with the next comparison suggested by *Selection Sort*. That is, when the second index increases to $l = 3$ and the schedule produces $\langle 4, 2, 3, 1 \rangle$.

In this case, we identify four modes of operation that have an impact on the order of nominating permutations:

1. *Multi-threading*.
2. *Single-threading*.
3. *Random*.
4. *Deterministic*.

The difference between 1 (*Multi-threading*) and 2 (*Single-threading*) is if the information about the state (that determines the continuation of the schedule of comparisons) is shared globally across chromosomes or not.

In a population-based approach, like Memetic algorithms, these two modes of operation are concerned with the proportion of nomination states to chromosomes in the population. With respect to the number of states, we can regard the nomination as multi-threaded (*MltThr*), with one thread per chromosome, or as a single thread (*SngThr*) for the entire population. In the first extreme each chromosome will carry, within its encoding, information regarding the state of its thread, and at the point where mutation is to be applied to it, the nomination step resumes accordingly. Thus, we have a different thread of the sorting algorithm per chromosome or, in other words, each chromosome records where it is in the schedule of comparisons. In the second extreme there is only one state, and although another chromosome is being mutated, the interaction with the schedule resumes at the point where the last globally performed mutation stopped.

As we explained before, the information about the state may only determine a family of states. So, where does information that determines the state come from? In our research we experiment with two modes of operation. The first one consists of randomly generating the current state of the schedule by the sorting algorithm (we call this *Rnd*). That is, among all states consistent with the information specifying a state, we select one uniformly. The second mode consists of defining a rule which deterministically generates a current state of the schedule by the sorting algorithm (we call this *Dtr*).

5.5.2 Amount of Progress

Along the dimension of amount of progress we consider four modes of operation:

1. *Minimal progress.*
2. *First improvement.*
3. *Maximal progress.*
4. *Best improvement.*

Consider a minimization problem. On *Minimal progress* mode (*MnmPrg*), if the neighbor $\nu(\pi_i)$ proposed by the sorting algorithm results in $f(\pi_i) \geq f(\nu(\pi_i))$, then the mutation is accepted and the sorting algorithm receives a success signal as if the corresponding comparison had resulted in *False*. For the purposes of the sorting algorithm, it now chooses the corresponding branch of operation of the evaluated comparison but stops there. If $f(\pi_i) < f(\nu(\pi_i))$, then the mutation is rejected (it is not applied). The sorting algorithm receives a signal as if the corresponding comparison has returned *True*. It now chooses the alternative branch of operation and halts again. In this mode of operation the sorting algorithm advances at most one level in the depth of its decision tree. The schedule of comparisons advances only to the next comparison.

A second alternative for this dimension is the *First improvement* mode (*FrsImp*). For this mode of operation the sorting mechanism repeatedly produces neighbors to test until it receives either a success signal or the advice that the list of neighbors

has been exhausted. In other words, if $f(\pi_i) < f(\nu(\pi_i))$, then another operation is requested. A sequence of operations is applied to progressively mutate the current permutation until $f(\pi_i) \geq f(\nu(\pi_i))$. In this mode of operation the mutation operator returns the chromosome responsible for the improvement.

However, for this same dimension, the sorting algorithm could be on *Maximal progress* mode (*MxmPrg*), so that, if $f(\pi_i) \geq f(\nu(\pi_i))$, then the mutation is accepted and another operation is requested. A sequence of operations is applied to progressively mutate and further improve the current permutation until $f(\pi_i) < f(\nu(\pi_i))$. At this point the mutation for that chromosome is terminated with the version that most improved the function f . The sorting algorithm advances its state until a comparison produces no improvement.

Finally, the *Best improvement* mode (*BstImp*) selects the best amongst all possible neighbors of $\nu(\pi_i)$. This mode of operation may be more costly than the others because it checks the corresponding value $f(\nu(\pi_i))$ of each possible neighbor of the current permutation.

Note that *Minimal progress* mode and *Best improvement* mode represent extremes. In minimal progress mode, only one hill-climber step in the direction suggested by the sorting algorithm is taken. In *Best improvement* mode, steps in the direction suggested by the sorting are taken until all the list of neighbors produced by $\nu(\pi)$ is exhausted and the best amongst them has been identified. Naturally, it is possible to mediate between these extremes in different ways. For example, allowing an argument to determine a (deterministic or random) amount of progress in the sorting algorithm per mutation.

During our research, with unconstraint combinatorial optimization problems, we carried out several experiments with the variants that result from applying *Insertion Sort* (*Ins*), *Quicksort* (*Qck*) and *Selection Sort* (*Slc*) with the following modes of operation: *MnmPrg*, *FrsImp*, *MxmPrg*, *BstImp*, *MltThr*, *SngThr*, *Rnd* and *Dtr*. Those variants are displayed in Tables 5.2, 5.3 and 5.4. The experimental results are described in the next Chapter.

Variant	Amount of Progress	Proportion of States	Current State
<i>Ins MnmPrgSngThr Rnd</i>	Minimal p ^{rogress}	Single-threaded	Random
<i>Ins MnmPrgSngThr Dtr</i>	Minimal p ^{rogress}	Single-threaded	Deterministic
<i>Ins MnmPrgMltThr Dtr</i>	Minimal p ^{rogress}	Multi-threaded	Deterministic
<i>Ins FrsImpSngThr Rnd</i>	First improvement	Single-threaded	Random
<i>Ins FrsImpSngThr Dtr</i>	First improvement	Single-threaded	Deterministic
<i>Ins FrsImpMltThr Dtr</i>	First improvement	Multi-threaded	Deterministic
<i>Ins Mx^mPrgSngThr Rnd</i>	Maximal progress	Single-threaded	Random
<i>Ins Mx^mPrgSngThr Dtr</i>	Maximal progress	Single-threaded	Deterministic
<i>Ins Mx^mPrgMltThr Dtr</i>	Maximal progress	Multi-threaded	Deterministic
<i>Ins BstImpSngThr Rnd</i>	Best improvement	Single-threaded	Random
<i>Ins BstImpSngThr Dtr</i>	Best improvement	Single-threaded	Deterministic
<i>Ins BstImpMltThr Dtr</i>	Best improvement	Multi-threaded	Deterministic

Table 5.2: *Variants using Insertion Sort.*

Variant	Amount of Progress	Proportion of States	Current State
<i>QckMnmPrgSngThrRnd</i>	Minimal p ^{rogress}	Single-threaded	Random
<i>QckMnmPrgSngThrDtr</i>	Minimal p ^{rogress}	Single-threaded	Deterministic
<i>QckMnmPrgMltThrDtr</i>	Minimal p ^{rogress}	Multi-threaded	Deterministic
<i>QckFrsImpSngThrRnd</i>	First improvement	Single-threaded	Random
<i>QckFrsImpSngThrDtr</i>	First improvement	Single-threaded	Deterministic
<i>QckFrsImpMltThrDtr</i>	First improvement	Multi-threaded	Deterministic
<i>QckMxmPrgSngThrRnd</i>	Maximal progress	Single-threaded	Random
<i>QckMxmPrgSngThrDtr</i>	Maximal progress	Single-threaded	Deterministic
<i>QckMxmPrgMltThrDtr</i>	Maximal progress	Multi-threaded	Deterministic
<i>QckBstImpSngThrRnd</i>	Best improvement	Single-threaded	Random
<i>QckBstImpSngThrDtr</i>	Best improvement	Single-threaded	Deterministic
<i>QckBstImpMltThrDtr</i>	Best improvement	Multi-threaded	Deterministic

Table 5.3: *Variants using Quicksort.*

Variant	Amount of Progress	Proportion of States	Current State
<i>SlcMnmPrgSngThrRnd</i>	Minimal p ^{rogress}	Single-threaded	Random
<i>SlcMnmPrgSngThrDtr</i>	Minimal p ^{rogress}	Single-threaded	Deterministic
<i>SlcMnmPrgMltThrDtr</i>	Minimal p ^{rogress}	Multi-threaded	Deterministic
<i>SlcFrsImpSngThrRnd</i>	First improvement	Single-threaded	Random
<i>SlcFrsImpSngThrDtr</i>	First improvement	Single-threaded	Deterministic
<i>SlcFrsImpMltThrDtr</i>	First improvement	Multi-threaded	Deterministic
<i>SlcMx^mPrgSngThrRnd</i>	Maximal progress	Single-threaded	Random
<i>SlcMx^mPrgSngThrDtr</i>	Maximal progress	Single-threaded	Deterministic
<i>SlcMx^mPrgMltThrDtr</i>	Maximal progress	Multi-threaded	Deterministic
<i>SlcBstImpSngThrRnd</i>	Best improvement	Single-threaded	Random
<i>SlcBstImpSngThrDtr</i>	Best improvement	Single-threaded	Deterministic
<i>SlcBstImpMltThrDtr</i>	Best improvement	Multi-threaded	Deterministic

Table 5.4: *Va^{riants} using Selection Sort.*

Chapter 6

Experiments with *MA-sorting*

6.1 Introduction

Our intention is to apply *MA-sorting* to harder problems than sorting, and in particular, to minimization problems searching for permutations. A benchmark problem emerges from the use of graphs in pattern matching [87, 134, 141]. Graphs are combinatorial objects that have been widely used in applications where structured objects emerge in a natural way. Remarkably, in the pattern-matching arena, modeling with graphs has been fruitfully used to match objects by matching the corresponding graph models [87, 134, 141]. Hence, the interest in finding efficient algorithms to deal with the Graph Isomorphism problem.

6.2 Harder Problems than Sorting

Graph Isomorphism (GI) [110, 143] is an important problem in Graph Theory, whose precise computational complexity remains unknown [87]. It requires to find a bijection of the vertices so that the edge structure is the same. Labeling the vertices from the same set corresponds to finding a permutation π . In what follows G is a graph, $V(G)$ denotes the vertices of G and $E(G)$ denotes the edges of G .

Definition 10 (GI) *An isomorphism from G to H is a bijection $\Upsilon : V(G) \rightarrow V(H)$ such that $yz \in E(G)$ if and only if $\Upsilon(y)\Upsilon(z) \in E(H)$. We say “ G is isomorphic to H ”, written $G \cong H$ if there is an isomorphism from G to H .*

However, Graph Isomorphism is not a minimization problem. The interesting aspect is that in practice, a close variant is a hard minimization problem. In real world applications of pattern matching, the existence of noise, distortion, uncertainty or measurement errors, together with weights associated with nodes and edges, translates the GI problem into its inexact version: the inexact Graph Isomorphism (iGI) or Error-Correcting Graph Isomorphism (ECGI) [134]. To define this problem we first need the notion of attributed graph [134].

Definition 11 (AG) *An attributed graph is a 4-tuple $G_a = (V, E, \alpha, \beta)$ where $V \neq \emptyset$ is a finite nonempty set of vertices; $E \subset V \times V$ is a set of distinct ordered pairs of distinct elements in V called edges; $\alpha : V \rightarrow \mathbb{R}$ is a function called vertex interpreter; and $\beta : E \rightarrow \mathbb{R}$ is a function called edge interpreter.*

Definition 12 (ECGI) *Given two attributed graphs $\mathbf{AGs}$,*

$$G_a = (V(G_a), E(G_a), \alpha_G, \beta_G)$$

and

$$H_a = (V(H_a), E(H_a), \alpha_H, \beta_H),$$

with $|V(G_a)| = |V(H_a)|$, the Error-Correcting Graph Isomorphism problem is to find a permutation $\pi : V(G_a) \rightarrow V(H_a)$ so that some metric of total dissimilarity between the graph $G'_a = (\pi[V(G_a)], \pi[E(G_a)], \alpha_{G'}, \beta_{G'})$ and the graph H_a is minimized.

To illustrate the virtues of our *MA-sorting* we will describe its application to this hard combinatorial minimization problem. This is also a benchmark because GAs have been previously shown to be effective [141].

6.3 A Benchmark of Graphs

We reproduce the experimental setting of Wang *et al.* [141]. Their method for the construction of instances of the ECGI problem provides the optimal solution, and these instances constitute a benchmark for testing the effectiveness of GAs. While this family of graphs represents only some instances of the ECGI problem, they constitute a parameterized model with respect to the size of the graph and the separation between the graphs that constitute a problem instance. Attributed undirected graphs are encoded as edge-weight adjacency matrices. Let $n = |G_a(V)|$ be the number of vertices of G_a and $M(G_a) = [m_{i,j}]$ be the $n \times n$ edge-weight matrix of the attributed undirected graph G_a given by $m_{i,i} = \alpha(v_i)$ and $m_{i,j} = \beta(v_i, v_j)$. Note

that two graphs are isomorphic only if their edge-weight adjacency matrices differ by a permutation of rows and columns, that is, if $M(H_a) = P \cdot (G_a) \cdot P^T$, where P and P^T are the matrix representation of a permutation π and its transpose, respectively (i.e. P is a permutation matrix).

The construction of Wang *et al.* assigns integer values in $[0, 100]$ to each cell $m_{i,j}$ of the matrix $M(G_a)$, independently and uniformly (each integer entry $m_{i,j}$ is chosen with uniform probability $1/101$ and independently of other entries). In order to build the graph H_a (the isomorphic graph), a random permutation π^* is uniformly constructed (each permutation has probability $1/n!$). The permutation matrix P is π^* applied to the columns of the identity matrix.

The construction of the graph H'_a starts with a copy of the graph G_a . We add noise to the graph H'_a for the matching to be inexact. For each entry in the upper triangle of $M(H'_a)$ an integer is selected uniformly in the interval $[-\varepsilon, +\varepsilon]$ (that is, with probability $1/(2\varepsilon + 1)$), where $\varepsilon \in [0, 20]$ is a parameter of the experiment regulating the approximate mismatch in the pair of graphs. In this process, noise is added only to the upper triangle and changes are reflected to keep an undirected graph. We then apply the permutation π^* by the use of the permutation matrix P . The resulting matrix is the edge-weight adjacency matrix for the attributed graph H_a ; that is $M(H_a) = P \cdot M(H'_a) \cdot P^T$. The minimization algorithms will receive as input the pair G_a, H_a and we hope that they will recover π^* . Measuring to what extent the algorithms recover π^* allows us to evaluate the quality of the optimization.

6.3.1 Fitness Function

In Definition 12, we left open the formulation of a criterion of dissimilarity between attributed graphs. Wang *et al.* conducted experiments with at least two fitness functions, and concluded that the absolute total error (**ATE**) in the entries of the edge-weight adjacency matrices is more accurate.

Definition 13 ($ATE(\pi)$) *Given two attributed graphs G_a and H_a , the Absolute Total Error of a permutation π is the 1-norm of the matrix $M(G_a) - P \cdot M(H_a) \cdot P^T$, where P is the permutation matrix of π . The explicit form of the Absolute Total Error is given by $ATE(\pi) = \sum_{i=1}^n \sum_{j=1}^n |m_{i,j} - m_{\pi(i),\pi(j)}|$.*

Wang *et al.* re-write this as a maximization problem: find the permutation π that maximizes $Mx_ATE(\pi)$ given by $Mx_ATE(\pi) = C'_{max} - ATE(\pi)$, where C'_{max} is the maximum possible value of $ATE(\pi)$.

6.3.2 Algorithms for ECGI

Wang *et al.* show that GAs perform much better than other optimization approaches for the ECGI. They use a heuristic named *status matching*. Essentially, vertices have a ‘status’ (the sum of the weights in their row of the adjacency matrix). The vertices in G and H are ranked by status and the initial permutation is the mapping between the rankings in G and in H . Here we reproduce their description [141].

“Let G be an attributed relational graph. The status $s(v)$ of a vertex v in G is defined as

$$s(v) = \alpha(v) + \sum_{v' \in V} \beta(v, v')$$

which is the sum of its attribute and the weighted values of edges which are from v to other vertices in G . The *status list* $S(G)$ is a list of status of vertices in G . The canonical status list $CanS(G)$ is a sorted status list in ascending order and is defined formally as

$$CanS(G) = \{s_i \mid \forall j > i, s_j > s_i\}$$

where s_i and s_j are the status in positions i and j . A permutation corresponding to $CanS(G)$ is defined as

$$\sigma_{CanS(G)}(i) = t, \quad v_t \in G \quad \text{and} \quad s(v_t) = s_i.$$

The status matching $p_{G,H}$ of graphs G and H is defined as:

$$\begin{aligned} p_{G,H}(i) &= t, \quad v_i \in G, \quad v_t \in H, \quad \sigma_{CanS(G)}^{-1}(i) \\ &= \sigma_{CanS(H)}^{-1}(t) \end{aligned}$$

where σ^{-1} is an inverse mapping of permutation σ . $p_{G,H}$ is also a permutation. However, it is a mapping of vertices between $CanS(G)$ and $CanS(H)$ ”.

While they left open how well this heuristic performs on its own, they do insert one chromosome obtained with status matching into the initial randomly-generated population of their GA. We have found that the heuristic gives a reasonably good approximated solution; better when noise levels are low (i.e. ε is close to 0). Wang *et al.* also left open how their GA performs without this initial seeding of the population. According to our experiments, the observed performance of the GA is better if the population is seeded with the chromosome that results from *status matching*.

6.3.3 Our implementation of *MA-sorting*

We implemented our *MA-sorting* and left fixed most of the common aspects with Wang *et al.* to demonstrate that hybridization as proposed here is the source of the much-improved optimization. That is, our experiments are designed to illustrate that our mutation operator *SORT* achieves a hybridization with sorting methods that speeds up convergence to much better solutions.

<i>Parameter</i>	<i>MA-sorting</i>
Encoding Scheme	Integer Permutation
Population Size	30
Selection	Inhibitive
Scaling	Ranking Scaling
Termination	Upon Thresh Convergence
Crossover	PMX
Mutation	<i>SORT</i>
Crossover Probability	0.9
Mutation Probability	.1
Number of Generations	300
Elitism	<i>True</i>

Table 6.1: *GA Parameters.*

The operators and parameters of the GAs used in our experiments are summarized in Table 6.1. Operators like *PMX* [45] are well known in the GA community when

applied to optimization problems searching for permutations. A description of the *PMX* operator and Ranking Scaling can be found in Chapter 2. Wang *et al.* propose an Inhibitive Selection operator to prevent premature convergence. Essentially, Inhibitive Selection limits the number k of copies each chromosome has in the population. We reproduce here their description [141].

“Suppose $P^t = (a_1^t, a_2^t, \dots, a_\lambda^t) \in I^\lambda$ denotes the population at generation t , where λ is the population size and I is the space of chromosomes a_i^t . An inhibitive selection operator is defined as

$$\exists k \in 1, 2, \dots, \lambda - 1, \quad \forall a_i^t \in I^t, \quad |\text{class}(a_i^t)| \leq k$$

where $\text{class}(a_i^t)$ is the set of chromosomes having the same value of fitness function as a_i^t , and $|\text{class}(a_i^t)|$ is the number of $\text{class}(a_i^t)$.

The parameter k , called *inhibition value*, ensures that for every chromosome in the population, its copies in the population is limited within k no matter how it is fitted”.

We use the fitness value of the best chromosome π_{found} in the final population to assess the quality of solutions. We compute the ratio between $Mx_ATE(\pi_{found})$ and the optimum value $Mx_ATE(\pi^*)$ (recall that permutation π^* was used to construct H_a from G_a). This ratio is called *correctness* and is given by

$$correctness = 1 - \frac{|Mx_ATE(\pi_{found}) - Mx_ATE(\pi^*)|}{Mx_ATE(\pi^*)}.$$

The stop condition applied by Wang *et al.* (that we call here Termination Upon Thresh Convergence) is based on the assumption that the expected mean of the absolute total error **ATE** (given by Definition 13) is very close to $(\varepsilon/2)n^2$. Thus, the GA should stop if $ATE(\pi)$ is less than its expected mean value. We apply the same stop condition but we add the requirement that the number of generations should not be greater than a given parameter.

6.4 Experimental Description

The performance of LS algorithms is highly determined by three of its components: neighborhood operator, number of neighbors explored, and schedule of exploration. The major contribution to the performance comes from the quality of the neighborhood structure, which is generated by the neighborhood operator. Thus, it is reasonable to expect that a fruitful neighborhood structure will provide solutions of better quality. In particular, we conjecture that a neighborhood structure is fruitful when the spread of *Focus* is more even and there is a locality in the swaps that is not distorted by other genetic operators (mainly by crossover). If we are dealing with a fruitful neighborhood and we explore a large number of neighbors (as long as the neighborhood is not too large), we will have even better results. Of course, if we explore a quite small number of neighbors the potential of the neighborhood structure will not be fully realised and the improvement will be marginal. Finally, the schedule of exploration may influence the performance. The effects of the schedule of exploration into the performance will be more evident when coping with a fruitful neighborhood structure and exploring more neighbors.

Our hypothesis is that a similar behavior can also be expected in Memetic Algorithms which apply local search to a small number of individuals in the population, as our *MA-sorting* does. Thus, we expect to witness a larger contribution to performance by the neighborhood structure implicitly generated by the sorting algorithm. We also expect the benefits of a fruitful neighborhood structure to be intensified by the exploration of a larger number of neighbors. Finally, we expect to witness that the impact of the schedule of exploration will be more evident when dealing with a fruitful neighborhood structure and exploring a larger number of neighbors.

We performed two sets of experiments. In the first set of experiments, we show the impact of sorting into mutation. In the second set of experiments, we compare the performance of *MA-sorting* with the approach of Wang *et al.* (we allude to it as GBSA, because originally Wang *et al.* referred to it as genetic-based search approach).

To evaluate the impact of the sorting algorithm into the mutation, we consider three components.

1. Neighborhood structure.
2. Amount of progress.
3. Generation of state.

In order to present the results obtained from the first set of experiments, we conceptualize those three components as the axes of a three-dimensional space. To facilitate the visualization of this three-dimensional space of results, we project such space on 3 hierarchical groups. Thus, we report our results in three groups, each group corresponds to each projected hierarchy.

The first hierarchy considers the neighborhood structure implicitly generated by sorting algorithms. We evaluate, in terms of effectiveness and efficiency, each sorting algorithm (*Insertion Sort*, *Quicksort* and *Selection Sort*) with each amount of progress (*MnmPrg*, *MxmPrg*, *FrsImp* and *BstImp*), across each mechanism for generating the state of the sorting algorithm (*SngThrRnd*, *SngThrDtr* and *MltThrDtr*).

The second hierarchy considers the amount of progress first. This parameter regulates the number of individuals in the local neighborhood to be explored. We evaluate, in terms of effectiveness and efficiency, each method for the amount of progress (*MnmPrg*, *MxmPrg*, *FrsImp* and *BstImp*) with each mechanism for generating the state of the sorting algorithm (*SngThrRnd*, *SngThrDtr* and *MltThrDtr*), across *Insertion Sort*, *Quicksort* and *Selection Sort*.

Finally, the third hierarchy considers the mechanism which generates the state of the sorting algorithm first. This parameter is relevant because it influences the schedule of the sorting algorithm. We compare, in respect to effectiveness and efficiency, each mechanism to generate the state of the sorting algorithm (*SngThrRnd*, *SngThrDtr* and *MltThrDtr*) with each amount of progress (*MnmPrg*, *MxmPrg*, *FrsImp* and *BstImp*), across *Insertion Sort*, *Quicksort* and *Selection Sort*.

In the second set of experiments, while we compare the quality of solutions provided by *MA-sorting* and GBSA, we test the impact of increasing the number of individuals explored by LS while inside *MA-sorting*. This time we increase the number of explorations by regulating the outer cycle in our LS model (described in Section 3.1). Also we evaluate the impact of providing a better quality initial solution (as explained in Section 3.1 and illustrated in Section 3.2.4) to our *MA-sorting*. For this second set of experiments, our hypothesis is that *MA-sorting* will also replicate the behavior of a local search mechanism. That is, the exploration of a larger number of neighbors will intensify the benefits of a fruitful neighborhood structure, and a better quality initial solution will improve the performance of *MA-sorting*.

Applying the methodology described in Section 6.3, we constructed 50 pairs of graphs for each integer value of $\varepsilon \in [0, 20]$, and each size $n \in \{10, 20, 30, 40\}$ (the largest graph size tested by Wang *et al.* was only $n = 20$). For each graph pair we executed three times the GA and selected the best solution found. So, we generated 4200 test cases, and executed 12600 times each Genetic Algorithm.

In order to carry out our experiments, we used the *GAlib* genetic algorithm software package, written by Matthew Wall at the Massachusetts Institute of Technology [140].

6.5 Impact of Sorting into Mutation

In the first set of experiments, we evaluate the impact of the sorting algorithm into the mutation with respect to effectiveness (correctness) and efficiency (computational cost). For this purpose, we implemented 36 variants of *MA-sorting*, as described in Section 5.5.

We have explained in Section 5.5.1 that the information which determines a state or a set of states is used to reactivate the schedule of comparisons. We have also mentioned that when the information about a state determines uniquely one state of the sorting algorithm, the schedule of comparisons is resumed unambiguously from that state of the sorting algorithm. In this first set of experiments the information provided to our algorithms about the state of sorting is partial; it consists of just one index. Thus, this partial information defines not one state but a family of states. Therefore, we apply a deterministic rule to complete the information that determines the state, and then the schedule of comparisons is resumed from it.

As we have explained in Section 5.4.1, given a comparison tree T and a sorting algorithm Q , an execution state of Q is any piece Ξ of information that uniquely identifies a branch of the comparison tree T . We describe now the partial information, which concerns the state, provided to *Insertion Sort*, *Quicksort* and *Selection Sort* during the first set of experiments.

For *Insertion Sort*, as we explained in Section 5.4.1, the state can be succinctly encoded as a pair $\Xi = (j, i)$ with $1 \leq j < i < n$. Recall that this state means that $x_1, \dots, x_i$ are sorted and x_{i+1} is less than x_j . Thus, $x_j, x_{j+1}, \dots, x_i$ is being shifted to $x_{j+1}, x_{j+2}, \dots, x_{i+1}$, and x_{i+1} is being placed in x_j . So the next comparison in the schedule is $< (j - 1, j)$ (unless $j = 1$ in which case i is incremented and the comparison is $< (i, i + 1)$). The partial information provided to *Insertion Sort* is the index i . The deterministic rule applied in this case consists of defining $j = i$, and the first comparison is $< (j, i + 1)$. Further comparisons follow those branches prescribed by the tree T of *Insertion Sort*.

Selection Sort, as we also explained in Section 5.4.1, uses the following schedule T of comparisons

$$\begin{array}{ccccccc}
 (1, 2) & (1, 3) & (1, 4) & \cdots & (1, n) \\
 & (2, 3) & (2, 4) & \cdots & (2, n) \\
 & & (3, 4) & \cdots & (3, n) \\
 & & & \ddots & \vdots \\
 & & & & (n - 1, n).
 \end{array}$$

Recall that in *Selection Sort* a state could be encoded as a pair $\Xi = (k, j)$ with $1 \leq k < j < n$. Such a state means that: 1) the prefix $x_1, \dots, x_k$ is sorted; 2) these are the k -smallest items in the input; and 3) the item x_{k+1} is the smallest

in $x_{k+1}, \dots, x_j$. The partial information provided to *Selection Sort* is the index k . The deterministic rule that we apply consists of defining $j = k + 2$, and the first comparison is $< (k + 1, j)$. Further comparisons follow those branches prescribed by the tree T of *Selection Sort*.

Quicksort, as we explained in Section 5.4, divides the sequence of items to be sorted into two subsequences, a left and a right part, and then calls the *Quicksort* procedure recursively to sort the two parts. This process is called partitioning. Recall that during partitioning, we choose a pivot ϑ and arrange that all the items in the left part are less than the pivot and all those in the right part are greater. Remember that this process keeps a left pointer l and a right pointer r ; pointer l scans from the left and pointer r scans from the right. The partial information provided to *Quicksort* is the index i to the pivot ϑ . The deterministic rule that we apply consists of defining $l = 1$ and $r = n$, and the first comparisons are $< (i, 1)$ and $< (i, n)$. Further comparisons follow those branches prescribed by the tree T of partitioning. Observe that for *Quicksort*, each call to mutation with *Best improvement* has a cost of n . In contrast, for *Selection Sort*, each call to mutation with *Best improvement* has an expected cost of $n/2$.

While we provide the same amount of partial information (one index) to *Insertion Sort*, *Selection Sort* and *Quicksort*, the quality of such partial information is not the same for those sorting algorithms. Note that for *Quicksort*, in contrast with *Insertion Sort* and *Selection Sort*, a pair of indices is not enough to uniquely determine its state, we also need information about the current state of the stack (if *Quicksort* is implemented recursively). This is why, during the first set of experiments, we make reference to *Partition* instead of making reference to *Quicksort*.

However, this makes all neighbor schedules tested in our experiments to rely on the same amount of information since it is specified by $\log n$ bits.

Figures that display correctness include plots that correspond, from left to right and from top to bottom, to the four values for the size n of the graph. In the plots, the x -axis shows the range of ε values. The y -axis is a correctness scale, from 70% correct to 100% correct. The plotted lines are average *correctness*, taken over the 50 problems. Labels in figures are strings with four letters. The first letter indicates the sorting method, with values: I (*Insertion Sort*), P (*Partition*) or S (*Selection Sort*). The second letter represents the amount of progress, with values: m (*MnmPrg*), M (*MxmPrg*), F (*FrsImp*) or B (*BstImp*). The third letter, stands for the number of threads, with values: t (*SngThr*) or T (*MltThr*). Finally, the fourth letter indicates the mechanism for generating the state: R (*Rnd*) or D (*Dtr*). Table 6.2 summarizes the translation for labels in all figures.

Tables that report computational cost display the number of evaluations of the objective function used by each algorithm. These values are the average taken over the whole range of noise levels for each size of the graph (recall that the value at each noise level is the average over 50 problems). Labels in tables are shown in Table 6.2.

6.5.1 Sorting Method

In Section 5.4 we explain that, in our method, sorting algorithms determine the neighborhood structure used by LS.

Name	Label in Tables	Label in Figures
<i>Insertion Sort</i>	<i>Ins</i>	I
<i>Partition</i>	<i>Prt</i>	P
<i>Selection Sort</i>	<i>Slc</i>	S
<i>Minimal progress mode</i>	<i>MnmPrg</i>	m
<i>Maximal progress mode</i>	<i>MxmPrg</i>	M
<i>First improvement</i>	<i>FrsImp</i>	F
<i>Best improvement</i>	<i>BstImp</i>	B
<i>Single-threading</i>	<i>SngThr</i>	t
<i>Multi-threading</i>	<i>MltThr</i>	T
<i>Deterministic</i>	<i>Dtr</i>	D
<i>Random</i>	<i>Rnd</i>	R

Table 6.2: *Translation for labels in Figures and Tables.*

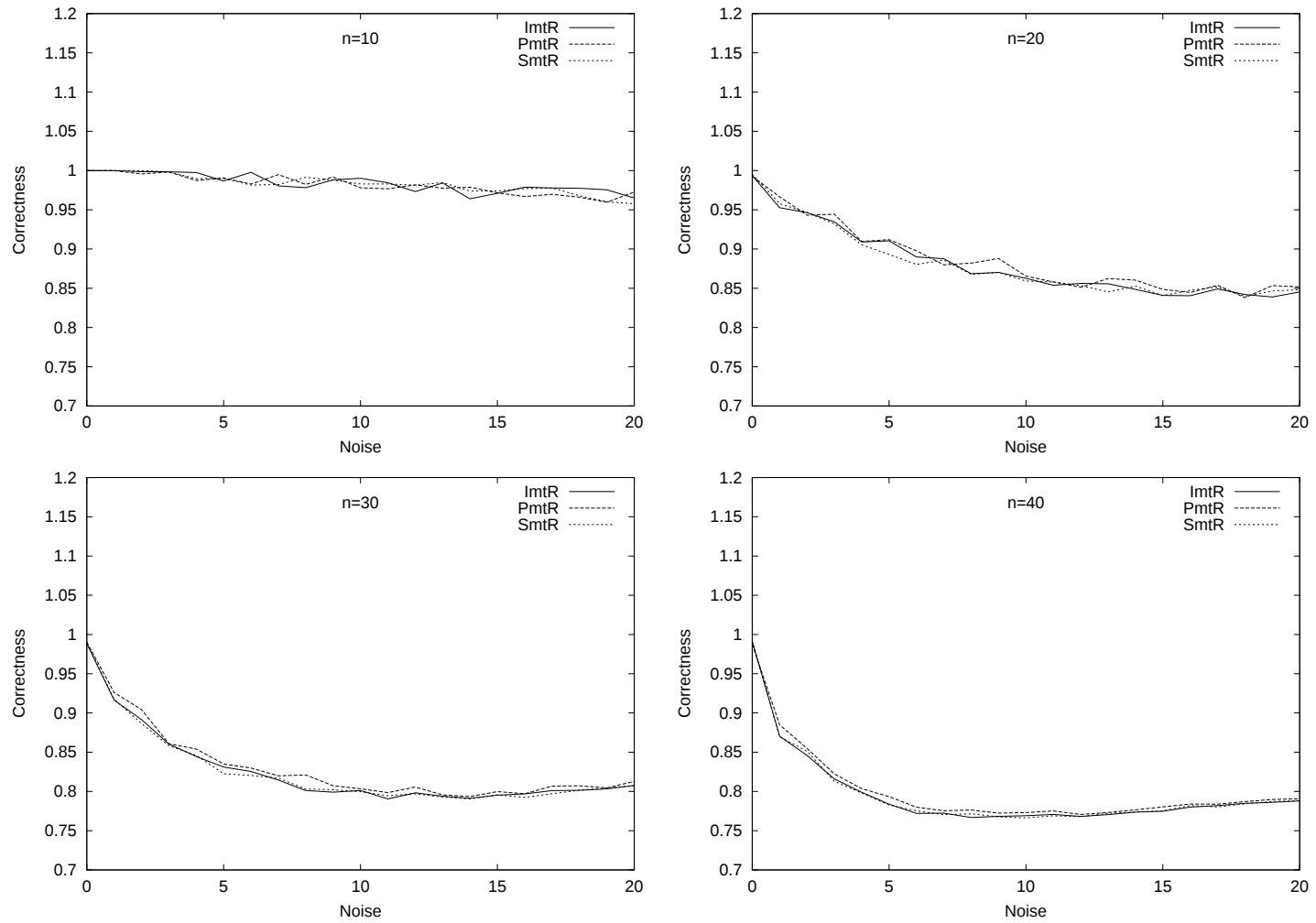
To evaluate the impact of the neighborhood structure on *MA-sorting*, we performed several experiments using three methods:

1. *Insertion Sort (Ins)*,
2. *Partition (Prt)* and
3. *Selection Sort (Slc)*.

We compare these three mechanisms in the context of four methods that regulate the amount of progress (*MnmPrg*, *MxmPrg*, *FrsImp* and *BstImp*), and using three different modes of generating the state of the sorting algorithm (*SngThrRnd*, *SngThrDtr* and *MltThrDtr*).

For *Minimal progress mode*, the dotted lines in Figure 6.1, Figure 6.2 and Figure 6.3 indicate results for our *MA-sorting* instantiated with *SlcMnmPrg*, the dashed lines indicate results for our *MA-sorting* instantiated with *PrtMnmPrg*, while the solid lines correspond to *InsMnmPrg*.

These figures illustrate that, for *Minimal progress mode* (for all tested graph sizes and different noise levels) differences in effectiveness are not important across *SngThrRnd*, *SngThrDtr* and *MltThrDtr*. Consequently, the quality of results is not largely influenced by changing the sorting method in the case of *Minimal progress mode*.

Figure 6.1: *Minimal progress mode. State randomly generated.*

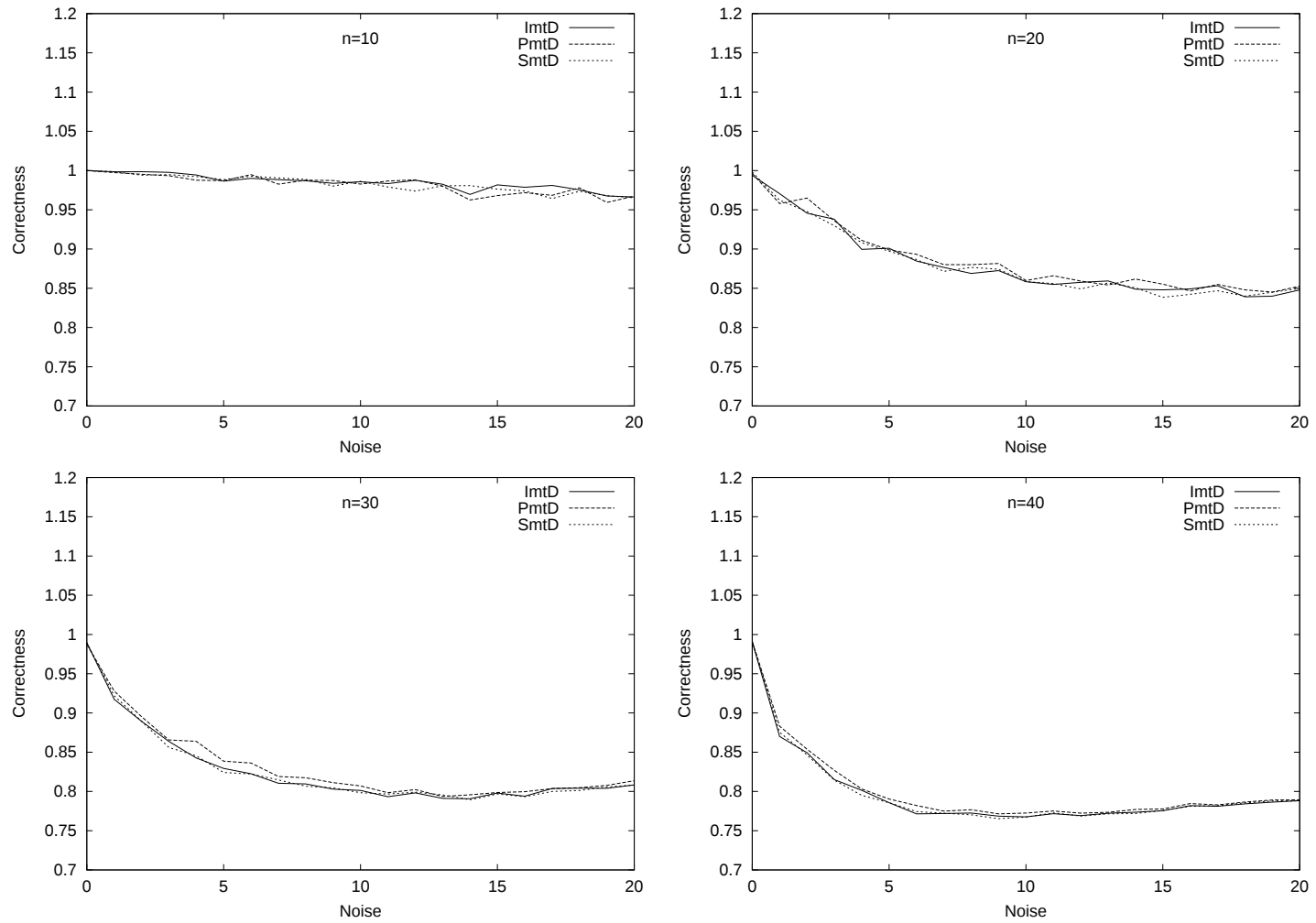


Figure 6.2: *Minimal progress mode. State deterministically generated, single-threading.*

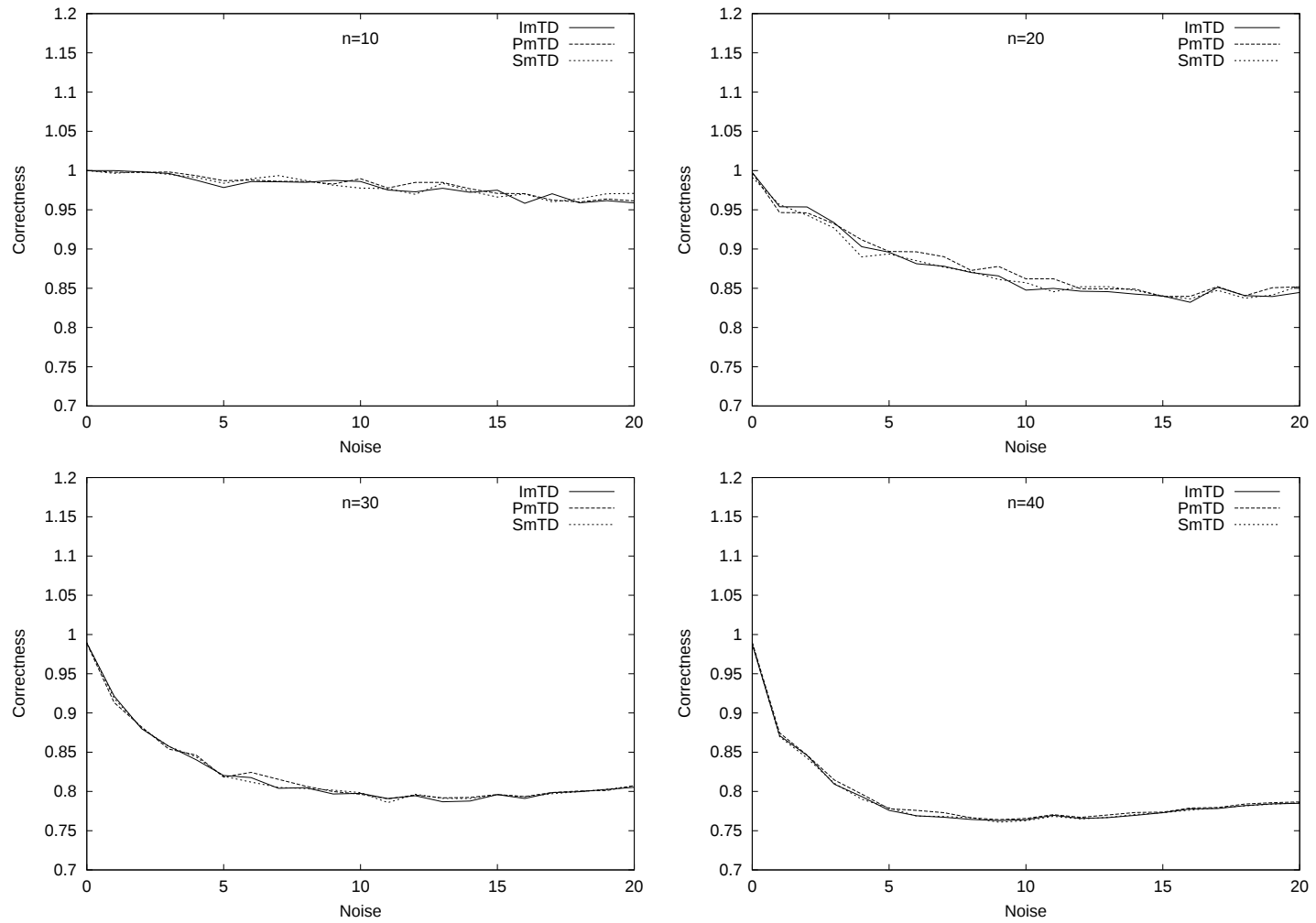


Figure 6.3: *Minimal progress mode. State deterministically generated, multi-threading.*

Table 6.3, Table 6.4 and Table 6.5 show that, for the *Minimal progress mode*, the variant *Partition* is the most expensive, in terms of number of evaluations of the objective function, and that the variants *Insertion Sort* and *Selection Sort* have nearly the same computational cost. These tables also show that the number of evaluations is almost the same as the size of the graph grows.

In effectiveness and efficiency, and according to our results, the variants *Insertion Sort* and *Selection Sort* are better than *Partition* when a *Minimal progress mode* is applied.

Sorting Algorithm	Size of the problem			
	$n = 10$	$n = 20$	$n = 30$	$n = 40$
<i>Insertion Sort</i>	24686	25865	26293	26502
<i>Partition</i>	30417	31363	31703	31875
<i>Selection Sort</i>	24682	25866	26290	26510

Table 6.3: *Number of evaluations of the objective function for Minimal progress mode with Single-threading (where the information about the state is deterministically generated), and for different graph sizes.*

For *Maximal progress mode*, the dotted lines in Figure 6.4, Figure 6.5 and Figure 6.6, indicate results for our *MA-sorting* instantiated with *SlcMxmPrg*, the dashed lines indicate results for our *MA-sorting* instantiated with *PrtMxmPrg*, while the solid lines correspond to *InsMxmPrg*.

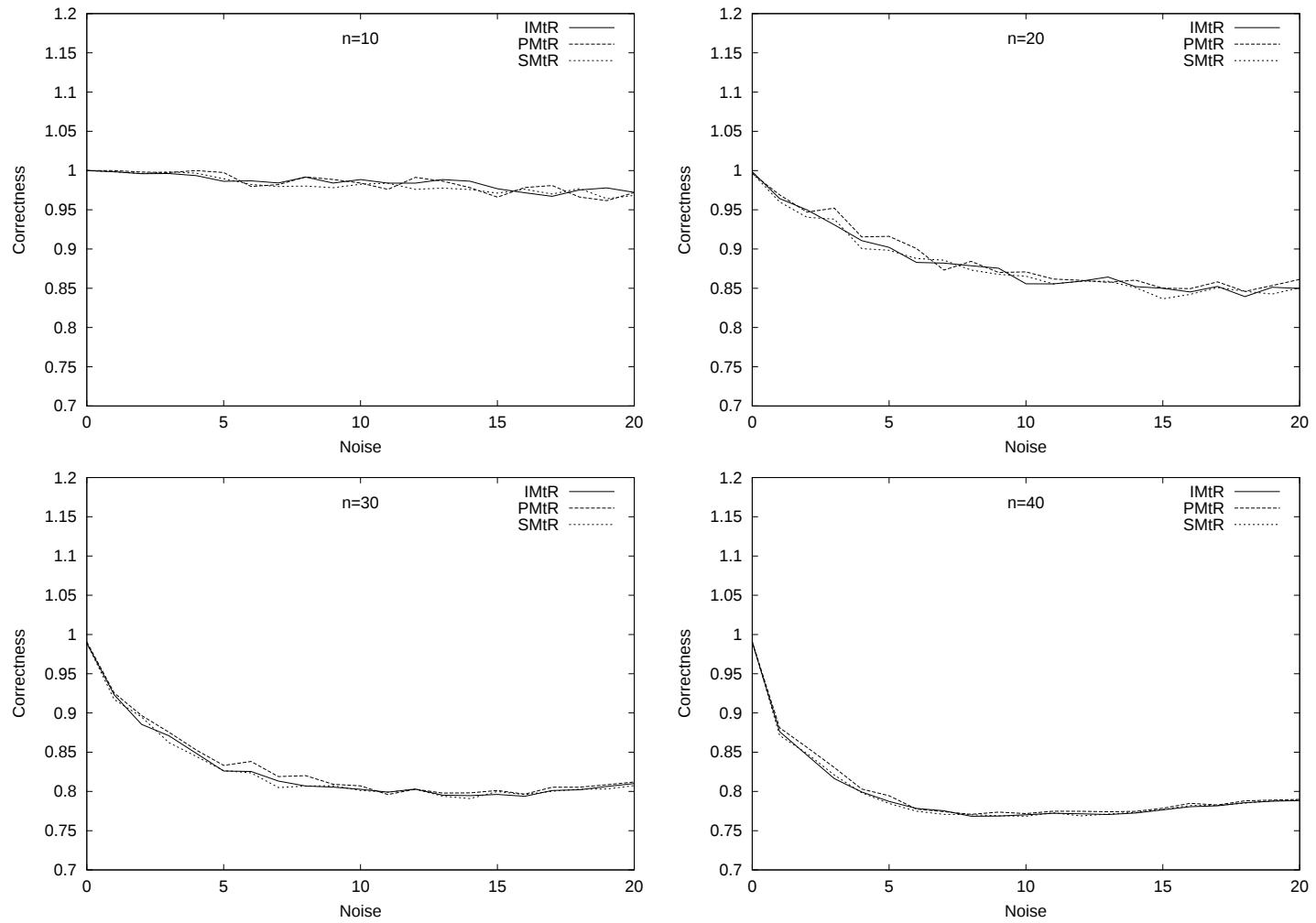
Sorting	Size of the problem			
Algorithm	$n = 10$	$n = 20$	$n = 30$	$n = 40$
<i>Insertion Sort</i>	24713	25900	26319	26535
<i>Partition</i>	30449	31393	31735	31902
<i>Selection Sort</i>	24709	25898	26323	26537

Table 6.4: *Number of evaluations of the objective function for Minimal progress mode with Single-threading (where the information about the state is randomly generated), and for different graph sizes.*

Sorting	Size of the problem			
Algorithm	$n = 10$	$n = 20$	$n = 30$	$n = 40$
<i>Insertion Sort</i>	24597	25808	26199	26392
<i>Partition</i>	30464	31341	31726	31949
<i>Selection Sort</i>	24764	25988	26386	26575

Table 6.5: *Number of evaluations of the objective function for Minimal progress mode with Multi-threading (where the information about the state is deterministically generated), and for different graph sizes.*

These figures illustrate that, for *Maximal progress mode* (for all tested graph sizes and different noise levels) differences in effectiveness are not important across *Sng-ThrRnd*, *SngThrDtr* and *MltThrDtr*. Consequently, the quality of results is not largely influenced by changing the sorting method in the case of *Maximal progress mode*.

Figure 6.4: *Maximal progress mode. State randomly generated.*

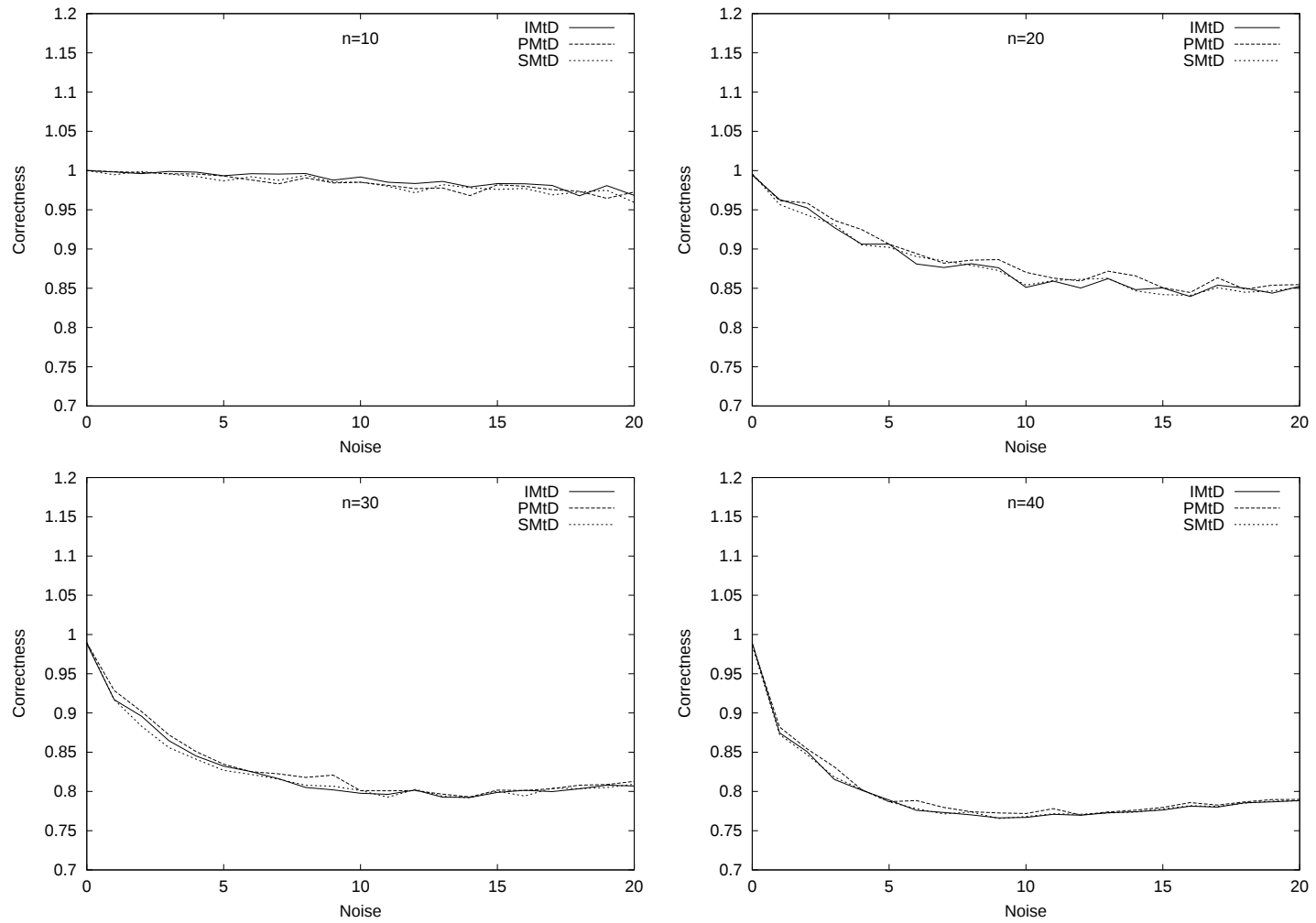


Figure 6.5: *Maximal progress mode. State deterministically generated, single-threading.*

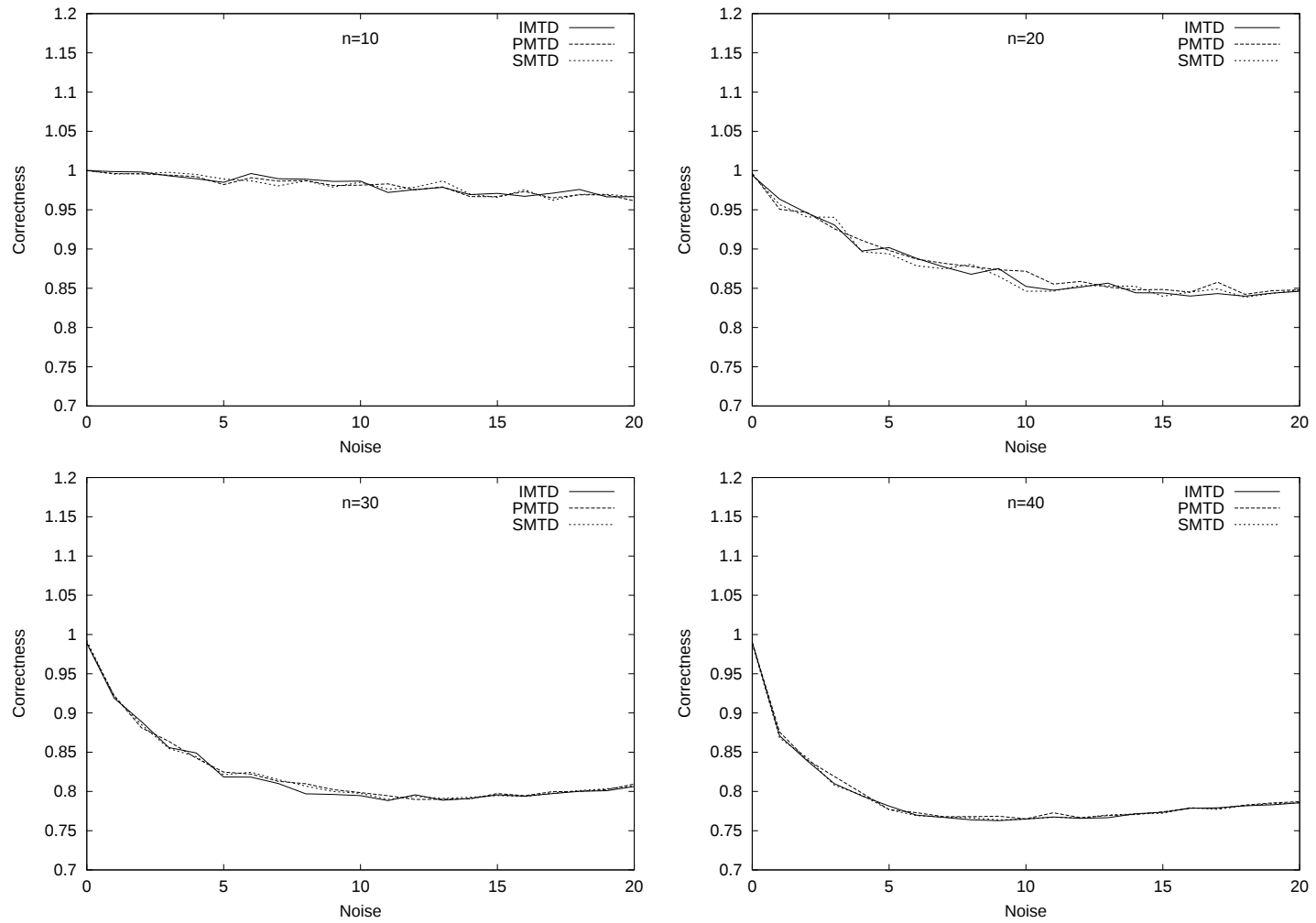


Figure 6.6: *Maximal progress mode. State deterministically generated, multi-threading.*

Table 6.6, Table 6.7 and Table 6.8 show that, for *Maximal progress mode*, the variant *Partition* is the most expensive in terms of number of evaluations of the objective function, and that the variant *Insertion Sort* is the cheapest. These tables also show that the number of evaluations slightly increases as the size of the graph grows.

In effectiveness and efficiency, and according to our results, the variants *Insertion Sort* and *Selection Sort* are better than *Partition* when a *Maximal progress mode* is applied.

Observe that given the number of neighbors explored by *Minimal progress mode* and *Maximal progress mode* the potential of the neighborhood structure is not fully displayed.

Sorting Algorithm	Size of the problem			
	$n = 10$	$n = 20$	$n = 30$	$n = 40$
<i>Insertion Sort</i>	24988	26507	27491	27969
<i>Partition</i>	31179	32143	32459	32648
<i>Selection Sort</i>	26716	28319	29035	29372

Table 6.6: *Number of evaluations of the objective function for Maximal progress mode with Single-threading (where the information about the state is deterministically generated), and for different graph sizes.*

For *First improvement*, the dotted lines in Figure 6.7, Figure 6.8 and Figure 6.9, indicate results for our *MA-sorting* instantiated with *SlcFrsImp*, the dashed lines indicate results for our *MA-sorting* instantiated with *PrtFrsImp*, while the solid lines correspond to *InsFrsImp*.

Sorting	Size of the problem			
Algorithm	$n = 10$	$n = 20$	$n = 30$	$n = 40$
<i>Insertion Sort</i>	25023	26555	27562	28026
<i>Partition</i>	31232	32166	32486	32684
<i>Selection Sort</i>	26755	28363	29069	29412

Table 6.7: *Number of evaluations of the objective function for Maximal progress mode with Single-threading (where the information about the state is randomly generated), and for different graph sizes.*

Sorting	Size of the problem			
Algorithm	$n = 10$	$n = 20$	$n = 30$	$n = 40$
<i>Insertion Sort</i>	24709	26002	26641	26905
<i>Partition</i>	31241	32029	32436	32691
<i>Selection Sort</i>	26976	28719	29411	29747

Table 6.8: *Number of evaluations of the objective function for Maximal progress mode with Multi-threading (where the information about the state is deterministically generated), and for different graph sizes.*

Figure 6.7, Figure 6.8 and Figure 6.9 illustrate that, for *First improvement* (for all tested graph sizes and different noise levels), *Partition* and *Selection Sort* are superior to *Insertion Sort*, across *SngThrRnd*, *SngThrDtr* and *MltThrDtr*.

Figure 6.9 also shows that, for *First improvement* with *Multi-threading* (for sizes $n = 30$ or $n = 40$, and different noise levels), *Selection Sort* is better than *Partition*.

First improvement explores a larger number of neighbors than both *MnmPrg* and *MxmPrg*, and makes it possible to observe that the neighborhood structure proposed by *Partition* and *Selection Sort* is superior to *Insertion Sort*.

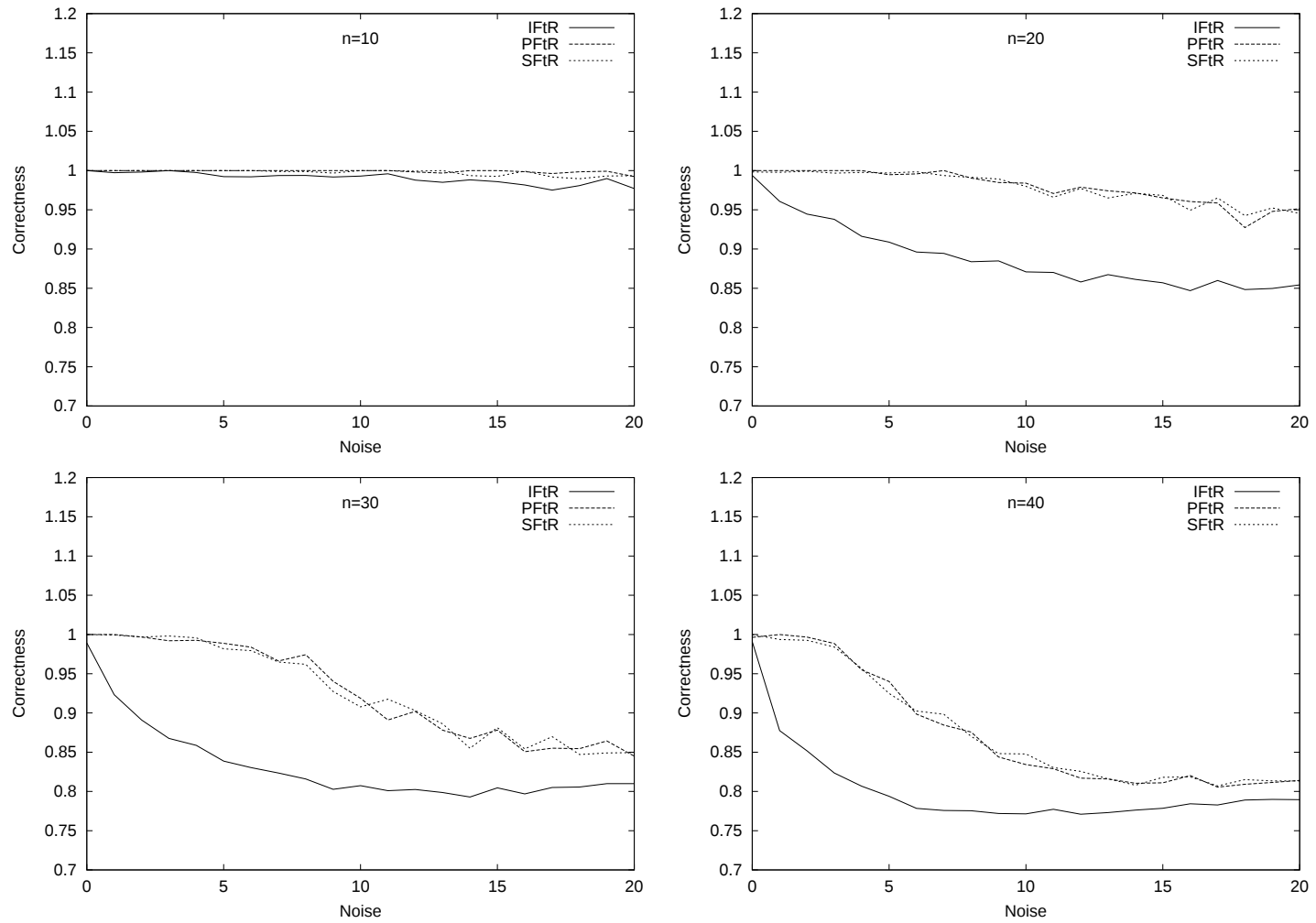
Table 6.9 and Table 6.10 show that, for *First improvement* with *Single-threading* (state deterministically or randomly generated), the variant *Partition* is the most expensive in the interval $n = 10$ to $n = 30$ (for size $n = 40$ *Insertion Sort* is more expensive than *Partition*). These tables also show that the variant *Selection Sort* is the cheapest (for all tested graph sizes and different noise levels). These tables illustrate that the number of evaluations increases linearly as the size of the graph grows.

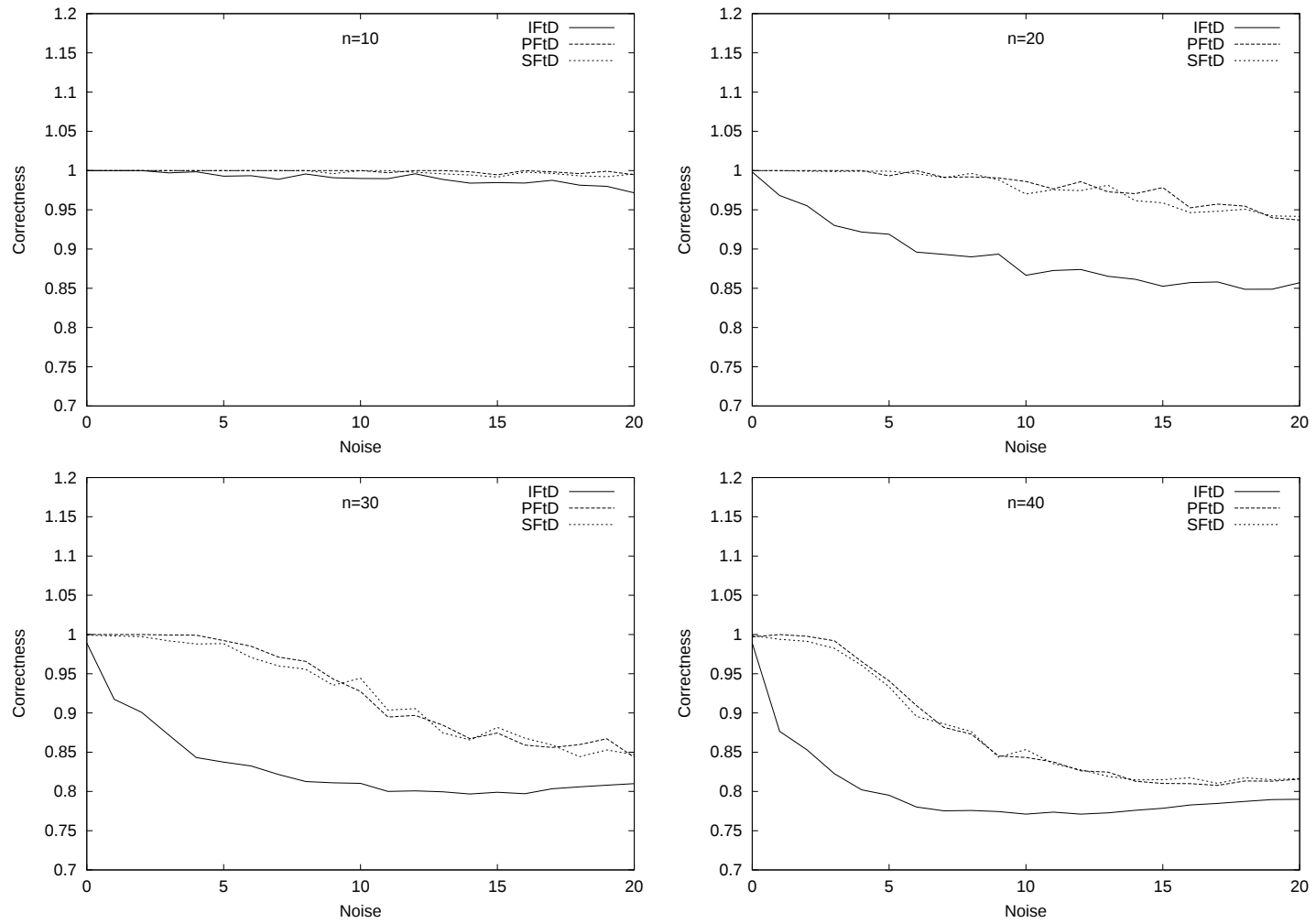
Sorting	Size of the problem			
Algorithm	$n = 10$	$n = 20$	$n = 30$	$n = 40$
<i>Insertion Sort</i>	33446	45448	53685	62771
<i>Partition</i>	45908	53172	58219	61589
<i>Selection Sort</i>	31041	41425	45812	50299

Table 6.9: *Number of evaluations of the objective function for First improvement with Single-threading (where the information about the state is deterministically generated), and for different graph sizes.*

Sorting	Size of the problem			
Algorithm	$n = 10$	$n = 20$	$n = 30$	$n = 40$
<i>Insertion Sort</i>	33462	45503	53757	62943
<i>Partition</i>	46016	53053	58307	61950
<i>Selection Sort</i>	31109	41535	45933	50417

Table 6.10: Number of evaluations of the objective function for First improvement with Single-threading (where the information about the state is randomly generated), and for different graph sizes.

Figure 6.7: *First improvement. State randomly generated.*

Figure 6.8: *First improvement. State deterministically generated, single-threaded.*

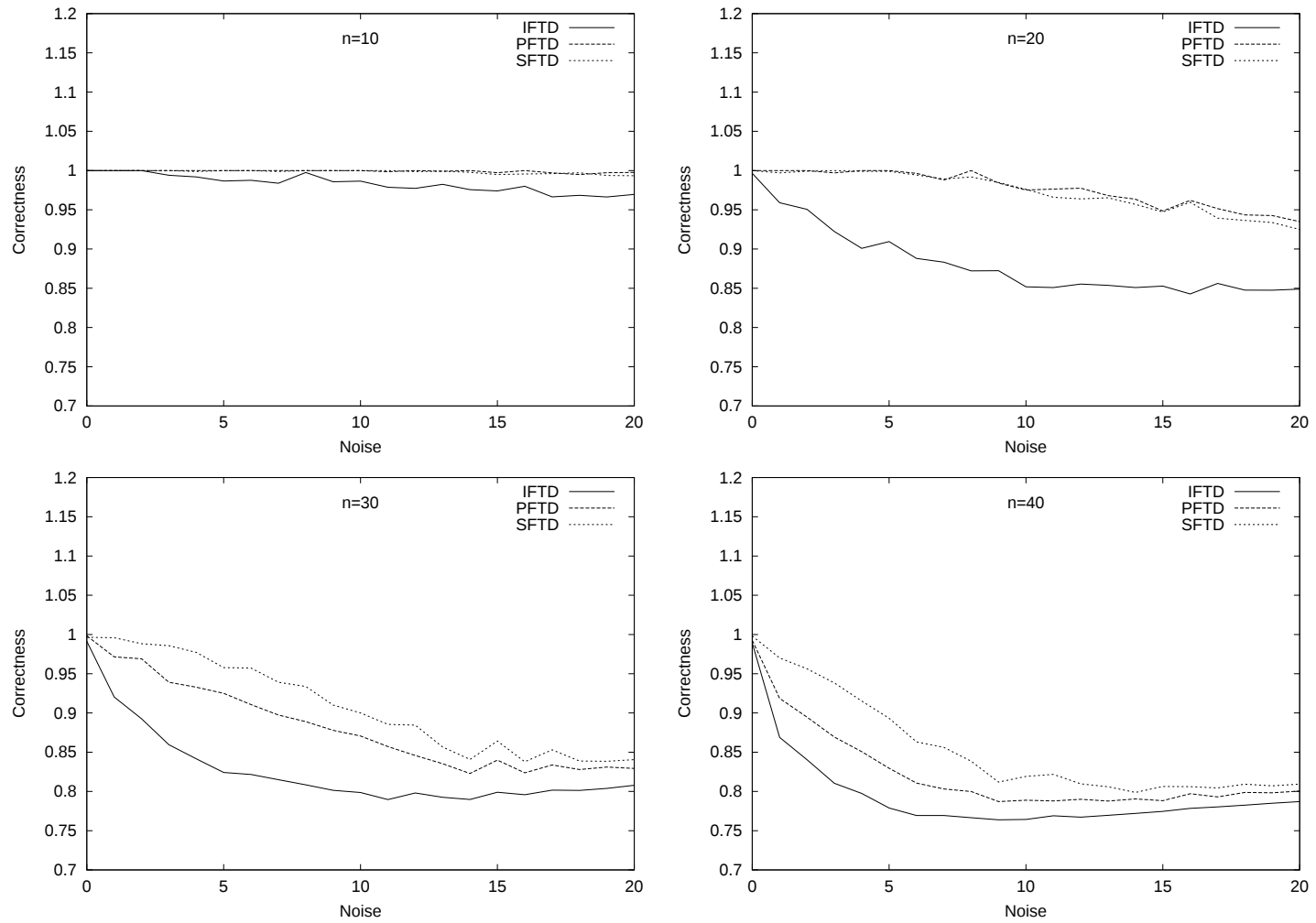


Figure 6.9: *First improvement. State deterministically generated, multi-threaded.*

Table 6.11 shows that, for *First improvement* with *Multi-threading*, the variant *Partition* is the most expensive in terms of number of evaluations, for problems with size $n = 10$ or $n = 20$, (the variant *Selection Sort* is the most expensive for problems with size $n = 30$ or $n = 40$). That means that the variant *Partition* scales better than *Selection Sort*. The variant *Insertion Sort* is the cheapest for problems with size $n = 20$, $n = 30$ or $n = 40$. These tables also show that the number of evaluations slightly increases as the size of the graph grows, faster with *Selection Sort*, slower with *Insertion Sort*.

In effectiveness and efficiency, and according to our results, for *First improvement*, the variant *Selection Sort* is better than *Partition* and this is better than *Insertion Sort*.

Sorting Algorithm	Size of the problem			
	$n = 10$	$n = 20$	$n = 30$	$n = 40$
<i>Insertion Sort</i>	32552	41989	42033	41993
<i>Partition</i>	45514	52575	53009	52892
<i>Selection Sort</i>	31755	43425	53234	61784

Table 6.11: *Number of evaluations of the objective function for First improvement with Multi-threading (where the information about the state is deterministically generated), and for different graph sizes.*

For *Best improvement*, the dotted lines in Figure 6.10, Figure 6.11 and Figure 6.12, indicate results for our *MA-sorting* instantiated with *SlcBstImp*, the dashed lines indicate results for our *MA-sorting* instantiated with *PrtBstImp*, while the solid lines correspond to *InsBstImp*.

Figure 6.10, Figure 6.11 and Figure 6.12, show that, for *Best improvement* (for all tested graph sizes and different noise levels), *Partition* and *Selection Sort* are superior to *Insertion Sort*, across *SngThrRnd*, *SngThrDtr* and *MltThrDtr*.

The figures also show that, for *Best improvement* (for size $n = 40$ and noise levels $\varepsilon = 7$ to $\varepsilon = 16$), *Selection Sort* is slightly better than *Partition*, across *SngThrRnd*, *SngThrDtr* and *MltThrDtr*.

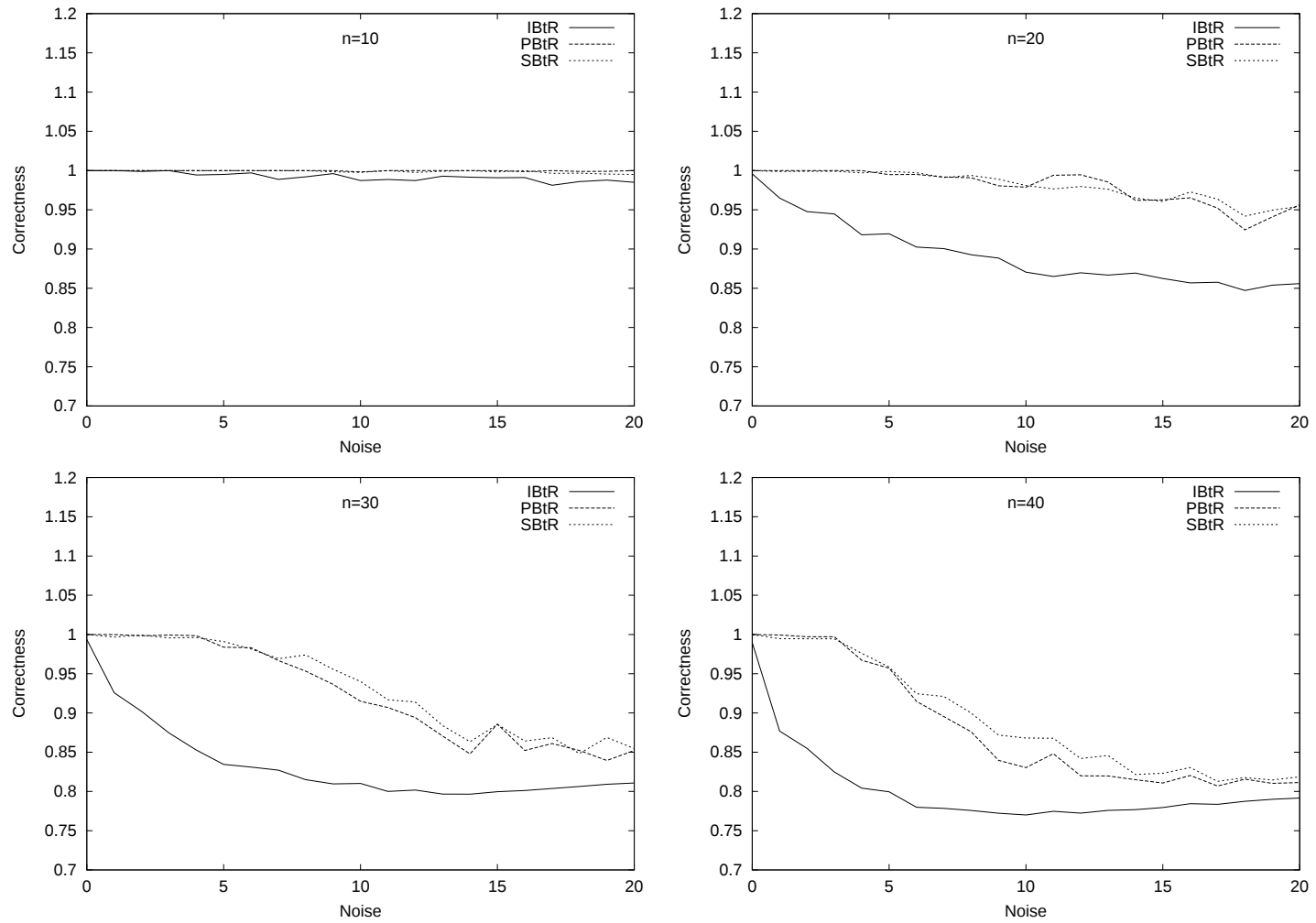
Table 6.12 and Table 6.13 show that, for *Single-threading* (with state deterministically or randomly generated), the variant *Partition* is the most expensive, and that the variant *Selection Sort* is the cheapest. These tables also show that *Selection Sort* and *Insertion Sort* scale better than *Partition*.

Sorting Algorithm	Size of the problem			
	$n = 10$	$n = 20$	$n = 30$	$n = 40$
<i>Insertion Sort</i>	34391	48911	62753	76358
<i>Partition</i>	59142	90789	121460	151590
<i>Selection Sort</i>	33939	48101	61600	75196

Table 6.12: *Number of evaluations of the objective function for Best improvement with Single-threading (where the information about the state is deterministically generated), and for different graph sizes.*

Sorting	Size of the problem			
Algorithm	$n = 10$	$n = 20$	$n = 30$	$n = 40$
<i>Insertion Sort</i>	34464	49065	63012	76799
<i>Partition</i>	56274	88087	118832	148917
<i>Selection Sort</i>	34017	48204	61728	75192

Table 6.13: Number of evaluations of the objective function for Best improvement with Single-threading (where the information about the state is randomly generated), and for different graph sizes.

Figure 6.10: *Best improvement. State randomly generated.*

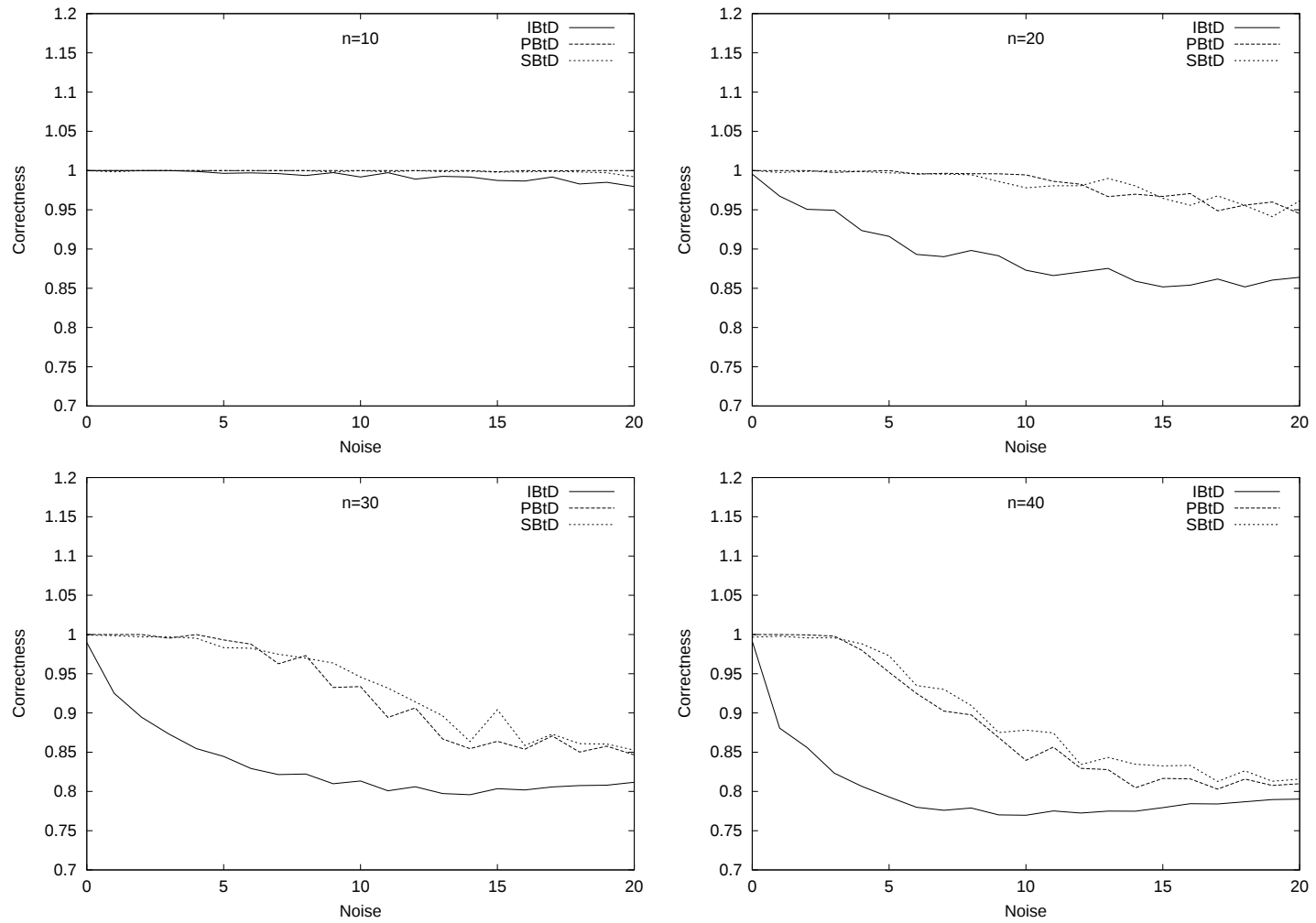


Figure 6.1: *Best improvement. State deterministically generated, single-threading.*

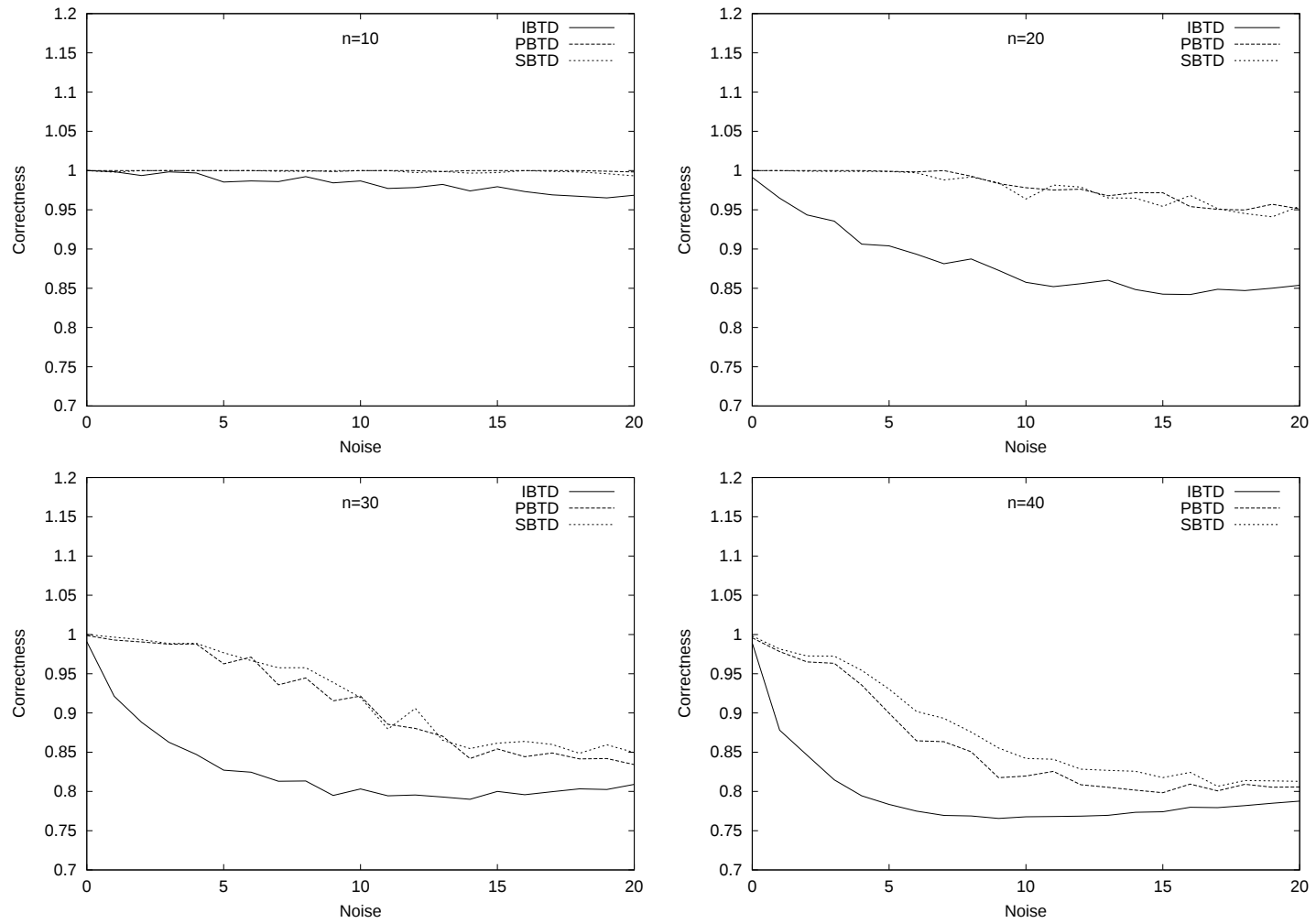
Figure 6.12: *Best improvement. State deterministically generated, multi-threading.*

Table 6.14 shows that, for *Multi-threading*, the variant *Partition* is the most expensive in terms of number of evaluations. The variant *Insertion Sort* is the cheapest. These table also shows that *Selection Sort* scales better than *Partition*.

In effectiveness and efficiency, and according to our results, for *Best improvement*, *Selection Sort* is better than *Partition* and this is better than *Insertion Sort*.

Sorting	Size of the problem			
Algorithm	$n = 10$	$n = 20$	$n = 30$	$n = 40$
<i>Insertion Sort</i>	32834	42761	43819	43993
<i>Partition</i>	59331	90893	121204	151241
<i>Selection Sort</i>	35323	53965	80265	107270

Table 6.14: *Number of evaluations of the objective function for Best improvement with Multi-threading (where the information about the state is deterministically generated), and for different graph sizes.*

Table 6.15 summarizes the impact of the sorting algorithm into *MA-sorting*. According to those results, we observe that, for this particular problem, the variant *Selection Sort* is superior to the variants *Insertion Sort* and *Partition*. We also observe that the variant *Partition* is better than the variant *Insertion Sort*.

In Section 5.4.5 we have shown that the measure of presortedness *Eng* does not increase its value when a swap is performed (after a comparison predicate returns *False*), and that *Selection Sort*, *Quicksort* and *Insertion Sort* are well behaved as local searchers for this measure of presortedness.

Amount of Progress	Method for generating the state		
	<i>SngThr Rnd</i>	<i>SngThr Dtr</i>	<i>MltThr Dtr</i>
<i>MnmPrg</i>	<i>Ins</i> or <i>Slc</i>	<i>Ins</i> or <i>Slc</i>	<i>Ins</i> or <i>Slc</i>
<i>MxmPrg</i>	<i>Ins</i> or <i>Slc</i>	<i>Ins</i> or <i>Slc</i>	<i>Ins</i> or <i>Slc</i>
<i>FrsImp</i>	<i>Slc</i>	<i>Slc</i>	<i>Slc</i>
<i>BstImp</i>	<i>Slc</i>	<i>Slc</i>	<i>Slc</i>

Table 6.15: *Impact of the sorting algorithm into MA-sorting.*

We observe that in our case study this measure of presortedness is also relevant, because each swap performed by *MA-sorting* improves the value of the objective function **ATE**.

Because we assume regularity of the landscape with respect to a measure of presortedness, and *Eng* is a valid measure of presortedness for *Insertion Sort*, *Quicksort* and *Selection Sort*, we assume the *Eng*-value reached by *MA-sorting* (instantiated with *Insertion Sort*, *Partition* and *Selection Sort*) provides relevant information about the quality (fruitfulness) of the neighborhood structure generated by those sorting algorithms.

Figure 6.13 displays the value of *Eng* (normalized by the graph size) and includes plots that correspond, from left to right and from top to bottom, to the four values for the size n of the graph. In the plots, the x -axis shows the range of ε values. The y -axis is an *Eng* scale, from 0 to 1. Lower values of *Eng* means better results. The plotted lines are average *Eng* taken over the 50 problems.

We compare the values of *Eng* for those variants of *MA-sorting* that have provided the best quality of results.

In Figure 6.13 the dotted lines indicate results for our *MA-sorting* instantiated with *SlcBstImpSngThrDtr*, the dashed lines indicate results for our *MA-sorting* instantiated with *PrtBstImpSngThrDtr*, while the solid lines indicate *InsBstImpSngThrDtr*.

Figure 6.13 shows that the *Eng*-values for *MA-sorting* instantiated with both *Selection Sort* and *Partition* are consistently lower than those provided by *MA-sorting* with *Insertion Sort*. We conclude that the neighborhood structures implicitly generated by *Selection Sort* and *Partition* are the most productive.

6.5.2 Amount of Progress

As we explain in Section 5.5.2 , we regulate the impact of the sorting algorithm into our *MA-sorting* by the amount of progress in the sorting process at each invocation of mutation.

With respect to the amount of progress we performed experiments with four variants:

1. *Minimal progress mode* (*MnmPrg*).
2. *Maximal progress mode* (*MxmPrg*).
3. *First improvement* (*FrsImp*).
4. *Best improvement* (*BstImp*).

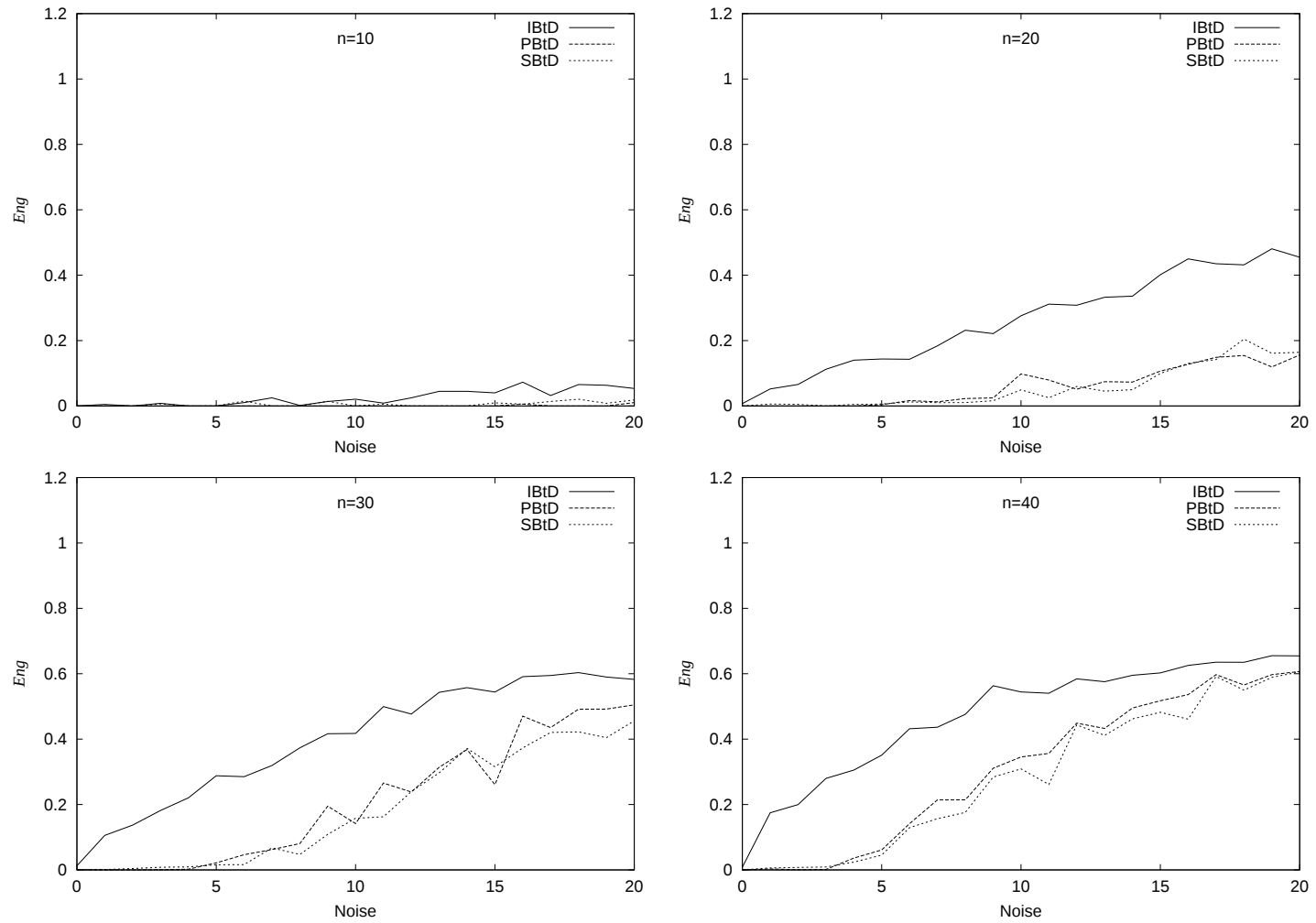


Figure 6.13: The *Eng*-values for MA-sorting instantiated with Insertion Sort, Partition and Selection Sort. Lower values of *Eng* means better results.

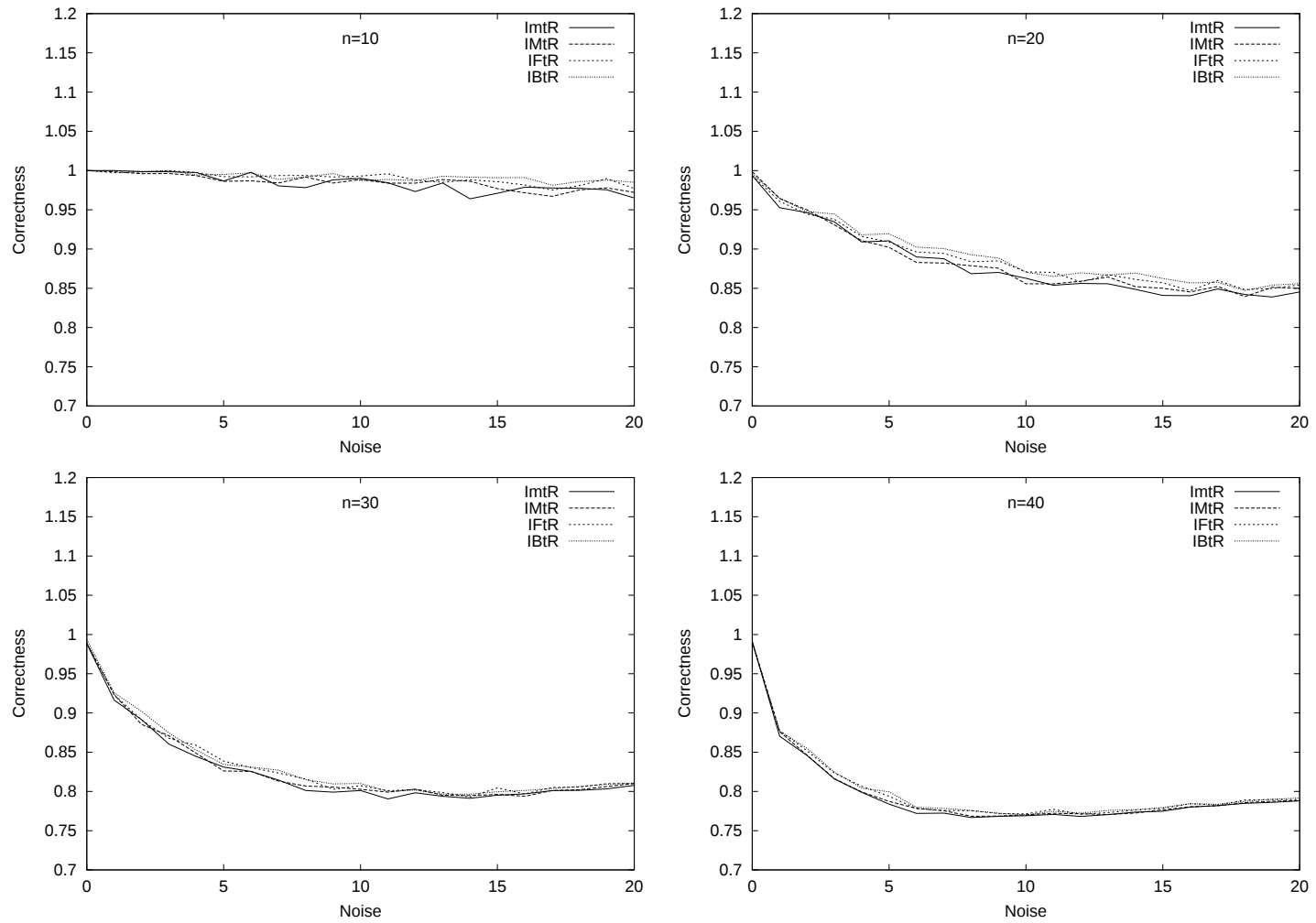
We compare these four variants for *Insertion Sort*, *Partition* and *Selection Sort* in the context of three modes of generating the state (*SngThrRnd*, *SngThrDtr* and *MltThrDtr*) that determine the continuation of the schedule of comparisons (those modes were explained in Section 5.5.1).

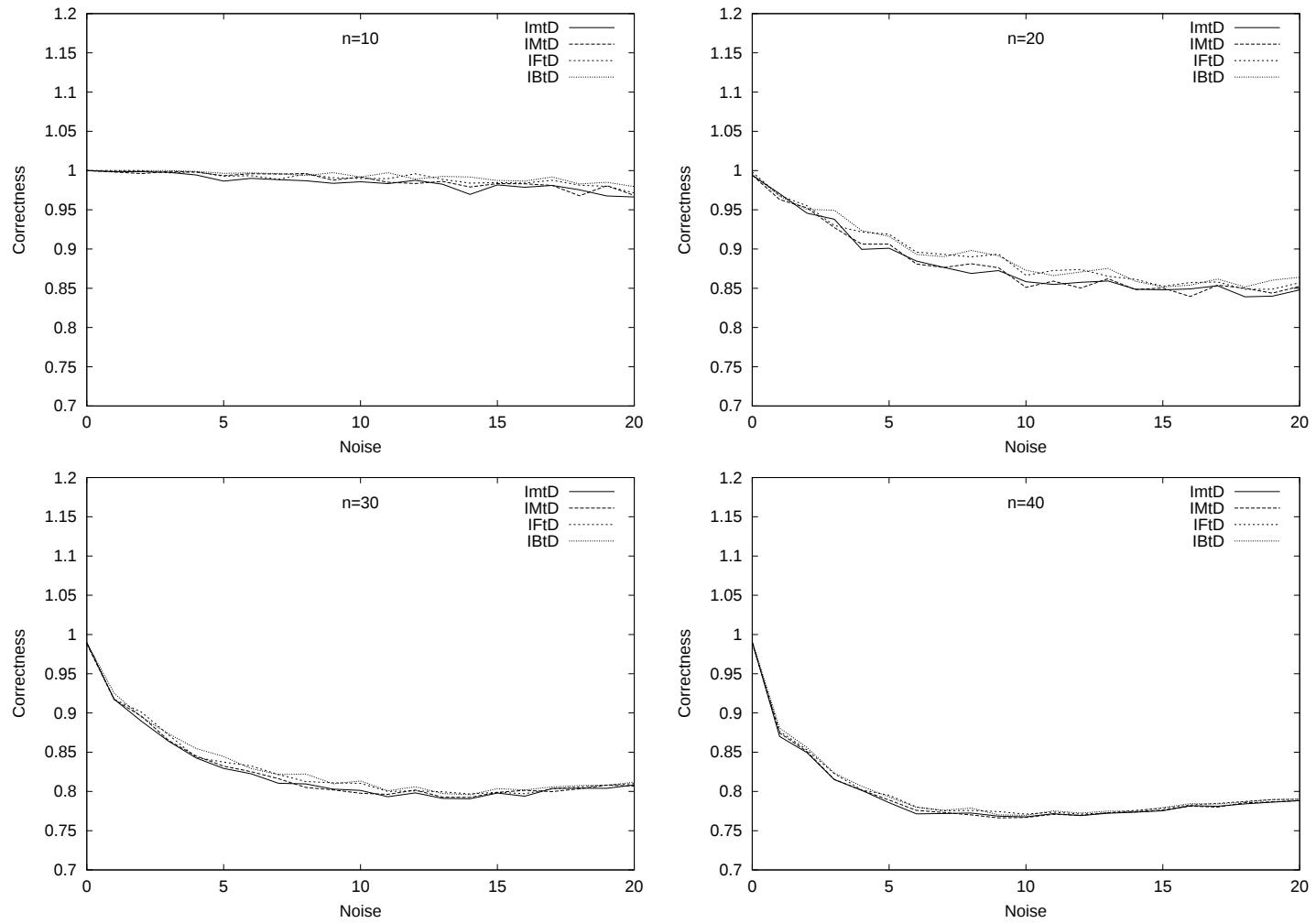
Our purpose is to evaluate the impact of the amount of progress into our *MA-sorting*.

For *Insertion Sort*, the dotted lines in Figure 6.14, Figure 6.15 and Figure 6.16, indicate results for our *MA-sorting* instantiated with *InsBstImp*, the dashed lines indicate results for our *MA-sorting* instantiated with *InsFrsImp*, the broken lines indicate results for our *MA-sorting* instantiated with *InsMxmPrg*, while the solid lines indicate *InsMnmPrg*.

Figure 6.14 and Figure 6.15 illustrate that, when using *Insertion Sort* with *Single-threading* (states randomly and deterministically generated), the variants *FrsImp* and *BstImp* give better results than the variants *MnmPrg* and *MxmPrg*, at least to problems with size $n = 20$.

Figure 6.16 illustrates that, when using *Insertion Sort* with *Multi-threading* (for all tested graph sizes and different noise levels), differences in effectiveness are not important across *MnmPrg*, *MxmPrg*, *FrsImp* and *BstImp*.

Figure 6.14: *Insertion Sort. State randomly generated.*

Figure 6.15: *Insertion Sort. State deterministically generated, single-threaded.*

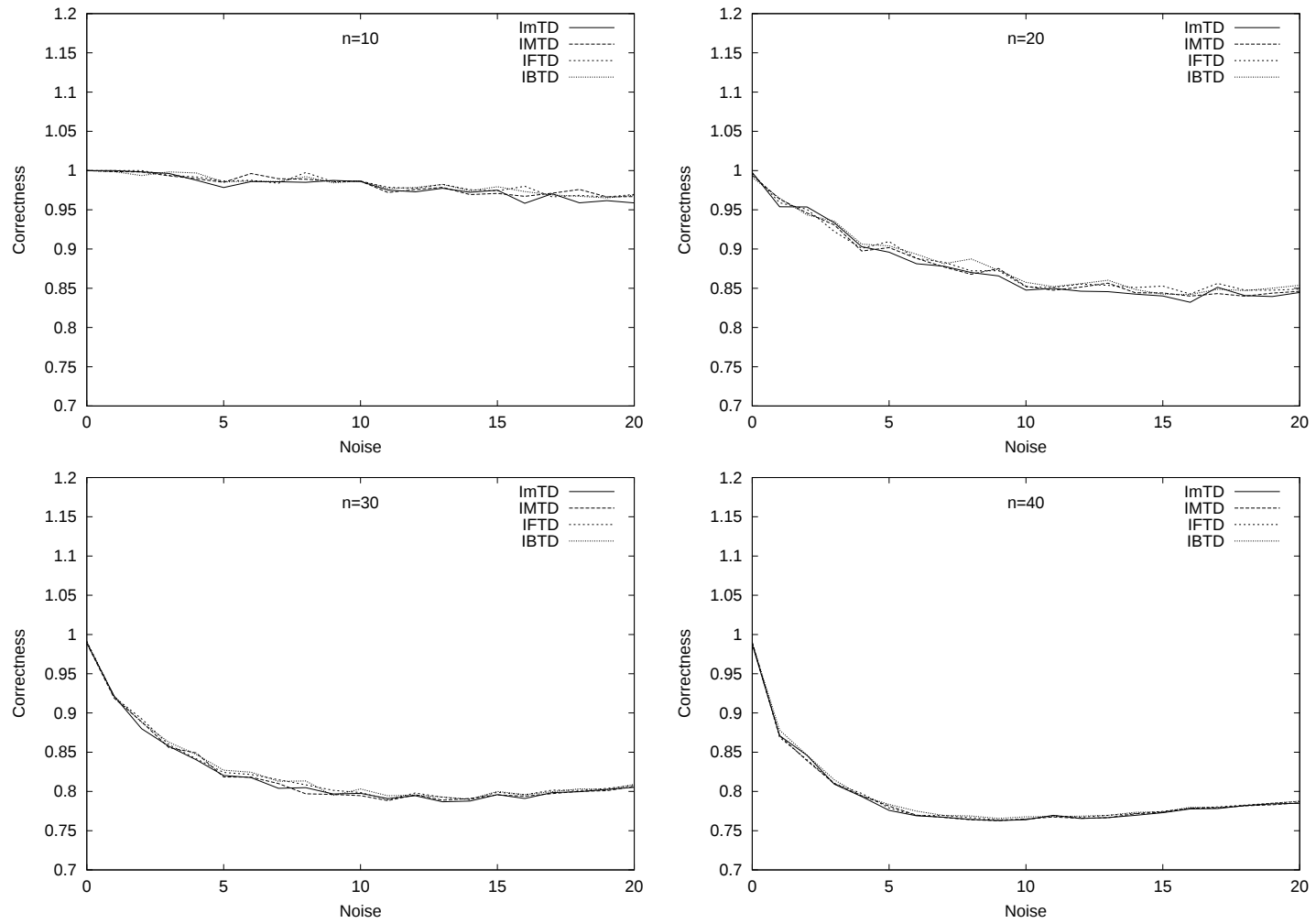
Figure 6.16: *Insertion Sort. State deterministically generated, multi-threading.*

Table 6.16, Table 6.17 and Table 6.18 show that, using *Insertion Sort*, the variant *MnmPrg* is the least expensive in terms of number of evaluations, and that the variant *BstImp* is the most expensive. These tables also show that the number of evaluations increases linearly as the size of the graph grows, although this growth is more noticeable in the case of *BstImp*.

Thus, with *Insertion Sort*, using *Single-threading* mode (random or deterministic), the variant *FrsImp* (for the amount of progress) is the best in terms of effectiveness and efficiency. In the case of *MltThrDtr*, the quality of results does not have a large impact across *MnmPrg*, *MxmPrg*, *FrsImp* and *BstImp*.

Amount of Progress	Size of the problem			
	$n = 10$	$n = 20$	$n = 30$	$n = 40$
<i>MnmPrg</i>	24686	25865	26293	26502
<i>MxmPrg</i>	24988	26507	27491	27969
<i>FrsImp</i>	33446	45448	53685	62771
<i>BstImp</i>	34391	48911	62753	76358

Table 6.16: *Number of evaluations of the objective function for Insertion Sort with Single-threading where the information about the state is deterministically generated, and for different graph sizes.*

For *Partition*, the dotted lines in Figure 6.17, Figure 6.18 and Figure 6.19, indicate results for our *MA-sorting* instantiated with *PrtBstImp*, the dashed lines indicate results for our *MA-sorting* instantiated with *PrtFrsImp*, the broken lines indicate results for our *MA-sorting* instantiated with *PrtMxmPrg*, while the solid lines indicate *PrtMnmPrg*.

Amount of Progress	Size of the problem			
	$n = 10$	$n = 20$	$n = 30$	$n = 40$
<i>MnmPrg</i>	24713	25900	26319	26535
<i>MxmPrg</i>	25023	26555	27562	28026
<i>FrsImp</i>	33462	45503	53757	62943
<i>BstImp</i>	34464	49065	63012	76799

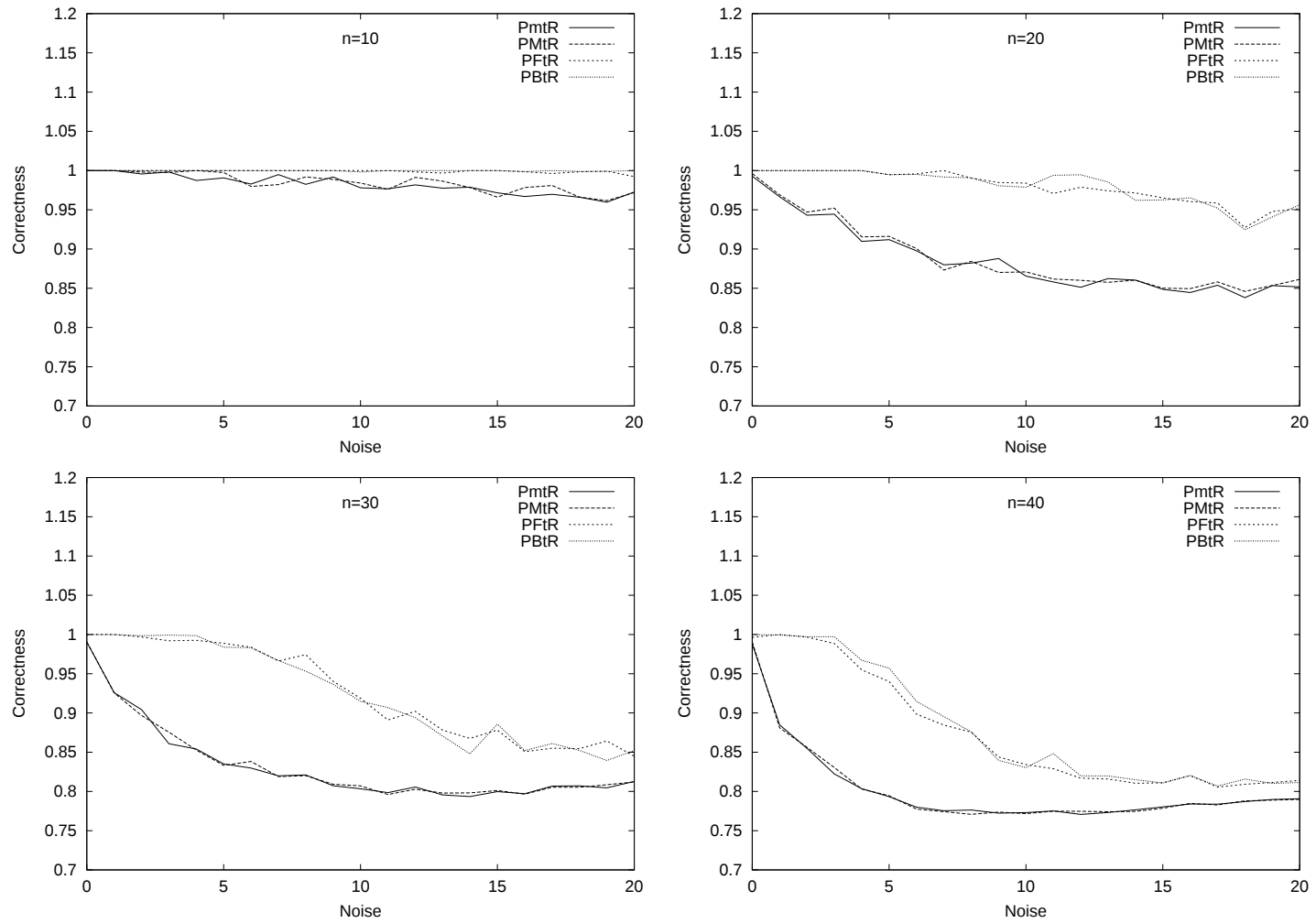
Table 6.17: Number of evaluations of the objective function for Insertion Sort with Single-threading where the information about the state is randomly generated, and for different graph sizes.

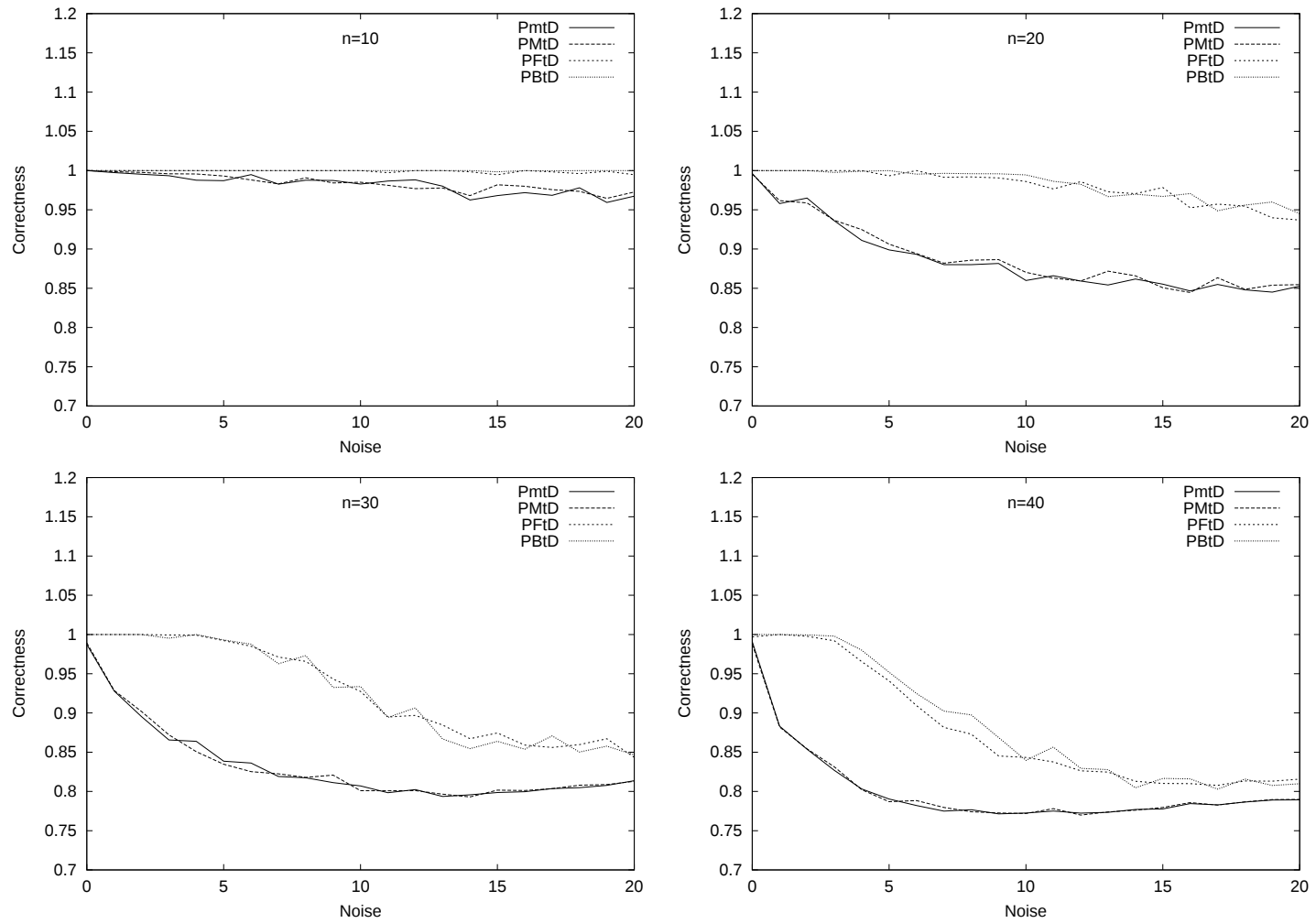
Figure 6.17, Figure 6.18 and Figure 6.19 illustrate that, for *Partition* (for all tested graph sizes and different noise levels) the variants *FrsImp* and *BstImp*, are better than the variants *MxmPrg* and *MnmPrg*, across *SngThrRnd*, *SngThrRnd* and *Mlt-ThrDtr*.

Figure 6.19 also shows that, using *Partition* with *Multi-threading* (for sizes $n = 30$ or $n = 40$, and for all tested noise levels), the variant *BstImp* is better than *FrsImp*.

Amount of Progress	Size of the problem			
	$n = 10$	$n = 20$	$n = 30$	$n = 40$
<i>MnmPrg</i>	24597	25808	26199	26392
<i>MxmPrg</i>	24709	26002	26641	26905
<i>FrsImp</i>	32552	41989	42033	41993
<i>BstImp</i>	32834	42761	43819	43993

Table 6.18: *Number of evaluations of the objective function for Insertion Sort with Multi-threading where the information about the state is deterministically generated, and for different graph sizes.*

Figure 6.17: *Partition. State randomly generated.*

Figure 6.18: *Partition. State deterministically generated, single-threading.*

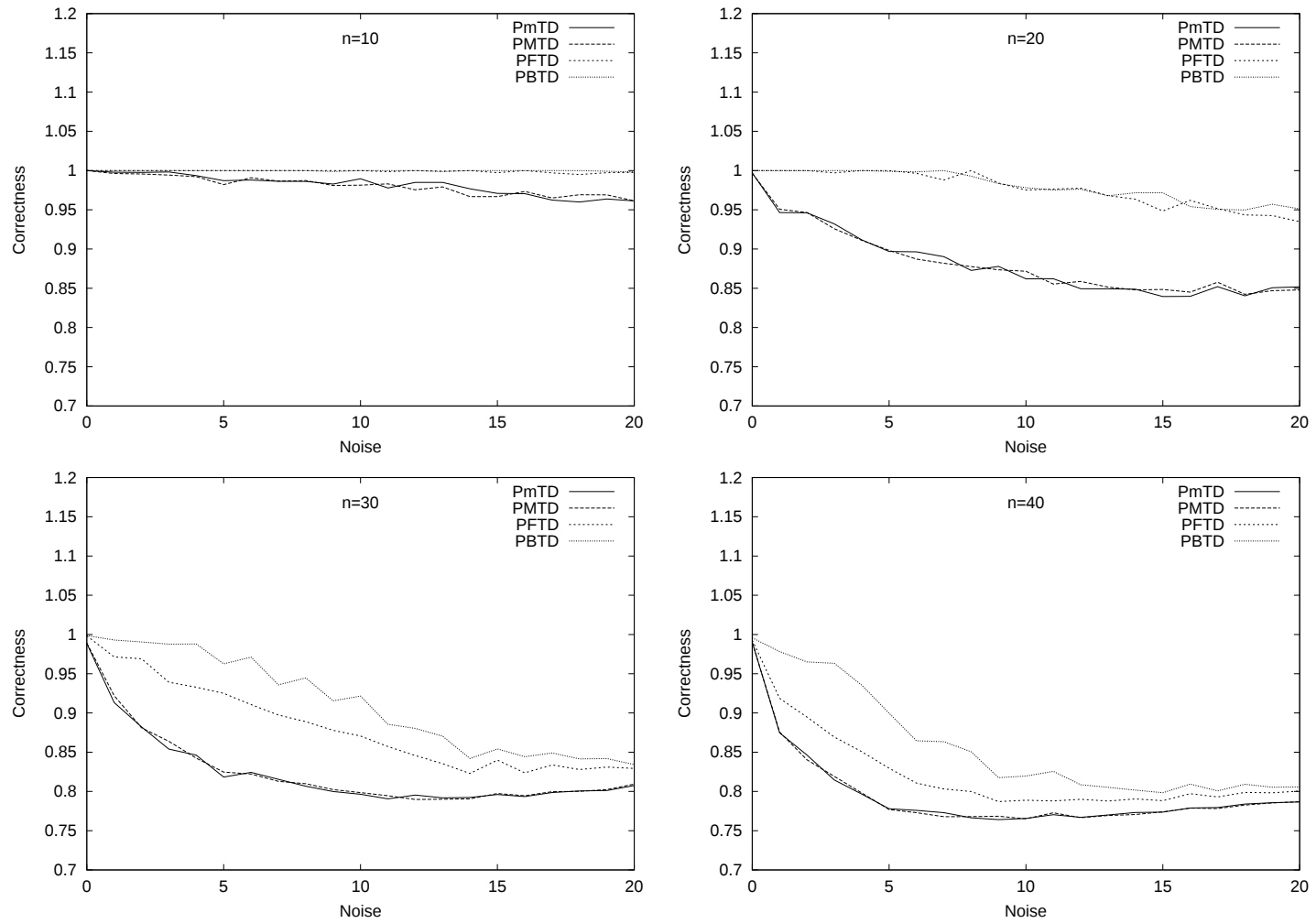
Figure 6.19: *Partition. State deterministically generated, multi-threading.*

Table 6.19, Table 6.20 and Table 6.21 show that, using *Partition*, the variant *MnmPrg* is the least expensive in terms of number of evaluations, and that the variant *BstImp* is the most expensive across *SngThrRnd*, *SngThrDtr* and *MltThrDtr*. These tables also show that the number of evaluations increases linearly as the size of the graph grows, although this growth is more noticeable in the case of *BstImp*.

In terms of effectiveness and efficiency, and according to our results, the variant *FrsImp* (for the amount of progress) is the best for *Partition* with *Single-threading* (random or deterministic). The variant *BstImp* is the best for *Partition* with *Multi-threading*.

Amount of Progress	Size of the problem			
	$n = 10$	$n = 20$	$n = 30$	$n = 40$
<i>MnmPrg</i>	30417	31363	31703	31875
<i>MxmPrg</i>	31179	32143	32459	32648
<i>FrsImp</i>	45908	53172	58219	61589
<i>BstImp</i>	59142	90789	121460	151590

Table 6.19: *Number of evaluations of the objective function for Partition with Single-threading where the information about the state is deterministically generated, and for different graph sizes.*

For *Selection Sort*, the dotted lines in Figure 6.20, Figure 6.21 and Figure 6.22, indicate results for our *MA-sorting* instantiated with *SlcBstImp*, the dashed lines indicate results for our *MA-sorting* instantiated with *SlcFrsImp*, the broken lines indicate results for our *MA-sorting* instantiated with *SlcMxmPrg*, while the solid lines indicate *SlcMnmPrg*.

Amount of Progress	Size of the problem			
	$n = 10$	$n = 20$	$n = 30$	$n = 40$
<i>MnmPrg</i>	30449	31393	31735	31902
<i>MxmPrg</i>	31232	32166	32486	32684
<i>FrsImp</i>	46016	53053	58307	61950
<i>BstImp</i>	56274	88087	118832	148917

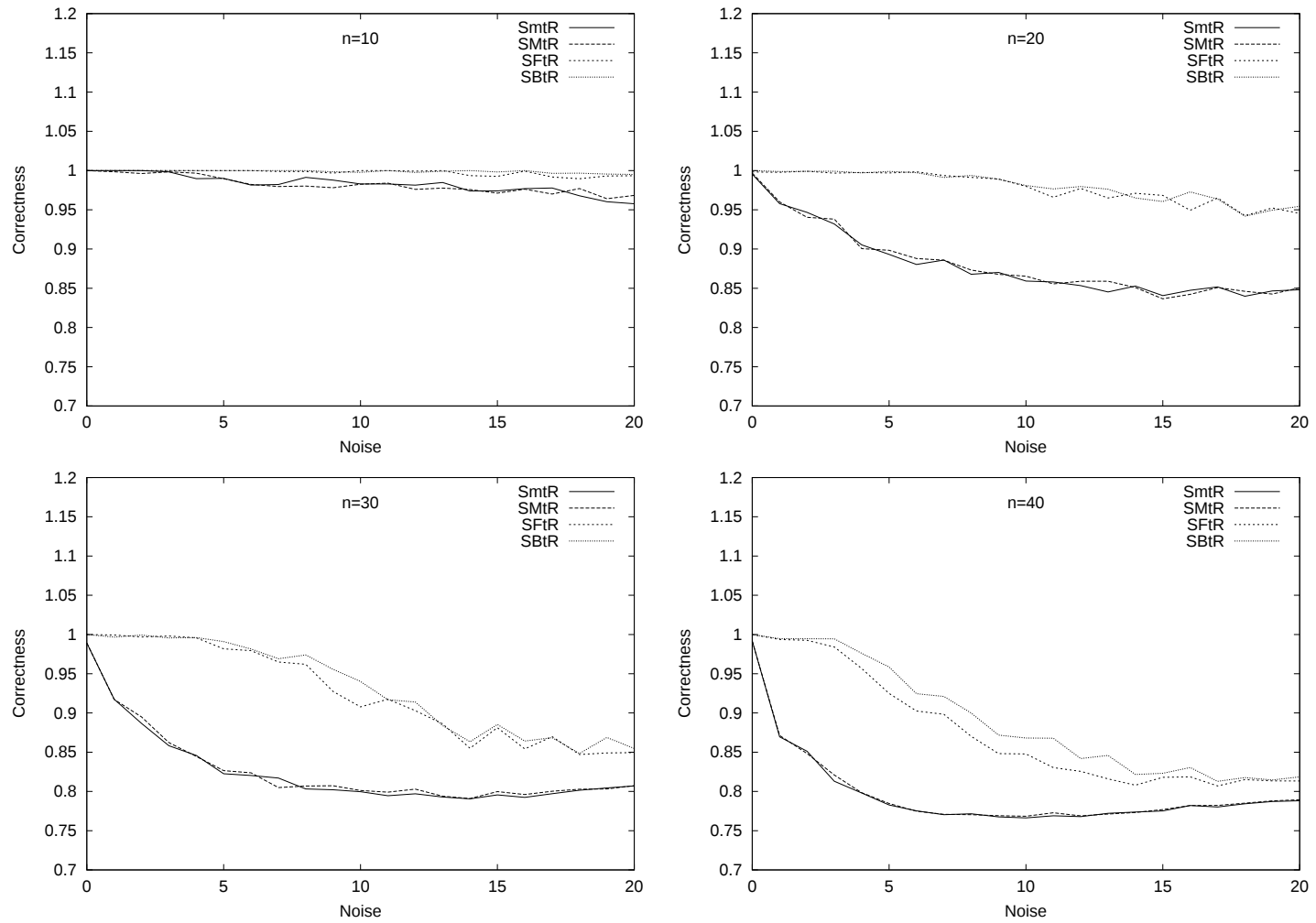
Table 6.20: *Number of evaluations of the objective function for Partition with Single-threading where the information about the state is randomly generated, and for different graph sizes.*

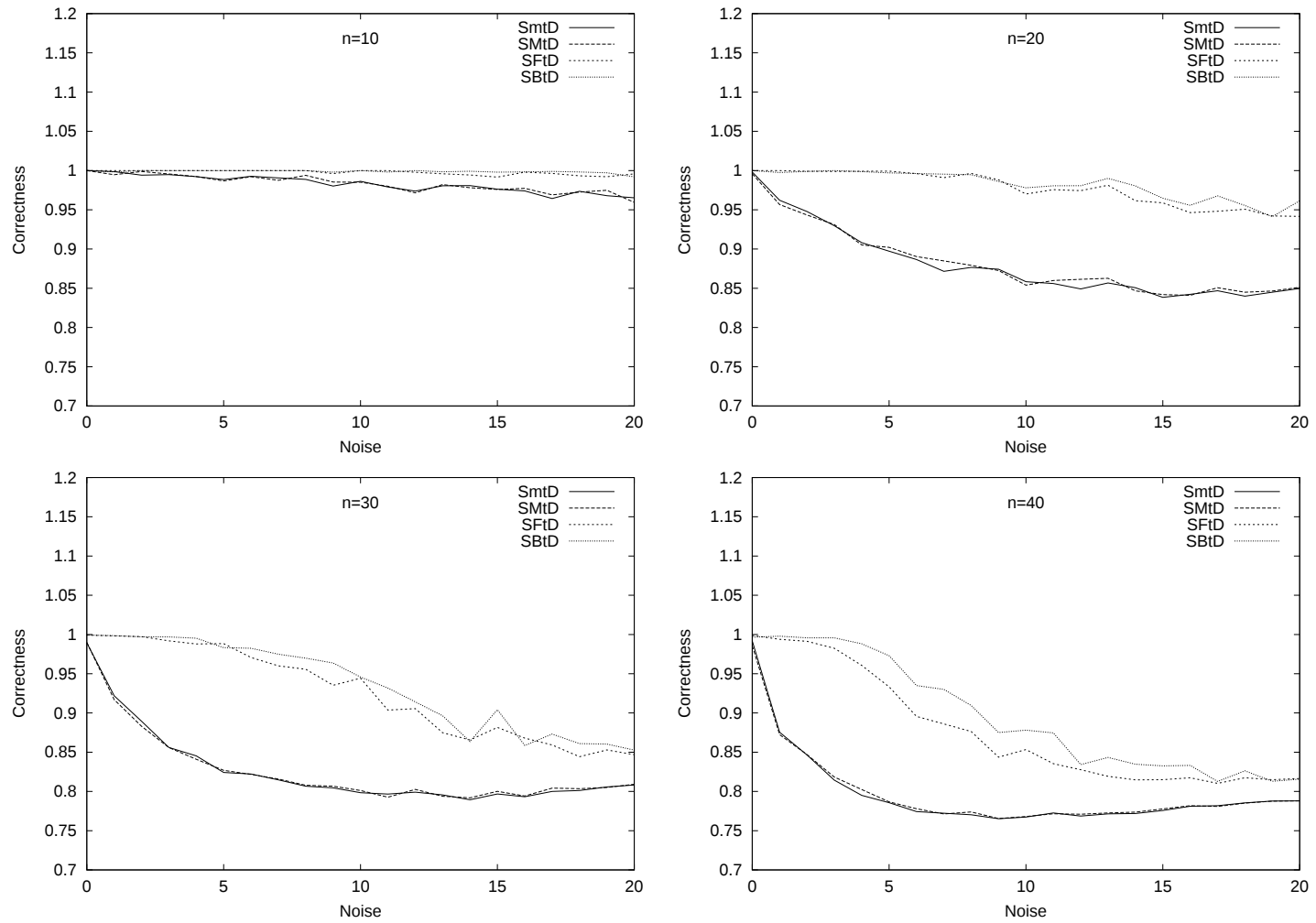
Figure 6.20, Figure 6.21 and Figure 6.22 show that, using *Selection Sort* (for all tested graph sizes and different noise levels), the variants *FrsImp* and *BstImp*, are better than the variants *MxmPrg* and *MnmPrg*, across *SngThrRnd*, *SngThrDtr* and *MltThrDtr*.

Figure 6.20, Figure 6.21 and Figure 6.22 also show that the variant *BstImp* is better than *FrsImp* (for size $n = 40$ and noise levels in the interval $\varepsilon = 3$ to $\varepsilon = 16$), across *SngThrRnd*, *SngThrDtr* and *MltThrDtr*.

Amount of Progress	Size of the problem			
	$n = 10$	$n = 20$	$n = 30$	$n = 40$
<i>MnmPrg</i>	30464	31341	31726	31949
<i>MxmPrg</i>	31241	32029	32436	32691
<i>FrsImp</i>	45514	52575	53009	52892
<i>BstImp</i>	59331	90893	121204	151241

Table 6.21: *Number of evaluations of the objective function for Partition with Multi-threading where the information about the state is deterministically generated, and for different graph sizes.*

Figure 6.20: *Selection Sort. State randomly generated.*

Figure 6.2: *Selection Sort. State deterministically generated, single-threaded.*

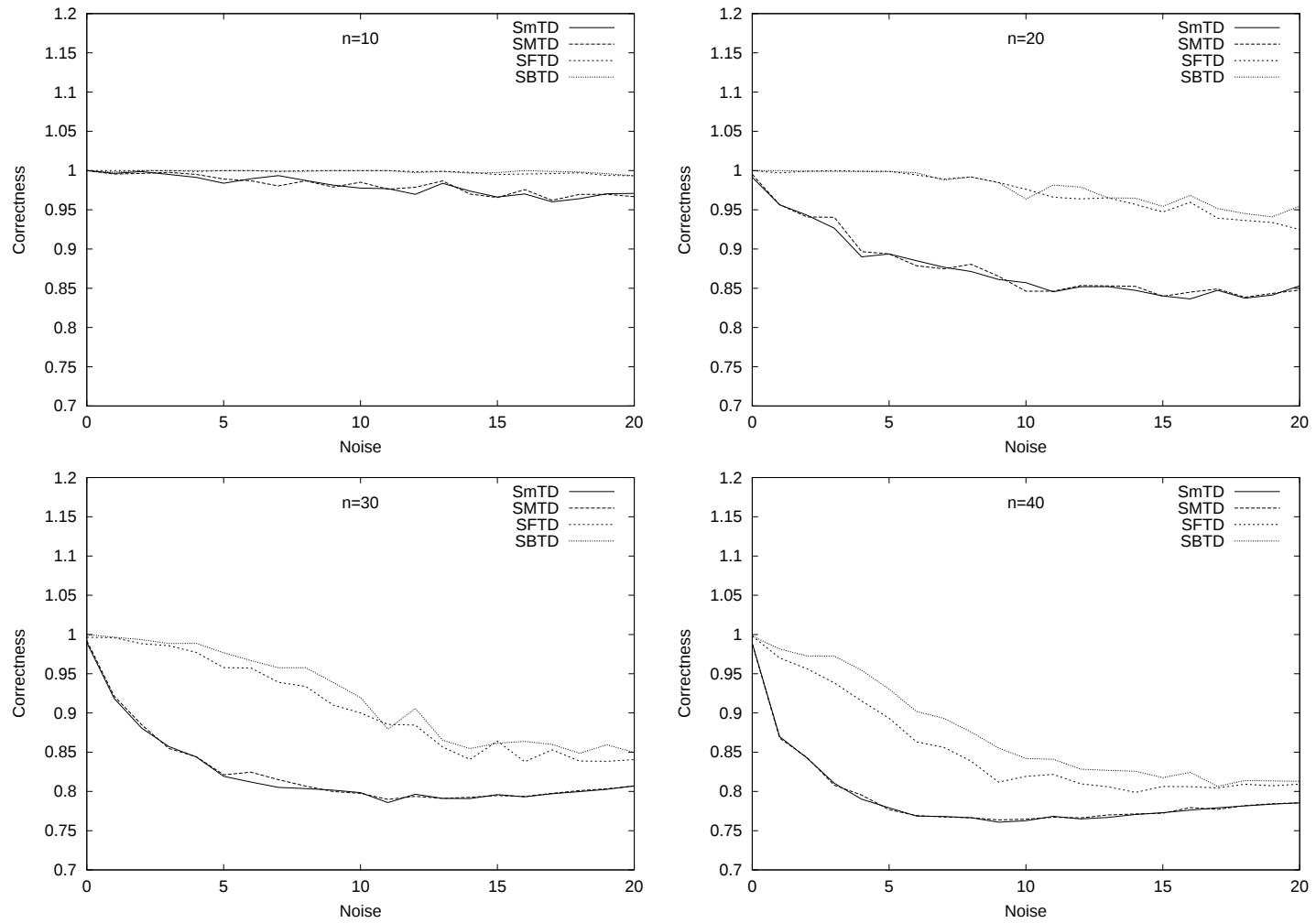
Figure 6.22: *Selection Sort. State deterministically generated, multi-threading.*

Table 6.22, Table 6.23 and Table 6.24 show that, using *Selection Sort*, the variant *MnmPrg* is the least expensive in terms of number of evaluations, and that the variant *BstImp* is the most expensive, across *SngThrRnd*, *SngThrDtr* and *MltThrDtr*. These tables also show that the number of evaluations increases linearly as the size of the graph grows, although this growth is more evident in the case of *BstImp*.

In terms of effectiveness and efficiency, and according to our results, the variant *BstImp* (for the amount of progress) is the best for *Selection Sort* with *Single-threading* (random or deterministic). The variant *FrsImp*, for *Selection Sort* with *Multi-threading*, is the best.

Amount of Progress	Size of the problem			
	$n = 10$	$n = 20$	$n = 30$	$n = 40$
<i>MnmPrg</i>	24682	25866	26290	26510
<i>MxmPrg</i>	26716	28319	29035	29372
<i>FrsImp</i>	31041	41425	45812	50299
<i>BstImp</i>	33939	48101	61600	75196

Table 6.22: *Number of evaluations of the objective function for Selection Sort with Single-threading where the information about the state is deterministically generated, and for different graph sizes.*

Table 6.25 summarizes the impact of the amount of progress into *MA-sorting*. According to these results we confirm our hypothesis that the performance of *MA-sorting* is intensified by increasing the number of neighbors explored, provided that we are dealing with a fruitful neighborhood structure. Moreover, the schedule of comparisons is also critical. As Table 6.25 shows, the impact of the amount of progress into

Amount of Progress	Size of the problem			
	$n = 10$	$n = 20$	$n = 30$	$n = 40$
<i>MnmPrg</i>	24709	25898	26323	26537
<i>MxmPrg</i>	26755	28363	29069	29412
<i>FrsImp</i>	31109	41535	45933	50417
<i>BstImp</i>	34017	48204	61728	75192

Table 6.23: *Number of evaluations of the objective function for Selection Sort with Single-threading where the information about the state is randomly generated, and for different graph sizes.*

MA-sorting is closely related to the method for generating the state of the sorting algorithm.

6.5.3 Order of Nominating Permutations

As explained in Section 5.5.1, the *Nomination* step may maintain a record of previous nominations. Also, we mentioned that a sorting algorithm maintains its state. Recall that this state determines the continuation of the schedule of comparisons.

According to the way of generating the state of the schedule by the sorting algorithm we experiment with three variants:

1. *Single-threading* and state deterministically generated (*SngThrDtr*),
2. *Single-threading* and state randomly generated (*SngThrRnd*) and
3. *Multi-threading* and state deterministically generated (*MltThrDtr*).

Amount of Progress	Size of the problem			
	$n = 10$	$n = 20$	$n = 30$	$n = 40$
<i>MnmPrg</i>	24764	25988	26386	26575
<i>MxmPrg</i>	26976	28719	29411	29747
<i>FrsImp</i>	31755	43425	53234	61784
<i>BstImp</i>	35323	53965	80265	107270

Table 6.24: Number of evaluations of the objective function for Selection Sort with Multi-threading where the information about the state is deterministically generated, and for different graph sizes.

Thus, we deterministically generate the state for both the *Multi-threading* and the *Single-threading* approaches, and we randomly generate the state only for the *Single-threading* approach. The reason is that in the random case the *Multi-threading* and the *Single-threading* approaches are equivalent.

We compare these three variants for *Insertion Sort*, *Partition* and *Selection Sort*, in the context of four modes of operation that regulate the amount of progress (which were explained in Section 5.5.2).

Sorting Algorithm	Method for generating the state		
	<i>SngThrRnd</i>	<i>SngThrDtr</i>	<i>MltThrDtr</i>
<i>Insertion Sort</i>	<i>FrsImp</i>	<i>FrsImp</i>	not relevant
<i>Partition</i>	<i>FrsImp</i>	<i>FrsImp</i>	<i>BstImp</i>
<i>Selection Sort</i>	<i>BstImp</i>	<i>BstImp</i>	<i>FrsImp</i>

Table 6.25: Impact of the amount of progress into MA-sorting.

Our purpose is to evaluate the impact of the order of nominating permutations on our *MA-sorting*.

For *Minimal progress mode*, the dotted lines in Figure 6.23, Figure 6.24 and Figure 6.25, indicate results for our *MA-sorting* instantiated with *MnmPrgMltThrDtr*, the dashed lines indicate results for our *MA-sorting* instantiated with *MnmPrgSngThrRnd*, while the solid lines indicate *MnmPrgSngThrDtr*.

These figures illustrate that, for the *Minimal progress mode* (for all tested graph sizes and different noise levels), differences in effectiveness are not important across *Insertion Sort*, *Partition* and *Selection Sort*. Consequently, the quality of results is not largely influenced by changing the method for generating the state of the sorting algorithm in the case of *Minimal progress mode*.

Table 6.26, Table 6.27 and Table 6.28 show that for *Minimal progress mode*, there are marginal differences in the number of evaluations as the size of the graph increases. This is valid for *Insertion Sort*, *Partition* and *Selection Sort*. Also, we observe that using different methods for generating the state of the sorting algorithm does not substantially change the number of evaluations.

In terms of effectiveness and efficiency, for *Minimal progress mode*, we do not detect differences when a different method for generating the state of the sorting algorithm is applied.

State	Size of the problem			
	$n = 10$	$n = 20$	$n = 30$	$n = 40$
<i>SngThrDtr</i>	24686	25865	26293	26502
<i>SngThrRnd</i>	24713	25900	26319	26535
<i>MltThrDtr</i>	24597	25808	26199	26392

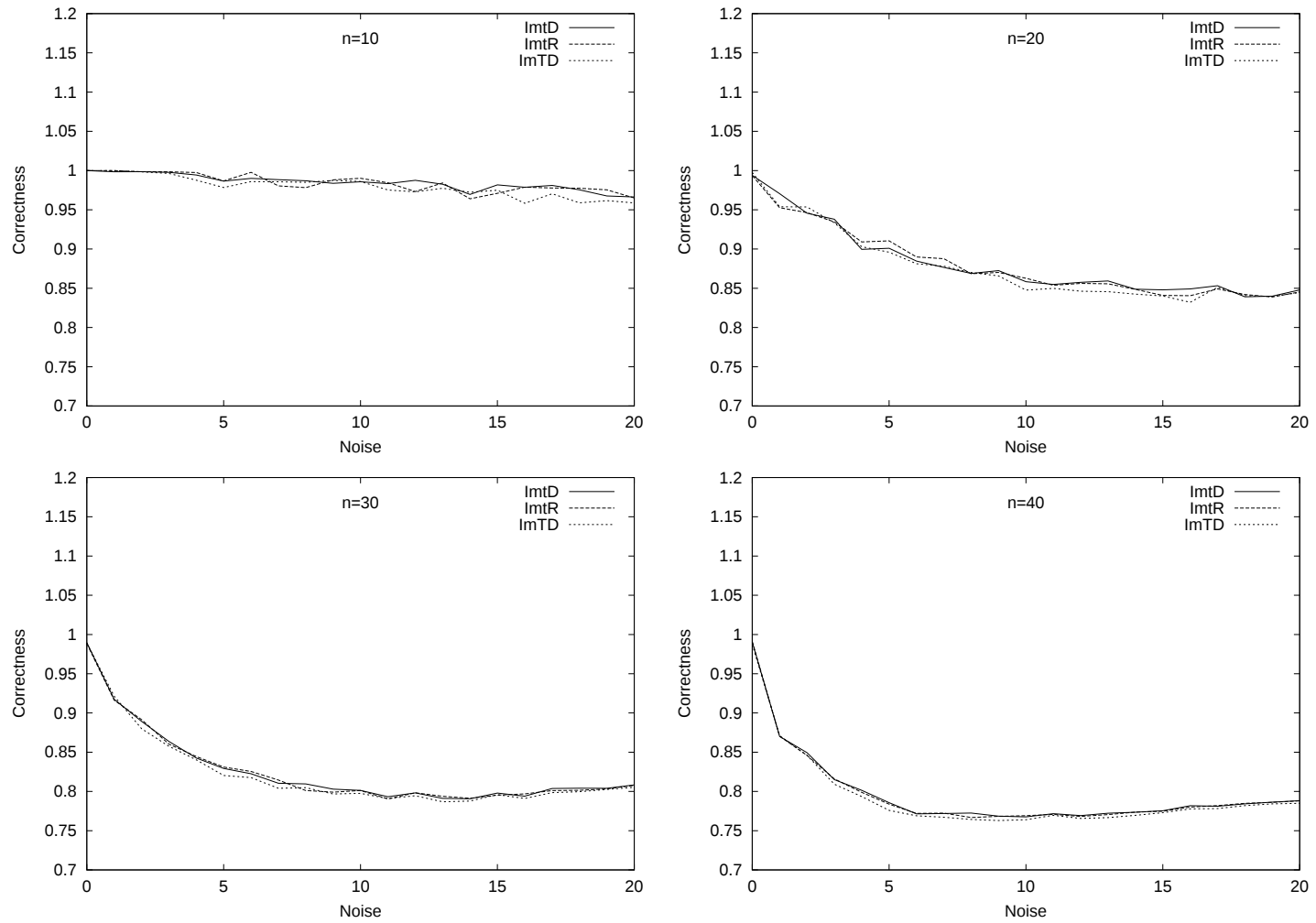
Table 6.26: *Number of evaluations of the objective function for Insertion Sort with Minimal progress mode, and for different graph sizes.*

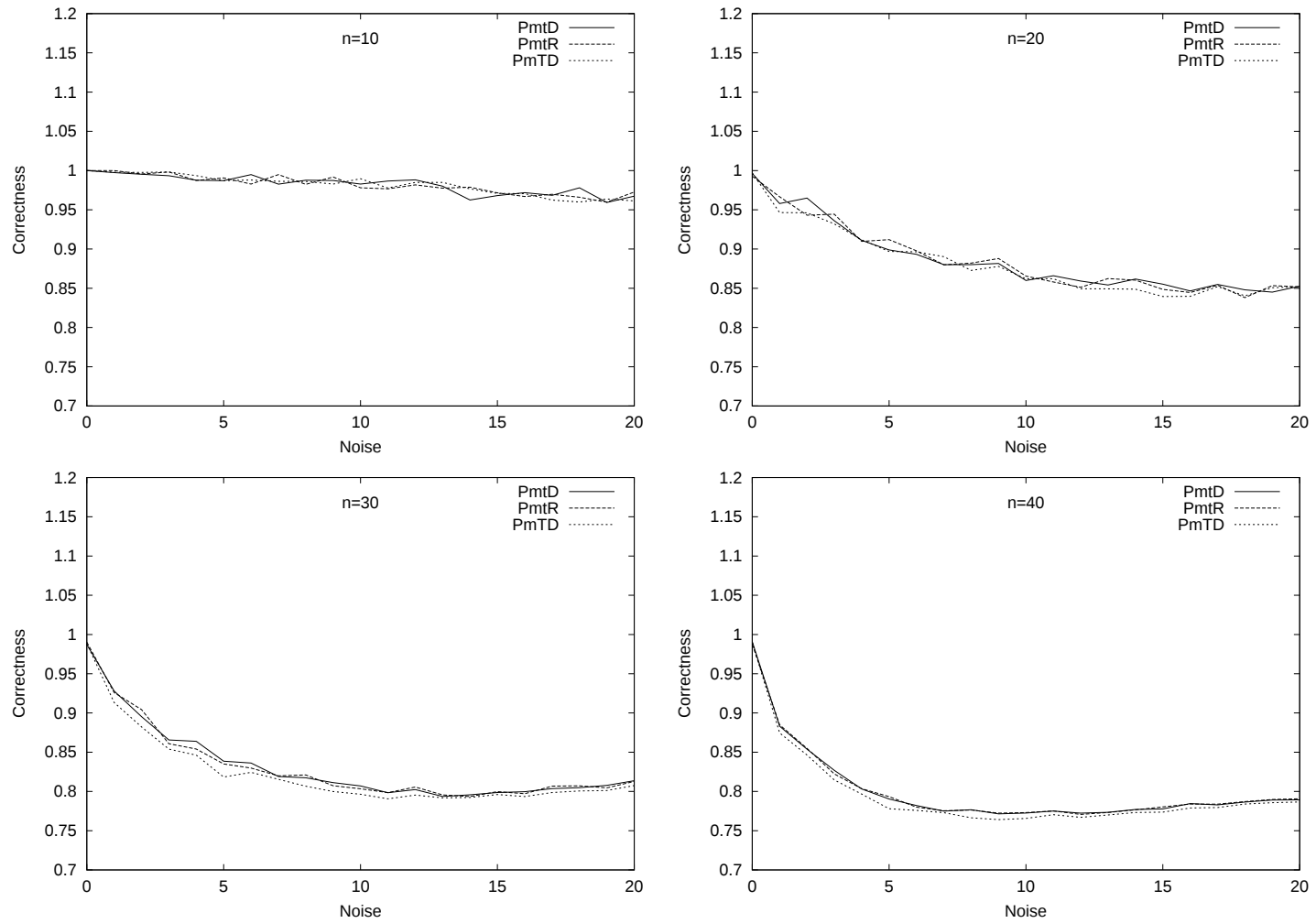
State	Size of the problem			
	$n = 10$	$n = 20$	$n = 30$	$n = 40$
<i>SngThrDtr</i>	30417	31363	31703	31875
<i>SngThrRnd</i>	30449	31393	31735	31902
<i>MltThrDtr</i>	30464	31341	31726	31949

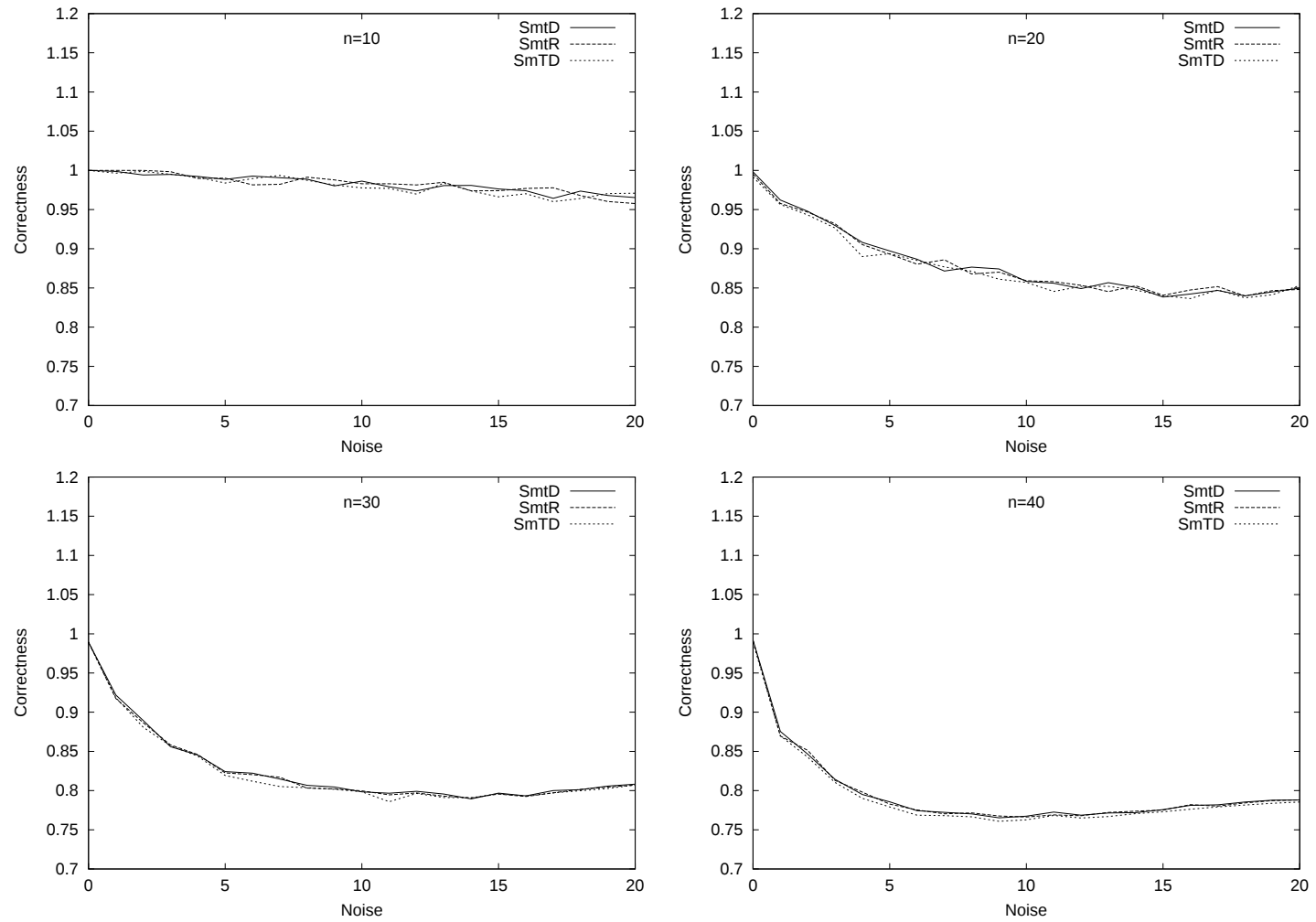
Table 6.27: *Number of evaluations of the objective function for Partition with Minimal progress mode, and for different graph sizes.*

State	Size of the problem			
	$n = 10$	$n = 20$	$n = 30$	$n = 40$
<i>SngThrDtr</i>	24682	25866	26290	26510
<i>SngThrRnd</i>	24709	25898	26323	26537
<i>MltThrDtr</i>	24764	25988	26386	26575

Table 6.28: *Number of evaluations of the objective function for Selection Sort with Minimal progress mode, and for different graph sizes.*

Figure 6.23: *Insertion Sort, minimal progress mode.*

Figure 6.24: *Partition, minimal progress mode.*

Figure 6.25: *Selection Sort, minimal progress mode.*

For *Maximal progress mode*, the dotted lines in Figure 6.26, Figure 6.27 and Figure 6.28, indicate results for our *MA-sorting* instantiated with *MxmPrgMltThrDtr*, the dashed lines indicate results for our *MA-sorting* instantiated with *MxmPrgSngThrRnd*, while the solid lines indicate *MxmPrgSngThrDtr*.

These figures illustrate that, for the *Maximal progress mode* (for all tested graph sizes and different noise levels), differences in effectiveness are not relevant across *Insertion Sort*, *Partition* and *Selection Sort*. Consequently, the quality of results is not largely influenced by changing the method for generating the state of the algorithm, in the case of *Maximal progress mode*.

Table 6.29, Table 6.30 and Table 6.31 show that for *Maximal progress mode*, we have results similar to those for *Minimal progress mode*. That is, there are marginal differences in the number of evaluations as the size of the graph increases, and the use of different methods for generating the state of the sorting algorithm does not substantially change the number of evaluations.

Thus, in terms of effectiveness and efficiency, for *Maximal progress mode*, we do not detect differences when a different method for generating the state of the sorting algorithm is applied.

The main explanation for the irrelevance of the method for generation the state of the sorting algorithm, across *Insertion Sort*, *Partition* and *Selection Sort*, is the small number of neighbors explored by both the *Minimal progress mode* and the *Maximal progress mode*. Recall that in *Maximal progress mode* the sorting algorithm stops as soon as a comparison produces no improvement.

State	Size of the problem			
	$n = 10$	$n = 20$	$n = 30$	$n = 40$
<i>SngThrDtr</i>	24988	26507	27491	27969
<i>SngThrRnd</i>	25023	26555	27562	28026
<i>MltThrDtr</i>	24709	26002	26641	26905

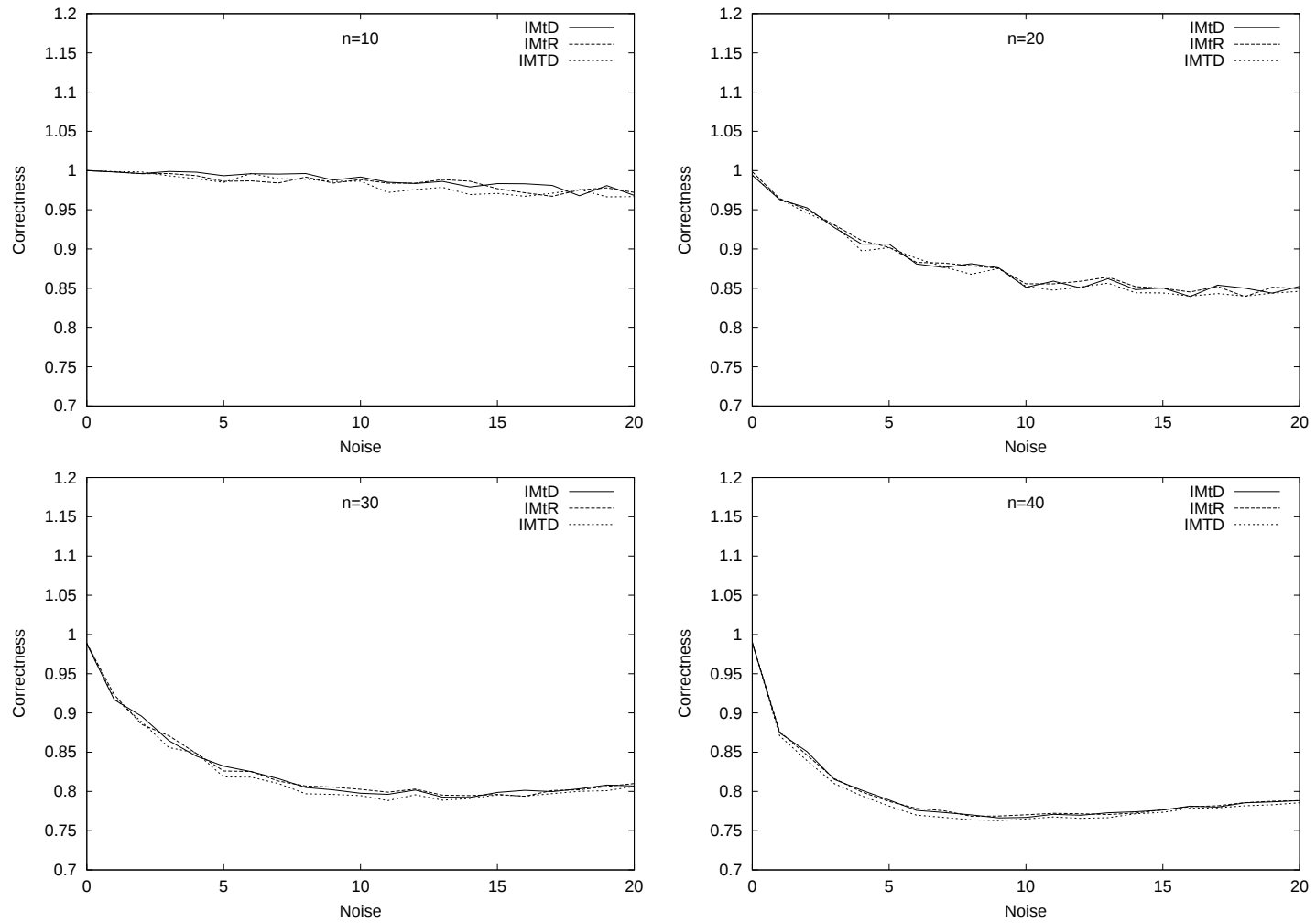
Table 6.29: *Number of evaluations of the objective function for Insertion Sort with Maximal progress mode, and for different graph sizes.*

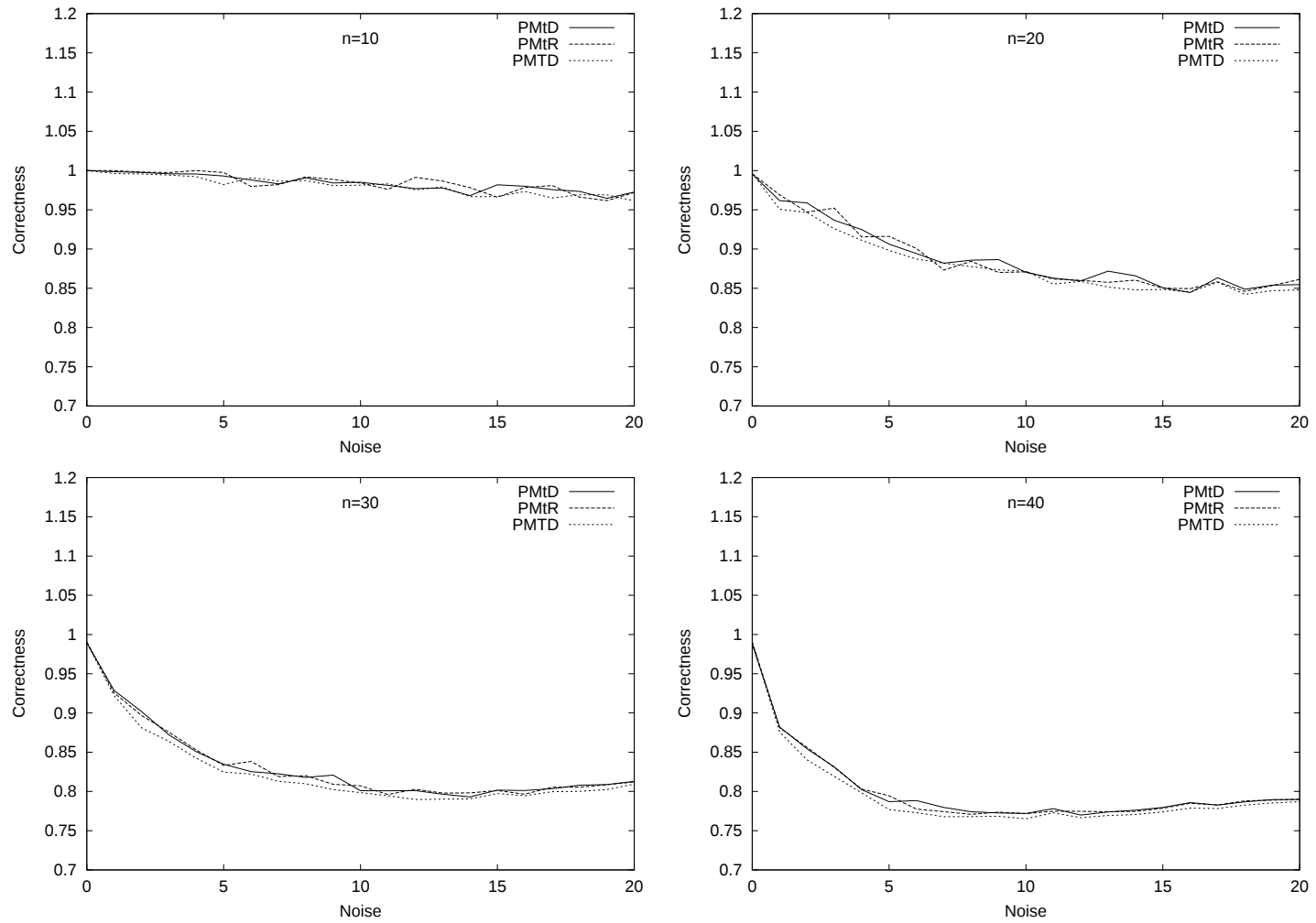
State	Size of the problem			
	$n = 10$	$n = 20$	$n = 30$	$n = 40$
<i>SngThrDtr</i>	31179	32143	32459	32648
<i>SngThrRnd</i>	31232	32166	32486	32684
<i>MltThrDtr</i>	31241	32029	32436	32691

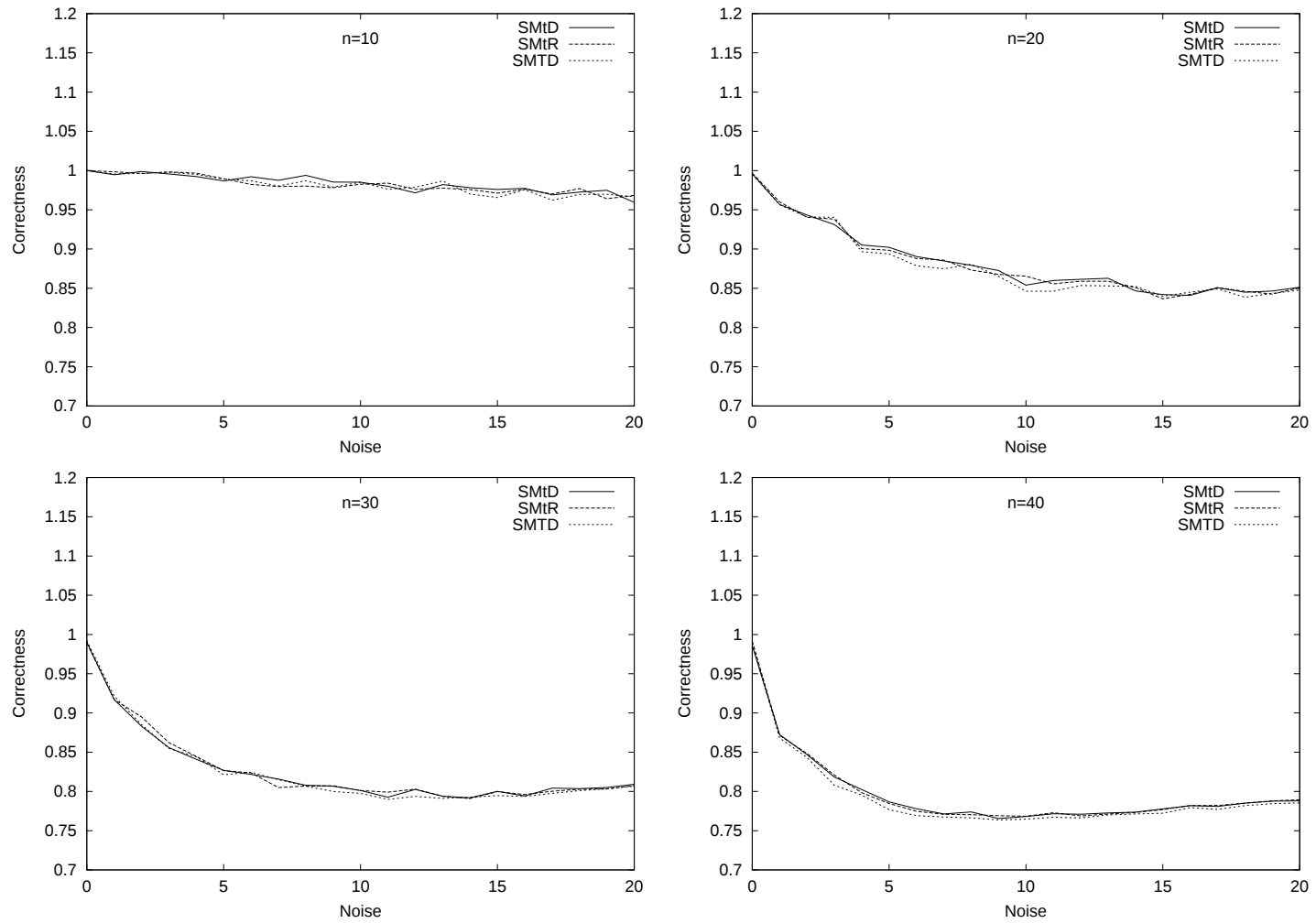
Table 6.30: *Number of evaluations of the objective function for Partition with Maximal progress mode, and for different graph sizes.*

State	Size of the problem			
	$n = 10$	$n = 20$	$n = 30$	$n = 40$
<i>SngThrDtr</i>	26716	28319	29035	29372
<i>SngThrRnd</i>	26755	28363	29069	29412
<i>MltThrDtr</i>	26976	28719	29411	29747

Table 6.31: *Number of evaluations of the objective function for Selection Sort with Maximal progress mode, and for different graph sizes.*

Figure 6.26: *Insertion Sort, Maximal progress mode.*

Figure 6.27: *Partition, Maximal progress mode.*

Figure 6.28: *Selection Sort, Maximal progress mode.*

For *First improvement*, the dotted lines in Figure 6.29, Figure 6.30 and Figure 6.31, indicate results for our *MA-sorting* instantiated with *FrsImpMltThrDtr*, the dashed lines indicate results for our *MA-sorting* instantiated with *FrsImpSngThrRnd*, while the solid lines indicate *FrsImpSngThrDtr*.

Figure 6.29 illustrates that, using *Insertion Sort* with *First improvement*, the variants *SngThrDtr* and *SngThrRnd* are slightly better than *MltThrDtr*, particularly for sizes $n = 20$ to $n = 40$.

In Figure 6.30 and Figure 6.31 we observe important differences that result from the application of different methods for generating the state of the sorting algorithm. In particular, Figure 6.30, and Figure 6.31 illustrate that, for sizes $n = 30$ or $n = 40$, the variants *PrtFrsImpMltThrDtr* and *SlcFrsImpMltThrDtr* are the least effective methods for generating the state of the sorting algorithm. Thus, in the case of *First improvement*, with *Partition*, the use of *Single-Threading* mode (random or deterministic) provides important benefits, in terms of effectiveness, at least to problems with size $n = 30$ or $n = 40$. Also, for *Selection Sort* these benefits are noticeable for those sizes, but only for noise levels $\varepsilon = 0$ to $\varepsilon = 17$.

Table 6.32 and Table 6.33 show that, using *Insertion Sort* and *Partition* with *First improvement*, the variant *MltThrDtr* is the least expensive in terms of number of evaluations, and that the variant *SngThrRnd* is the most expensive. These tables also show that the number of evaluations increases linearly as the size of the graph grows, although this growth is less noticeable in the case of *MltThrDtr* (moreover, in this case and for size $n = 40$ the number of evaluations is slightly lower than for size $n = 30$).

State	Size of the problem			
	$n = 10$	$n = 20$	$n = 30$	$n = 40$
<i>SngThrDtr</i>	33446	45448	53685	62771
<i>SngThrRnd</i>	33462	45503	53757	62943
<i>MltThrDtr</i>	32552	41989	42033	41993

Table 6.32: *Number of evaluations of the objective function for Insertion Sort with First improvement, and for different graph sizes.*

State	Size of the problem			
	$n = 10$	$n = 20$	$n = 30$	$n = 40$
<i>SngThrDtr</i>	45908	53172	58219	61589
<i>SngThrRnd</i>	46016	53053	58307	61950
<i>MltThrDtr</i>	45514	52575	53009	52892

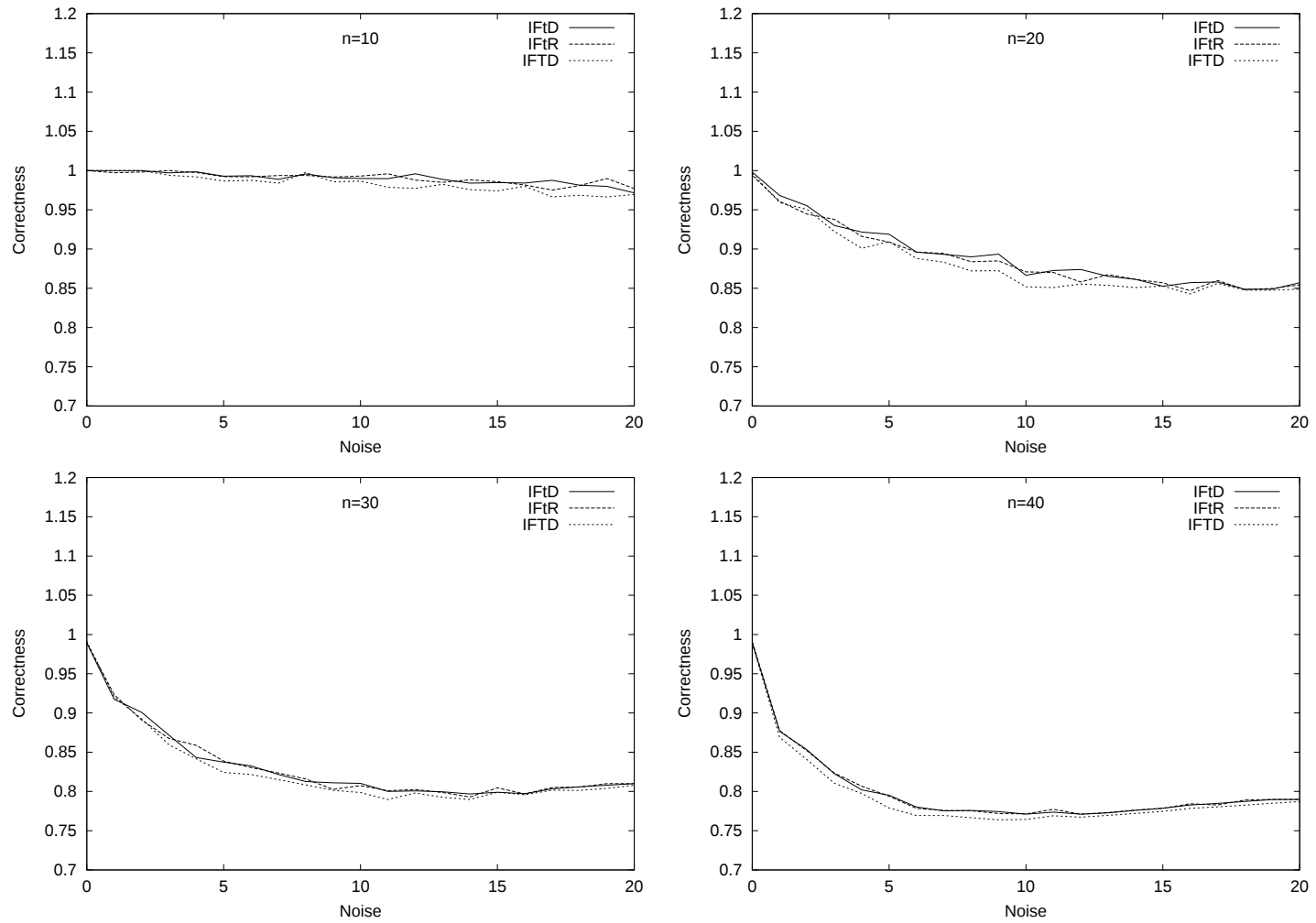
Table 6.33: *Number of evaluations of the objective function for Partition with First improvement, and for different graph sizes.*

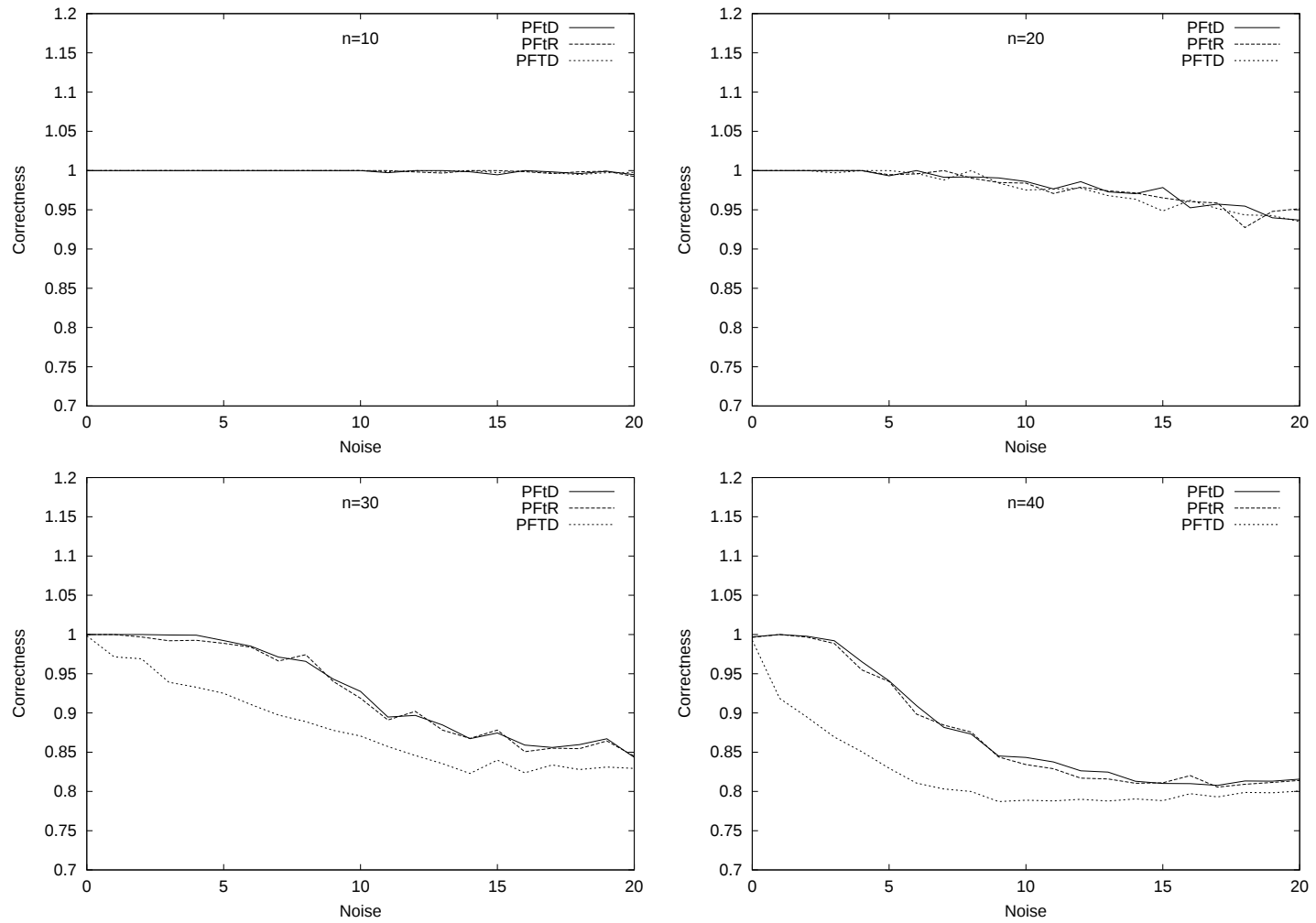
Table 6.34 shows that, using *Selection Sort* with *First improvement*, the variant *SngThrDtr* is the least expensive in terms of number of evaluations, and that the variant *MltThrDtr* is the most expensive. These tables also show that the number of evaluations increases linearly as the size of the graph grows, although this growth is more evident in the case of *MltThrDtr*.

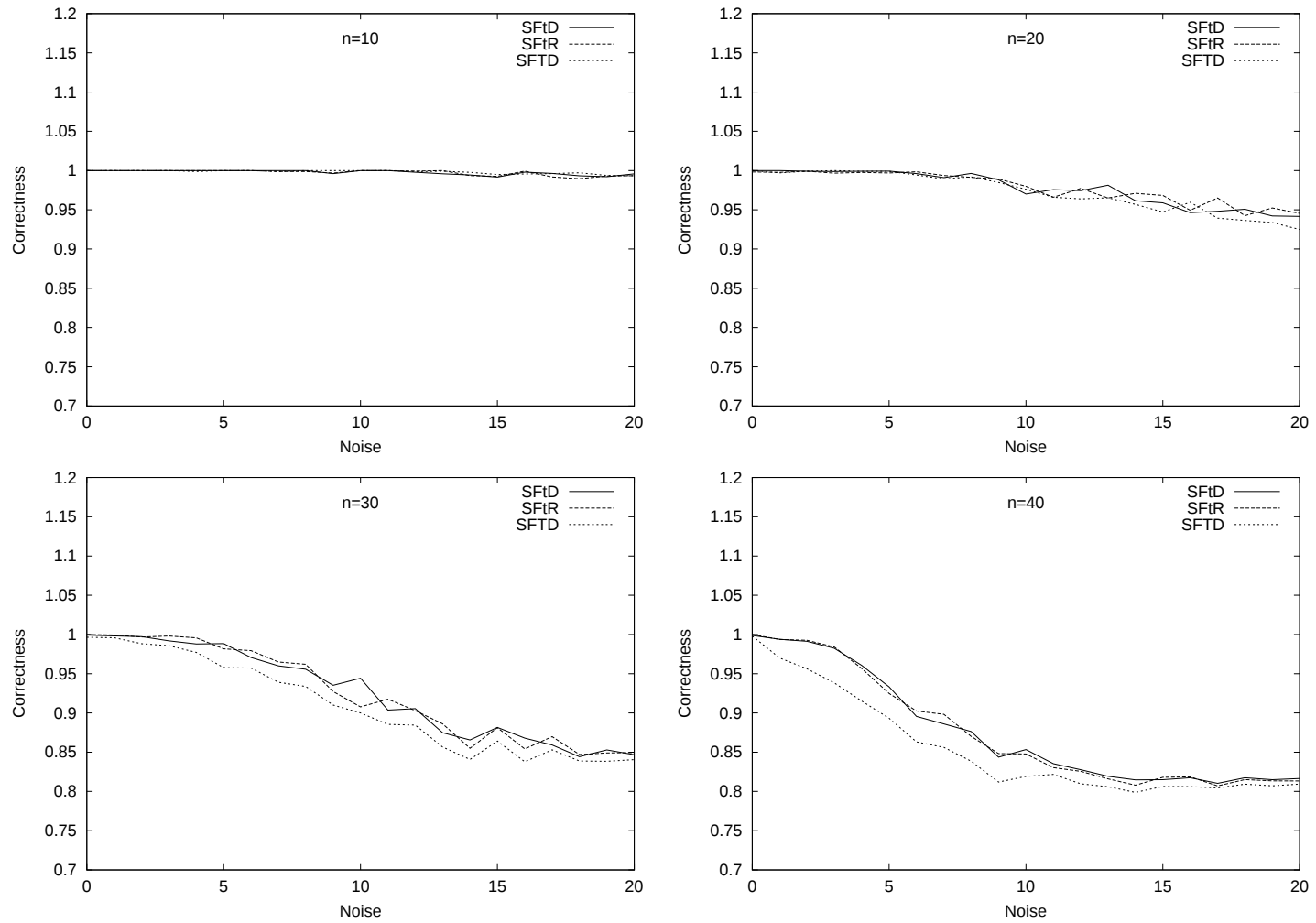
In effectiveness and efficiency, for *First improvement*, the variant *SngThrDtr* is better than *SngThrRnd*, and the latter is better than *MltThrDtr*.

State	Size of the problem			
	$n = 10$	$n = 20$	$n = 30$	$n = 40$
<i>SngThrDtr</i>	31041	41425	45812	50299
<i>SngThrRnd</i>	31109	41535	45933	50417
<i>MltThrDtr</i>	31755	43425	53234	61784

Table 6.34: *Number of evaluations of the objective function for Selection Sort with First improvement, and for different graph sizes.*

Figure 6.29: *Insertion Sort, First improvement.*

Figure 6.30: *Partition, First improvement.*

Figure 6.3: *Selection Sort, First improvement.*

For *Best improvement*, the dotted lines in Figure 6.32, Figure 6.33 and Figure 6.34, indicate results for our *MA-sorting* instantiated with *BstImpMltThrDtr*, the dashed lines indicate results for our *MA-sorting* instantiated with *BstImpSngThrRnd*, while the solid lines indicate *BstImpSngThrDtr*.

Figure 6.32 illustrates that, using *Insertion Sort* with *Best improvement*, the variants *SngThrDtr* and *SngThrRnd* are slightly better than the variant *MltThrDtr* (for all tested graph sizes).

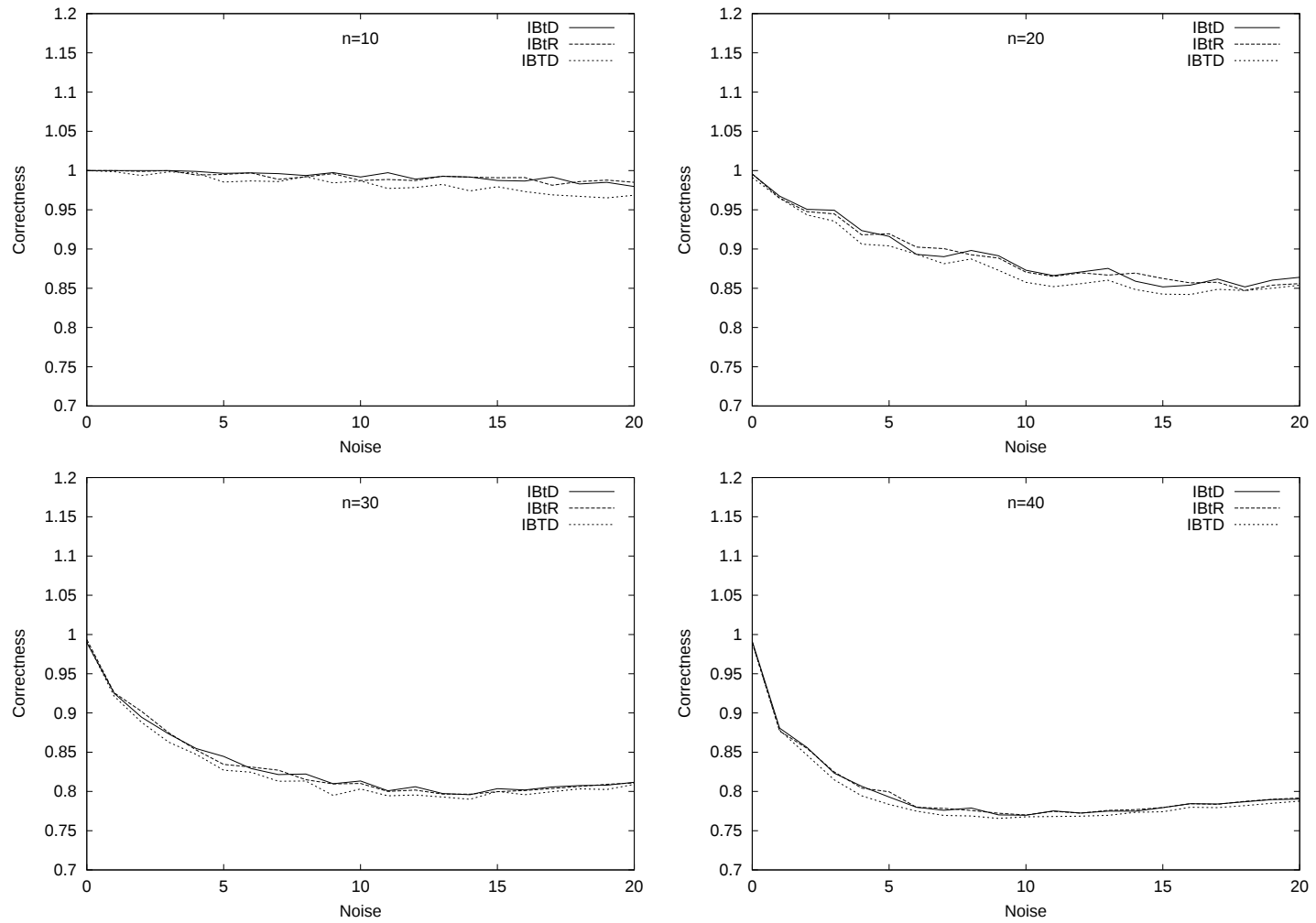
Figure 6.33 illustrates that, using *Partition* with *Best improvement* (for size $n = 40$ and noise levels in the interval $\varepsilon = 0$ to $\varepsilon = 13$), the variant *MltThrDtr* is the least effective method for generating the state of the sorting algorithm.

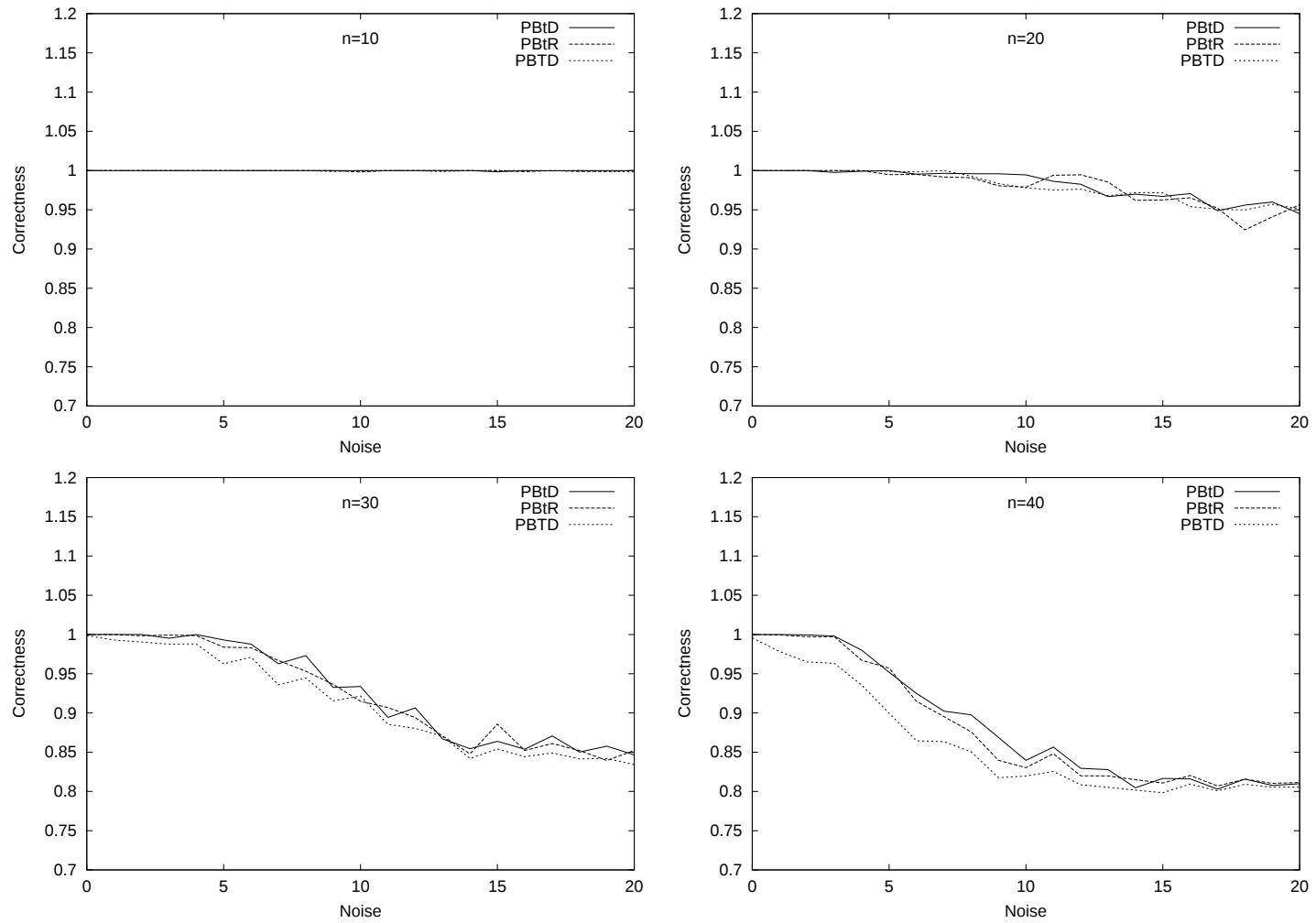
Figure 6.34 illustrates that, using *Selection Sort* with *Best improvement* (for sizes $n = 30$ or $n = 40$, and noise levels in the interval $\varepsilon = 0$ to $\varepsilon = 11$), the variant *MltThrDtr* is the least effective method for generating the state of the sorting algorithm. Thus, the *Multi-threading* mode is not superior to the variants *SngThrDtr* and *SngThrRnd*, using *Best improvement*, across *Insertion Sort*, *Partition* and *Selection Sort*.

Table 6.35 shows that, using *Insertion Sort* with *Best improvement*, the variant *MltThrDtr* is the least expensive in terms of number of evaluations, and that the variant *SngThrRnd* is the most expensive. This table also shows that the number of evaluations increases linearly as the size of the graph grows, although this growth is less noticeable in the case of *MltThrDtr*.

State	Size of the problem			
	$n = 10$	$n = 20$	$n = 30$	$n = 40$
<i>SngThrDtr</i>	34391	48911	62753	76358
<i>SngThrRnd</i>	34464	49065	63012	76799
<i>MltThrDtr</i>	32834	42761	43819	43993

Table 6.35: *Number of evaluations of the objective function for Insertion Sort with Best improvement, and for different graph sizes.*

Figure 6.32: *Insertion Sort, Best improvement.*

Figure 6.33: *Partition, Best improvement.*

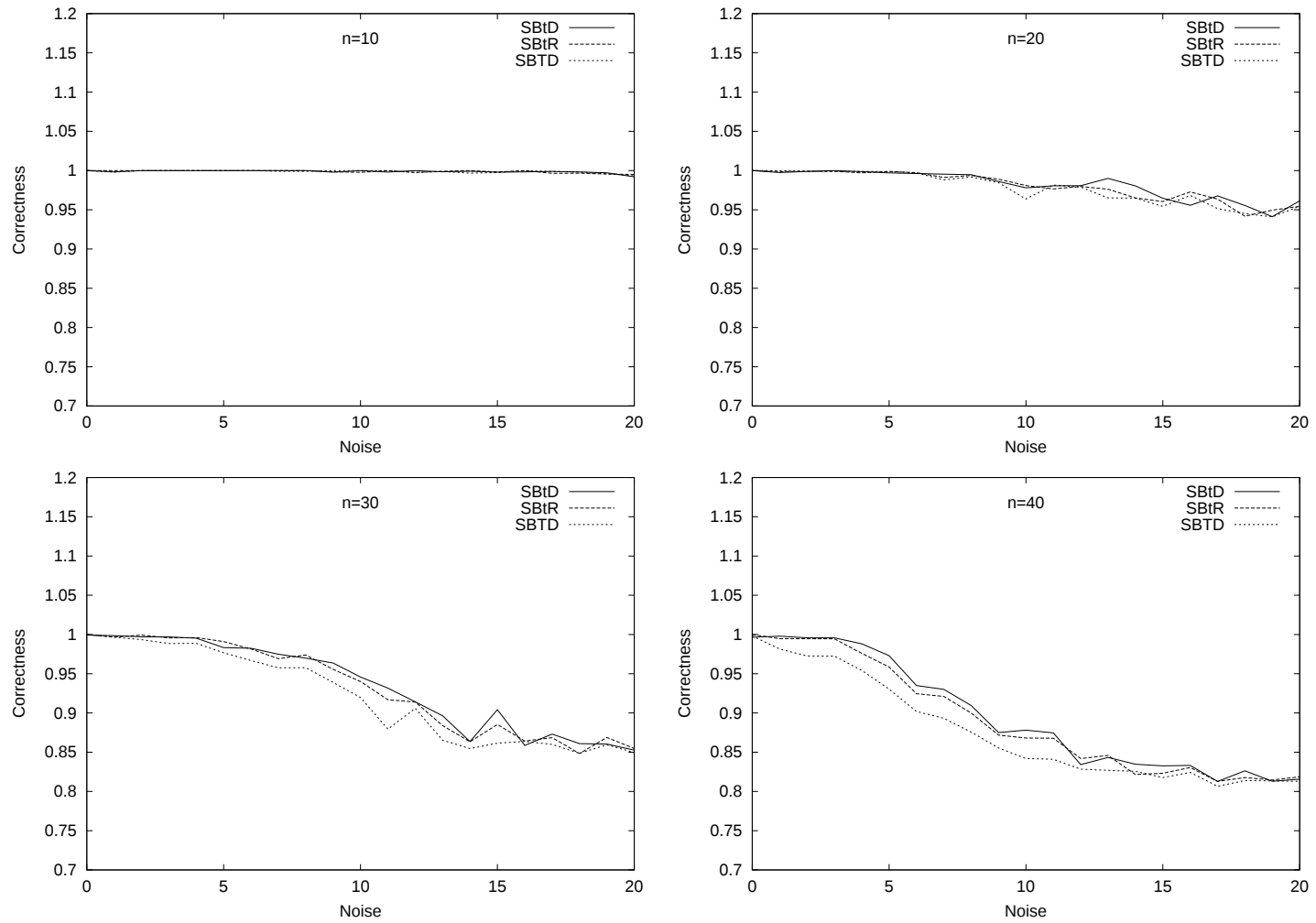
Figure 6.34: *Selection Sort, Best improvement.*

Table 6.36 shows that, using *Partition* with *Best improvement*, the variant *SngThrRnd* is the least expensive in terms of number of evaluations. This table also shows that the number of evaluations increases linearly as the size of the graph grows.

State	Size of the problem			
	$n = 10$	$n = 20$	$n = 30$	$n = 40$
<i>SngThrDtr</i>	59142	90789	121460	151590
<i>SngThrRnd</i>	56274	88087	118832	148917
<i>MltThrDtr</i>	59331	90893	121204	151241

Table 6.36: *Number of evaluations of the objective function for Partition with Best improvement, and for different graph sizes.*

Table 6.37 shows that, using *Selection Sort* with *Best improvement*, the variant *SngThrDtr* is the least expensive in terms of number of evaluations, and that the variant *MltThrDtr* is the most expensive. This table also shows that the number of evaluations increases linearly as the size of the graph grows, although this growth is more noticeable in the case of *MltThrDtr*.

In conclusion, for *Best improvement*, the trade-off effectiveness-efficiency favors a *Single-threading* approach: random for *Partition*; and deterministic for *Insertion Sort* and *Selection Sort*.

Table 6.38 summarizes the impact of the method of generating the state of the sorting algorithm into *MA-sorting*. According to these results, if the potential of the neighborhood structure is not fully displayed (or if we are dealing with an unfruitful neighborhood structure), the impact of the method for generating the state of the

State	Size of the problem			
	$n = 10$	$n = 20$	$n = 30$	$n = 40$
<i>SngThrDtr</i>	33939	48101	61600	75196
<i>SngThrRnd</i>	34017	48204	61728	75192
<i>MltThrDtr</i>	35323	53965	80265	107270

Table 6.37: Number of evaluations of the objective function for Selection Sort with Best improvement, and for different graph sizes.

sorting algorithm on the performance is minimum. We have found that for *Partition* and *Selection Sort*, and *FrsImp* and *BstImp*, *Single-threading* is superior to *Multi-threading*.

One of the reasons for these results is that *Single-threading* distributes more evenly, in the search space of permutations, the chances of being part of improvement by local search. In contrast, *Multi-threading* strongly biases the search by a parameter (the current state) which is not regulated by the evolutionary search.

We confirm our hypothesis that the major improvement comes from the neighborhood structure proposed by the sorting algorithm. We also confirm that the benefits of a fruitful neighborhood structure can be intensified by increasing the number of neighbors explored. Finally, we have found that the method for generating the state of the sorting algorithm influences the performance, particularly in the context of fruitful neighborhood structures and larger local neighborhoods. In our case study, the *Single-threading* approach is better than *Multi-Threading*.

Amount of Progress	Sorting Algorithm		
	<i>Insertion Sort</i>	<i>Partition</i>	<i>Selection Sort</i>
<i>MnmPrg</i>	not relevant	not relevant	not relevant
<i>MxmPrg</i>	not relevant	not relevant	not relevant
<i>FrsImp</i>	<i>SngThr Dtr</i>	<i>SngThr Dtr</i>	<i>SngThr Dtr</i>
<i>BstImp</i>	<i>SngThr Dtr</i>	<i>SngThr Rnd</i>	<i>SngThr Dtr</i>

Table 6.38: *Impact of the method of generating the state of the sorting algorithm into MA-sorting.*

6.6 A Comparison between *MA-sorting* and GBSA

In this second set of experiments, while we compare the performance obtained by the execution of *MA-sorting* and GBSA [141], we evaluate the impact of increasing the number of explorations and the impact of providing a better quality initial solution to our *MA-sorting*. We expect that exploring a larger number of neighbors will intensify the benefits of a fruitful neighborhood structure, and that a better quality initial solution will improve the performance of the *MA-sorting*.

A first comparison arises from the results of the first set of experiments. Thus, we compare GBSA with those variants of *MA-sorting* (for *Insertion Sort*, *Partition* and *Selection Sort*) which provide the best quality of solutions. In general, those variants which include *BstImp*, *SngThr* and *Dtr* are the best.

In Figure 6.35, the dotted lines indicate results for our *MA-sorting*, instantiated with *InsBstImpSngThrDtr* (labeled in the figure as IBtD), the dashed lines indicate

results for Wang *et al.* approach (labeled in the figures as GBSA), the broken lines indicate results for our *MA-sorting* instantiated with *PrtBstImpSngThrDtr* (labeled in the figure as PBtD), while the solid lines indicate *SlcBstImpSngThrDtr* (labeled SBtD).

Figure 6.35 shows that the variants based on *Partition* (PBtD) and *Selection Sort* (SBtD), are much better than both GBSA and the variant based on *Insertion Sort* (IBtD). This figure also shows the statistical significance of our results. The lines represent the average of the 50 solutions produced by each method. The intervals plotted are 99% confidence intervals, and show that SBtD and PBtD are better than both IBtD and GBSA, for sizes $n = 20$ (for noise levels $\varepsilon > 8$), $n = 30$ (for all noise levels) and $n = 40$ (for all noise levels). These results have a very high statistical significance (because the intervals do not overlap the statistical significance is even higher than 99% significant). Thus, even a multi-start GBSA with a similar number of evaluations as a run of *MA-sorting* has an extremely small opportunity to produce better results.

Table 6.39 displays the number of evaluations performed by each algorithm for the tested graph sizes. This table shows that the number of evaluations made by GBSA is the lowest. The main reason is that, in contrast with *MA-sorting*, GBSA does not call to the evaluation function each time a mutation operation is requested. The number of evaluations (for PBtD, SBtD and IBtD) increases linearly.

As explained in Section 6.5, the partial information provided to *Partition* is an index to the “pivot”. Thus, in this case, each call to mutation with *Best improvement* has cost of n . For *Selection Sort*, the partial information indicates a position in the

permutation. Thus, in this case, each call to mutation with *Best improvement* has an expected cost of $n/2$. It explains why the number of evaluations for *Partition* is almost the double those performed by *Selection Sort*.

Genetic Algorithm	Size of the problem			
	$n = 10$	$n = 20$	$n = 30$	$n = 40$
<i>PrtBstImpSngThrDtr</i>	59142	90789	121460	151590
<i>SlcBstImpSngThrDtr</i>	33939	48101	61600	75196
<i>InsBstImpSngThrDtr</i>	34391	48911	62753	76358
GBSA	22755	23704	24036	24209

Table 6.39: Comparison of the number of evaluations of the objective function between GBSA and some variants of MA-sorting for different graph sizes.

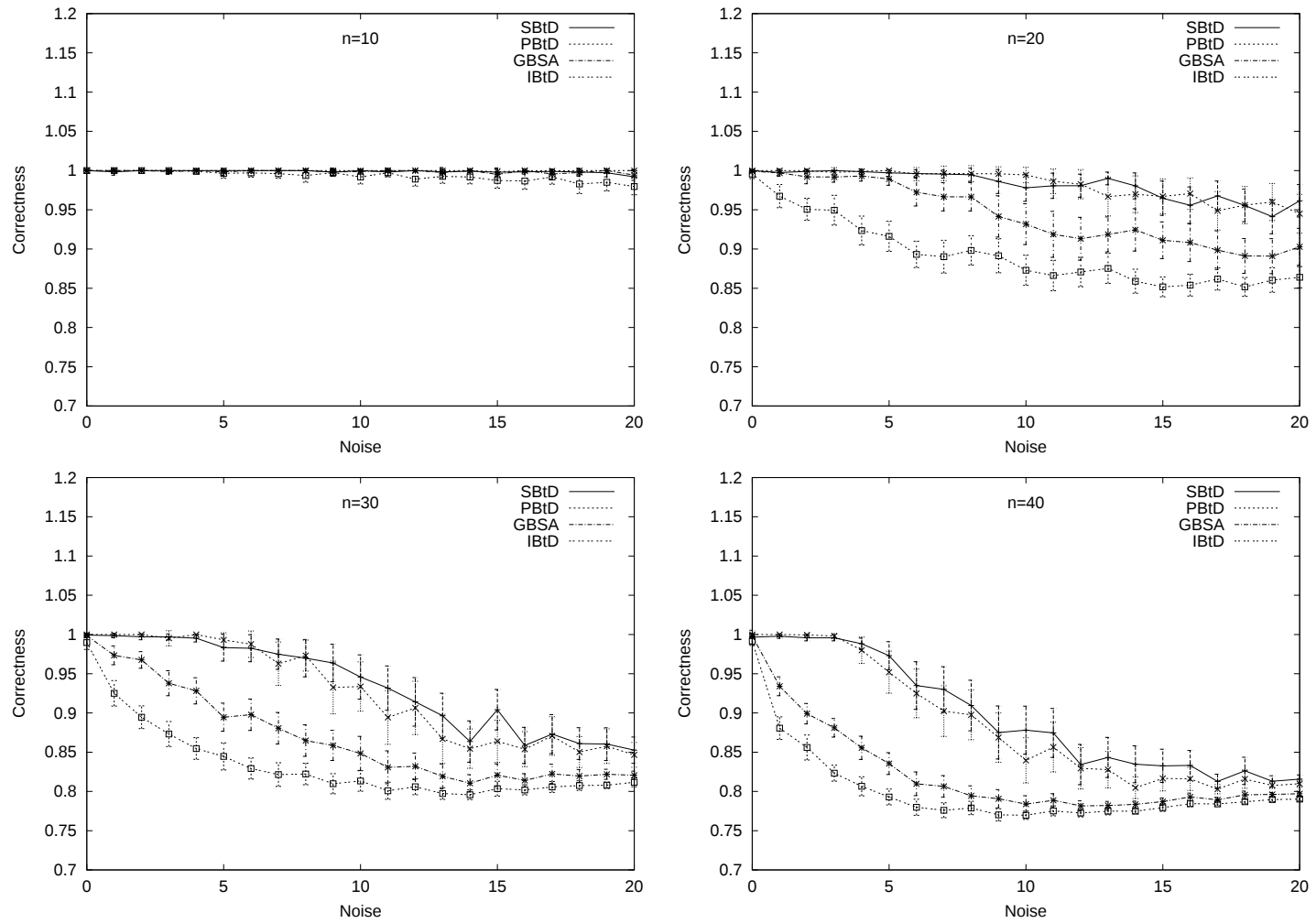


Figure 6.35: Comparison of quality of solutions between *MA-sorting* and *GBSA* for different graph sizes.

6.6.1 Increasing the Number of Explorations

Recall that in our LS model (Section 3.1), two nested cycles regulate the number of explorations to be performed in the search space. The inner cycle regulates the number of explorations in the local neighborhood. Each iteration of the inner cycle produces a good local neighbor (or the best if a *BstImp* approach is applied). After obtaining such a good neighbor, and in accordance with our LS model, it becomes the current solution and we perform a new iteration of the inner cycle. The outer cycle regulates the length of the path traveled in the search space. In the first set of experiments we dealt with the inner cycle by using four modes for the amount of progress: *MnmPrg*, *MxmPrg*, *FrsImp* and *BstImp*. This time we deal with the outer cycle.

We regulate the number of explorations performed by the outer cycle by requiring the ordering of a substring of the input sequence. Thus, each time a mutation operation is requested, *MA-sorting* generates, uniformly at random, two positions q, t (where $q < t$) that delimit the subsequence to be ordered by the sorting algorithm. These random positions are used by *Insertion Sort*, *Selection Sort* and *Quicksort* to establish their initial state and the extent of their sorting.

Insertion Sort considers index q as its index i and defines $j = i$, thus the first comparison is $< (j, i + 1)$. Further comparisons follow those branches prescribed by the tree T of *Insertion Sort*. The mutation operator of *MA-sorting* halts until it inserts the element pointed by the index $i = t$.

Selection Sort considers index q as its index k and defines $j = k + 2$, thus the first comparison is $< (k + 1, j)$. Further comparisons follow those branches prescribed by the tree T of *Selection Sort*. The mutation operator of *MA-sorting* halts until it places the element pointed by the index $k = t - 1$ in this best location.

Quicksort considers index q as its index l , and considers index t as its index r . The comparisons follow those branches prescribed by the tree T of *Quicksort*. This time, while ordering the subsequence, we keep complete information about the state of sorting. Thus, we maintain a record of indices l , r , pivot ϑ and the state of the stack.

Considering the results obtained so far, we expect *MA-sorting* instantiated with *Quicksort* or *Selection Sort* will improve their performance. We expect marginal improvements for *MA-sorting* instantiated with *Insertion Sort* (mainly because this variant does not generate a fruitful neighborhood structure, as shown in Section 6.5.1).

In this second set of experiments, we use the same parameters (given in Table 6.1) and test cases as those applied in the first set of experiments (Section 6.5). For the amount of progress, which is regulated by the inner cycle, the tested variants of *MA-sorting* use *Best improvement*.

In Figure 6.36, the dotted lines indicate results for our *MA-sorting* instantiated with *Insertion Sort* (labeled in the figure as MA-Isort), the dashed lines indicate results for Wang *et al.* (labeled in the figures as GBSA), the broken lines indicate results for our *MA-sorting* instantiated with *Quicksort* (labeled in the figure as MA-Qsort), while the solid lines correspond to our *MA-sorting* instantiated with *Selection Sort* (labeled MA-Ssort).

Figure 6.36 shows remarkable improvements for *MA-sorting* when instantiated with both *Quicksort* and *Selection Sort*, in comparison with the instantiation with *Insertion Sort*. We also observe that the improvement for the variant with *Selection Sort* is larger than the improvement for the variant with *Quicksort*. These results confirm that a larger number of explorations intensifies the benefits of a fruitful neighborhood structure.

Figure 6.36 also illustrates the statistical significance of our results. The lines represent the average of the 50 solutions produced by each method. The intervals plotted are 99% confidence intervals, and show that MA-Ssort and MA-Qsort are better than both MA-Isort and GBSA, for sizes $n = 20$ (for noise levels $\varepsilon > 5$), $n = 30$ (for all noise levels) and $n = 40$ (for all noise levels). Note that MA-Ssort is better than MA-Qsort for sizes $n = 30$ (for noise levels $\varepsilon > 10$) and $n = 40$ (for noise levels $\varepsilon > 5$). These results have a very high statistical significance (because the intervals do not overlap the statistical significance is even higher than 99% significant).

Table 6.40 illustrates that the number of evaluations performed by GBSA is the lowest and it is almost constant. Also, that MA-Qsort scales better than both MA-Ssort and MA-Isort.

Table 6.40 also shows that, for the range of n values, *MA-sorting* adds overhead, but the table shows that it is linear for MA-Qsort and polynomial (with a small constant exponent) for MA-Ssort and MA-Isort. Observe that although MA-Isort and MA-Ssort perform almost the same number of evaluations, the quality of MA-Ssort is clearly superior to MA-Isort. The increment in the number of evaluations made by *MA-sorting* is worthwhile, particularly for MA-Qsort and MA-Ssort.

Genetic Algorithm	Size of the problem			
	$n = 10$	$n = 20$	$n = 30$	$n = 40$
MA-Qsort	74322	163890	281947	423130
MA-Ssort	72516	218145	453290	779275
MA-Isort	64966	201397	427974	745363
GBSA	22755	23704	24036	24209

Table 6.40: *Comparison of the number of evaluations of the objective function between GBSA and some variants of MA-sorting for different graph sizes. MA-sorting orders a subsequence of the input sequence.*

According to these results, we confirm that by increasing the number of explorations performed by *MA-sorting*, in the context of a fruitful neighborhood structure, we obtain improved results. That is, *MA-sorting* replicates the expected behavior of a LS mechanism.

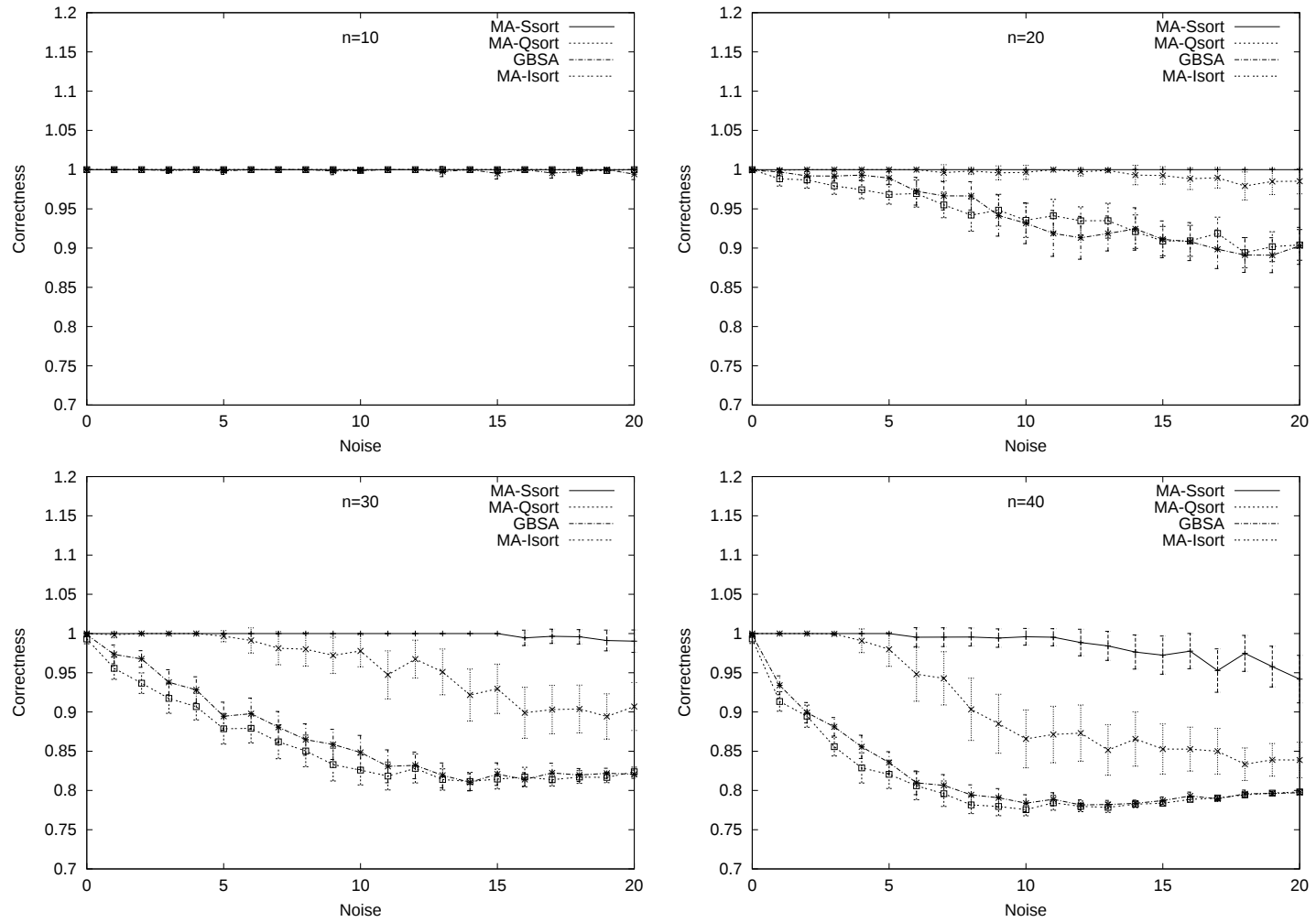


Figure 6.36: Comparison of quality of solutions between MA-sorting and GBSA for different graph sizes. MA-sorting orders a subsequence of the input sequence.

6.6.2 Providing a Better Initial Solution

Despite the increment in the number of explorations performed by *MA-sorting* instantiated with *Insertion Sort*, the quality of its solutions is lower than that provided by GBSA. For that reason, we consider this variant a good candidate to evaluate the impact of providing a better initial solution.

Until now the initial solution to *MA-sorting* is the permutation to be mutated. This time, and only for *MA-sorting* with *Insertion Sort*, we generate the initial solution in a different way. We proceed as follows. Each time a mutation operation is requested, we first exchange the positions that delimit the subsequence to be ordered. The best permutation, between the original and that resulting from the exchange, is the input to *MA-sorting* with *Insertion Sort*. As it occurs with local search, we expect to observe improved performance when a better initial solution is provided to the *MA-sorting* with *Insertion Sort*.

In Figure 6.37, the dotted lines indicate results for our *MA-sorting* instantiated with *Insertion Sort* (labeled in the figure as MA-Isort), the dashed lines indicate results for Wang *et al.* (labeled in the figures as GBSA), the broken lines indicate results for our *MA-sorting* instantiated with *Quicksort* (labeled in the figure as MA-Qsort), while the solid lines correspond to our *MA-sorting* instantiated with *Selection Sort* (labeled MA-Ssort).

Figure 6.37 shows an important improvement in the case of *MA-sorting* with *Insertion Sort*. Figure 6.37 also shows that *MA-sorting* with *Insertion Sort* is better than GBSA at least for sizes $n = 10$ and $n = 20$ [31]. For sizes $n = 30$ and $n = 40$, *MA-sorting* with *Insertion Sort* provides the same quality of solutions as GBSA.

While Table 6.41 illustrates that the increase in the number of evaluations is marginal (in comparison with those reported in Table 6.40), Figure 6.37 shows that the improvement in quality is noticeable.

Genetic Algorithm	Size of the problem			
	$n = 10$	$n = 20$	$n = 30$	$n = 40$
MA-Qsort	74322	163890	281947	423130
MA-Ssort	72516	218145	453290	779275
MA-Isort	69613	205899	431678	747902
GBSA	22755	23704	24036	24209

Table 6.41: *Comparison of the number of evaluations of the objective function between GBSA and some variants of MA-sorting for different graph sizes. MA-sorting orders a subsequence of the input sequence, and a better initial solution is the input to MA-Isort.*

According to these results, we confirm that a better quality initial solution improves the performance of *MA-sorting* with *Insertion Sort*. That is, *MA-sorting* with *Insertion Sort* replicates the expected behavior of a LS mechanism.

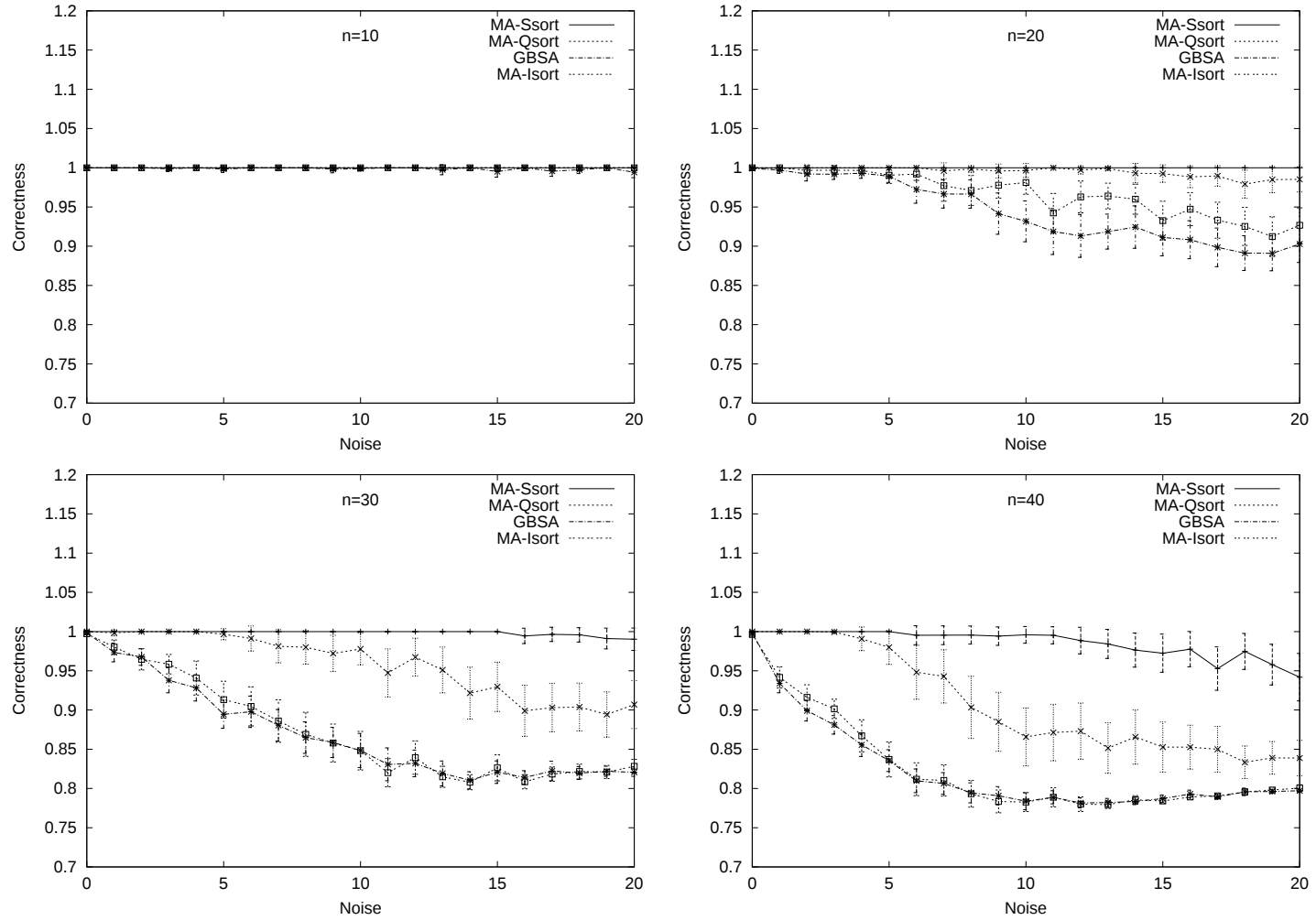


Figure 6.37: Comparison of quality of solutions between MA-sorting and GAB for different graph sizes. MA-sorting orders a subsequence of the input sequence, and a better initial solution is given to MA-Isort.

6.7 Discussion

A remarkable point derived from these experiments is the outstanding performance obtained by *MA-sorting* with *Selection Sort* and *Quicksort*.

Our *MA-sorting* does assume something about the nature of the optimization problem on permutations. It assumes regularity of the neighborhood structure with respect to a measure of presortedness. Problems should be such that a permutation that is very similar in sorted order to the optimal solution has also a good value for the objective function. Thus, it makes sense for the optimizer to learn heuristically: that is, record information “*k* before *l*” suggested by the sorting algorithm. Knowing which measure of presortedness is suitable for an optimization problem would go a long way in suggesting the sorting algorithms for the hybridization.

We have shown that *Eng* is a valid measure of presortedness for *Insertion Sort*, *Quicksort* and *Selection Sort*. We have also shown that *Selection Sort* improves *Eng* more effectively than *Quicksort* and *Insertion Sort*. Moreover, we have observed that *Eng* is compatible with the objective function for the ECGI problem. Thus, for the case study, a permutation that is a close order to the optimal permutation is a reasonable match between the two given graphs to reduce the error in the match.

We claim there are two important reasons for the outstanding performance of *MA-sorting* with *Selection Sort*. First, there is compatibility between optimizing the objective function (which guides the evolutionary search) and optimizing the measure of presortedness (which guides the sorting algorithm). Second, there is the use of an effective sorting algorithm, *Selection Sort*, to optimize such a measure of presortedness.

Moreover, we confirm (as pointed out by W. Hart [51]) that the behavior of a Memetic Algorithm which applies local search to a small number of individuals in the population, as our *MA-sorting* does, resembles the expected behavior of a LS algorithm.

Thus, we observe that the performance of *MA-sorting* is highly determined by three of the components of the LS embedded in it: neighborhood operator, number of neighbors explored, and schedule of exploration.

We have confirmed that the major contribution to the performance of *MA-sorting* comes from the quality of the neighborhood structure generated by the neighborhood operator. In our case study, the neighborhood structure proposed by *Selection Sort* is the best, and the neighborhood structure generated by *Quicksort* is better than that generated by *Insertion Sort*. While in *Selection Sort* and *Quicksort* the spread of *Focus* is more even, in *Insertion Sort* the spread of *Focus* is more unequal. Moreover, *Selection Sort* avoids the risk of being distorted by crossover because of its locality in the swaps performed. Thus, for the ECGI problem, *Selection Sort* and *Quicksort* propose fruitful neighborhood structures.

Our results show that, if we are dealing with a fruitful neighborhood structure and we perform a larger number of explorations, *MA-sorting* will obtain even better results. In our case of study, the benefits obtained by increasing the number of explorations are more important for *Selection Sort* and *Quicksort* than for *Insertion Sort*.

In our case study, we have found that the *Single-threading* method (random and deterministic) provides better results to generate the state of the sorting algorithm. Moreover, we have found that the method for generating such states influences

the performance of *MA-sorting*, particularly when exploring a fruitful neighborhood structure and performing more explorations.

Finally, we corroborate that, as it occurs with local search algorithms, the performance of *MA-sorting* can be improved by using a better initial solution.

Chapter 7

A Baldwinian Approach for Constraint Combinatorial Optimization

7.1 Introduction

Constraint problems pose an important question to the field of Genetic Algorithms (GAs): How should the concept of feasibility be handled? In this respect, several proposals have been suggested, from the numerical perspective [88] to the combinatorial perspective [23]. Two well known results from the field of GAs are relevant to this question. First, the improvement of the genotype, usually by a local search mechanism [137], has been recognized as an efficient way of coping with hard combinatorial

optimization problems. Second, for many combinatorial optimization problems it is relatively easy to transform an infeasible genotype into a feasible genotype by a fixing process [88]. These results are a window into our proposal.

Genetic Algorithms based on the Baldwin effect (also called *Baldwinian* approach) use the result of the improvement to the genotype without modifying the genotype. We introduce the *Baldwinian* approach to effectively deal with feasibility on the class of Constraint Combinatorial Optimization Problems (CCOPs) that admit formulation as permutation problems [124]. As a distinctive feature, our proposal includes a constructive deterministic stage to repair (without backtracking) infeasible genotypes.

The *n-queens problem* [135] and the *valued n-queens problem* [104] are cited frequently as prototypical CSPs and CCOPs, respectively. Also, the *n-queens problem* is a well-investigated example of a permutation problem, although practical examples also occur in scheduling and allocation problems. It has been proposed that, for the *n-queens problem*, GAs are not as effective as local-search mechanisms based on conflict minimization [58], (the trade-off on the time required to find a solution favors the latter). A recent GA approach [120] for the *valued n-queens problem* piles up high execution times, with CPU requirements increasing exponentially with the size of the problem.

In this research we efficiently solve the *valued n-queens problem*, as a proof-of-concept example. For this problem we obtain good quality solutions while requiring several orders of magnitude less execution time than the most recent previous approach [120]. Moreover, because we apply conventional order-based crossover operators, our ap-

proach is easier to implement than others. Although this paper illustrates our approach with the *valued n -queens problem* we indicate how our techniques extend in a natural fashion to a more general CCOP framework.

7.2 Our *Baldwinian* Approach

The *Baldwinian* approach proposes to assess a genotype after the improvements from local search, but not to incorporate them into the genotype.

When feasibility is not an issue, the obvious candidate to guide the improvement is the objective function f [146]. However, for Constraint Optimization Problems we additionally have the penalty function. The most common approach utilizes the combination of the objective function and the penalty function [88]. Our thesis is that there are efficiency reasons to guide the improvement in stages.

Our *Baldwinian* approach operates in two stages. During the first stage, which we called *constructor*, we set up an almost feasible phenotype from the genotype. In the second stage, named here *learner*, we improve this phenotype using the objective function f . The purpose of the first stage is producing feasible solutions in an efficient and effective way. In the second stage (*learner*) we may apply local search in the landscape of the evaluation function. As prescribed by the *Baldwinian* approach, we assign the resulting fitness value to the original genotype without affecting its structure.

Figure 7.1 displays the stages that our proposal comprises.

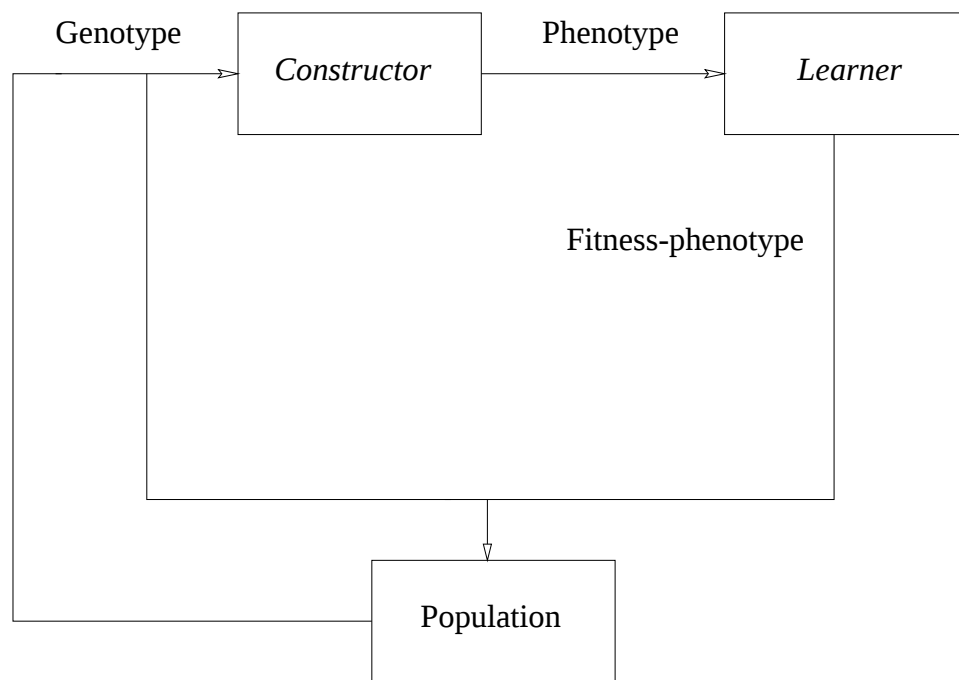


Figure 7.1: *Our Baldwinian approach comprises two stages: Constructor and Learner. The fitness-phenotype, returned by the Learner, is assigned to the original genotype.*

The rationale behind our proposal is to integrate an effective and efficient mechanism, that produces almost feasible solutions, into a general search mechanism (the GA) that is particularly effective in dealing with hard optimization problems.

A remarkably additional benefit of this approach is that it makes it possible to incorporate problem specific knowledge by the use of a different version of *constructor* and *learner*. Recall S stands for the search space and $F \subseteq S$ is the set of feasible solutions. We use the word *constructor* to emphasize the goal of this stage as moving a genotype $d \in S \setminus F$ into a genotype closer to F . The generic operator *constructor* applied to a genotype $d \in S \setminus F$ reduces the value of the penalty function ρ ; thus, $\rho(d) \geq \rho(\text{constructor}(d))$. The name *learner* is applied to the local search that improves the objective function, since in both *Baldwinian* and *Lamarckian*, it is assumed that it is the living phenotype who discovers and learns behaviors that make it fitter.

Our contribution is that, against previous applications of the *Baldwinian* approach, in the case of CCOPs, the *constructor* is crucial and perhaps more relevant than the *learner*, for two reasons. First, because without it, the search is costly. It consumes resources on evaluating infeasible solutions $d \in S \setminus F$ that are far from feasible solutions. The effort of the GA is really spent on trying to learn what is feasible, rather than on what is best among the feasible; remember that fitness evaluation is costly. Second, because it is easier to define the *constructor* operator than the *learner* operator. Recall (Section 2.4.3) that the contexts of CCOPs is such that it is hard to find an operator $\omega : F \rightarrow F$ (that is, $\omega(a) \in F, \forall a \in F$). Thus it is typically difficult to define the local search around the phenotype, and if an operator is found, it is because it involves a *constructor*, or it is again CPU-costly.

7.2.1 The Constructor Stage

The *constructor* : $S \rightarrow S$ then is an operator that reduces in feasibility with respect to ρ , a penalty function. Which penalty function ρ ? If the feasibility criteria ϕ is decomposable, then it is typically possible to consider partial assignments (partial solutions) [23, 104]. What does decomposable ϕ mean? For CCOPs, decomposable means we can separate constraints. More formally [23]:

Definition 14 (PS) *A Constraint Problem (CSP or CCOP) is decomposable provided that ϕ is given by a set of constraints $R = \{r_1, \dots, r_m\}$. A Partial Solution s_p only satisfies a subset of R .*

Thus, we can check separately the constraints involving variables and build a partial solution s_p . Namely, if all the constraints involving a variable are satisfied with its current value, we consider this variable assigned. However, if one of the constraints is violated we consider this variable to have its value not assigned. Then, we can count a ratio of constraints violated in an infeasible (total or partial) assignment, and the penalty function is any monotone function ρ on the number of variables without specified variables. Alternatively, the penalty function can be a monotone function on the number of constraints not satisfied. Note than, in the resulting partial solution s_p not all of the n variables are given values, only a fraction of them. Also constraints involving variables with assigned values are satisfied.

In this case, the *constructor* is a local search that increases the number of constraints satisfied, or increases the number of variables assigned a value and satisfying more

constraints. These are efficient procedures, for which we mention two examples next. The first example [58] is a mechanism that iteratively attempts to complete a partially defined permutation. We will refer to it as *iterative prefix extension* for future reference and we now explain the algorithm. Let $\langle \sigma_1, \sigma_2, \dots, \sigma_k \rangle$ be a partially defined assignment ($k < n$), that encodes that the value of variable i is σ_i for $i = 1, \dots, k$; but the value of variable i is not defined for $k < i \leq n$. It is partial in the sense that all constraints involving the first k variables are satisfied, and is a valid permutation in that $\{\sigma_1, \sigma_2, \dots, \sigma_k\} \subset \{1, \dots, n\}$ (and not two variables have been assigned a repeated value).

The iteration is initialized by looking at a permutation π (a genotype) and extracting the largest prefix that constitutes a partial assignment. The remaining genes of the genotype are kept in order in an initial *proposals list*.

The body of the iteration attempts to extend the current partial assignment by determining values for the $(k + 1)$ -th variable and beyond. The appended values must be different from all previous values and each must satisfy the constraints of the corresponding variable. A scan is performed of the *proposals list*, appending the next value in the *proposals list* to the partial assignment if the corresponding constraints are satisfied, or otherwise appending it in a *rejects list*. When the *proposals list* is exhausted, the *rejects list* replaces it, and the body of the iteration is repeated. An empty *rejects list* or an iteration that does not reduce the *proposals list* (does not increase the partial assignment) signals termination of the procedure. In the latter case for termination, the *rejects list* could be appended to the partial assignment to reconstruct a phenotype.

Clearly, this *constructor* complies with what we have required. This *constructor* will reduce the number of non-satisfied constraints. It typically is fast, since in the body of the iterations constraints are checked for one variable. Although it can potentially require $O(n^2)$ constraint checks, we show in the case study that: 1) it typically requires the same order ($O(n)$) of effort as evaluating the objective function once; 2) and more importantly even if this may be more costly than evaluating the objective function, the savings in reducing evaluation of infeasible solutions compensate the effort. This is an important point of our ideas. Our approach would be even more beneficial when the evaluation of the objective function is more costly. Despite the fact that the number of constraint checks performed by the *constructor* is potentially $O(n^2)$, it is worth the effort. We reduce the waste of resources on evaluating infeasible solutions. Our evolutionary search, while not restricted to the feasible region, is focused on it and its neighborhood.

Figure 7.2 displays this type of *constructor*.

Our second example of a *constructor* is based on using hill-climbing on the space of permutations with objective function the penalty function ρ . In generic terms, this is based on considering a family L of reversible transformations and the search space as a discrete graph. Two permutations are adjacent if there is a transformation in L that takes one into the other. Then, the current genotype is a node in a graph, and its neighborhood is computed by applying the transformations to it. The penalty function ρ is evaluated in the neighbors and the one that results in a smaller number of unsatisfied constraints is adopted as the new permutation. The hill-climber stops when no neighbor suggests an improvement.

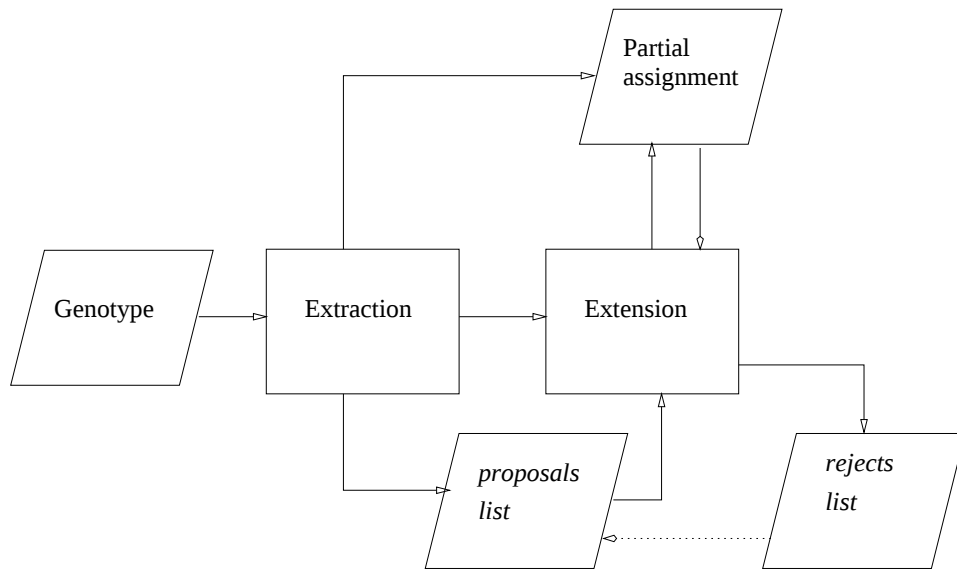


Figure 7.2: *The constructor stage extracts from the genotype the largest prefix that constitutes a partial assignment, the rest of the genotype is stored in a proposals list. The partial assignment is extended with the scanned value in the proposals list if it satisfies the constraints, otherwise the value is appended to the rejects list. At the end of this stage we have set up an almost feasible phenotype.*

A common example of a family of transformations is all pair-wise swaps. Unfortunately, this results in a quadratic number of neighbors for node in the search space, so we can add the specifics of a CCOP to make the *constructor* more efficient. For example, using the notion of a partial assignment described before, it should be clear that swapping the values of two variables that already satisfy their constraints is more likely to increase the value of ρ , so we may consider swapping only those variables whose values violate constraints. Using domain knowledge of the *valued n-queens problem*, this is the type of *constructor* proposed in [126]. We refer to this mechanism as *restricted swap*.

Note that the *constructor* is not concerned about optimality of the objective function; its main interest is in producing feasible phenotypes instead.

7.2.2 The Learner Stage

It is only after feasible phenotypes have been encouraged that we make use of the objective function. The *learner* applies an evaluation function ψ to obtain a fitness value and uses local search in the landscape of ψ . For CCOPs, we think that this may be rather costly.

Usually, for unconstraint problems, the evaluation function and the optimization function are the same. However, our evaluation function is a combination of a penalty function and the objective function. This is simply because the *constructor* may not have reduced ρ all the way down to zero. Also, since the objective function may have been defined only on the feasible space F and not on S , some adjustments may be

necessary. Different ways of combining ρ with f have been proposed [88] and several taxonomies of ρ have been suggested [22, 88]. Unfortunately, the factors [88] that influence the design of ψ are problem dependent. In Section 7.3.2 we explain in more detail the evaluation function applied to our particular case study.

7.3 A case study: the *valued n-queens problem*

The *n-queens problem* consists of placing n queens on n different squares on a chess board with n rows and n columns, satisfying the constraint that no two queens can threaten each other. This problem has been studied for more than a century and there are many proposals to solve it [24]. It is well known that for this problem there are many solutions when $n > 3$ (discounting the trivial case of one solution when $n = 1$). However, no closed-form formula has been found to compute the total number of solutions for the general case (experimental results show that the number of solutions grows exponentially as the size of the problem increases) [24]. Moreover, this problem is a kind of the maximal coverage problem and it has many practical scientific and engineering applications. Amongst them are VLSI testing, communication systems, computer task scheduling, computer resource management, optical parallel processing, and data compression [126]. The *n-queens problem* is easily solved (in expected polynomial time) by repairing methods [126]), when just a solution is required. GAs are too inefficient for this type of problem [58]. However, if our purpose is to obtain all solutions, a systematic search must be applied at the cost of exponential time.

25	16	20	18	1	27	23	21
16	9	15	27	5	12	1	7
20	15	16	31	30	18	23	26
18	27	31	5	16	29	10	12
1	5	30	16	6	30	25	23
27	12	18	29	30	9	22	26
23	1	23	10	25	22	26	30
21	7	26	12	23	26	30	0

$n=8$

optimum value=190

Figure 7.3: *An example of a valued 8-queens problem and its optimum value.*

Former results for the *n-queens problem*, using GAs, apply mutation as the predominant operator [15,57], or *ad hoc* asexual operators [21]. However, they do not outperform local-search mechanisms [21,54,57] and demand a prohibitive computational effort [15,54].

The *valued n-queens problem* is an extension of the *n-queens problem*. In this case each square of the chess board is weighted. The purpose is to find a solution to the *n-queens problem* that maximizes the sum of the weighted squares occupied by the queens. If the global optimum is required, this problem must be solved by

a systematic search, provided that the problem is small. Otherwise, with large problems, it can become extremely difficult to find a solution. Figure 7.3 shows an example of a *valued 8-queens problem* and its optimum value.

For the *valued n -queens problem* specialized representation and genetic operators, based on constraint-solving techniques, demand a high execution time, that shows rapid growth when the size of the problem increases [120]. Moreover, they do not guarantee to produce only feasible solutions (the output of their operators may contain infeasible assignments, even if the input is feasible) [104, 120]. Also, these specialized genetic operators could affect properties necessary for genetic operators (like assortment [107]).

In this section we apply our proposed *Baldwinian* approach to deal with these problems.

7.3.1 Genotype Representation

Letting $q_i \in \{1, \dots, n\}$ denote the column position of the queen in the i^{th} row we ensure that the genotype $\langle q_1, \dots, q_n \rangle$ has no two queens on the same row. If all the q_i are distinct integers in $\{1, \dots, n\}$ then no two queens share a column [126]. Thus, this representation, now popular [15, 21, 54, 57], ensures that all row and column constraints are satisfied but diagonal conflicts may exist. This order-based representation means that the GA search space (with size $n!$) consists of integer permutations (some of which are not feasible).

7.3.2 Fitness Evaluation

Before we proceed to describe the mechanism of obtaining the fitness value, it is convenient to mention how we deal with diagonal conflicts. Diagonal conflicts are used to compute penalty functions. To calculate the number of diagonal conflicts we apply the method proposed in [126]. This method allows the identification of diagonal collisions in $O(n)$ time by the construction of a pair of arrays, the positive and the negative array. These arrays have size $2n - 1$. Each time a queen is placed on row i and column q_i , a counter is incremented on the $n + i - q_i$ position of the negative array to account for this diagonal of negative slope. Also on the $i + q_i - 1$ position of the positive array. Figure 7.4 displays the arrays (positive and negative) after a queen has been placed on a chess board with size 4×4 . A non-conflictive placement of the n queens is achieved if the counter on each position of each array is 0 or 1.

During our genetic search, a genotype is evaluated by the *Baldwinian* approach proposed in the previous section. The *constructor* is a combination of the local search mechanisms for finding a feasible solution described in Section 7.2.1. The particular *constructor* we have designed for the *valued n -queens problem* corrects almost feasible solutions for the *n -queens problem*. It first applies the *iterative prefix extension* to a partial solution, and if it does not succeed it then continues with the *restricted swap* also described in that section. It should be mentioned that experimental evidence suggests that the *iterative prefix extension* produces solutions for the *n -queens problem* with a very low number of diagonal conflicts regardless of the problem size [58]. We emphasize that this is a very efficient local search since after placing a queen in lo-

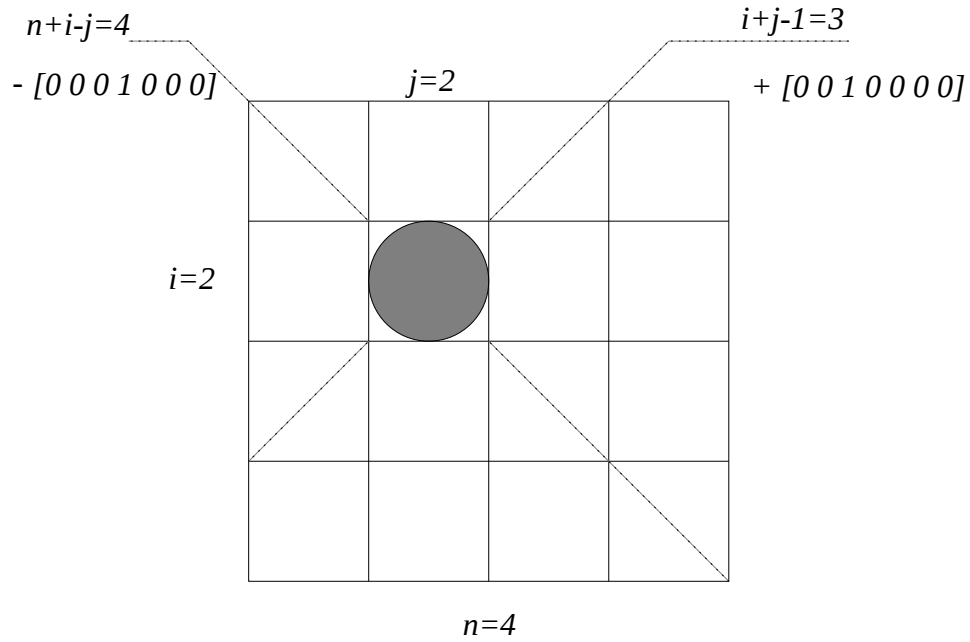


Figure 7.4: *Positive and negative arrays after a queen has been placed on a chess board (row 2, column 2) with size 4×4 .*

cation (i, q_i) , the number of diagonal conflicts (dc) on each array is updated easily. Consequently it is easy to test if the new placement has enlarged the current partial solution for both the *iterative prefix extension* and the *restricted swap* search for a feasible solution. At the end of this *constructor*, with a serialization of two local searches for feasibility, we obtain a feasible solution for the *n-queens problem* with high probability. While we have no mathematical proof, our experiments show that the number of times the composed *constructor* we have described fails to produce a feasible solution is low and decreases rapidly as the size of the problem grows. This is not surprising since we are using the two fastest and most effective strategies known to find feasible solutions to the *n-queens problem*.

As *learner*, we use *SORT* guided by *Insertion Sort*.

We finally arrive at the evaluation of the fitness function. Our implementation computes the fitness of phenotypes as follows. A chess board can be represented as a matrix M with $n \times n$ cells (squares) and the weights w associated with each board location as $w(i, j)$, where $1 \leq i, j \leq n$. The permutation of values (the location of queens) is given by $\pi = \langle q_1, \dots, q_i, \dots, q_n \rangle$. The function to optimize is defined by $f(\pi) = \sum_{i=1}^n w(i, q_i)$. Nevertheless, the possibility exists for an infeasible solution to reach this stage. In this case f only takes valid location of queens into account and a slightly different penalty function is applied. While during the *constructor* stage the penalty function was the length of the largest prefix that causes no constraint conflict, the penalty function we designed for this stage is based on computing the effort r of repairing the genotype [88]. We increase by one the value of r each time the *iterative prefix extension* mechanism adds an element to the *rejects list* and each time a swap is performed by the *restricted swap* mechanism. This penalty function is not applied to feasible solutions. We obtain, $fitness = \frac{f(\pi)}{r}$.

This fitness value is assigned to the genotype that was the input to the *constructor* without applying the structural changes which may have been suggested by the *constructor* itself.

7.3.3 Genetic Operators

Our proposal, named here *Baldwin-GA*, uses the operators and parameters reported in Table 7.1. Because *PMX*, Roulette Wheel Selection and Linear Scaling [44] have

been described in Chapter 2 we will skip details here. Hybridization with a hill-climber at the level of the fitness landscape is achieved by the use of *SORT* guided by *Insertion Sort* [31], which has been described in Section 6.6.2.

<i>Parameter</i>	<i>Baldwin-GA</i>
Representation Scheme	Integer Permutation
Population Size	20
Number of Generations	100
Selection	Roulette Wheel
Scaling	Linear Scaling
Crossover	PMX
Mutation	<i>SORT</i>
Crossover Probability	0.8
Mutation Probability	0.2
Elitism	<i>true</i>

Table 7.1: Baldwin-GA *parameters*.

7.3.4 Experimental Results

The benchmark of Ruiz-Andino *et al.* assigns integer values (weights) in the interval $[0, 31]$ to each cell $m_{i,j}$ of the matrix M , independently and uniformly (each integer entry $m_{i,j}$ is chosen with uniform probability $1/32$ and independently of other entries). In the construction of our test cases we apply this method. We compare *Baldwin-GA* with a branch and bound (*B&B*) algorithm (as Ruiz-Andino *et al.* do)

in terms of cost and quality of solutions. Implementation of *Baldwin-GA* is based on the software *GAlib* genetic algorithm package written by M. Wall at the Massachusetts Institute of Technology. The implementation of the *B&B* algorithm is totally our own.

We first show results in the trade-off of quality versus resource requirements (most importantly, CPU-time). We next show results that display the high success rate of the *constructor* stage in finding feasible solutions. This is reported as the proportion of feasible solutions produced by *constructor* over the number of its invocations. Last, we will report the benefits obtained when using *SORT* mutation.

Execution Time and Quality of Solutions

To compare the quality of our solutions with the optimal values obtained from methods that guarantee optimality, we contrasted them with the well-known *B&B* method. It is the usual optimization method applied by constraint programming languages [120]. Because this method is exponential, we only report results with size n in the integer interval $[4, \dots, 20]$. Quality is measured as a competitive ratio. We obtain an optimal value $Q_{B\&B}$ applying *B&B*, as well as the best value Q_{BGA} (after ten runs) applying *Baldwin-GA*. Then, we compute $Quality = 1 - \frac{Q_{B\&B} - Q_{BGA}}{Q_{B\&B}}$. This is the competitiveness ratio of the multi-start *Baldwin-GA*. The value of the competitive ratio is in the real interval $[0, 1]$ (larger values means better quality). Table 7.2 displays quality values obtained for each board size. The x -axis represents the problem size and the y -axis the quality obtained with *Baldwin-GA*. The lowest quality value (0.859) occurs when $n = 19$. We conclude that our *Baldwin-GA* produces good quality solutions along these sizes.

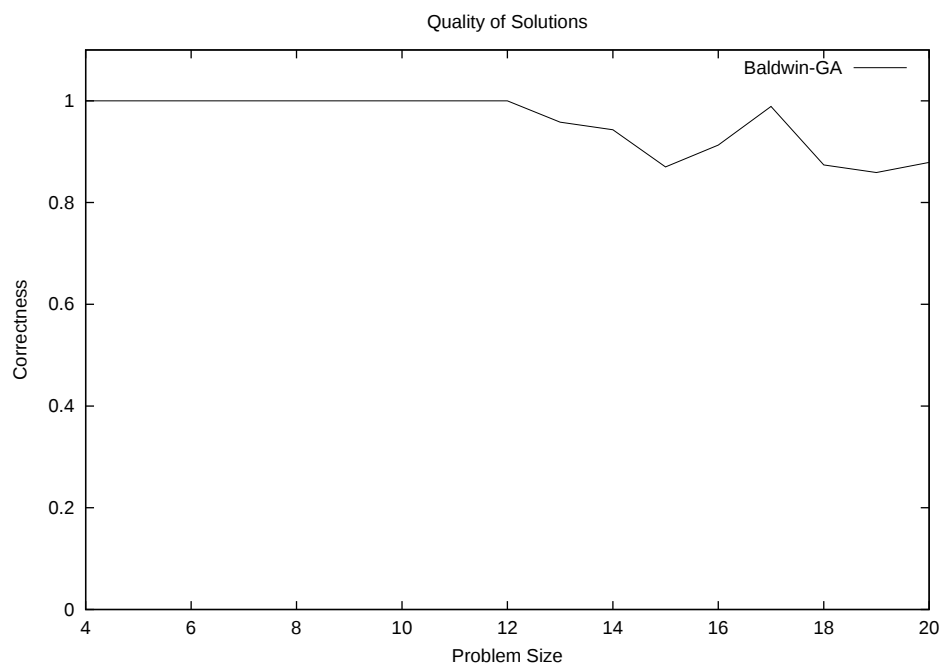


Figure 7.5: Baldwin-GA *produces good quality solutions for problem sizes between 4 and 20.*

Table 7.2 also displays the average time over ten runs for *Baldwin-GA*. The contrast with the execution time required by the *B&B* algorithm is dramatic. Even if we allow 10 times the average time to account for the multi-start, the *Baldwin-GA* is about 10 thousand times faster by the time the problem size is $n = 20$. In particular, the cost of our algorithm is less than one second for the tested problem sizes.

Proportion of Feasible Solutions

To assess the quality of the composed *constructor* we ran our algorithm three times for each size $n \in \{50, 100, 150, \dots, 500\}$. Each time, we recorded the number ph_f of feasible phenotypes and the number ph_u of infeasible phenotypes produced by the invocations to the *constructor*. We computed the proportion $\frac{ph_f}{ph_f + ph_u}$. Figure 7.6 displays the proportion of feasible phenotypes produced by *Baldwin-GA* for tested problem sizes. The x -axis represents the problem size n and the y -axis the proportion of feasible phenotypes when a *Baldwin-GA* is applied. While for $n = 50$ the proportion of feasible solutions is 0.22, starting with $n \geq 200$, this proportion increases consistently and it almost reaches 1 by the time the problem size is 500. These results show that our *constructor* is effective in producing feasible solutions.

Benefits of *SORT* Mutation

We present results for four different sizes: 16, 32, 48 and 64. Ten runs were carried out for each size, using different random seeds. We compare two versions of our *Baldwin-GA*. The first one uses *SORT* mutation [31], the second swaps two ran-

n	<i>B&B</i> : Time in seconds	<i>Baldwin-GA</i> : Time in seconds	<i>Baldwin-GA</i> : Quality of solutions
4	0	0.024	1
5	0	0.029	1
6	0	0.041	1
7	0	0.048	1
8	0	0.054	1
9	0.01	0.067	1
10	0.03	0.082	1
11	0.25	0.097	1
12	0.53	0.112	1
13	1.93	0.137	0.958
14	2.47	0.159	0.943
15	17.63	0.172	0.870
16	56.9	0.195	0.913
17	84.37	0.215	0.989
18	1392.57	0.245	0.874
19	7073.9	0.261	0.859
20	21117.9	0.29	0.879

Table 7.2: Comparison of a *B&B* algorithm with a Baldwin-GA.

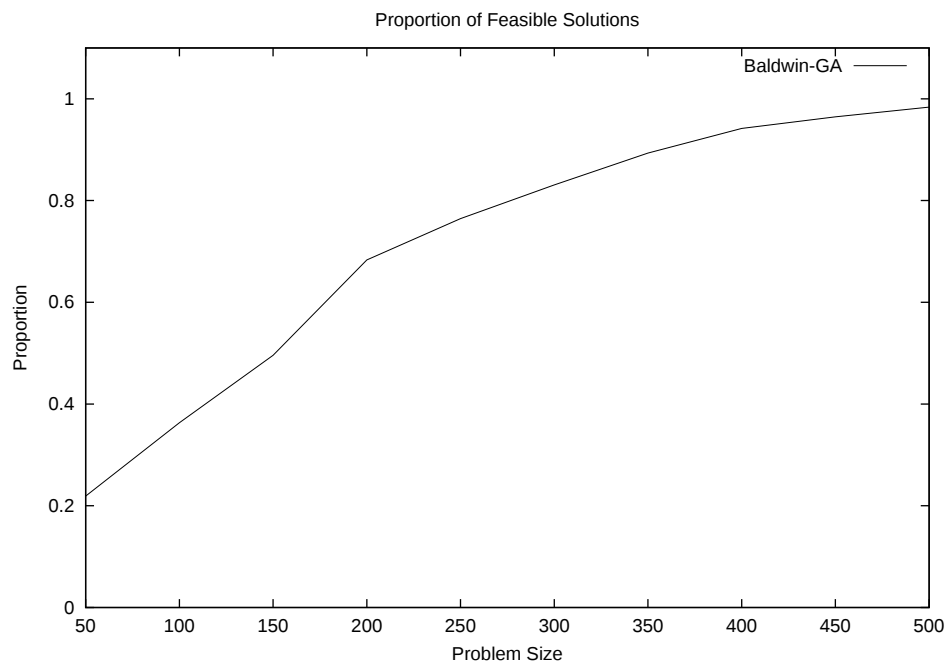


Figure 7.6: *The proportion of feasible solutions produced by a Baldwin-GA increases as the problem size grows.*

domly selected variable assignments. The swap is carried out with a given mutation probability (defined as a parameter of the GA). This is a simple randomized swap mutation.

A plot of the results is displayed in Figure 7.7. The x -axis is for the problem size. The y -axis represents the fitness value (computed as described in Section 7.3.2). The plotted lines interpolate the average fitness values over the 10 runs. The dotted line corresponds to results for *Baldwin-GA* without *SORT* mutation (labeled in the figure as *Baldwin-GA-No-SORT*), while the solid line corresponds to *Baldwin-GA* instantiated with *SORT* mutation (labeled as *Baldwin-GA-SORT*). This figure illustrates that a *Baldwin-GA* with *SORT* mutation performs much better. The improvement is statistically significant across the entire range tested of problem sizes (because we calculated confidence intervals shown in Figure 7.7). We conclude that *SORT* mutation is consistently better.

7.4 Discussion

An important point derived from this work is the remarkable performance obtained when a Genetic Algorithm based on the Baldwin effect is applied to the *valued n -queens problem*, a typical representative of Constraint Combinatorial Optimization Problems.

Secondly, our approach facilitates the inclusion of problem-specific knowledge by the use of different versions of *constructor*, and *learners*; while it benefits from the use of a general and well-known framework (the *order-based GAs*).

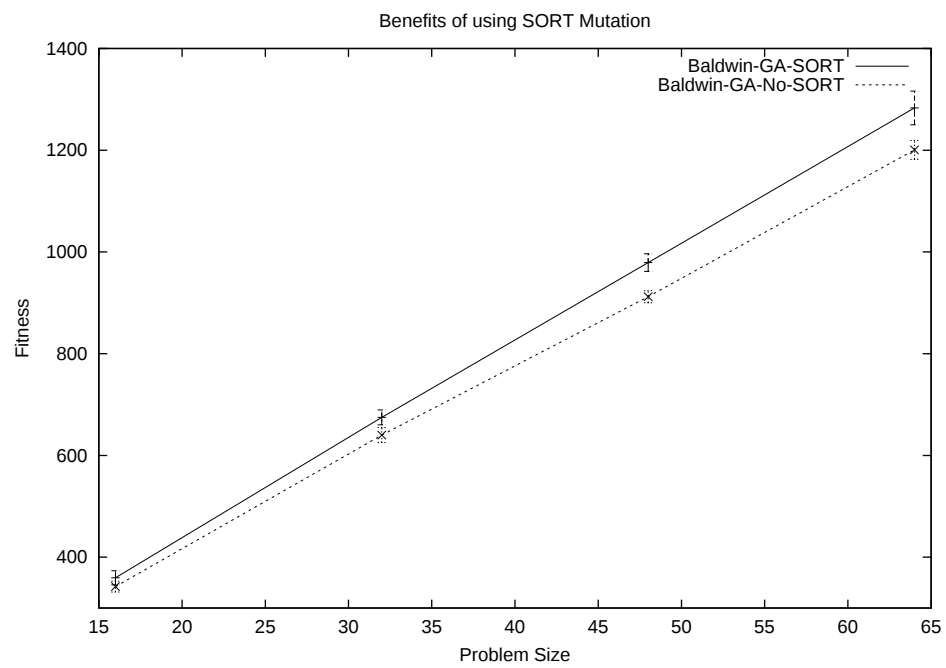


Figure 7.7: SORT *mutation provides benefits statistically significant.*

Our approach is neither a purely local search nor a simple mapping from genotype to phenotype. It includes a deterministic stage (the *constructor*) that does not perform backtracking.

Even though we do not guarantee to create the whole phenotype space, its high success in producing feasible phenotypes keeps the problem of infeasibility under control. Actually, as a result of our evolutionary search, for the tested problem sizes, we have always been able to obtain a feasible solution: a key issue that Ruiz-Andino *et al.* do not report.

We conjecture that one of the reasons for the better performance is the effectiveness of the *constructor* in producing feasible solutions. Another important reason is the sole use of functions based on feasibility criteria to guide the *constructor*. One of the difficulties in relying only on $\psi(\rho, f)$ for improvement of individuals is its high dependency on the particular instance of a problem. For example, it is conceivable that we could deal with an instance of the *valued 4-queens problem* as depicted in Figure 7.8. In the example, the two feasible solutions are highlighted with rectangles. Observe that all feasible solutions are not better than all infeasible solutions. This is an important reason to consider the *constructor* as of great significance.

Finally, in the literature there has been interest also in partial solutions s_p , mainly because satisfying all the constraints may be impossible. That is, the Constraint Problem has no solution (for example the *valued 3-queens problem*). Our methods have a natural extension to this type of problem.

20	0	0	14
0	17	6	0
0	8	19	0
23	0	0	3

$n=4$

Figure 7.8: *An instance of the valued 4-queens problem where all feasible solutions are not better than all infeasible solutions.*

Chapter 8

Conclusion

Section 8.1 will summarize and substantiate our contributions while Section 8.2 will provide some pointers and new avenues that our research opens for follow-up research.

8.1 A Summary of the Main Conclusions

Clustering a matrix can be reduced to the <i>Longest Hamiltonian Path Problem</i> .

In the context of clustering visitations paths of web-users, we have illustrated that Local Search (LS) is a powerful model to cope with hard combinatorial optimization problems. We have dealt with the problem of clustering a matrix because it is a crucial step in clustering visitation paths. In order to cluster a matrix, McCormick *et al.* [81] propose the Bond Energy Algorithm (*BEA*) which optimizes a measure of

effectiveness $e(A)$. We have shown that clustering a matrix, using such a measure of effectiveness, can be reduced to solving the *Longest Hamiltonian Path Problem*.

Neighborhood structure in LS for clustering a matrix is crucial.

The heuristic usually referred to as *BEA* has been implemented in different ways by several authors [81, 102, 149]. Those implementations generate and exploit different neighborhood structures. The results obtained, in terms of quality of solutions and computational effort, are dissimilar; they depend on the particular implementation. In our experiments the variant proposed by McCormick *et al.* [81] (*BEAm*) provides the best quality of solutions. The variant proposed by Özsu and Valduriez [102] (*BEAo*) is the fastest but provides solutions of less quality than those provided by *BEAm*.

For the problem of clustering a matrix, the algorithm 2-opt provides better solutions than three variants of *BEA*.

Because we have shown that the problem of clustering a matrix can be reduced to the *Longest Hamiltonian Path Problem*, we have compared the performance of the variants of *BEA* with methods suitable for the *Longest Hamiltonian Path Problem*, in particular with the 2-opt algorithm proposed by Croes [16]. Our experiments have shown that the algorithm 2-opt, usually applied in the solution of the Traveling Salesman Problem (TSP), is superior, in quality of solutions, to all tested variants of *BEA*. This result is remarkable given that *BEA* has been historically applied in clustering a matrix using the measure of effectiveness $e(A)$. However, 2-opt is more expensive, in computational effort, than both *BEAm* and *BEAo*. Thus, an

opportunity for a better trade-off between quality of clustering and CPU time was discovered.

We developed a better alternative algorithm for clustering a matrix.

We have proposed to cluster a matrix by a two-step procedure. In the first step, a starting solution is obtained by a constructive heuristic. The second step improves this solution by a local-search mechanism. In particular, we have implemented three variants of *BEA* to build the starting solution, then the *2-opt* algorithm improves this preliminary solution. The application of the three variants of *BEA* to build the starting solution produces dissimilar results in performance. The sequential application of the variant proposed by Özsu and Valduriez [102] (*BEAo*) and the *2-opt* algorithm provides the best trade-off between quality and computational effort amongst the tested algorithms. This combination provides better results than those obtained by the same algorithms when they are executed separately. Thus, our approach to cluster a matrix is better than both *2-opt* and *BEA*.

A good place for hybridization of GA and Local Search is as a mutation operator.

Our research has provided empirical evidence, which is consistent with previous results [6, 13, 19, 30, 49, 115, 133, 145, 150], that shows that the application of local search in the mutation operator will provide good results. In particular, for the *p*-median problem, we have proposed a memetic algorithm that incorporates exploitation characteristics of a hill-climber into the mutation operator. In this case, our mutation operator is a heuristic, which we call TAB, proposed in 1968 by Teitz and Bart [129]. Our hybridization is a tight integration that preserve those aspects

of TAB that make it the most effective hill-climber for the the p -median problem. Thus, a non-representative u_i that fails to become a representative is banned until all other non-representatives have also been attempted. Our mutation operator remembers the index i where the last promotion of a u_i to a representative took place. By adjusting the mutation rate (and the population size) the MA can be configured to resemble TAB or to be more independent (TAB is the case population size 1 and mutation rate 1).

We developed a better alternative algorithm for the p -median problem.

For the p -median problem as a case study, we have presented results that show the benefits of our memetic algorithm approach. Our proposal applies with all crossover operators but it demands delicate balancing of the impact of hill-climbing so that the MA avoids early convergence. As a result, we have shown that our proposed MA optimization is robust and effective and balances effort/quality of solution better than a plain GA.

For the *valued n -queens problem*, a Constraint Combinatorial Optimization Problem, hybridization can be achieved effectively in a *Baldwinian* mode.

Our research has utilized [32] the *Baldwinian* approach to effectively deal with feasibility on the *valued n -queens problem*, which belongs to the class of Constraint Combinatorial Optimization Problems (CCOPs) that admit formulation as permutation problems [124]. Our *Baldwinian* approach operates in two stages. During the first stage, which we have called *constructor*, we set up an almost feasible phenotype from the genotype. In the second stage, named here *learner*, we improve this phe-

notype using the objective function f . Our approach is neither a purely local search nor a simple mapping from genotype to phenotype; it includes a deterministic stage (the *constructor*) that does not perform backtracking. Our approach facilitates the inclusion of problem-specific knowledge by the use of different versions of *constructor*, and *learners*; while it benefits from the use of a general and well-known framework (the *order-based GAs*).

The *constructor* is more relevant than the *learner*.

In this respect our contribution is that, against previous applications of the *Baldwinian* approach, in the case of CCOPs, the *constructor* is crucial and perhaps more relevant than the *learner*, for two reasons. First, because without it, the search is costly. It consumes resources on evaluating infeasible solutions that are far from the feasible region F . The effort of the GA is really spent on trying to learn what is feasible, rather than on what is best among the feasible; remember that fitness evaluation is costly. Second, because it is easier to define the *constructor* operator than the *learner* operator. In the contexts of CCOPs it is hard to find an operator $\omega : F \rightarrow F$ (that is, $\omega(a) \in F, \forall a \in F$). Thus, it is typically difficult to define the local search around the phenotype, and if an operator is found, it is because it involves a *constructor*, or it is again CPU-costly. We reduce the waste of resources on evaluating infeasible solutions. Our evolutionary search, while not restricted to the feasible region, is focused on it and its neighborhood. Moreover, it is conceivable that we could deal with an instance of a problem where no feasible solution is better than any infeasible solution. This is an important reason to consider the *constructor* as of great significance. Finally, in the literature there has been interest also in partial

solutions s_p , mainly because satisfying all the constraints may be impossible. That is, the Constraint Problem has no solution. Our methods have a natural extension to this type of problem.

We developed a better alternative algorithm for the *valued n -queens problem*.

For a set of experiments of the *valued n -queens problem*, a typical CCOP, our approach proves to be superior to the most recent proposal [120]. Our experiments have shown that we successfully produce feasible phenotypes, so the problem of infeasibility is kept under control. Moreover, for the tested problem sizes, we have always been able to obtain a feasible solution: a key issue that Ruiz-Andino *et al.* [120] do not report. We conjecture that the reasons for a better performance are: the effectiveness of the *constructor* in producing feasible solutions; and the sole use of functions based on feasibility criteria to guide the *constructor*.

Classical sorting algorithms provide a neighborhood structure and a Local Search for permutation problems.

During our research we have shown that sorting algorithms [69] can be conceptualized as LS mechanisms which optimize a measure of presortedness. Thus, in our approach [31, 32, 131, 132], which we call *MA-sorting*, classical sorting works as a map for the search space S_n of $n!$ permutations. We propose to guide a memetic algorithm with a new family of mutation operators. Our mutation operators perform local search following the path traced by classical sorting mechanisms. Our mutation operator does not randomly swap items of a chromosome encoding a solution. Our mutation actually requests from the sorting algorithm which permutation to construct from the current permutation. Thus, the neighborhood operator ν sug-

gested by a sorting algorithm is applied to mutate the current permutation π_i into π_j . In our approach the comparison predicate and the evaluation function are made to correspond and guide the evolutionary search.

We have identified five attributes of sorting algorithms which influence the neighborhood structure that is used for a LS: *Swap constraint*, *Record of the state*, *Focus*, *Dependency on input* and *Presortedness measure*. In particular, they control what is to be understood as the neighbor operator ν in the LS part. *Swap constraint* is relevant because it determines the neighborhood structure. That is, the most common operation performed by sorting algorithms when the comparison predicate returns *False* is the swap of the elements involved in the comparison: the swap determines the number of neighbors to a current solution and establishes a possible new current solution. Depending on the sorting algorithm (which fixes the type of comparison, and consequently the type of swap) the swap operation may restrict or expand the neighborhood structure. The *Record of the state* is important because our definition of *state* serves to identify where the schedule of comparisons can be restarted. *Focus* is a relevant characteristic of a sorting method because it express the number of times a particular element is repeatedly involved in the comparison predicate, so it influences the schedule structure of the neighborhood and its orientation in the search space. *Dependency on input* accounts for the fact that either the sorting mechanism predefines the order of comparisons, independently of the value returned by the comparison predicate, or the order of comparisons depends on the previous result of the comparison predicate. Finally, a specific *Presortedness measure* may be more appropriate than others for the sorting algorithm at hand, thus we may distinguish sorting methods depending on the suitability of the measure of presortedness.

With respect to *Presortedness measure* we have demonstrated that *Eng*, a measure of presortedness, does not increase its value when a swap is performed after a comparison predicate returns *False*. Thus, *Selection Sort*, *Quicksort* and *Insertion Sort* are well behaved as local searchers for the *Eng* measure of presortedness. This presortedness measure is based on the total difference between the position occupied by the element x_i after sorting the sequence X (that we call the $rank(x_i)$), and the current value of i . Formally, $Eng = \sum_i^n |rank(x_i) - i|$. When the sequence X is ordered, $Eng = 0$, the minimum.

Memetic algorithms using classical sorting as Local Search are effective.

We have shown that for the Error Correcting Graph Isomorphism and the *valued n-queens problem*, the combination of Memetic Algorithms with classical sorting improves permutation search [31, 32, 131, 132]. For these case studies our *MA-sorting* achieves a better quality versus effort trade-off.

Hybridization with classical sorting has many flavors but the combination of:

- *Single-threading*,
- *Deterministic* and
- *Best improvement*,

is generally a robust and effective starting point.

Our research has provided useful information for the design of our *MA-sorting*. In particular, we have shown that the performance of our *MA-sorting* is influenced by three of the components of the LS embedded in it: neighborhood operator, number

of neighbors explored, and schedule of exploration. With respect to the first component, we have found that the major contribution to the performance of *MA-sorting*, comes from the quality of the neighborhood structure generated by the neighborhood operator. For the ECGI problem, the neighborhood structure implicitly generated by *Selection Sort* is the best, and the neighborhood structure implicitly generated by *Quicksort* is better than that generated by *Insertion Sort*. In particular, for *Selection Sort* and *Quicksort*, the spread of *Focus* is more even and there is a locality in the swaps performed by *Selection Sort* that is not distorted by other genetic operators. This explains the fruitful nature of the neighborhood proposed by *Selection Sort*.

Regarding the second component, we have explained that while the inner cycle (in our model of LS) regulates the number of explorations in the local neighborhood, the outer cycle regulates the length of the path traveled in the search space. For the ECGI problem we performed experiments regulating the inner cycle by using four modes of operation: *Minimal progress*, *First improvement*, *Maximal progress* and *Best improvement*. Our experiments have shown that for this problem the *Best improvement* mode of operation provides the best quality of solutions. In our case study, the benefits obtained by increasing the number of explorations regulated by the outer cycle are more important for *Selection Sort* and *Quicksort* than for *Insertion Sort*.

For the third component we have made experiments with three modes of operation that have an impact on the order of nominating permutations: *Multi-threading-Deterministic*, *Single-threading-Deterministic*, and *Single-threading-Random*. The difference between *Multi-threading* and *Single-threading* is whether or not the infor-

mation about the state (that determines the continuation of the schedule of comparisons) is shared globally across chromosomes. For the ECGI problem we have found that the *Single-threading* method (random and deterministic) provides the best results to generate the state of the sorting algorithm. Moreover, we have found that the influence of the method for generating such state on the performance of *MA-sorting* is more evident when exploring a fruitful neighborhood structure and performing more explorations.

Other improvements are also reflected in *MA-sorting*.

Finally, we have found that two techniques which are commonly applied to improve the performance of local-search algorithms (increasing the number of approximate solutions explored in the search space, and using a “good” approximate solution to start the exploration of the search space), are also beneficial for our *MA-sorting*.

MA-sorting is applicable to permutation problems.

Our *MA-sorting* is specific to problems involving permutations, but in any other sense it is domain-independent. Therefore, it is applicable to the Error-Correcting Graph Isomorphism and the *valued n-queens problem*, but also to other problems involving permutations. *MA-sorting* is generic in the sense that it can incorporate other classical sorting algorithms besides those reported in our research. Thus, one can actually expect that the hybridization would work with Mergesort, Heapsort and so on.

It is important to analyze to what extent our algorithms provide a general black-box that does not make assumptions about the landscape of the search space or about the nature of the problem, beside the fact that it involves permutations.

Specially, the No Free Lunch (NFL) Theorems [147] imply that a good algorithm would pay the price by being bad in many of the other optimization problems (those problems that have search space S_n and as objective function some function $f : S_n \rightarrow \mathfrak{R}$). First, one has to review the assumptions of the NFL theorems. Searching algorithms for NFL have no cost for remembering where they have been before, an issue that certainly implies a space-time trade-off in any practical algorithm. Also, there are some restrictions over the *a priori* distribution of all optimization problems (the space of all functions $f : S_n \rightarrow \mathfrak{R}$). Otherwise, exhaustive search (which meets the requirements of an algorithm that does not repeat function evaluations) would be a widely used algorithm in practice for optimization problems. Notice that exhaustive search is a variant of LS; simply let $\nu(i) = S_n$ and produce a list of all permutations in $O(1)$ amortized time (by classical algorithms like generating all permutations lexicographically [112]). Exhaustive search would perform better maximization than other algorithms in functions like $High_K(\pi) = 0$ for all $\pi \in S_n \setminus \{\pi_K\}$ and $High_K(\pi_K) = 1$ where π_K is a permutation with high Kolmogorov Complexity (assuming n is large enough, such permutations exist [73]). But certainly, the optimization problems on permutations we expect to arise in practice would have their objective function encoded in a (short) program; thus, these problems could not be like optimizing $High_K$. So the distribution of optimization problems we expect our algorithms to face has a very marked non-flat distribution.

Moreover, our *MA-sorting* does assume something about the nature of the optimization problem on permutations. It assumes regularity of the neighborhood structure with respect to both a measure of presortedness and a neighborhood operator. Problems should be such that a permutation that is very similar in sorted order with re-

spect to the optimal solution has also a good value for the objective function. Thus, it makes sense for the optimizer to learn heuristically: that is, record information “ k before l ” suggested by the sorting algorithm. Knowing which measure of presortedness is suitable for an optimization problem would go a long way in suggesting the sorting algorithms for the hybridization.

In *MA-sorting*, distance and operators are highly coupled.

In the context of fitness landscape (which is given by a triplet (S, f, d) , where S is the set of approximate solutions, f is an objective function, and d is a measure of distance), Jones and Forrest [63] point out a crucial issue: the analysis of fitness landscape would be more accurate if it were based on the distance between points in the space of states according to the operator in use by the algorithm. In contrast with previous proposals [85, 86, 152] in our *MA-sorting* the local-search operators and the measures of distance d are highly coupled: the measures of presortedness correspond to the concept of distance d and sorting algorithms are the operators which transform one approximate solution into another.

We developed a better alternative algorithm for ECGI.

We have shown that *Eng* is a valid measure of presortedness for *Insertion Sort*, *Quicksort* and *Selection Sort*. We have also shown that, for our case study, *Selection Sort* improves *Eng* more effectively than *Quicksort* and *Insertion Sort*. Moreover, we have observed that *Eng* is compatible with the objective function for the ECGI problem. Thus, for the case study, a permutation that is a close order to the optimal permutation is a reasonable match between the two given graphs to reduce the error in the match.

We claim that there are two important reasons which explain the outstanding performance of *MA-sorting* with *Selection Sort* for the ECGI problem. First, there is compatibility between optimizing the objective function (which guides the evolutionary search) and optimizing the measure of presortedness (which guides the sorting algorithm). Second, it is the use of an effective sorting algorithm, *Selection Sort*, to optimize the measure of presortedness *Eng*.

We developed a methodology for the design of our *MA-sorting*.

The design of our *MA-sorting* involves three processes. First, we identify a measure of disorder, and algorithms that minimize this measure effectively. Second, we establish that the optimization of the measure of disorder is in close association with optimizing the objective function. The third process merges the heuristic to optimize the measure of disorder as a local search into a Memetic Algorithm. We have emphasized that classical sorting algorithms optimize measures of disorder. We have identified three sorting algorithms that monotonically optimize a new measure of disorder *Eng*. We have also shown that the measure of disorder *Eng* has close association to our case studies. Thus, the effectiveness of *MA-sorting*.

8.2 A Window to the Future

This connection to classical sorting opens new avenues that deserve research, amongst them:

The application of *MA-sorting* to other problems involving permutations for which the corresponding disorder measure and sorting algorithm are identified.

So far we have successfully applied our *MA-sorting* to the ECGI problem and to the *valued n -queens problem*. However, more experiments are required to confirm that our approach can be successfully extended to other combinatorial problems that involve permutations.

Landscape analysis has been found useful to characterize the difficulty of problems [63, 85, 86, 152]. We believe our approach can be helpful in improving landscape analysis.

In particular, we consider that several measures of disorder can be successfully applied to characterize the difficulty of problems. A practical benefit of this characterization is that we can dynamically tune (because the landscape of the search space changes dynamically along the evolutionary search) our *MA-sorting* by choosing an appropriate LS operator (based on sorting algorithms) for the current landscape of the search space.

Other algorithmic approaches, mainly those amenable to being modeled as searching in a space of states, can also be benefited from the process of design we have followed for *MA-sorting*.

In particular, we believe our techniques can be directly applied to the design of plain LS algorithms by the construction of neighborhood operators (based on sorting algorithms) that optimize a suitable measure of disorder.

Bibliography

- [1] E. Aarts and J. K. Lenstra. Introduction. In E. Aarts and J. K. Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 1–17. John Wiley & Sons, 1997.
- [2] E. Aarts and J. K. Lenstra, editors. *Local Search in Combinatorial Optimization*. John Wiley & Sons, 1997.
- [3] T. Arita and R. Suzuki. Interactions between learning and evolution: The outstanding strategy generated by the Baldwin effect. In *Proceedings of Artificial Life VII*, pages 196–205. MIT Press, 2000.
- [4] J. E. Baker. Adaptive selection methods for genetic algorithms. In J.J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and their Applications*, pages 101–111, Carnegie Mellon University, 1985. Lawrence Erlbaum Associates.
- [5] R. Baraglia, J. I. Hidalgo, and R. Perego. A hybrid heuristic for the traveling salesman problem. *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION*, 5(6):613–622, 2001.

- [6] J. Berger, M. Sassi, and M. Salois. A hybrid genetic algorithm for the vehicle routing problem with time windows and itinerary constraints. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honovar, M. Jakiela, and R. E. Smith, editors, *GECCO-99. Proceedings of the Genetic and Evolutionary Conference*, volume 1, pages 44–51, Orlando, Florida, 1999. Morgan Kaufmann Publishers.
- [7] G. Bianchi and R. Church. A non-binary encoded GA for a facility location problem. Working Paper, D. Geography, U. California, Santa Barbara, 1992.
- [8] D. Bonachea, E. Ingberman, J. Levy, and S. McPeak. An improved adaptive multi-start approach to finding near-optimal solutions to the euclidean TSP. In D. Whitley, D. Goldberg, E. Cantú-Paz, L. Spector, I. Parmee, and H.-G. Beyer, editors, *GECCO-2000. Proceedings of the Genetic and Evolutionary Conference*, pages 143–150, Las Vegas, Nevada, 2000. Morgan Kaufmann Publishers.
- [9] B. Bozkaya, J. Zhang, and E. Erkut. An effective genetic algorithm for the p -median problem. In *INFORMS*, Dallas, USA, October 1997.
- [10] T. N. Bui and P. H. Eppley. A hybrid genetic algorithm for the maximum clique problem. In L. J. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 478–484, University of Pittsburgh, 1995. Morgan Kaufmann Publishers, Inc.
- [11] E. K. Burke and J. P. Newall. A multistage evolutionary algorithm for the timetable problem. *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION*, 3(1):63–74, 1999.

- [12] B. Chandra, H. Karloff, and C. Tovey. New results on the old k -opt algorithm for the traveling salesman problem. *SIAM Journal on Computing*, 28(6):1998–2029, 1999.
- [13] C.-H. Chu and C.-C. Tsai. A heuristic genetic algorithm for grouping manufacturing cells. In *Congress on Evolutionary Computation CEC2001*, pages 310–317. IEEE Press, 2001.
- [14] G. Cornuejols, M. Fisher, and G. Nemhauser. Location of bank accounts to optimize float: An analytic study of exact and approximate algorithms. *Management Science*, 23:789–910, 1997.
- [15] K. D. Crawford. Solving the n-queens problem using genetic algorithms. In *Proceedings ACM/SIGAPP, Symposium on Applied Computing, Volume 2, Technological Challenges of the 1990's*, pages 1039–1047, 1992.
- [16] G. A. Croes. A method for solving traveling-salesman problems. *Operations Research*, 5:791–812, 1958.
- [17] J. Culberson and J. Lichtner. On searching α -ary hypercubes and related graphs. In R. K. Belew and M. D. Vose, editors, *Foundations of Genetic Algorithms 4*, pages 263–290. Morgan Kaufmann Publishers, 1997.
- [18] L. Davis. Job shop scheduling with genetic algorithms. In J. J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and their Applications*, pages 136–140, Carnegie Mellon University, 1985. Lawrence Erlbaum Associates.

- [19] L. Davis, editor. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, 1991.
- [20] P. Densham and G. Rushton. A more efficient heuristic for solving large p -median problems. *Papers in Regional Science*, 71:307–329, 1992.
- [21] A. Eiben, P.-E. Raué, and Z. Ruttkay. GA-easy and GA-hard constraint satisfaction problems. In *Proceedings of the ECAI-94, Workshop on Constraint Processing*, Lecture Notes in Computer Science 923, pages 267–283. Springer-Verlag, 1995.
- [22] A. E. Eiben, J.K. van der Hauw, and J.I. van Hemert. Graph coloring with adaptive evolutionary algorithms. *Journal of Heuristics*, 4(1):25–46, 1998.
- [23] A. E. Eiben and Z. Ruttkay. Constraint satisfaction problems. In T. Bäck, D. Fogel, and M. Michalewicz, editors, *Handbook of Evolutionary Computation*. IOP Publishing Ltd. and Oxford University Press, 1997.
- [24] C. Erbas, S. Sarkeshik, and M. M. Tanik. Different perspectives of the n-queens problem. In *ACM Annual Computer Science Conference. Proceedings of the 1992 ACM Annual Conference on Communications*, pages 99–108, 1992.
- [25] V. Estivill-Castro. Hybrid genetic algorithms are better for spatial clustering. In R. Mizoguchi and J. Slaney, editors, *PRICAI 2000 Topics in Artificial Intelligence. 6th Pacific Rim International Conference on Artificial Intelligence*, pages 424–434. Springer Verlag LNAI 1886, 2000.
- [26] V. Estivill-Castro. Personal communication, March 2002.

- [27] V. Estivill-Castro, H. Mannila, and D. Wood. Right invariant metrics and measures of presortedness. *Discrete Applied Mathematics*, 42:1–16, 1993.
- [28] V. Estivill-Castro and A. T. Murray. Discovering associations in spatial data - an efficient medoid based approach. In X. Wu, R. Kotagiri, and K. K. Korb, editors, *Proc. of the 2nd Pacific-Asia Conf. on Knowledge Discovery and Data Mining (PAKDD-98)*, pages 110–121, Melbourne, Australia, 1998. Springer-Verlag, LNAI 1394.
- [29] V. Estivill-Castro and A. T. Murray. Spatial clustering for data mining with genetic algorithms. In *Int. ICSC Symp. Engineering of Intelligent Systems EIS-98*, 1998.
- [30] V. Estivill-Castro and R. Torres-Velázquez. Hybrid genetic algorithm for solving the p -median problem. In X. Yao, R. I. McKay, C. S. Newton, and J. H. Kim, editors, *Proceedings of the Second Asia Pacific Conference on Simulated Evolution and Learning (SEAL-98)*, pages 18–25. Springer Verlag LNAI 1585, 1999.
- [31] V. Estivill-Castro and R. Torres-Velázquez. Classical sorting embedded in genetic algorithms for improved permutation search. In *Congress on Evolutionary Computation CEC2001*, pages 941–948, Seoul, Korea, 2001. IEEE Press.
- [32] V. Estivill-Castro and R. Torres-Velázquez. How should feasibility be handled by genetic algorithms on constraint combinatorial optimization problems? the case of the *valued n -queens problem*. In *Second Workshop on Memetic Algorithms. WOMA II. Genetic and Evolutionary Computation Conference GECCO-2001*, pages 146–151, San Francisco CA, July 2001.

- [33] V. Estivill-Castro and D. Wood. A survey of adaptive sorting algorithms. *Computing Surveys*, 24:441–476, 1992.
- [34] V. Estivill-Castro and D. Wood. Randomized adaptive sorting. *Random Structures and Algorithms*, 4:26–51, 1993.
- [35] E. Falkenauer. A new representation and operators for genetic algorithms applied to grouping problems. *Evolutionary Computation*, 2(2):123–144, 1994.
- [36] G. Folino, C. Pizzuti, and G. Spezzano. Parallel hybrid method for SAT that couples genetic algorithms and local search. *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION*, 5(4):323–334, 2001.
- [37] M. L. Fredman, D. S. Johnson, L. A. McGeoch, and G. Ostheimer. Data structures for traveling salesmen. *Journal of Algorithms*, 18:432–479, 1995.
- [38] M. R. Garey and D. S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.
- [39] R. S. Garfinkel. Motivation and modeling. In E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, editors, *The Traveling Salesman Problem*, pages 17–36. John Wiley & Sons, 1985.
- [40] M. Georges-Schleuter. Asparagos an asynchronous parallel genetic optimization strategy. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 422–427, George Mason University, 1989. Morgan Kauffmann Publishers, Inc.
- [41] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research*, 5:533–549, 1986.

- [42] F. Glover. Tabu search: part I. *ORSA Journal on Computing*, 1:190–206, 1989.
- [43] F. Glover. Tabu search: part II. *ORSA Journal on Computing*, 2:4–32, 1990.
- [44] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Publishing Company, 1989.
- [45] D. E. Goldberg and R. Jr. Lingle. Alleles, loci, and the traveling salesman problem. In J. J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and their Applications*, pages 154–159, Carnegie Mellon University, 1985. Lawrence Erlbaum Associates.
- [46] D. E. Goldberg and S. Voessner. Optimizing global-local search hybrids. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honovar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computing Conference (GECCO-99)*, pages 220–228. Morgan Kaufmann Publishers, 1999.
- [47] M. Goodchild and V. Noronha. Location-allocation for small computers. Monograph 8, U. of Iowa, 1983.
- [48] J. Grefenstette, R. Gopal, B. Rosmaita, and D. van Gucht. Genetic algorithms for the traveling salesman problem. In J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and their Applications*, pages 160–168. Lawrence Erlbaum Associates, 1985.
- [49] J. J. Greffenstette. Incorporating problem specific knowledge into genetic algorithms. In L. Davis, editor, *Genetic Algorithms and Simulated Annealing*, pages 42–60. Pitman Publishing, 1987.

- [50] L. Hakimi. Optimum locations of switching centers and the absolute centers and medians of a graph. *Operations Research*, 12:450–459, 1964.
- [51] W. E. Hart. *Adaptive Global Optimization with Local Search*. PhD thesis, University of California, San Diego, 1994.
- [52] J. H. Holland. *Adaptation in Natural and Artificial Systems*. Complex Adaptive Systems. A Bradford Book, The MIT Press, 1992.
- [53] A. Homaifar, S. Guan, and G. E. Liepins. A new approach on the traveling salesman problem by genetic algorithms. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 460–466, University of New Mexico, 1993. Morgan Kaufmann Publishers, Inc.
- [54] A. Homaifar, J. Turner, and S. Ali. The n-queens problem and genetic algorithms. In *Proceedings IEEE Southeast Conference*, volume 1, pages 262–267, 1992.
- [55] J. J. Hopfield and D. W. Tank. 'Neural' computation of decisions in optimization problems. *Biological Cybernetics*, 52:141–152, 1985.
- [56] C. Hosage and M. Goodchild. Discrete space location-allocation solutions from genetic algorithms. *Annals of Operations Research*, 6:35–46, 1986.
- [57] J. Hynek. An improved genetic algorithm for the n-queens problem. In *Proceedings of the International Conference on Artificial Intelligence IC-AI'2000*, pages 517–522, Las Vegas NV, 2000.
- [58] J. Hynek. The n-queens problem revisited. In *Proceedings of the ICSC. Sym-*

- posia on Intelligent Systems and Applications (ISA'2000)*. ICSC Academic Press, 2000.
- [59] H. Ishibuchi, T. Murata, and S. Tomioka. Effectiveness of genetic local search algorithms. In T. Bäck, editor, *Proceedings of the Seventh International Conference on Genetic Algorithms*, pages 505–512, Michigan State University, East Lansing, 1997. Morgan Kaufmann Publishers, Inc.
- [60] P. Jog, J. Y. Suh, and D. van Gucht. The effect of population size, heuristic crossover and local improvement on a genetic algorithm for the traveling salesman problem. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 110–115, George Mason University, 1989. Morgan Kauffmann Publishers, Inc.
- [61] D. S. Johnson and L. A. McGeoch. The traveling salesman problem: a case study. In E. Aarts and J. K. Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 215–310. John Wiley & Sons, 1997.
- [62] R. C. Johnson. Record travelling salesman solution. TechWeb, <http://www.techweb.com/>, June 1998.
- [63] T. Jones and S. Forrest. Fitness distance correlation as a measure of problem difficulty for genetic algorithms. In L. J. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 184–192, University of Pittsburgh, 1995. Morgan Kaufmann Publishers, Inc.
- [64] K. Katayama and H. Narihisa. Iterated local search approach using genetic transformation to the traveling salesman problem. In W. Banzhaf, J. Daida,

- A. E. Eiben, M. H. Garzon, V. Honovar, M. Jakiela, and R. E. Smith, editors, *GECCO-99. Proceedings of the Genetic and Evolutionary Conference*, volume 1, pages 321–328, Orlando, Florida, 1999. Morgan Kaufmann Publishers.
- [65] K. Katayama, M. Tani, and H. Narihisa. Solving large binary quadratic programming problems by effective genetic local search algorithm. In D. Whitley, D. Goldberg, E. Cantú-Paz, L. Spector, I. Parmee, and H.-G. Beyer, editors, *GECCO-2000. Proceedings of the Genetic and Evolutionary Conference*, pages 643–650, Las Vegas, Nevada, 2000. Morgan Kaufmann Publishers.
- [66] L. Kaufman and P. J. Rousseuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley & Sons, NY, US, 1990.
- [67] G. A. P. Kindervater and M. W. P. Savelsbergh. Vehicle routing: handling edge exchanges. In E. Aarts and J. K. Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 337,360. John Wiley & Sons, 1997.
- [68] S. Kirkpatrick, C. D. Jr. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.
- [69] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley Publishing Company, 1973.
- [70] M. W. S. Land. *Evolutionary Algorithms with Local Search for Combinatorial Optimization*. PhD thesis, University of California, San Diego, 1998.
- [71] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, editors. *The Traveling Salesman Problem*. John Wiley & Sons, 1985.

- [72] J. K. Lenstra. Clustering a data array and the traveling-salesman problem. *Operations Research*, 22:413–414, 1974.
- [73] M. Li and P. M. B. Vitanyi. A theory of learning simple concepts under simple distributions and average case complexity for the universal distribution. In *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science*, pages 34–39, Research Triangle Park, NC., 1989.
- [74] K.-H. Liang, X. Yao, and C. Newton. Combining landscape approximation and local search in global optimization. In *Proceedings of the 1999 Congress on Evolutionary Computation*, volume 2, pages 1514–1520. IEEE Press, 1999.
- [75] K.-H. Liang, X. Yao, and C. Newton. Evolutionary search of approximated N-dimensional landscapes. *International Journal of Knowledge-Based Intelligent Engineering Systems*, 4(3):172–183, 2000.
- [76] G. E. Liepins and M. R. Hilliard. Greedy genetics. In J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithm and Their Applications*, pages 90–99. Lawrence Erlbaum Associates, 1987.
- [77] S. Lin and B. W. Kernighan. An effective heuristic for the traveling-salesman problem. *Operations Research*, 21:498–516, 1973.
- [78] G. Magyar, M. Johnsson, and O. Nevalainen. An adaptive hybrid genetic algorithm for the three-matching problem. *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION*, 4(2):135–146, 2000.
- [79] H. Mannila. Measures of presortedness and optimal sorting algorithms. *IEEE Transactions on Computers*, C-34:318–325, 1985.

- [80] J. E. Maranzana. On the location of supply points to minimize transport costs. *Operational Research Quarterly*, 15:261–270, 1964.
- [81] W. T. McCormick Jr., P. J. Schweitzer, and T. W. White. Problem decomposition and data reorganization by a clustering technique. *Operations Research*, 20:993–1009, 1972.
- [82] P. Merz and B. Freisleben. A genetic local search approach to the quadratic assignment problem. In T. Bäck, editor, *Proceedings of the Seventh International Conference on Genetic Algorithms*, pages 465–472, Michigan State University, East Lansing, 1997. Morgan Kaufmann Publishers, Inc.
- [83] P. Merz and B. Freisleben. Genetic local search for the TSP: New results. In *Proceedings of the IEEE Conference on Evolutionary Computation*, pages 159–164. IEEE Press, 1997.
- [84] P. Merz and B. Freisleben. Genetic algorithms for binary quadratic programming. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honovar, M. Jakiela, and R. E. Smith, editors, *GECCO-99. Proceedings of the Genetic and Evolutionary Conference*, volume 1, pages 417–424, Orlando, Florida, 1999. Morgan Kaufmann Publishers.
- [85] P. Merz and B. Freisleben. Fitness landscape analysis and memetic algorithms for the quadratic assignment problem. *IEEE Transactions on Evolutionary Computation*, 4(4):337–352, 2000.
- [86] P. Merz and B. Freisleben. Fitness landscapes, memetic algorithms and greedy

- operators for graph bipartitioning. *Evolutionary Computation*, 8(1):61–91, 2000.
- [87] B. T. Messmer and H. Bunke. A decision tree approach to graph and subgraph isomorphism detection. *Pattern Recognition*, pages 1979–1998, 1999.
- [88] Z. Michalewicz. A survey of constraint handling techniques in evolutionary computation methods. In *Proceedings of the 4th Annual Conference on Evolutionary Programming*, pages 135–155. MIT Press, 1995.
- [89] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, third edition, 1996.
- [90] M. Mitchell. *An Introduction to Genetic Algorithms*. A Bradford book, The MIT Press, 1996.
- [91] A. Moffat, G. Eddy, and O. Petersson. Splaysort. fast, versatile, practical. *Software – Practice and Experience*, 26(7):781–797, July 1996.
- [92] P. Moscato. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. Technical Report 826, California Institute of Technology, 1989.
- [93] H. Mühlenbein. Parallel genetic algorithms, population genetics and combinatorial optimization. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 416–421, George Mason University, 1989. Morgan Kaufmann Publishers, Inc.
- [94] H. Mühlenbein. Evolution in time and space - the parallel genetic algorithm.

- In G. J. E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 316–337, Indiana University, 1991. Morgan Kaufmann.
- [95] H. Mühlenbein, M. Gorger-Schleuter, and O. Krämer. Evolution algorithms in combinatorial optimization. *Parallel Computing*, 7:65–85, 1988.
- [96] A. T. Murray and R. L. Church. Applying simulated annealing to location-planning models. *J. of Heuristics*, 2:31–53, 1996.
- [97] A. T. Murray and V. Estivill-Castro. Cluster discovery techniques for exploratory spatial data analysis. *Int. J. of GIS*, 12(5):431–443, 1998.
- [98] Y. Nagata and S. Kobayashi. Edge assembly crossover: A high-power genetic algorithm for the traveling salesman problem. In T. Bäck, editor, *Proceedings of the Seventh International Conference on Genetic Algorithms*, pages 450–457, Michigan State University, East Lansing, 1997. Morgan Kaufmann Publishers, Inc.
- [99] S. Narula, U. Ogbu, and H. Samuelsson. An algorithm for the p -median problem. *Operations Research*, 25:709–713, 1977.
- [100] Rice News. Researchers forge new optimal path for traveling salesman problem. Rice University, June 25th 1998.
- [101] R. T. Ng and J. Han. Efficient and effective clustering methods for spatial data mining. In J. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. of the 20th Conf. on Very Large Data Bases (VLDB)*, pages 144–155, Santiago, Chile, 1994. Morgan Kaufmann.

-
- [102] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 2nd edition, 1999.
- [103] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Inc., 1982.
- [104] J. Paredis. Genetic state-space search for constrained optimization problems. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, volume 2, pages 967–972. Morgan Kaufmann, 1993.
- [105] C. Peterson and B. Söderberg. Artificial neural networks. In E. Aarts and J. K. Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 173–213. John Wiley & Sons, 1997.
- [106] N. J. Radcliffe. Genetic set recombination. In L. D. Whitley, editor, *Foundations of Genetic Algorithms 2, FOGA-92*, pages 203–219. Morgan Kaufmann, 1993.
- [107] N. J. Radcliffe. The algebra of genetic algorithms. *Annals of Mathematics and Artificial Intelligence*, 10:339–384, 1994.
- [108] N. J. Radcliffe and F. A. W. George. A study of set recombination. In *Proc. Fifth Int. Conf. Genetic Algorithms*, pages 23–30. Morgan Kaufmann, 1993.
- [109] N. J. Radcliffe and P. D. Surry. Formal memetic algorithms. In T. Fogarty, editor, *Evolutionary Computing: AISB Workshop*, LNCS 865. Springer-Verlag, 1994.
- [110] R. C. Read and D. G. Corneil. The graph isomorphism disease. *Journal of Graph Theory*, 1:339–363, 1977.

- [111] C. Reeves. Hybrid genetic algorithms for bin-packing and related problems. *Annals of Operations Research*, 63:371–396, May 1996.
- [112] E. M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms, Theory and Practice*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1977.
- [113] C. ReVelle and R. Swain. Central facilities location. *Geographical Analysis*, 2:30–42, 1970.
- [114] J. T. Richardson, M. R. Palmer, G. Liepins, and M. Hilliard. Some guidelines for genetic algorithms with penalty functions. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 191–197, George Mason University, 1989. Morgan Kauffmann Publishers, Inc.
- [115] M. Rocha, R. Mendes, P. Cortez, and J. Neves. Sitting guests at a wedding party: Experiments on genetic and evolutionary constrained optimization. In *Congress on Evolutionary Computation CEC2001*, pages 671–678, Seoul, Korea, 2001. IEEE Press.
- [116] D. Rolland, E. Schilling, and J. Current. An efficient tabu search procedure for the p -median problem. *European J. of Operations Research*, 96:329–342, 1996.
- [117] K. Rosing. An optimal method for solving the (generalized) multi-Weber problem. *European J. of Operations Research*, 58:414–426, 1992.
- [118] K. Rosing, E. Hillsman, and H. Rosing. A note comparing optimal and heuristic solutions to the p -median problem. *Geographical Analysis*, 11:86–89, 1979.
- [119] K. E. Rosing, C. S. ReVelle, and H. Rosing-Voyelaar. The p -median and its

- linear programming relaxation: An approach to large problems. *J. of the Operational Research Society*, 30:815–823, 1979.
- [120] A. Ruiz-Andino, L. Araujo, F. Sáenz, and J. Ruz. A hybrid evolutionary approach for solving constrained optimization problems over finite domains. *IEEE Transactions on Evolutionary Computation*, 4(4):353–372, November 2000.
- [121] J. Sakuma and S. Kobayashi. Extrapolation-directed crossover for the job-shop scheduling problems: Complementary combination with jox. In D. Whitley, D. Goldberg, E. Cantú-Paz, L. Spector, I. Parmee, and H.-G. Beyer, editors, *GECCO-2000. Proceedings of the Genetic and Evolutionary Conference*, pages 973–980, Las Vegas, Nevada, 2000. Morgan Kaufmann Publishers.
- [122] V. Schnecke and O. Vornberger. Hybrid genetic algorithms for constrained placement problems. *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION*, 1(4):266–277, 1997.
- [123] R. Sedgewick. *Algorithms in C++*. Addison-Wesley Publishing Company, 1992.
- [124] B. M. Smith. Modelling a permutation problem. Research Report 2000.18, University of Leeds. School of Computer Studies, June 2000. Presented at the ECAI 2000, Workshop on Modelling and Solving Problems with Constraints.
- [125] P. Sorensen. Analysis and design of heuristics for the p -median location-allocation problem. Master’s thesis, D. Geography, U. California, Santa Barbara, 1994.

- [126] R. Sosič and J. Gu. Efficient local search with conflict minimization: A case study of the n-queens problem. *IEEE Transactions on Knowledge and Data Engineering*, 6(5):661–668, October 1994.
- [127] T. Starkweather, S. McDaniel, K. Mathias, and D. Whitley. A comparison of genetic sequencing operators. In R. K. Belew and L. B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 69–76, San Mateo California, 1991. Morgan Kaufmann Publishers.
- [128] J. Suh and D. van Gucht. Incorporating heuristic information into genetic search. In J. J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms and their Applications*, pages 100–107. Lawrence Erlbaum Associates, 1987.
- [129] M. B. Teitz and P. Bart. Heuristic methods for estimating the generalized vertex median of a weighted graph. *Operations Research*, 16:955–961, 1968.
- [130] R. Torres-Velázquez and V. Estivill-Castro. Local search for Hamiltonian path with applications to clustering visitation paths. In *Local Search, Two Day Workshop*, City University, London UK 16-17 April 2002, 2002. OR Society (UK): Local Search Study Group.
- [131] R. Torres-Velázquez and V. Estivill-Castro. A memetic algorithm guided by quicksort for the error-correcting graph isomorphism problem. In S. Cagnoni, J. Gottlieb, E. Hart, M. Middendorf, and G. R. Raidl, editors, *Applications of Evolutionary Computing, Lecture Notes in Computer Science LNCS 2279*, pages 173–182, Kinsale Ireland, April 3-4, 2002, 2002. Springer Verlag.

- [132] R. Torres-Velázquez and V. Estivill-Castro. A memetic algorithm instantiated with selection sort consistently finds global optima for the error-correcting graph isomorphism. In *Congress on Evolutionary Computation CEC2002*, pages 1958–1963, Honolulu, Hawaii, May 12-17, 2002, 2002. IEEE Press.
- [133] H.-K. Tsai, J.-M. Yang, and C.-Y. Kao. A genetic algorithm for traveling salesman problems. In L. Spector, E. D. Goodman, A. Wu, W. B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke, editors, *GECCO-2001. Proceedings of the Genetic and Evolutionary Conference*, pages 687–693, San Francisco, California, 2001. Morgan Kaufmann Publishers.
- [134] W.-H. Tsai and K.-S. Fu. Error-correcting isomorphisms of attributed relational graphs for pattern analysis. *IEEE Transactions on Systems, Man and Cybernetics*, 9(12):757–768, December 1979.
- [135] E. Tsang. A glimpse of constraint satisfaction. *Artificial Intelligence Review*, 13:215–227, 1999.
- [136] P. Turney. How to shift bias: Lessons from the Baldwin effect. *Evolutionary Computation*, 4(3):271–295, 1996.
- [137] N. L. J. Ulder, E. H. L. Aarts, H.-J. Bandelt, P. J. M. van Laarhoven, and E. Pesch. Genetic local search algorithms for the travelling salesman problem. In *First International Conference on Parallel Problem Solving from Nature*, pages 109–116. Springer Verlag, 1991.

- [138] J. Valenzuela and A. E. Smith. A seeded memetic algorithm for large unit commitment problems. *Journal of Heuristics*, 8:173–196, 2002.
- [139] M. D. Vose. *The Simple Genetic Algorithm: Foundations and Theory*. A Bradford Book. The MIT Press, 1999.
- [140] M. Wall. GALib: A c++ library of genetic algorithms components. <http://lancet.mit.edu/ga/>, August 1996.
- [141] Y.-K. Wang, K.-C. Fan, and J.-T. Horng. Genetic-based search for error-correcting graph isomorphism. *IEEE Transactions on Systems, Man and Cybernetics, Part B: Cybernetics*, 27(4):588–597, August 1997.
- [142] J. Weaver and R. Church. A median location model with nonclosest facility service. *Transportation Science*, 19:107–119, 1985.
- [143] D. B. West. *Introduction to Graph Theory*. Prentice Hall, 1996.
- [144] D. Whitley. Modeling hybrid genetic algorithms. In G. Winter and P. Cuesta, editors, *Genetic Algorithms in Engineering and Computer Science*, pages 191–201. John Wiley, 1995.
- [145] D. Whitley, S. Dominic, and R. Das. Genetic reinforcement learning with multilayer neural networks. In R. K. Belew and L. B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 562–569, University of California San Diego, 1991. Morgan Kaufmann Publishers, Inc.
- [146] D. Whitley, V. S. Gordon, and K. Mathias. Lamarckian evolution, the Baldwin effect and function optimization. In Y. Davidor, H. P. Schwefel, and

- R. M. Maenner, editors, *International Conference on Evolutionary Computation, Parallel Problem Solving from Nature III*, pages 6–15. Springer Verlag, 1994.
- [147] D. H. Wolpert and W.G. MacReady. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1:67–82, 1997.
- [148] J. Xiao. Personal communication, June 2001.
- [149] J. Xiao, Y. Zhang, X. Jia, and T. Li. Measuring similarity of interests for clustering web-users. In *Australian Computer Science Communications. Database Technologies. Proceedings of the 12th Australasian Database Conference ADC 2001*, volume 23, pages 107–114. IEEE Computer Society, 2001.
- [150] Y. Xu and S.-C. Xu. Heuristic operators, redundant mapping and other issues in genetic algorithms. In *Congress on Evolutionary Computation CEC2001*, pages 1398–1405. IEEE Press, 2001.
- [151] T. Yamada and R. Nakano. A genetic algorithm with multi-step crossover for job-shop scheduling problems. In *First IEE/IEEE International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications (GALESIA '95)*, pages 146–151, Sheffield, UK, 1995.
- [152] T. Yamada and C. R. Reeves. Permutation flowshop scheduling by genetic local search. In *Genetic Algorithms in Engineering Systems: Innovations and Applications*, pages 232–238, 1997.
- [153] X. Yao. Optimization by genetic annealing. In M. Jabri, editor, *Proceedings of*

the Second Australian Conference on Neural Networks., pages 94–97, Sydney, Australia, 1991.

- [154] X. Yao. Simulated annealing with extended neighbourhood. *International Journal of Computer Mathematics*, 40:169–189, 1991.