DESIGN AND IMPLEMENTATION OF A GENETIC ALGORITHM FOR A PACKING WITH AN UNCERTAIN FUTURE PROBLEM

Cristiano Saturni

Dicember 2000

To my parents

Contents

Declaration **vii**

Aknowledgements **ix**

In Breve **xi**

1 About this Project **1**

1.1 Introduction 1

1.2 Summary 2

2 Constraint Satisfaction Problems (CSPs) **3**

2.1 Overview 3

2.2 Constraint Satisfaction Problems (CSPs) 3

2.3 K-Consistency 5

2.4 Algorithms to solve CSPs 9

2.4.1 Pruning algorithms 9

2.4.2 Solving a CSP by maximizing a function 11

2.4.3 Heuristic and stochastic algorithms 13

2.4.4 Consistency techniques 13

2.5 Optimization CSPs (OCSPs) 13

2.5.1 Exact algorithms 14

2.5.2 Aproximation algorithms 15

3 Branching Constraint Satisfaction Problems (BCSPs) **17**

3.1 Overview 17

3.2 Branching Constraint Satisfaction Problems (BCSPs) 17

3.3 Assignments and feasible assignments 18

3.4 Expected Utility (EU) 19

3.5 BCSPs as CSPs 22

3.6 The extended BCSPs (eBCSPs) 23

4 Genetic Algorithms (GAs) 25
4.1 Overview 25
4.2 Introduction 25
4.3 Evolutionary Programs (EPs) 26
4.4 Genetic Algorithms (GAs) 28
4.4.1 The typical GA 28
4.4.2 The problem of the feasibility 30
4.4.3 Classical GAs 31
4.5 $(\mu + \lambda)$-Evolution Strategies ($(\mu + \lambda)$-ESs) 32
4.6 (μ, λ)-Evolution Strategies ((μ, λ)-ESs) 32

5 The Harbour Packing Problem 33
5.1 Overview 33
5.2 Introduction 33
5.3 The problem simplified 34
5.4 The problem formalized 37
5.4.1 Branching Packing Problems (BPPs) 37
5.4.2 Assignments and feasible assignments 38
5.4.3 Expected Utility (EU) 42
5.5 BPPs with one path trees 45
5.6 NP-hardness 46
5.7 The BPP as a BCSP 49
5.8 The eBCSPs and the requirement of balance 49

6 The Genetic Algorithm 51
6.1 Overview 51
6.2 Introduction 51
6.3 Basic concepts and operations 52
6.3.1 The ordering of the positions 52
6.3.2 The ordering of the nodes 54
6.3.3 The Fill operation 54
6.3.4 The repair operations 54
6.3.5 The functions returning a random position 56
6.4 The main structure 57
6.5 The creation of the initial feasible population 59
6.6 The mating 61
6.6.1 The Upward Gentle Crossover 62
6.6.2 The Downward Gentle Crossover 64
6.6.3 The Ordered Brutal Crossover 66
6.6.4 The Random Brutal Crossover 66
6.6.5 The choice of the crossover node 69

6.7 The mutation 69
6.7.1 The Ordered Gentle Mutation 70
6.7.2 The Random Gentle Mutation 72
6.7.3 The Ordered Brutal Mutation 72
6.7.4 The Random Brutal Mutation 72
6.8 The selection 75
6.8.1 The Fixed Selection 77
6.8.2 The $f(\frac{val}{MAX})$-selection 78
6.8.3 The Proportional Selection 79
6.9 The Shake feature 80

7 A GA for BCSPs 83
7.1 Overview 83
7.2 Introduction 83
7.3 Basic concepts and operations 83
7.4 The main structure 85
7.5 The changed Initial-feasible-population 86
7.6 The changed crossovers 86
7.7 The mutations 89

8 The Parameters Tuning 93
8.1 Overview 93
8.2 Introduction 93
8.3 The experiments 94
8.4 The experiments on single genetic components 95
8.4.1 The first set of experiments 95
8.4.2 The second set of experiments 100
8.5 The experiments on mixed genetic components 101
8.6 The experiments on the shake feature 104
8.7 Conclusions 105

9 The Evaluation of the GA 107
9.1 Overview 107
9.2 Introduction 107
9.3 Problems with known optimal solution 109
9.3.1 Problems with linear trees 109
9.3.2 A modified problem 109
9.3.3 A problem with reached UB 111
9.4 Problems with unknown optimal solution 112
9.5 Conclusions 112

10 The Application **115**
10.1 Overview . . . 115
10.2 Introduction . . . 115
10.3 The parameters file . . . 116
10.4 The input file . . . 116
10.5 The output file . . . 121
10.6 The average and best files . . . 122
10.7 The seed file . . . 122
10.8 The run time output . . . 122

A The Support Applications **123**
A.1 The random problem generator . . . 123
A.2 The random-solver . . . 126

B The Test Problems **127**
B.1 rinput0 . . . 127
B.2 rinput4 . . . 129
B.3 rinput5 . . . 130
B.4 rinput6 . . . 132
B.5 rinput7 . . . 133
B.6 rinput8 . . . 134
B.7 art7 . . . 136
B.8 lin16 . . . 137
B.9 lin73 . . . 138

Bibliography **141**

Declaration

I declare that this thesis has been composed by myself and describes my own work. All the sources of information has been aknowledged.

Cristiano Saturni

Dicember 13, 2000.

Department Of Computing Science
University Of Aberdeen
Kings Colledge
Aberdeen
UK

Dipartimento di Elettronica e Informatica
Universita' di Padova
Padova
ITALY

Aknowledgements

I want to thank all the people who helped me to make this thesis and expecially the people who have been close to me during these years in the university.

First of all I want to thank my parents, who made possible this degree with their work and sacrifices: I feel that this degree is also theirs. I thank my sister and grandmother.

I want to thank Dr. Ken Brown who helped me with attention and sympathy in the project for this thesis and from whom I have learned a lot and Dr. David Fawler for his gentle help. I thank Prof. Matteo Fischetti for his advices, trust and esteem.

I thank Prof. Luigi Donà Dalle Rose and Mr. Dirk Nikodem for making possible my final project in the stimulating Computing Science Department of the University of Aberdeen and Prof. Maristella Agosti for her important encouragement and confidence.

I want to thank my friends who have always been close to me in these years of study, in particular Andrea Cola, Giuliano, Federico, Alberto, Corrado, Paolo, Luca, Andrea Giaretta, Stefano. I thank my trainer Alby and all the friends of the gym. I thank Sofia for her love. I thank the many other people who helped me with their affection.

In Breve

Introduzione

Questa tesi è stata svolta nel dipartimento di Computing Science dell' università di Aberdeen in Scozia nell'ambito del progetto ERASMUS con la supervisione del Dr. Ken Brown. La tesi è stata scritta in Italia con la supervisione del Prof. Matteo Fischetti.

Lo scopo della tesi è stato quello di progettare e implementare in C un Algoritmo Genetico (GA) per risolvere un particolare problema di ottimizzazione, derivato dalla semplificazione e idealizzazione di un problema di caricamento di navi reale e più complesso.

La tesi è stata svolta nelle seguenti fasi:

1. lo studio dei Constraint Satisfaction Problems (CSPs), dei Branching Constraint Satisfaction Problems (BCSPs) e degli algoritmi genetici e l'analisi del problema del porto (capitoli 2, 3, 4 e 5 rispettivamente)

2. la progettazione di un GA per BCSPs (Capitolo 7), la progettazione e l'implementazione in C di un GA per il problema del porto (capitoli 6 e 10 rispettivamente)

3. la progettazione e l' implementazione di un generatore casuale di problemi e di un algoritmo di ricerca casuale delle soluzioni (Appendice A)

4. gli esperimenti per trovare una buona combinazione di parametri per il GA (Capitolo 8) e quelli per la sua valutazione (Capitolo 9)

Il problema

Il problema ideale da noi affrontato è il seguente. Viene dato il ponte rettangolare di una nave, sul quale è disposta una griglia, e un insieme di container, ciascuno con una sua utilità, che possono essere posizionati sul ponte in modo che, dopo una eventuale rotazione, lo spigolo in basso a sinistra coincida con uno dei punti della griglia. I container arrivano alla nave in tempi diversi e la sequenza degli arrivi non è nota. Alcune possibili sequenze d'arrivo con le rispettive probabilità vengono date mediante l'albero degli arrivi, cioè un albero con un container associato ad ogni nodo ed una probabilità ad ogni lato in modo tale che per ogni nodo:

1. la somma delle probabilità dei lati che portano ai figli è ≤ 1
2. i container associati ai nodi figli sono tutti diversi
3. i container dei nodi del percorso dalla radice a quel nodo sono tutti diversi

Ogni nodo rappresenta un arrivo e i possibili arrivi successivi sono rappresentati dai figli, ciascuno con la probabilità del rispettivo lato. Se in un nodo la somma delle probabilità dei lati che portano ai figli è $\sigma < 1$, significa che, con probabilità $1 - \sigma$, dopo quell' arrivo non ce ne saranno altri. Un tale tipo di nodo è detto *terminale* e la sua probabilità è definita come $1 - \sigma$. Le foglie sono nodi terminali con probabilità 1, ma possono esistere anche nodi terminali interni. Le possibili sequenze di arrivo sono cosìtutte le sequenze associate ai percorsi dal nodo radice ad un nodo terminale e le rispettive probabilità si possono calcolare moltiplicando le probabilità dei lati dei rispettivi percorsi e del corrispondente nodo terminale.

Appena un container arriva deve essere caricato o rifiutato: non può essere parcheggiato nel molo in attesa che altri container arrivino, postponendo la decisione sul caricamento, e una volta caricato non può essere spostato. Il problema consiste nel decidere, prima che qualunque container sia arrivato, cosa fare ad ogni possibile arrivo, in modo che, quando i container cominciano ad arrivare, se vengono caricati rispettando le decisioni prese in precedenza, la somma delle utilità dei contanier caricati sarà probabilmente alta. (Il problema reale è più complesso, in quanto nella realtà, per esempio, i container devono essere disposti sul ponte in modo da non sbilanciare eccesivamente la nave e in modo da rispettare alcune regole di sicurezza dipendenti dalle merci trasportate e le caratteristiche della gru che carica i container.)

Più precisamente i dati del problema sono il ponte della nave con la griglia, l'insieme dei container e l'albero degli arrivi. Una soluzione del problema è una funzione che ad ogni nodo dell'albero associa una decisione per il corrispondente container, cioè una posizione sul ponte (un punto della griglia e l'eventuale rotazione) o NULL che rappresenta il non caricamento del container corrispondente, in modo tale che i container caricati siano dentro il ponte e che per ogni nodo i container caricati del percorso dalla radice a quel nodo non si sovrappongono tra di loro. Il valore di una soluzione (detto EU) è ottenuto nel seguente modo: per ogni nodo terminale si calcola la somma delle utilità dei container che la soluzione ha caricato del percorso dalla radice a quel nodo; si calcola il prodotto delle probabilità dei lati del percorso dalla radice a quel nodo e lo si moltiplica per la probabilità del rispettivo nodo terminale; si moltiplicano la somma delle utilità e il prodotto delle probabilità e si sommano i prodotti cosi' ottenuti per tutti i nodi terminali. I nostro problema è quello di trovare la soluzione con il valore più alto.

È importante notare che questo problema è un' estensione del normale problema di caricamento (cutting stock bidimensionale): infatti un problema con un albero lineare e con tutti gli archi etichettati con probabilità 1 equivale al problema di scegliere dai dati container alcuni da posizionare sulla nave e di trovare per questi un adeguato posizionamento in modo che la somma delle utilità dei container caricati sia massima.

È stato da noi dimostrato che il problema è NP-hard e quindi meglio risolvibile mediante un algoritmo di approssimazione. Abbiamo deciso cosi' di utilizzare un algoritmo genetico.

L' algoritmo

Un algoritmo genetico imita il processo evolutivo di una specie di esseri viventi che si adattano progressivamente all'ambiente grazie ai meccanismi dell'ereditarietà, della riproduzione, della mutazione e della selezione naturale.

Il GA da noi sviluppato genera una sequenza finita di n-uple di soluzioni (dette *popolazioni* e i cui elementi sono detti *individui* o *cromosomi*) e ritorna in output la soluzione generata con il valore più alto. La prima popolazione viene generata in modo casuale e ogni altra viene prodotta dalla precedente in tre fasi:

1. la riproduzione
2. la mutazione
3. la selezione

La difficoltà di questa tesi è stata la progettazione e l' implementazione degli operatori genetici usati nelle varie fasi – i 4 crossover, le 4 mutazioni e le 3 selezioni – che sono molto complessi e hanno richiesto mesi di lavoro. Il numero di popolazioni generate e il numero di individui per popolazione sono parametri dell' algoritmo.

La riproduzione

Nella fase di riproduzione alcuni individui vengono scelti per l' accoppiamento e vengono raggruppati in coppie. Il numero di coppie è variabile in modo casuale, ma è controllato da un parametro dell' algoritmo. Ciascuna coppia (detta di *genitori*) genera due individui (detti *figli*) simili a entrambi i genitori che vanno a sostituire i genitori nella popolazione. Nel nostro GA vengono usati 4 meccanismi per la generazione dei figli (detti *crossover*) ciascuno associato ad una frequenza che vengono applicati con la frequenza corrispondente. Le frequenze dei crossover sono un parametro del GA. I crossover sviluppati sono i seguenti:

1. Upward Gentle Crossover

2. Downward Gentle Crossover

3. Random Brutal Crossover

4. Ordered Brutal Crossover

Un crossover produce i figli dai genitori nel seguente modo:

1. sceglie un nodo A dell'albero in modo casuale

2. scambia nei genitori i valori dei nodi del sottoalbero che ha A come radice, producendo due funzioni non necessariamente ammissibili, a causa di sovrapposizioni

3. ripara le due funzioni, cioè modifica queste due funzioni rendendole ammissibili e cercando di mantenerne l'aspetto

Ciò che distingue i 4 crossover è il metodo di riparazione.

Per esempio l' Upward Gentle Crossover ripara le due funzioni possibilmente non ammissibili nel seguente modo:

1. pone a NULL quei nodi del percorso dalla radice ad A i cui container nella corrispondente posizione si sovrappongono con container di nodi del sottoalbero nella corrispondente posizione

2. esegue una operazione di riempimento della soluzione cosi' ottenuta, cioè cerca di sostituire nei nodi del percorso dalla radice ad A i valori nulli con posizioni non nulle compatibili con il corrispondente percorso e sottoalberi

Il nodo su cui viene fatto il crossover viene scelto usando uno di 3 algoritmi di scelta, che assegnano probabilita' diverse ai vari nodi. Per esempio uno di questi algoritmi esclude dalla scelta la radice e le foglie e sceglie solo nodi interni; ogni nodo interno viene scelto con una probabilità $p = \frac{1}{k}$, con k il numero di nodi interni del suo livello. Un parametro del GA decide quale algoritmo utilizzare.

La mutazione

Nella fase di mutazione gli individui vengono più o meno mutati, cioè modificati cercando di mantenerne le caratteristiche. In questo GA vengono usati 4 metodi di mutazione, ciascuno associato ad una frequenza, che vengono applicati con la corrispondente frequenza. Le mutazioni sono anche associate ad un parametro che ne determina l' intesità. Questi parametri e le frequenze dei vari metodi sono parametri dell' algoritmo.

Gli operatori di mutazione producono nuovi individui nel seguente modo:

1. vengono selezionati in nodi da mutare
2. viene loro assegnato un valore casuale, producendo una funzione non necessariamente ammissibile
3. viene riparata la funzione riportando l'ammissibilita' cercando di mantenerne l'aspetto

Ciò che distingue i 4 metodi di mutazione è il metodo di riparazione. Anche qui i metodi sono molto complessi e hanno richiesto mesi di lavoro.

Per esempio il Random Gentle Mutation inizialmente pone a NULL i nodi selezionati per la mutazione; successivamente considera questi nodi in ordine casuale; a ciascuno assegna un valore casuale e se questo non e' compatibile con il corrispondente percorso e sottoalberi viene scelto il valore compatibile piu' vicino.

La selezione

La fase di selezione ha lo scopo di far sopravvivere gli individui migliori e di sopprimere quelli peggiori. Nel nostro GA si possono usare 3 meccanismi di selezione.

1. la proportional selection
2. la fixed selection
3. la $f(\frac{val}{max})$ selection

Uno soltanto di questi metodi viene usato nelle varie iterazioni. Un parametro dell'algoritmo decide quale metodo viene utilizzato. Gli individui della popolazione selezionata vengono scelti tra quelli della popolazione mutata ciacuno con una probabilità dipendente dal suo valore e crescente con esso. Alcuni individui possono essere scelti più volte e alcuni mai. Ciò che distingue i 3 metodi di selezione è la regola che associa il valore di un individuo con la sua probabilità di essere selezionato. Per esempio nella proportional selection, la probabilità di selezione è proporzionale al valore dell' individuo.

Lo shake

L' algoritmo sviluppato per questa tesi presenta poi un meccanismo da me inventato e chiamato *shake* per impedirne la convergenza prematura, cioè per impedire che l'algoritmo concentri la sua ricerca in una zona di massimo locale non globale. Il meccanismo consiste nell'intensificare la mutazione per qualche generazione quando il valore medio e il valore massimo delle ultime generazioni sono troppo vicini. Anche questo meccanismo viene controllato da alcuni parametri dell' algoritmo.

Gli esperimenti

Nell'ultima parte del progetto sono stati fatti molti esperimenti per trovare la combinazione di parametri che faccia funzionare il GA nel modo migliore. Questi esperimenti sono stati fatti risolvendo problemi di test generati in modo casuale da un algoritmo da me progettato e implementato. Gli esperimenti hanno portato ad alcune combinazioni di parametri che si sono rivelate molto buone e migliori di quelle che si usavano all'inizio.

Infine numerosi esperimenti sono stati fatti per valutare la capacità del GA di risolvere il problema. La valutazione è stata molto difficile perchè di pochi problemi difficili si aveva la soluzione esatta e non era disponibile alcun altro algoritmo per questa classe di problemi.

Abbiamo generato in modo casuale alcuni problemi difficili e abbiamo confrontato la soluzione trovata dal nostro algoritmo genetico con quella fornita da un algoritmo da noi implementato che genera in modo casuale lo stesso numero di individui e ritorna in output l'individuo con il valore più alto. Il risultato è che il nostro GA ritorna sempre una soluzione molto migliore.

Infine abbiamo affrontato con il nostro GA alcuni problemi di cutting stock bidimensionale risolti in modo esatto in letteratura – che, come detto i precedenza, si possono considerare un caso particolare del problema risolvibile dal nostro GA – e alcuni problemi la cui soluzione ottima puo' essere dedotta con un ragionamento. Il risultato è che l' algoritmo genetico ritorna spesso la soluzione ottima e comunque sempre una molto vicina a quella ottima.

Il GA da noi sviluppato è stato quindi considerato molto buono e valido anche per un uso professionale.

Chapter 1

About this Project

1.1 Introduction

This project has been developed as a final thesis for the "Corso di Laurea in Ingegneria Informatica" in the University of Padua under the ERASMUS program in the Computing Science Department of the University of Aberdeen with the supervision of Dr. Ken Brown.

The project consisted in studying and implementing a computer program to solve a particular optimization problem. The problem is an idealized packing problem motivated by a real one encountered at the Aberdeen harbour and is called Branching Packing Problem (BPP). The problem arises when boats must be loaded with containers of goods without knowing the exact details about the cargo and is encountered by vessels all over the world. It differs from a classical packing problem because of the uncertain information about the loading.

After studying the problem and the theoretical tools necessary to describe and solve it – such as Constraint Satisfaction Problems (CSPs), Branching Constraint Satisfaction Problems (BCSPs), Genetic Algorithms (GAs) presented in the chapters 2, 3 and 4 respectively – the problem has been proven to be NP-hard and, as such, better solvable by an approximation algorithm. The description of the problem, its formal model and the proof of its NP-hardness can be found in Chapter 5.

A Genetic Algorithm for BPPs has been designed and implemented in ANSI C. The description of this GA can be found in Chapter 6 and one of its implementation in Chapter 10. During the design phase we found convenient first to design a GA for BCSPs and then to adapt it to the case of BPPs. The GA for BCSPs can be found in Chapter 7.

Extensive experiments have been carried out in order to find the best parameters for the GA for BPPs. This task was very difficult and the search has been guided mainly by good sense, as discussed in Chapter 8.

Experiments have been done to estimate the goodness of the GA in solving the problem and are reported in Chapter 9.

Finally this report has been written with the supervision of Prof. Matteo Fischetti of the "Dipartimento di Ingegneria Elettronica e Informatica" of the University of Padua.

1.2 Summary

In summary, the project has been carried out in the following phases:

- the study of CSPs, BCSPs and GAs and the analisys of the problem of the harbour
- the design of the GA for BCSPs and for BPPs and the implementation of the GA for BPPs
- the experiments to find the best parameters and to test the goodness of the GA for BPPs
- the writing out of this report by using LaTeX

Chapter 2

Constraint Satisfaction Problems (CSPs)

2.1 Overview

In this Chapter the Constraint Satisfaction Problems ([9],[10]) and the main ways to solve them are quickly outlined. CSPs have great practical importance. Many real life problems, in particular scheduling, timetabling, packing and other combinatorial problems can be modeled as CSPs ([9]). The problem faced in this project can be viewed as a particular CSP, as it is shown in Chapter 5.

2.2 Constraint Satisfaction Problems (CSPs)

Definition 1 A *Constraint Satisfaction Problem (CSP)* is

$$(D_1, .., D_m, \mathcal{C})$$

with

- $D_1, .., D_m$ finite and non empty sets, with $m \in \mathbb{N}_0$
- $\mathcal{C}$ a set of $(I, \mathcal{I})$ such that $I \subset \{1, .., m\}, I \neq \emptyset$ and $\mathcal{I} \subset \{f \mid f : I \rightarrow \bigcup_{i \in I} D_i, i \longmapsto d \in D_i\}$, $\mathcal{C} \neq \emptyset$

The elements of $X = \{1, .., m\}$ are called *variables* of the CSP; for each variable i, D_i is called the *domain* of variable i; the elements of $\mathcal{C}$ are called *constraints* of the CSP; for each $(I, \mathcal{I}) \in \mathcal{C}$, $|I|$ is the *arity* of constraint $(I, \mathcal{I})$;

we say that a constraint with arity 1 (2) is an *unary* (*binary*) constraint; if $I \subset X, I \neq \emptyset$ an *assignment* of the variables I is a function

$$\begin{array}{c} f : I \longrightarrow \bigcup_{i \in I} D_i \\ i \longmapsto d \in D_i \end{array}$$

which is *partial* iff $|I| < |X|$; we also say that a constraint c *concerns* variables I iff $c = (I, \mathcal{I})$; the set $D_1 \times .. \times D_m$ is called *search space*. $\square$

Definition 2 Given a constraint $(I, \mathcal{I})$ and f an assignment of the variables J such that $I \subset J$, we say that f *satisfies* $(I, \mathcal{I})$ iff $f_I \in \mathcal{I}$ with f_I the restriction of f to I. Given a CSP C and f an assignment of variables I, we say that f is *feasible* for C iff $\forall I' \subset I f$ satisfies all the constraints concerning I'. Given a partial assignment f of variables I, variable $j \notin I$ and $v \in D_j$ we call the *extension of f to j with v* the function $\bar{f}$ defined in $\bar{I} = I \cup \{j\}$ such that $\bar{f} = f$ in I and $\bar{f}(j) = v$. An f is a *solution* of a given CSP iff f is an assignment of the variables X that satisfies all the constraints. $\square$

Note that the solutions of a CSP are a subset of the cathesian product[1] $D_1 \times .. \times D_m$ caracterized by the constraints; each constraint restricts the possible solutions by allowing only the m-tuple some of whose restrictions satisfy some conditions, i.e. the m-tuples that satisfy the constraint; the solutions are then the intersection all the sets of m-tuples that satisfy some constraint. The constraints, then, define a subset of the search space $D_1 \times .. \times D_m$.

Note that $\mathcal{C}$ is finite, because the set of $I \subset X$ is finite and the set of assignmets of variables I is finite too and so is the number of subsets of the assignments of variables I. The number of solutions is $\leq |D_1| \cdot .. \cdot |D_m|$ and there may be no solution at all. Given a CSP we are interested in finding whether it has solutions and if it does, in finding one (or all) of its solutions.

The following property holds ([11]).

Proposition 1 *For each CSP C, there exists a CSP C' with only binary or unary constraints whose solutions are in biunivocal correspondence with those of C.*

Proof. Let $C = (D_1, .., D_m, \mathcal{C})$ and $C' = (D_1, .., D_m, D_{m+1}, \mathcal{C}')$ where

$$D_{m+1} = D_1 \times .. \times D_m$$

[1] An element $(x_1, .., x_m)$ of a cathesian product $X_1 \times .. \times X_m$ is in fact a function that associates each $i \in \{1, .., m\}$ a value in X_i.

and

$$\mathcal{C}' = \{(\{m+1\}, I_{m+1})\} \cup \{(\{i, m+1\}, I_i) \mid i = 1..m\}$$

with I_{m+1} = the set of assignments of variable $m+1$ that associate it with a solution of C and $\forall i = 1..m : I_i = \{f \mid f$ is an assignment of variables $\{i, m+1\}$ such that $f(i) = i$-th component of $f(m+1)\}$. Now, C is a CSP – note that D_{m+1} is finite – and $(x_1, .., x_m, u) \in \mathcal{S}_{C'}$ iff $\forall c' \in \mathcal{C}' : (x_1, .., x_m, u)$ satisfies c' that is iff $u \in \mathcal{S}_C$ and $\forall i : x_i = i$-th component of u that is iff $(x_1, .., x_m) = u$ with $u \in \mathcal{S}_C$ that is iff $(x_1, .., x_m) \in \mathcal{S}_C$ and $u = (x_1, .., x_m)$. Then the map

$$\begin{aligned} f : \mathcal{S}_{C'} &\longrightarrow \mathcal{S}_C \\ (x_1, .., x_m, (x_1, .., x_m)) &\longmapsto (x_1, .., x_m) \end{aligned}$$

is a one to one correspondence between $\mathcal{S}_C$ and $\mathcal{S}_{C'}$. $\square$

So we can concentrate our study on only with only binary or unary constraints.

2.3 K-Consistency

Definition 3 A CSP is said to be *1-consistent* iff $\forall i and \forall v_i \in D_i$, the assignment that associates variable i to v_i satisfies all the constraints concerning variable i. Given a CSP and $k \in \{2, .., m\}$, we say that the CSP is *k-consistent* ([10]) iff $\forall I \subset \{1, .., m\}, |I| = k-1$ and $\forall f$ assignment of variables I feasible for C and $\forall j \in \{1, .., m\} \setminus I$, there $\exists v \in D_j$ such that the extension of f to j with v is feasible for C. We say that the CSP is *node consistent (NC)* if it is 1-consistent; that it is *arc consistent (AC)* if it is 2-consistent; that it is *path consistent (PC)* if it is 3-consistent. A CSP is said to be *strongly k-consistent* iff $\forall j \in \{1, .., k\}$ it is j-consistent. $\square$

In other words a CSP is NC iff for each variable i, each value of domain D_i satisfies[2] all the constraints concerning variable i. A CSP is AC iff for each variable i and for each value v_i of domain D_i such that v_i satisfies all the constraints concerning variable i and for each variable $j \neq i$, there exists a value v_j such that v_j satisfies all the constraints concerning variable j and the f assignment of variables $\{i, j\}$ such that $f(i) = v_i$ and $f(j) = v_j$ satisfies all the constraints concerning variables $\{i, j\}$. A CSP is PC iff for every couple of different variables i and j and for each f assignment of variables $\{i, j\}$

[2]When we say that the value v of domain D_i satisfies the constraints c we mean that the trivial function that associates i to v satisfies c.

that satisfies all the constraints concerning variables i, j and $\{i,j\}$ and for each variable $k \in \{1,..,m\} \setminus \{i,j\}$ there exists a value $v_k \in D_k$ satisfying all the constraints concerning k such that the assignment g of variables $\{i,k\}$ such that $g(i) = f_i$ and $g(k) = v_k$ satisfies all the constraints concerning variables $\{i,k\}$ and the assignment h of variables $\{j,k\}$ such that $h(j) = f_j$ and $h(k) = v_k$ satisfies all the constraints concerning variables $\{j,k\}$.

Proposition 2 *If a CSP is strongly m-consistent, then it has a solution and a solution $(x_1,..,x_m)$ can be simply computed by the algorithm of figure 2.1, where $D_j = \{v_1^j,..,v_{n_j}^j\}, \forall j \in \{1,..,m\}$.*

Proof. The proof is simple, as $\forall j$ there $\exists v_i^j$ such that $\forall I \subset \{1,..,j\}, j \in I(x_1,..,x_j)$ satisfies all the constraints concerning I, because the CSP is j-consistent and if $j \geq 2$, $(x_1,..,x_{j-1})$ is a partial assignment satisfying all the constraints concerning any subset of its domain. $\square$

```
{     for(j ← 1 to m)
      {     i ← 0;
            repeat
            i ← i + 1;
            x_j ← v_i^j;
            until((x_1,..,x_j) is feasible for C);
      }
      return (x_1,..,x_m);
}
```

Figure 2.1: The procedure to make a CSP C strongly k-consistent.

Proposition 3 *Given a CSP C and $k \geq 2$ then there exists a $k-consistent$ CSP C' with the same solutions and the same domains obtained by eventually adding some constraints concerning new sets of variables or by removing some assignments from the sets of some constraints.*

Proof. If C is k-consistent, then C'=C. Else the set $G = \{g \mid \exists J$ set of $k-1$ variables, g assignment of J feasible for C, $\exists j \in X \setminus J$ such that $\forall v \in D_j$ the extension of g to j with v is not feasible for $C\}$ is not empty. Let C' be the CSP obtained from C by adding the set of constraints $\{(J, \mathcal{J}) \mid \exists g \in G$

such that J is the domain of g and $\mathcal{J}$ is the set of all the assignments f of J such that $f \neq g\}$. Then this C' is k-consistent, because it is C without all the assignments that prevented it from being k-consistent. And it has the same solutions, because we have only eliminated partial assignmts of $k-1$ variables that could not be the restriction of any solution.

Note that C' may have more than one constraint concerning the same set of variables. In this case we can replace all the constraints concerning the same set of variables by one that concerns the same variables and whose set of assignments is the intersection of all of the their set of assignments. So the new C' is just C with some new constraints and with some constraints with less elements in its set of assignments. $\square$

Many algorithms have been designed to build such a C' from a given C and k. All of them have an exponential running time.

Proposition 4 *If we have a k-consistent CSP and we make it h-consistent with $2 \leq h < k$ by the method of Proposition 3, we obtain a CSP which is still k-consistent.*

Proof. Let C be a k-consistent CSP and $G = \{g \mid \exists J$ set of $h-1$ variables, g assignment of J feasible for C, $\exists j \in X \setminus J$ such that $\forall v \in D_j$ the extension of g to j with v is not feasible for $C\}$ is not empty. Let C' be the CSP obtained from C by adding the set of constraints $\{(J, \mathcal{J}) \mid \exists g \in G$ such that J is the domain of g and $\mathcal{J}$ is the set of all the assignments f of J such that $f \neq g\}$. Then C' is still k-consistent. As a matter of fact for every f assignment of a set of $k-1$ variables I feasible for C', f is feasible for C as well. Thus for the k-consistency of $C, \forall j \in X \setminus I$ there $\exists v_j \in D_j$ such that the $\bar{f}$ extension of f to j with v_j is fesible for C. Now, every restriction of $\bar{f}$ is $\notin G$. (As a matter of fact let f' be a restriction of $\bar{f}$ to a $I' \subset \bar{I}$ with $|I'| = h-1$. Then there exists I'' such that $I' \subset I'' \subset \bar{I}$ and $|I''| = k-1$. The restriction of $\bar{f}$ to I'' is feasible for C. Thus we know that $\forall j \notin I''$ there exists v''_j such that the extension of f'' to j with v''_j is feasible for C. So for every $j \notin I'$ if $j \in \bar{I}$ the extension of f' to j with $\bar{f}(j)$ is feasible for C; else the extension of f' to j with v''_j is feasible for C. So $f' \notin G$.) So every restriction of $\bar{f}$ satisfies all the constraints of C concerning its domain and is $\notin G$. So $\bar{f}$ is feasible for C' too. Hence the k-consistency of C'. $\square$

Proposition 5 *Given a CSP C and $k \in \{2, .., m\}$, then there exists a CSP C', j-consistent $\forall j \in \{2, .., k\}$, with the same solutions and the same domains and obtained by eventually addying some constraints concerning new sets of variables or by removing some assignments from the sets of some constraints.*

Proof. We just have to apply the method of Proposition 3 iteratively for $j = k, k-1, .., 2$ and we will obtain C'. Then $\forall j \in \{2, .., k\}$, C' is j-consistent because of Proposition 4 and, because of proposition 3 it has the same solutions and the same domains and it is obtained from C by adding some new constraints or by removing some elements from the set of assignments of some constraints . □

```
{    find j ∈ X and v ∈ D_j such that all the constraints
                 concerning j are satisfied by v;
     let {j_1, .., j_m} be an ordering of the variables X
           such that j_1 = j;
     v_{j_1} ← v;
     for(h ← 2 to m)
     {    i ← 0;
          repeat
          i ← i + 1;
          x_{j_h} ← v_i^{j_h};
          until((x_{j_1}, .., x_{j_h}) is feasible for C);
     }
     return (x_1, .., x_m);
}
```

Figure 2.2: The procedure to solve a $2, .., m$-consistent CSP with solution.

Proposition 6 *Let C be a CSP j-consistent, $\forall j \in \{2, .., m\}$. Then C has a solution iff $\exists j \in X$ and $v \in D_j$ such that v satisfies all the constraints concerning variable j. If C has a solution the algorithm of Figure 2.2 quicly finds a solution.*

Proof. If C has a solution x, then if we choose a $j \in X$ obviously $x(j)$ satisfies all the constraints concerning variable j. On the other hand, if $\exists j \in X$ and $v \in D_j$ such that v satisfies all the constraints concerning variable j, we can build a solution by the algorithm of Figure 2.2 where $D_j = \{v_1^j, .., v_{n_j}^j\}, \forall j \in \{1, .., m\}$. Note that in every iteration of the while loop the $|I|$-consistency of the CSP guarantes that v_i^h is found for some i. □

2.4 Algorithms to solve CSPs

The problem of finding a solution of a CSP or proving that there are no solutions is simple from the theoretical point of view. As a matter of fact we can order all the elements of $D_1 \times .. \times D_m$, generate them one after the other and for each of them check if it satisfies all the constraints until one that satisfies them is found or until we have checked all the m-tuples. This approach is called Generate and Test (GT). Yet from the practical point of view the GT is often useless, as real problems have very large domains and the time it takes us to generate and check all the m-tuples is too large. So we need more clever ways to address this task. However every algorithm that guarantes to find a solution if one exists or to prove that the problem is unsolvable must explore systematically the search space and so it may run for a long time.

2.4.1 Pruning algorithms

The fact that some constraints do not concern all the variables often lets us understand that a whole set of m-tuples is unfeasible. In other words the presence of more than one constraint and of constraints with an arity less than m gives us the possibility to understand that some partial assignments of the variables bring necessarily to unfeasible total assignments. Then if we avoid the generation and the constraint checking of all the solutions that are a completion of an unfeasible partial assignment (i.e. if we prune the branches of the tree of the possible solutions that we have understood lead to unfeasible leaves) we save a lot of time. Many algorithms that do this have been developed. The Back Tracking (BT) and the Forward Checking (FC) algorithms ([9]) are the most commonly used.

Backtracking

This algorithm returns all the solutions (if one exists) otherwise it states that there are no solutions.

We first assign a value v_1 to variable 1 that satisfies the constraints concerning variable 1; then we look for a value v_2 for variable 2 such that (v_1, v_2) is feasible for C; if none is found, we go back and change the previous value for variable 1 and do the same again; else we look for a value v_3 for 3 such that (v_1, v_2, v_3) is feasible for C; if no v_3 is found we go back to find another v_2 and repeat the same; if no v_2 exists, we go back again to find a new v_1. And so on.

A possible pseudocode for Backtracking is in Figure 2.3.

This algorithm is better than the GT because it avoids the useless generation and test of the m-tuples derived from the completion of an unfeasible partial assignment of variables $\{1, 2, .., i\}$ for some i.

```
f(v, i)
{     x_i ← v;
      if ((x_1, .., x_i) is feasible for C)
            if (i = m) put (x_1, .., x_m) in I;
            else for each (v ∈ D_{i+1}) f(v, i + 1);
}

BT
{     read the CSP C;
      let I be an initially empty set of solutions;
      let (x_1, .., x_m) be an assignment of variables X;

      for each (v ∈ D_1) f(v, 1);

      if (I = ∅) return "No Solution";
      else return I;
}
```

Figure 2.3: The Backtracking algorithm in recursive form.

Forwardchecking

This algorithm is an evolution of the previous one. It assumes that each value is associated with a set of variables initially empty. The set of variables corresponding to a value can be modified during a run of the algorithm. And variables con be assignd only to values with an empty set at the moment of the assignment.

Initially all the sets are empty. We first assign a value v_1 to variable 1 that satisfies the constraints concerning variable 1; then for all $i = 2, .., m$ we consider all the values $v \in D_i$ and if the extension of (v_1) to i with value v is not feasible for C, we put variable 1 on the set of value v; if one domain has all its values with non empty sets or if no value $v_2 \in D_2$ with empty set such that (v_1, v_2) is feasible for C is found, we delete all the 1's from the sets of the values of domains $D_2, .., D_m$ and we go back, choose another value for variable 1 and repeat the same; else we assign v_2 to variable 2 and for all $i = 3,..,m$ we consider all the values $v \in D_i$ and if the extension of (v_1, v_2) to i with v is not feasible for C, we put variable 2 on the set of value v; if one domain has all its values with non empty sets or if no value $v_3 \in D_3$ with empty set such that (v_1, v_2, v_3) is feasible for C is found, we delete all the 2's from the sets of the values of domains $D_3,..,D_m$ and we go back, choose another value for variable 2 and repeat the same; if no value for variable 2 is found, we delete all the 1's from the sets of the values of domains $D_2,..,D_m$ and we to go back again and change again variable 1. And so on.

The forward checking algorithm as well returns all the solutions if one exists otherwise it states that there are no solutions. A pseudocode for Forwardchecking is in Figure 2.4.

Apparently the FC does more constraint checkings than the BT, because it must update the sets of variables associated to the values; but sometimes it avoids some checkings that the BT would do, as it prunes bigger branches. As a matter of fact the FC realizes that a partial assignment is not part of any solution earlier than the BT. More precisely, given a CSP in input to BT, if it realizes that $(v_1,..,v_i)$ cannot be part of any solution, then the FC, if run on the same CSP, will realize that $(v_1,..,v_j)$ cannot be part of any solution with $j \leq i$. So FC will avoid the generation and valuation of the sequences of values that BT has generated and checked between $(v_1,..,v_j)$ and $(v_1,..,v_i)$.

Which is faster depends on the particular CSP.

2.4.2 Solving a CSP by maximizing a function

The problem of finding a solution of a CSP can be viewed as that of finding a point of maximum of a function. As a matter of fact, given a CSP $(D_1,..,D_m,\mathcal{C})$, we can consider the function

$$\begin{gathered} f : D_1 \times .. \times D_m \to \mathbb{R} \\ (x_1,..,x_m) \longmapsto f(x_1,..,x_m) \end{gathered}$$

where $f(x_1,..,x_m)$ is the number of constraints not satisfied by $(x_1,.,x_m)$ or, more generally, $f(x_1,..,x_m) = \sum_{c \in K} w(c)$ with K the set of constraints violated by $(x_1,..,x_m)$ and $w(c)$ a real number > 0 associated to c. Then $\vec{x}$ is a solution of the CSP iff $\vec{x}$ is a point of maximum of f. So we can solve the CSP by using an algorithm that tries to find a point of maximum of this function. In this case imf is finite and $|imf| = |\mathcal{C}|$.

```
f(v, i)
{   x_i ← v;
    if ((x_1, .., x_i) is feasible for C)
        if (i = m) put (x_1, .., x_m) in I;
        else
        {   for each(j ∈ {i + 1, .., m})
                for each(v ∈ D_j)
                    if (the extension of (x_1, .., x_i) to j with v
                    is unfeasible for C)
                    add i to the set of value v;

            for each (v ∈ D_{i+1} with set of v empty) f(v, i + 1);

            for each(j ∈ {i + 1, .., m})
                for each(v ∈ D_j) remove i from the set of v;
        }
}

FC
{   read the CSP C;
    let I be an initially empty set of solutions;
    set to ∅ the set of all the values;
    let (x_1, .., x_m) be an assignment of variables X;

    for each (v ∈ D_1) f(v, 1);

    if (I = ∅) return "No Solution";
    else return I;
}
```

Figure 2.4: The Forwardchecking algorithm in recursive form.

2.4.3 Heuristic and stochastic algorithms

Instead of exploring systematically the search space by considering all the elements of $D_1 \times .. \times D_m$ in some way, some algorithms explore the search space in a more or less random fashion. They do not guarante to find a solution nor they prove that the problem is unsolvable and they may run forever. During the exploration the choice of the next possible solution to consider is influenced in part by the chance and in part by a deterministic heuristic rule. These algorithms usually face the solution of a CSP from the point of view of the maximization of a real function as presented in section 2.4.2. The most common heuristic and stochastic approaches are those of Taboo Search, Hill Climbing, Min conflict, Genetic Algorithms and Evolution Strategies ([10]). Yet many others are used and new ones are designed by mixing different algorithms.

2.4.4 Consistency techniques

In order to solve a given CSP, we can make it $2, 3, .., m$-consistent by the algorithm of Proposition 5, check if there exists $j \in X$ such that $\exists v \in D_j$ that satisfies all the constraints that concern j and then, if it exists, a solution exists and we can find one by the algorithm of Proposition 6; else no solution exists.

Yet making the CSP $2, .., m$-consistent is a very complex process and even with the quickest algorithms it takes often too long, so this procedure is rarely used. Usually we only make the CSP $2, .., k$-consistent for some $k \in \{2, .., m\}$ by the algorithm of Proposition 5 and then we solve the new problem by one of the previous search algorithms. We do so, because the new problem is simpler to solve, as the tightened constraints and the new added ones make that the search algorithm refuses bigger subsets of unfeasible total arrignements.

2.5 Optimization CSPs (OCSPs)

Definition 4 An *Optimization CSP (OCSP)* is a (P, f) with P a CSP and $f : D_1 \times .. \times D_m \to \mathbb{R}$. $\square$

Definition 5 A *solution* of a given OCSP (P, f) is a solution of P. A solution $\bar{x}$ of a given OCSP is *optimal* iff $\forall x$ solution of the OCSP, $f(\bar{x}) \geq f(x)$. $\square$

Note that if an OBCSPs has solutions it has an optimal solution, because the set of solutions is finite. We are interested in finding whether a solution exists and if one exists in finding an aptimal one or all the optimal ones. Sometimes we are just interested in a good solution.

2.5.1 Exact algorithms

The problem of finding an optimal solution of an OCSP is obvioulsy at least as hard as the problem of finding a solution of a CSP.

The simplest algorithm we can imagine is a variant of the GT: we generate all the values of $D_1 \times .. \times D_m$ in some order and for each of them we check if it is a solution of the corresponding CSP; if it is not, we go on; else we calculate its value and if this is the best value found so far we memorize the solution and we go on. We can do the same with BT and FC: we find a solution x of the correspondent CSP by one of these algorithms and we memorize it if $f(x)$ is better than the values found so far and then we continue the search for a new solution. We stop when we have checked all the solutions of the CSP.

The standard algorithm

The *standard* algorithm is slightly more sophisticated and is shown in Figure 2.5. It receives in input a CSP and it returns an optimal solution if it exists else it states that there is no solution.

This algorithm, in order to solve the CSPs built before returning the output, makes use of one of the algorithms presented in Section 2.4 .

The dicotomic algorithm

In order to use the *dicotomic* algorithm we need to know an upper bound of the possible values assumed by the solutions of the correspondent CSP, i.e. a value U_0 such that $\forall x$ solution of the CSP, $f(x) < U_0$. This algorithm receives in input a CSP, an $\epsilon > 0$ and an upper bound U_0 and if the correspondent CSP has no solution it states that theres no solution, else it returns a solution x such that if $\bar{x}$ is the optimal solution, then $f(\bar{x}) \in [f(x), f(x)+\epsilon[$.

The dicotomic algorithm is then an approximation algorithm, as it returns a solution as good as we want, but it does not guarante that it is optimal. Yet, because the search space is finite, if f is not constant and the search space is made of more than one element, there exists the number

$$\epsilon_0 = \min\{|f(x) - f(y)| \mid x, y \in D_1 \times .. \times D_m, f(x) \neq f(y)\}$$

and if we give $\epsilon < \epsilon_0$ to the dicotomic algorithm, it returns an optimal solution or it states that the problem has no solution.

```
{   read (C, f);

    s ← NO;
    while(C has a solution)
    {   s ← YES;
        let x_0 be a solution of C;
        add to C the constraint
            (X, {x|x assignment of variables X, f(x) > f(x_0)});
    }
    if(s=NO) return "No Solution";
    else return x_0;
}
```

Figure 2.5: The standard algorithm.

The pseudocode for the dicotomic algorithm is presented in Figure 2.6. Note that the dicotmic algorithm, in order to solve the CSPs built before returning the output, makes use of one of the algorithms presented in Section 2.4.

2.5.2 Aproximation algorithms

Sometimes the OCSP is too difficult to be solved by an exact search algorithm. In this case we accept to find a good solution instead of the best one. A good solution could be defined informally as one that is much better than a randomly found solution. To do so we use algorithms based on heuristic and stochastic techniques in the same way as the algorithms of Section 2.4.3. The search space is explored in a more or less clever and random way in the search for better and better solutions of the corresponding CSP.

Taboo Search, Hill Climbing, Genetic Algorithms, Evolution Strategies are the most commonly used aproximation approaces to solve OCSPs. Genetic Algorithms are presented in Chapter 4.

```
{   read (C, f), ε, U_0;

    if(C has no solution) return "No Solution";
    let x_0 be a solution of C;
    L ← g(x_0);
    U ← U_0;
    repeat
        M ← (U+L)/2;
        add to C the constraint
            (X, {x|x assignment of variables X, g(x) ≥ M});
        if(C has a solution)
        {   let x_0 be a solution of C;
            L ← g(x_0);
        }
        else U ← M;
    until(U − L ≤ ε);
    return x_0;
}
```

Figure 2.6: The dicotomic algorithm.

Chapter 3

Branching Constraint Satisfaction Problems (BCSPs)

3.1 Overview

In this Chapter the Branching Constraint Satisfaction Problems defined in [8] are introduced. It is shown that they can be viewed as particular CSPs. A simple extension of this model is then presented. The problem tackled in this work can be considered as a particular BCSP, as shown in Section 5.7.

3.2 Branching Constraint Satisfaction Problems (BCSPs)

Definition 6 A *Branching Constraint Satisfaction Problem (BCSP)* is

$$(P, \vec{u}, T, p, c)$$

where

- $P = (D_1, .., D_m, \mathcal{C})$ is a CSP such that $\forall i = 1, .., m, NULL \notin D_i$
- $\vec{u} \in \mathbb{R}^{+m}$
- $T = (N, E, \rho)$ a rooted tree with $n = |N| \geq 2$
- $p : E \rightarrow]0, 1]$ such that $\forall \alpha \in N$, if O is the set of the outgoing edges of α, then $\sum_{a \in O} p(a) \leq 1$;
- $c : N \rightarrow V$, with V the variables of P, such that $\forall \alpha \in N$:

1. if J is the set of the children of α, then $\forall \gamma, \delta \in J$ such that $\gamma \neq \delta, c(\gamma) \neq c(\delta)$;
2. if H is the set of the nodes of the path from the root to α, then $\forall \gamma, \delta \in H$ such that $\gamma \neq \delta, c(\gamma) \neq c(\delta)$;

The rooted tree T is called the branching tree; for $a \in E$, $p(a)$ is called the probability of edge a; for $\alpha \in N$, $c(\alpha)$ is called the variable of node α. $\square$

As a consequence of the constraints on p and c, it is clear that the number of nodes $n = |N|$, the depth d of T, the degree $\delta(\alpha)$ and the level $l(\alpha)$ of each node α and the number of variables $m = |V|$ of problem P are slightly related one to the other[1]. In particular the following relations hold:

- $n \leq (m-1)!$
- $d \leq m$
- $\forall \alpha \in N, \delta(\alpha) \leq m + 1 - l(\alpha)$

3.3 Assignments and feasible assignments

Definition 7 An *assignment* of a given BCSP is a function

$$\psi : N \to \bigcup_{i=1}^{m} D_i \cup \{NULL\}$$

such that $\forall \alpha \in N, \psi(\alpha) \in D_{c(\alpha)} \cup \{NULL\}$. An assignment ψ is *feasible* iff $\forall \alpha$ leaf, if $B = \{\beta \in N \mid \psi(\beta) \neq NULL$ and β is of the path from the root to $\alpha\}$ and $I = \{c(\beta) \mid \beta \in B\}$ then the assignment of variables I such that $\forall i \in I$ it is $f(i) = \psi(\beta)$ with $\beta \in B$ and $c(\beta) = i$ is feasible for P. A solution is a feasible assignment. $\square$

Note that the function $\psi \equiv NULL$ is always a feasible assignment. If ψ is a feasible assignment then $\forall S \subset N$ the assignment ψ_S such that $\psi_S = \psi$ in $N \setminus S$ and $\psi_S = NULL$ in S is feasible too. And $\forall \psi$ assignment there exists $S \subset N$ such that ψ_S is feasible. Setting to $NULL$ some nodes of an

[1] In this thesis it will be used the terminology of [4]. In particular the level of a node is defined recursively as follows: the level of the root is 0; the level of a node different from the root is the level of the father plus 1. The depth of a node is the level of the node plus 1; the depth of a tree is the maximum depth of its nodes. The outgoing edges of a node are those that connect it to its children; the degree of a node is the number of its outgoing edges.

unfeasible assignment is then a way to make it feasible, i.e. it is a way to find a feasible assignment similar to it. This properties will be used in Chapter 7.

The number of assignments is

$$\prod_{\alpha \in N} (1 + |D_{c(\alpha)}|)$$

The number of feasible assignments is obviously less than or equal to this number.

3.4 Expected Utility (EU)

Definition 8 Given a BCSP, a node α of its tree is said to be *terminal* iff $\sum_{a \in O(\alpha)} p(a) < 1$ with $O(\alpha)$ the outgoing edges of α. If α is a terminal node, we define $P_\alpha = 1 - \sum_{a \in O(\alpha)} p(a)$. $\square$

As a consequence of Definition 8, the following properties hold:

1. all the leaves are terminal nodes, as if α is a leaf $O(\alpha) = \emptyset$ and so $\sum_{a \in O(\alpha)} p(a) = 0$
2. $\forall \alpha$ leaf it is $P_\alpha = 1$
3. $\forall \alpha$ node it is $0 < P_\alpha \leq 1$

Definition 9 Given a BCSP and one of its feasible assignments ψ, the *expected utility (EU)* of ψ is the number

$$EU(\psi) = \sum_{\alpha \in L} \Big(\sum_{\beta \in I(\alpha)} u(c(\beta)) \Big) \Big(\prod_{a \in A(\alpha)} p(a) \Big) P_\alpha$$

with

- L the set of the terminal nodes of T
- $\forall \alpha \in L : I(\alpha) =$ the set of the nodes β of the path from the root to α such that $\psi(\beta) \neq NULL$;
- $\forall \alpha \in L : A(\alpha) =$ the set of the edges of the path from the root to α

$\square$

Because the number of feasible assignments is finite, the set

$$I = \{EU(\psi) | \psi \text{ is a fesable assignment}\}$$

is also finite. Hence there exists $\bar{\psi}$ a feasible assignment such that $\forall\psi$ feasible assignment, $EU(\bar{\psi}) \geq EU(\psi)$. It is then not unreasonable to be interested in finding a feasible assignment with the highest EU. In paper [8] some algorithms for this pourpose are presented. In Chapter 7 a GA that tries to find solutions with a good EU will be presented. A feasible assignment with the highest EU is called *optimal solution*.

Given a BCSP, the set I defined above has an upper bound given by the formula

$$UB = \sum_{\alpha \in L} \Big(\sum_{\beta \in J(\alpha)} u(c(\beta)) \Big) \Big(\prod_{a \in A(\alpha)} p(a) \Big) P_\alpha$$

with

- L the set of the terminal nodes of T
- $\forall \alpha \in L, J(\alpha) =$ the set of the nodes β of the path from the root to α;
- $\forall \alpha \in L, A(\alpha) =$ the set of the edges of the path from the root to α

which is a number depending only on the BCSP and which corrisponds to the EU of an hypothetical feasible assignment that assign all the nodes with a value different from $NULL$. Thus $\forall\psi$ feasible assignment of the BCSP, $EU(\psi) \leq UB$ and if we find a feasible assignment ψ such that $EU(\psi) = UB$ then ψ is a solution with the highest expected utility.

Recursive computation of $EU(\psi)$

Given a BCSP and one of its feasible assignments ψ, $EU(\psi)$ can be calculated recursively. In fact we can define $v_\psi : N \to \mathbb{R}$ such that $\forall \alpha \in N$

1. if α is a leaf of T then

$$v_\psi(\alpha) = \begin{cases} 0 & \text{if } \psi(\alpha) = NULL \\ u_{c(\alpha)} & \text{else} \end{cases}$$

2. else

$$v_\psi(\alpha) = \begin{cases} \sum_{\beta \in C(\alpha)} v_\psi(\beta) p_\beta & \text{if } \psi(\alpha) = NULL \\ u_{c(\alpha)} + \sum_{\beta \in C(\alpha)} v_\psi(\beta) p_\beta & \text{else} \end{cases}$$

   with $C(\alpha)$ the children of α and p_β the probability of the edge that connects β with its father α.

Then $EU(\psi) = v_\psi(\rho)$ with ρ the root of T.

This is the original definition of $EU(\psi)$ in the paper [8].

Computation of $EU(\psi)$ by matrices

If we fix an ordering of the nodes $N = \{N_1, .., N_n\}$ and one of the leaves $F = \{F_1, .., F_k\}$ of T, then there exists a vector $\vec{y}$, dependent only on the BCSP and on the orderings of N and of F, such that $\forall\psi$ feasible assignment of the BCSP, $EU(\psi) = \vec{y}^T\vec{b}$ with $\vec{b} = (b_1, .., b_n)$ whereas $\forall i = 1..n$

$$b_i = \begin{cases} 0 & \text{if } \psi(N_i) = NULL \\ 1 & \text{else} \end{cases}$$

Thus if we have $\vec{y}$, the calculation of $EU(\psi)$ consists on the simple computations of $\vec{b}$ and of the product $\vec{y}^T\vec{b}$. The vector $\vec{y}$ is given by the formula:

$$\vec{y} = F^T\vec{p}$$

where

1. $p = (p_1, .., p_k)$ with p_i the product of the probabilities of the edges in the path from the root to the terminal node F_i and of P_{F_i}
2. F the matrix of order $k \times n$ such that $\forall(i, j) \in \{1, .., k\} \times \{1, .., n\}$

$$F_{i,j} = \begin{cases} u_{c(N_j)} & \text{if } N_j \text{ is in the path from the root to } F_i \\ 0 & \text{else} \end{cases}$$

In other words, given a BCSP C, there exists a function $G : N \to \mathbb{R}^+$ such that $\forall\psi$ solution of C it is

$$EU(\psi) = \sum_{\alpha\in N} G(\alpha)b_\psi(\alpha)$$

with

$$b_\psi(\alpha) = \begin{cases} 0 & \text{if } \psi(\alpha) = NULL \\ 1 & \text{else} \end{cases}$$

Moreover it is $UB = \sum_{\alpha\in N} G(\alpha)$, with UB the upper bound defined above. The function G is given by the formula

$$G(\alpha) = u(c(\alpha)) \cdot \sum_{\tau\in T(\alpha)} \Big(\prod_{a\in A(\tau)} p(a)\Big)P_\tau$$

with $T(\alpha) = \{\tau|\tau$ terminal node such that α is in the path[2] from the root to $\tau\}$ and $A(\tau) =$ the set of the edges of the path from the root to τ.

Note that function G is dependent only on the BCSP. So given the solution ψ in order to compute $EU(\psi)$, once we have function G, we just have to

{ $\forall \alpha \in N : G(\alpha) \leftarrow 0;$

$\forall \tau$ terminal node:
{ $\pi_\tau \leftarrow \prod_{a \in A(\tau)} p(a) \cdot P_\tau$
(with $A(\tau)$ the set of the edges of the path from the root to τ);

$\forall \alpha$ node of the path from the root to τ:
$G(\alpha) \leftarrow G(\alpha) + \pi_\tau$
}

$\forall \alpha \in N : G(\alpha) \leftarrow G(\alpha) \cdot u(c(\alpha));$
}

Figure 3.1: The algorithm to compute G.

compute the function b_ψ and the sum $\sum_{\alpha \in N} G(\alpha) b_\psi(\alpha)$. Function G can be calculated by the simple algorithm of Figure 3.1.

In spite of its notational complexity, this method is simple and it is the fastest way to compute $EU(\psi)$, once we have calculated G. It is particularly convenient when we have to compute the EU of several feasible assignments of the same BCSP.

3.5 BCSPs as CSPs

It is evident that for each BCSP B there exists a CSP C whose feasible assignments are in one to one correspondence with the feasible assignments of B. As a matter of fact, given a BCSP B, let $\{N_1, .., N_n\}$ be an ordering of its nodes. Let

$$C = (D'_1, .., D'_n, \mathcal{C}')$$

where $D'_i = D_{c(N_i)} \cup \{NULL\}$ and $\mathcal{C}' = \{(\{1, .., n\}, \mathcal{I})\}$ with $\mathcal{I} = \{(x_1, .., x_n) \in D'_1 \times .. \times D'_n |$ the function ψ such that $\forall i, \psi(N_i) = x_i$ is feasible for $B\}$. Then C is a CSP and the function

$$\phi : \text{feasible assignments of C} \rightarrow \text{feasible assignments of B}$$
$$(x_1, .., x_n) \longmapsto \psi | \forall i = 1..n, \psi(N_i) = x_i$$

[2]Note that τ is in the path from the root to τ and if α is a terminal node, then it is $\alpha \in T(\alpha)$.

is biunivocal. We call C the *CSP correspondent* to B.

Thus, we can find a solution of a given BCSP by one of the algorithms for finding a solution of a CSP. And we can find an optimal solution of a BCSP by one of the algorithms that find a solution of an OBCSP. As a matter of fact, given B if we define

$$\begin{aligned} f : D'_1 \times .. \times D'_n &\to \mathbb{R} \\ (x_1, .., x_n) &\longmapsto EU(\phi(x_1, .., x_n)) \end{aligned}$$

we have that $P = (C, f)$ is an OCSP and if $\vec{x}$ is an optimal solution of P then $\phi(\vec{x})$ is one of the optimal solutions of B.

Yet it does not mean that studying algorithms for the BCSPs is useless. Being a BCSP a very special CSP, we can exploit its peculiarities to reach our purpose faster: in other words there is the possibility that an algorithm for BCSPs solves a BCSP faster than an algorithm for CSPs solves the correspondent CSP.

3.6 The extended BCSPs (eBCSPs)

Sometimes the BCSP model seems unable to describe the whole complexity of a practical situation (see section 5.8). In some cases we need that the constraints concern the $NULL$ value as well and that the EU still treats it as a particular value. In these cases we can use a model slightly different from the BCSP that we can call *extended BCSP (eBCSP)*. In this model the $NULL$ value is necessarily in all the D_i and an assignment of an eBCSP is feasible iff $\forall\alpha$ leaf, if $B =$ the set of the nodes of the path from the root to $\alpha\}$ and $I = \{c(\beta) \mid \beta \in B\}$ then the assignment of variables I such that $\forall i \in I, f(i) = \psi(\beta)$ with $\beta \in B$ and $c(\beta) = i$ is feasible for P. All the other definitions remain the same.[3]

For every BCSP there exists an eBCSP whose solutions are those of the BCSP and have the same EU. As a matter of fact if $(D_1, .., D_m, \mathcal{C}, \vec{u}, T, p, c)$ is a BCSP, then $(D'_1, .., D'_m, \mathcal{C}, \vec{u}, T, p, c)$ with $D'_i = D_i \cup \{NULL\}$ is an eBCSP with the same feasable solutions and with the same EU. So the descriptive power of the eBCSP model is at least as good as that of the BCSP.

Note that the properties of section 3.3 do not hold for the eBCSPs. In fact there are eBCSPs for which the $NULL$ solution is not feasible. There are eBCSPs with a feasible solution ψ and a set of nodes S such that the solution ψ_S is not feasible. Thus the GA for BSCP that can be derived from

[3]Note that in this case some contraints of P may concern the NULL value as well, in the sense that some constraints may not be satisfied by some partial assignments that assign NULL to some variables.

the GA of Chapter 6 cannot be used to find an optimal solution of a given eBCSP.

Chapter 4

Genetic Algorithms (GAs)

4.1 Overview

In this Chapter a breaf and informal introduction to GAs is provided. The broader class of Evolutionary Programs (EPs) (see [1]) are introduced. GAs are presented as a subset of EPs and classical GAs as a subclass of GAs. Evolution Strategies (ES) are also introduced as a subset of EPs. The algorithm designed in this thesis is an EP and with special parameters – $shake_r = 1$, see Section 6.4 – it is a GA not classical. Anyway, whatever are the parameters, it has very strong similarities with GAs. For this reason it wil be called GA.

4.2 Introduction

GAs can be considered as a subclass of EPs. An EP can be viewed as an algorithm to find the optimal solution of a problem or equivalently as an algorithm to find the point of maximum of a real function defined on some set[1].

EPs are approximation algorithms as they do not guarantee to find the optimal solution, but they usually give a very good suboptimal solution.

They are stochastic algorithms in the sense that they more or less randomly visit the elements (called *solutions*) of the domain (called *search space*) of the function (called *objective function*) and return the visited solution with the highest value of the objective function. However the exploration of the search space is not totally random and these algorithms tend to concentrate the exploration in the promising areas of the search space, i.e. the areas with

[1]For some authors this is too restrictive a vision, see [1, page 16].

good visited solutions.

EPs imitate living beings in their fight for survival. All living beings manage to survive in a difficult environment by adapting their bodies to the environment. The adaptation to the environment is obtained through the Darwinian Natural Selection: the environment kills the individuals without the features necessary to survive in it; those who survive give their winning features to their offsprings; new features are introduced in the species by the mutation and reproduction phenomenona. Thanks to this simple mechanism some organisms have managed to survive in environments once forbidden to them.

An EP creates a population of individuals – representing solutions of the search space – living in a difficult environment. The measure of the adaptation to the environment of an individual (called *fitness* of the individual) is the value of the objective function on the solution represented by that individual. Some of these individuals mate producing offsprings with genetic heritage which is a mixture of that of their parents and new genes are introduced in the population by a mutation phenomenon. Some individuals with low fitness are killed by the environment. After some generations the average fitness of the population should increase. In this way the EP explores the search space and tries to find solutions with higher fitness.

EPs are useful to solve very hard optimization problems and in the last decade their importance has been growing, as they can be naturally implemented in parallel computers ([1]) that are now becoming available.

4.3 Evolutionary Programs (EPs)

An EP tries to find the point of maximum of a function $f : S \to \mathbb{R}$. Each EP is associated with an injective function $\phi : S \to \bar{R}$ called *encoding* of the solutions. The EP directly deals with the elements of the set $im\psi = R$ whose elements are called the *representations* of the solutions. Also $\forall x \in R, val(x)$ is defined as $f(\phi^{-1}(x))$ and is called the *fitness* (or the *value*) of x. An *individual* (also called *chromosome* or *phenotype*) is an element of $\bar{R}$. An individual is *feasible* iff it is in R, i.e. iff it represents a solution. A *population* is a tuple of individuals. An EP is also associated with a *population size* $\mu \in \mathbb{N}_0$, an *offspring size* $\lambda \in \mathbb{N}_0$.

The Evolutionary Program creates an initial population of μ feasible individuals $\vec{v}^0 = (v_1^0, .., v_\mu^0)$; then it creates new feasible populations $\vec{v}^1, \vec{v}^2, .., \vec{v}^k$ one after the other until a termination condition occurs; finally the individual with the highest fitness created until then is returned. The creation of a new population $\vec{v}^i$ is obtained in the following way: a population of λ

feasible individuals $\vec{w} = (w_1, .., w_\lambda)$ called the offspring population is created; the creation of $\vec{w}$ depends on the past populations $\vec{v}^0, .., \vec{v}^{i-1}$ and on random events; then a population $\vec{v}^i$ of μ individuals is created such that $\forall j \in \{1, .., \mu\}, v^i_j \in \{w_j | j = 1, .., \lambda\} \cup \{v^{i-1}_j | j = 1, .., \mu\}$ – i.e. $\vec{v}^i$ is obtained by choosing, even more than once, some individuals from those of the previous population $\vec{v}^{i-1}$ and of the offspring population $\vec{w}$. The pseudocode for the EPs is given in Figure 4.1

{ $\quad i \leftarrow 0$;
create $\vec{v}^0 \in R^\mu$;
$\beta \leftarrow$ the element of $\vec{v}^0$ with the highest fitness;

while(**not** termination condition)
{ $\quad i \leftarrow i + 1$;
$\vec{w} \leftarrow$ a population of λ feasible individuals depending on
populations $\vec{v}^0, .., \vec{v}^{i-1}$ and on random events;
$\vec{v}^i \leftarrow$ a population of μ individuals taken from those
of $\vec{w}$ and $\vec{v}^{i-1}$, even more than once;
if$\Big(\exists j$ such that $val(v^i_j) > val(\beta)\Big)$ $\beta \leftarrow v^i_j$;
}

return β;
}

Figure 4.1: The structure of an EP.

The creation of $\vec{v}^i$ out of $\vec{w}$ and $\vec{v}^{i-1}$ is called *selection*; the creation of $\vec{w}$ out of the past populations and of random events is called *recombination*.

The goodness of the EP in finding a good solution depends on the goodness of the selection and recombination procedures. In order to imitate the evolutionary process of nature, the genetic information of the offspring population must be both similar and different to that of the last populations – so as to keep the good features found until then and experiment new ones that may reveal good – and the new population $\vec{v}^i$ must have individuals taken from the best of those of the offspring and of the previous population.

The GAs, the (μ, λ)-ESs, the $(\mu + \lambda)$-ESs are particular EPs.

4.4 Genetic Algorithms (GAs)

GAs are EPs such that:

- $\lambda = \mu$
- $\vec{w}$ depends only on $\vec{v}^{i-1}$ and on random events
- $\vec{v}^i$ is a population of individuals only taken from those of $\vec{w}$, even more than once
- the recombination is a sequence of two operations: the *mating* and the *mutation*. The mating consists of choosing some couples of individuals of the population $\vec{w}$ and for each chosen couple (u, v) of replacing it in $\vec{w}$ with two feasible individuals (ψ, χ) depending on (u, v) and on random events, each similar to both (u,v); the mutation consists of choosing some individuals of the population $\vec{w}$ and for each chosen individual x of replacing it with a feasible chromosome y depending on it and on random events and similar to it

The pseudocode of a GA is given in Figure 4.2.

The individuals u, v are called the *parents* of ψ, χ; and ψ, χ are called the *children* of u, v. The procedure that creates the children from the parents is usually called *crossover*. More that one crossover can be used in the mating operation: some couples may produce their children by using a crossover procedure, some others may do it by using onother one. However the parents always die after mating and their place in the population is taken by their children. The procedure that creates y from x is called *mutation* and more than one mutation can be used.

4.4.1 The typical GA

In the typical GA, $\bar{R} = D_1 \times .. \times D_n$ with D_i sets. The chromosomes are then tuples. The positions of the chromosomes are called *genes*. But $\bar{R}$ may be a set of trees or other more complex structures.

The typical mating procedure consists of selecting the individuals to mate and of grouping them into couples; the selection of the individuals to mate is done in the following way: for each individual we take a random number $r \in]0, 1]$ and if $r \leq P_c$ – where $P_c \in]0, 1]$ is a constant of the GA –we select the individual; if the number of selected individuals is odd, we remove from the selected individuals the last selected one; the couples are formed by grouping by two in some way the selected individuals. Typically a couple of parents produce a couple of children in the following way: we choose some of the genes

{ $\quad i \leftarrow 0$;
 create $\vec{v}^0 \in R^\mu$;
 $\beta \leftarrow$ the element of $\vec{v}^0$ with the highest fitness;
 while(**not** termination condition)
 { $\quad i \leftarrow i + 1$;
  choose $\mathcal{I} \subset 2^{\{1,..,\mu\}}$ such that $\forall A, B \in \mathcal{I}, A \neq B$:
   $|A| = |B| = 2$ and $A \cap B = \emptyset$;
  $\forall I = \{h, k\} \in \mathcal{I}$: replace (v_h^{i-1}, v_k^{i-1}) with (x_h, x_k)
   feasible individuals depending on (v_h^{i-1}, v_k^{i-1}) and
   on random events;
  $\forall h$: replace v_h^{i-1} with a feasible individual
   depending on v_h^{i-1} and on random events;
  $\vec{w} \leftarrow \vec{v}^{i-1}$;
  $\forall h \in \{1, .., \mu\}$: choose $k \in \{1, .., \mu\}$ according to some
   criteria and $v_h^i \leftarrow w_k$;
  if$\Big(\exists j$ such that $val(v_j^i) > val(\beta)\Big)$ $\beta \leftarrow v_j^i$;
 }
 return β;
}

Figure 4.2: The structure of a GA.

of the parents and we swap them in the two chromosomes, thus obtaining two individuals similar two both the parents, but possibly unfeasible; if they are unfeasible, we repair them, i.e. we change them in few genes in order to make them feasible.

The typical mutation procedure consists of randomly changing the values of some genes of some individuals and of repairing the chromosomes that have become unfeasible after this operation: for each chromosome and for each gene we generate a random number $r \in]0,1]$ and if $r \leq P_m$ – where $P_m \in [0,1]$ is a constant of the algorithm – we randomly modify that gene; finally we repair the chromosomes that have become unfeasible after this operation. A mutation is said to be more or less strong if the mutated individuals are more or less different from the chromosomes from which they are obtained. P_m influences the strength of the mutation operation.

Many selection algorithms have been developed and typically only one is used in one GA. The most used selection algorithms are the *q-tournament*, the *proportional* and the *ranking* selection. With the q-tournament ([5]) selection $\forall h \in \{1,..,\mu\}$ a set $I \subset \{1,..,\mu\}$ such that $|I| = q$ is randomly chosen, the index $j \in I$, such that $\forall k \in I$ it is $val(w_k) \leq val(w_j)$, is found and it is set $v_h^i = w_j$. The proportional selection ([5]) can be used if $f \geq 0$; with this selection method each index $j \in \{1,..,\mu\}$ is associated with a probability p_j proportional to $val(w_j)$; $\forall h \in \{1,..,\mu\}$ an index $j \in \{1,..,\mu\}$ is randomly chosen[2] with probability p_j and v_h^i is set to w_j. With the ranking selection the set $\{val(w_j)|j = 1,..,\mu\}$ is ordered and each position $j \in \{1,..,\mu\}$ is associated with a probability p_j depending on the rank of $val(w_j)$ in that ordering; $\forall h \in \{1,..,\mu\}$ an index $j \in \{1,..,\mu\}$ is randomly chosen with probability p_j and v_h^i is set to w_j.

4.4.2 The problem of the feasibility

Usually we have a situation in which

1. $\bar{R} \neq R$

2. given two feasible individuals u, v, if we swaps correspondent parts of u, v we still obtain two elements of $\bar{R}$ not necessarily in R

3. given a feasible individual u, if we change some of its parts randomly we still obtain an element of $\bar{R}$ not necessarily of R

[2]More precisely, if $\forall j$ it is $val(w_j) = 0$ then $\forall j \in \{1,..,\mu\}$ we define $p_j = 1$; else $\forall j$ we define $p_j = \frac{val(w_j)}{\sum_{j=1}^{\mu} val(w_j)}$; we also define $q_1 = p_1$ and $\forall j \in \{2,..,\mu\}$ we define $q_j = q_{j-1} + \frac{p_j}{\sum_{h=1}^{\mu} val(w_h)}$; $\forall h$ we generate a random number $r \in]0,1]$ and we calculate the index k such that $k \in]q_k - 1, q_k]$; we set $v_h^i = w_k$.

In such a situation we can design a reparation algorithm that makes feasible an unfeasible individual by changing some of its parts trying to keep as many parts of the initial individual as possible. A crossover procedure can initially swap correspondent parts of the parents, thus producing intermediate individuals not necessarily feasible, and then repair them by a reparation algorithm. A mutation procedure can randomly change some parts of an individual, thus producing an intermediate unfeasible chromosome, and then repair it by a reparation algorithm. This is the most common way a genetic operator works. Yet a repair algorithm is very difficult to design and it slows down the GA as it often requires long computations ([1, page 5]).

In order to avoid the difficult design of reparation procedures and the slow speed of a GA using repair procedures, we can use the method of the penalty function ([1, page 97]) that exploits the fact that no reparation is needed when $\bar{R} = R$ as, in such a situation, whatever modification we make on a chromosome, we obtain a feasible chromosome.

The method consists of creating the new function $\bar{f} : \bar{R} \to \mathbb{R}$ such that $\forall x \in R, \bar{f}(x) = f(\phi^{-1}(x))$ and $\forall x \in \bar{R} \setminus R, \bar{f}(x) = V - p(x)$, with V a value $<$ than the optimal value of f and p a positive function increasing with the distance of x from R, and of designing a GA to maximize this function. An optimal solution of $\bar{f}$ is the rapresentation of an optimal solution of f. A GA that uses the identity function of $\bar{R}$ as encoding can be quickly designed with simple and fast genetic operators. Of course, in order to use this method we need to define the concept of distance of $x \in \bar{R} \setminus R$ from R.

The advantage of this technique is that in this GA crossover and mutation operators need no reparation; the disadvantage is that this GA risks to spend most of its time in valuating individuals unfeasible for the initial problem and as soon as a chromosome feasible for the initial problem is found to converge to it without any improvement ([1, page 98]). Another disadvantage is the difficulty of finding reasonable values for V and function p ([1, page 98]).

4.4.3 Classical GAs

The *classical GA* is a GA such that

- $\bar{R} = \{0, 1\}^n$
- the mating is like that of the typical GA of section 4.4.1 and uses only one crossover; the crossover modifies the parents $u = u_1..u_n$, $v = v_1..v_n$ in the following way: a random number $i \in \{1, .., n-1\}$ is generated; the possibly unfeasible intermediate individuals are created $u' = u_1..u_i v_{i+1}..v_n$ and $v' = v_1..v_i u_{i+1}..u_n$; these individuals are repaired and become the children of u, v.

- the mutating is like that of the typical GA of section 4.4.1 and uses only one mutation operator; in this case the random change of a gene consists of just setting it to 0, if it is 1, and to 1, if it is 0.

Most of the studies and of the theoretical results on GAs concern the classical GA. Yet, recently many researchers have obtained better results by developing GAs with more complex representations of solutions and with more varied genetic operators ([1]).

4.5 $(\mu + \lambda)$-Evolution Strategies ($(\mu + \lambda)$-ESs)

$(\mu + \lambda)$-ESs are EPs such that

- the offspring population $\vec{w}$ of iteration i depends only on $\vec{v}^{i-1}$
- the selection consists of keeping for the next generation the μ best individuals of the offspring $\vec{w}$ and of the previous generation $\vec{v}^{i-1}$

4.6 (μ, λ)-Evolution Strategies ((μ, λ)-ESs)

(μ, λ)-ESs are EPs such that:

- $\lambda > \mu$
- the offspring population $\vec{w}$ of iteration i depends only on $\vec{v}^{i-1}$
- the selection consists of keeping for the next generation the μ best individuals of the offspring $\vec{w}$

Chapter 5

The Harbour Packing Problem

5.1 Overview

The present project consists of studying and implementing a computer program to tackle an ideal packing problem that models a simplification of a real problem faced by boat managers of many harbours. In this chapter this problem is described and formalized. It is demonstrated that the problem is NP-hard, thus justifying our choice for its solution by an approximation algorithm such as a Genetic Algorithm.

5.2 Introduction

In every harbour there are ships that bring goods to destinations. The goods are carried to the vessels in containers by lorries which arrive to the boat in different moments of the day from different places. The containers have different shapes, utility and weights and carry different types of goods and must be placed on the hold of the ship.

The possible positions of a container are limited by many constraints (see [13]). Some goods cannot be too close to others because of safety reasons (for example, fireworks or explosives cannot be placed close to acids, as the acids accidentally can break out of the container and interact with the explosive, thus exposing the crew to a big risk and possibly loosing the cargo). Containers should be placed on the hold in a way that keeps the boat balanced. The crane cannot move very heavy containers too far, and they must be put close to the base of the crane. Some containers cannot be placed above others.

Often there is not enough room on the hold to load all the containers satisfying all the constraints and some of them must be left on the harbour, thus loosing the income due to their transportation to the destination.

As soon as a container arrives the boat manager must decide whether to load it or not. If he/she decides to load it, he/she must choose where. Sometimes outside of the boat there is a little space where the boat manager can park containers postponing the decision of their loading. Also often some containers, after they have been loaded, cannot be moved or unloaded.

The problem is complicated by the fact that lorries seldomly arrive in time and some of them arrive later or earlier than expected or do not arrive at all. Often, right in the middle of the day, after some containers have already been loaded, the boat manager gets to know that some lorries will not arrive or that new unexpected lorries will bring their containers to the boat. Moreover the boat often has to reach several destinations in one trip and in some of them not only it must bring but also receive containers and these arrivals are not sure too.

The boat manager should do his/her decisions maximizing the utility of the containers brought to destination. The task of the boat manager is then nontrivial and of great responsibility.

5.3 The problem simplified

We imagined though that the boat manager must face a simpler problem, which is the one tackled in this thesis. Then the problem dealt of in this dissertation is ideal, but of great importance.

In this simpler situation the boat manager must load a boat with a rectangular hold. The containers have a parallelepipedal shape of variable dimensions and utility and they have different weights, but the ship will be balanced whichever disposition they will have on the hold. They must be placed side by side, and it is not possible to place a container on the top of another. The containers may hold different types of goods, but they are all mutually compatible, according to the safety regulations. The crane is strong enough to put each containers in whichever position wanted, and after a container is loaded on the ship, it is fixed on the hold and cannot be unloaded or moved to another position. In the quay there is not enough room for containers to be parked. Hence the decision of whether to refuse a container or not and of where to place it on the hold cannot be postponed: as soon as a container arrives to the boat, it must be loaded or definitely refused. The vessel must bring these containers to only one destination where all the containers will be completely unloaded. The edges of the loaded containers must be parallel to those of the hold and the lower left edge of each container must be placed in one of the points of a given grid. Containers can be rotated to optimize the space.

The boat manager does not know exactly the set of containers that he/she will have to load, nor the sequence of their arrivals, but he/she has got some information about this unknown future which reduces its uncertainty.

He knows the dimensions of the rectangular bases and the utilities of the containers of a given set, a subset of which will arrive; he/she has a probabilistical description of the arrivals, that tells him/her for each arrived container which other can arrive next and with which probability. More precisely, the possible arrivals are described by a tree (see figure 5.1): each node of the tree is associated with a known container and each edge is associated with a probability. The arrivals will obey to this tree in this sense: the container of the root will arrive first and with total certainty; the next arrival will be one of the containers associated with the children of the root; for each of them the probability of arrival after the first is given by the probability associated with the edge that brings to them; and so on. Each actual arrival will be then represented by a node of the tree and the next arrival will be represented by one of its children.

For each node the sum σ of the probabilities of the edges bringing to its children is ≤ 1; yet if it is $\sigma < 1$ it means that with probability $1 - \sigma$ no other container will arrive after the one associated with that node. A terminal node is a leaf node or an inner node with $\sigma < 1$.

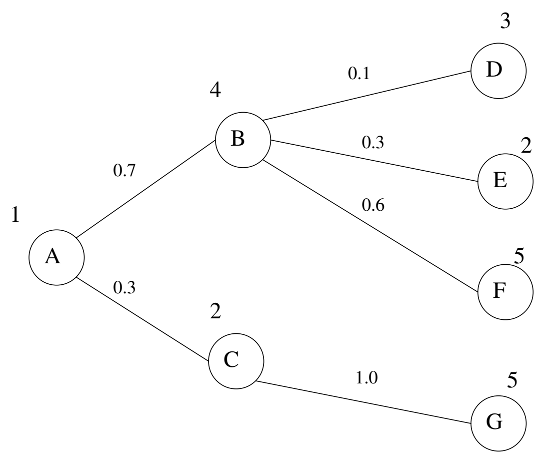


Figure 5.1: The problem tree.

The set of possible arriving containers is then the set of containers associated with the nodes of the tree, and the actually arrived containers will be the set of containers associated with one path to the root to a terminal node. The possible sequences of arrivals are all the sequences associated with the paths from the root to a terminal node, and the actual sequence of arrivals will be one of them. The probability that a sequence of arrival will occur is given by the product of all probabilities associated with the edges of the path for that sequence multiplied by $1-\sigma$ with σ the sum of the probabilities of the outgoing edges of the terminal node of that sequence.

Before any container arrives, the boat manager must decide what to do at every possible arrival. That is for each node of the tree, he/she must decide whether to load the container of that node or not, and where. More precisely he/she must prepare a plan of action which associates each node of the tree with the decision for the correspondent container (see figure 5.2). The decision is the position of the container on the hold – that is the coordinates of the point of the grid in which the lower left corner of the container is placed and the orientation of the container –, if it will be loaded, or the information that it will not be loaded. This plan will be used to refuse or load the containers as soon as they arrive.

The boat manager must find a plan that will load the boat with a big total utility with great probability.

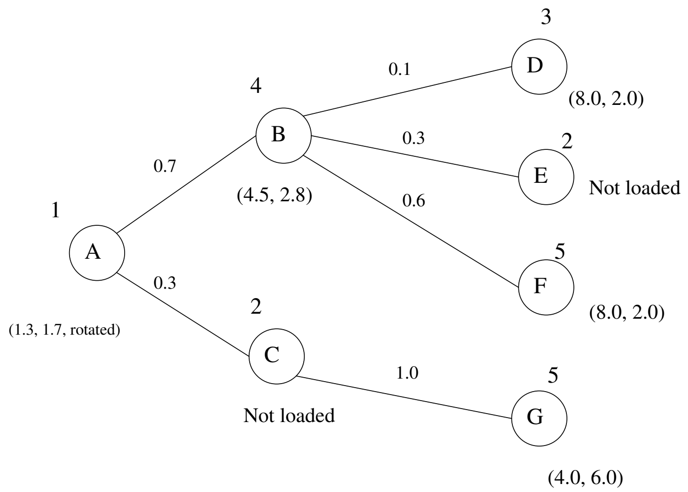


Figure 5.2: The plan of action.

Each terminal node of a plan can be associated with a value obtained by multiplying the product of the probabilities of the edges of the path from the root to that node multiplied by $1-\sigma$ with σ the sum of the probabilities of the outgoing edges of that terminal node and the sum of the utilities of the loaded containers of that path in the plan. The value of a plan can then be defined as the sum of the values of its terminal nodes.

If the same situation occurs several times and we always use the same plan to load the boat, then the average of the sum of the utilities of the loaded containers tends to be the value of the plan. Therefore if the situation occurs only once and we use a plan to load the boat, the most probable sum of the utilities of the loaded containers is the value of that plan. Hence the boat manager must find the plan with the highest value.

This problem will be formalized in the following section.

5.4 The problem formalized

5.4.1 Branching Packing Problems (BPPs)

Definition 10 In this project a *Branching Packing Problem (BPP)* is defined as a

$$(d_x, d_y, e, l_x, l_y, u, T, p, c)$$

where [1]

- $l_x, l_y, u : C \to \mathbb{R}_0^+$, with $C = \{1, .., m\}$ and $m \in \mathbb{N}_0$
- $e \in \mathbb{R}_0^+$
- $d_x, d_y \in \mathbb{R}_0^+$, with $d_x, d_y > e$
- $T = (N, E, \rho)$ a rooted tree with $n = |N| \geq 2$
- $p : E \to]0, 1]$ such that $\forall \alpha \in N$, if O is the set of the outgoing edges of α, then $\sum_{a \in O} p(a) \leq 1$;
- $c : N \to C$ such that $\forall \alpha \in N$:
    1. if J is the set of the children of α, then $\forall \gamma, \delta \in J$ such that $\gamma \neq \delta$ it is $c(\gamma) \neq c(\delta)$;
    2. if H is the set of the nodes of the path from the root to α, then $\forall \gamma, \delta \in H$ such that $\gamma \neq \delta$ it is $c(\gamma) \neq c(\delta)$;

[1] In the following pages the symbol $\mathbb{R}_0^+$ represents the set of real numbers > 0, $\mathbb{R}^+$ the set of real numbers ≥ 0, $\mathbb{N}_0$ the set of natural numbers > 0, and $\mathbb{Q}^+$ the set of rational numbers ≥ 0.

The elements of C are called *containers*; for $a \in C$, $l_x(a)$, $l_y(a)$ and $u(a)$ are called the *width*, the *length* and the *utility* of container a respectively; e is the *precision* of the grid; d_x and d_y are the *width* and the *length* of the hold; T is the *branching tree*; for $a \in E$, $p(a)$ is the *probability* of edge a; for $\alpha \in N$, $c(\alpha)$ is the *container of* node α. □

Less formally we can imagine a BPP as a set of objects C called containers with a rectangular base, each $a \in C$ with width $l_x(a)$ and length $l_y(a)$ and utility $u(a)$; a rectangular container called hold with width d_x and length d_y with a grid of horizontal and vertical lines parallel to the borders and far e one from the other, the leftmost of which is coincident with the left border line and lowest with the lower border line; a tree T where each edge a is labelled with a positive number $p(a)$ and each node α is labelled with a container $c(\alpha)$ in a way that for each node the sum of the probabilities of the edges outgoing from the node is less than one and each node has a container different from that of its brothers and ancestors. We can imagine that there is a cartesian system whose origin is placed in the lower left corner of the hold and whose axes are parallel to the lower border and to the left border of the hold – see figure 5.3. The points of the grid are identified by their coordinates on the cartesian system. If a container is placed on the hold, it can be rotated and its lower left corner must be placed on one of the points of the grid and its sides must be parallel to those of the hold. Two object, when placed on the hold cannot overlap – see figure 5.3.

A BPP represents the information about the ship space and the future arrivals of containers that the boat manager receives in the morning, before any container arrives.

5.4.2 Assignments and feasible assignments

Definition 11 Given a BPP, we define $D_x = \{ne | n \in \mathbb{N}, ne < d_x\}$ and $D_y = \{ne | n \in \mathbb{N}, ne < d_y\}$. The elements of the set $(D_x \times D_y \times \{0, 1\}) \cup \{NULL\}$ are called *positions* of the BPP. □

Definition 12 Given a BPP, a container i and a position $P = (x, y, z) \neq NULL$ we define

$$\Delta_x(i, P) = \begin{cases} l_x(i) & \text{if } z = 0 \\ l_y(i) & \text{else} \end{cases}$$

and

$$\Delta_y(i, P) = \begin{cases} l_y(i) & \text{if } z = 0 \\ l_x(i) & \text{else} \end{cases}$$

□

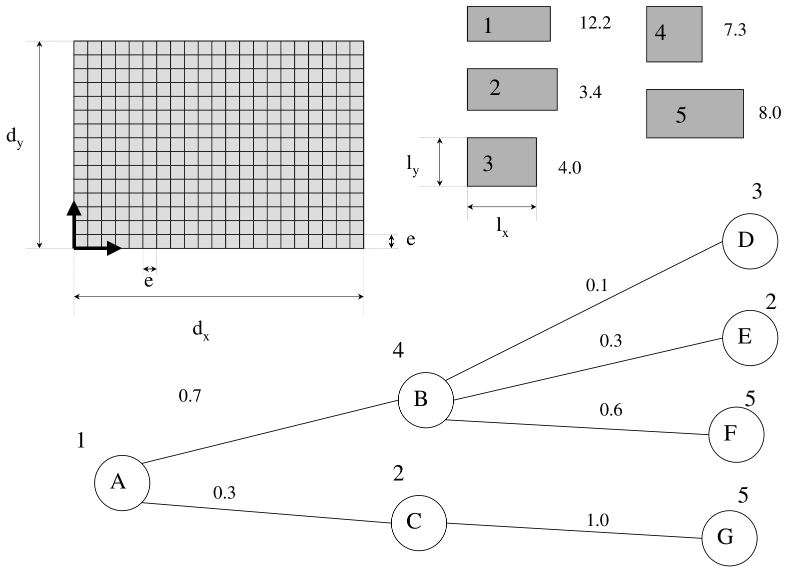


Figure 5.3: A rapresentation of a BPP.

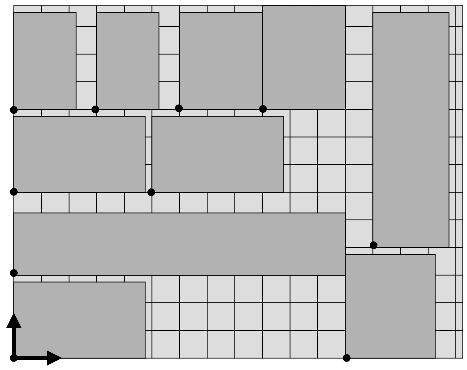


Figure 5.4: The positioning of containers on the hold.

Definition 13 Given a BPP, we say that a container i in position P is *inside* of the hold iff $P = NULL$ or

$$(x + \Delta_x(i, P), y + \Delta_y(i, P)) \in [0, d_x] \times [0, d_y]$$

with $P = (x, y, z)$.

We say (see Figure 5.5) that a container i in position P_i *overlaps* with container j in position P_j iff $P_i = (a_i, b_i, c_i) \neq NULL$ and $P_j = (a_j, b_j, c_j) \neq NULL$ and

$$(a_j, b_j) \in]a_i - \Delta_x(j, P_j), a_i + \Delta_x(i, P_i)[\times]b_i - \Delta_y(j, P_j), b_i + \Delta_y(i, P_i)[$$

$\square$

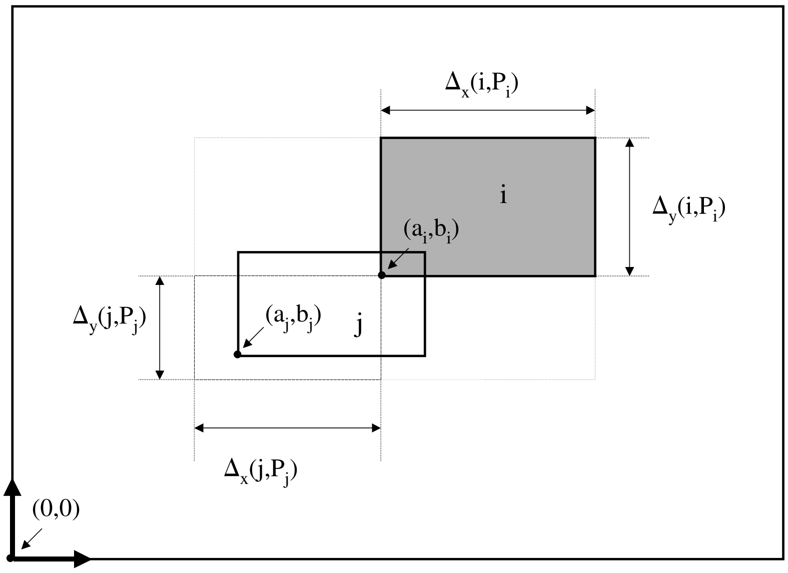


Figure 5.5: The rectangle of overlapping.

Definition 14 A *assignment* of a given BPP is a function

$$\psi : N \to (D_x \times D_y \times \{0, 1\}) \cup \{NULL\}$$

an assignment ψ is called *feasible* iff $\forall \alpha \in N$:

1. container $c(\alpha)$ in position $\psi(\alpha)$ is inside of the hold;

2. $\forall \beta \in N$, β of the path from the root to α, $\beta \neq \alpha$, container $c(\beta)$ in position $\psi(\beta)$ does not averlap with container $c(\alpha)$ in position $\psi(\alpha)$.

An assignment which is not feasible is called *unfeasible*. □

Definition 15 Given BPP, an assignment ψ, a node α and a position P, we say that position P in the node α of the assignment ψ is *feasible* iff

1. container $c(\alpha)$ in position P is inside of the hold;
2. $\forall \beta \in N$, $\beta \neq \alpha$, β of the path from the root to α, $c(\alpha)$ in position P does not overlap with $c(\beta)$ in position $\psi(\beta)$.

□

Less formally the sets D_x and D_y represent the abscissas and ordinates of the points of the grid in a cartesian system. The set $D_x \times D_y$ is then the set of coordinates of these points.

An *assignment* is a function that associates to each node α of the tree a position $\psi(\alpha)$ for the correspondent container $c(\alpha)$ that must be interpreted in this way: if $\psi(\alpha) = NULL$, the container is not loaded on the boat; else if $\psi(\alpha) = (x, y, z)$, the corresponding container is loaded on the hold in the following way: if $z = 1$ it is rotated of 90 degrees – clockwise or anticlockwise, it is the same – else it is kept with the same orientation; then it is placed with the lower left corner on the point of the grid of coordinates (x, y) and with sides parallel to the borders of the hold.

If position $P = (x, y, 1)$, then $\Delta_x(i, P)$ and $\Delta_y(i, P)$ represent the width and the length of container i after it is rotated and placed with the lower left corner in the point of the grid of coordinates (x, y); if $P = (x, y, 0)$, then $\Delta_x(i, P)$ and $\Delta_y(i, P)$ represent the width and the length of container i after it is placed with the same orientation and with the lower left corner in the point of the grid of coordinates (x, y).

Not all the possible assignments are feasible. In order to be *feasible* an assignment must satisfy some constraints: if the container associated with a node α is loaded, after it is eventually rotated and positioned in the corresponding point of the grid, it must be inside the hold; also it must not overlap with any container of the nodes of the path from the root to α loaded by the same assignment.

As can be deduced by the definition, a feasible assignment can place the containers of two nodes in an overlapping position. But one of these nodes must not be in the path from the root to the other.

A feasible assignment represents the plan of actions of the boat manager.

Note that if container i in position P_i overlaps with container j in position P_j then j in position P_j overlaps with i in position P_i.

Given a BPP, the number of its assignments is

$$\Big(|D_x| \cdot |D_y| \cdot 2 + 1\Big)^{|N|}$$

The number of feasible assignments is obviously less and is greater than one, for the trivial assignment which associates each note to $NULL$ is always a feasible assignment . We are interested in finding feasible assignments. In the following section we will define the goodness of a feasible assignment. We will then be interested in finding the best feasible assignments.

Note then that $n_x = |D_x| \geq 2$ and $n_y = |D_y| \geq 2$ and $D_x = \{0, .., e(n_x - 1)\}$ and $D_y = \{0, .., e(n_y - 1)\}$.

5.4.3 Expected Utility (EU)

Definition 16 Given a BPP, a node α of its tree is said to be *terminal* iff $\sum_{a \in O(\alpha)} p(a) < 1$ with $O(\alpha)$ the outgoing edges of α. If α is a terminal node, we define $P_\alpha = 1 - \sum_{a \in O(\alpha)} p(a)$. $\square$

As a consequence of Definition 16, the following properties hold:

1. all the leaves are terminal nodes, as if α is a leaf $O(\alpha) = \emptyset$ and so $\sum_{a \in O(\alpha)} p(a) = 0$
2. $\forall \alpha$ leaf it is $P_\alpha = 1$
3. $\forall \alpha$ node it is $0 < P_\alpha \leq 1$

Definition 17 Given a BPP and one of its feasible assignments ψ, the *expected utility (EU)* of ψ is the number

$$EU(\psi) = \sum_{\alpha \in L} \Big(\sum_{\beta \in I(\alpha)} u(c(\beta)) \Big) \Big(\prod_{a \in A(\alpha)} p(a) \Big) P_\alpha$$

with

- L the set of the terminal nodes of T
- $\forall \alpha \in L, I(\alpha) =$ the set of the nodes β of the path from the root to α such that $\psi(\beta) \neq NULL$;
- $\forall \alpha \in L, A(\alpha) =$ the set of the edges of the path from the root to α

□

Because the number of feasible assignments is finite, the set

$$I = \{EU(\psi) | \psi \text{ is a feasible assignment}\}$$

is also finite. Hence there exists $\bar{\psi}$ a feasible assignment such that $\forall \psi$ feasible assignment, $EU(\bar{\psi}) \geq EU(\psi)$. It is then not unreasonable to be interested in finding a feasible assignment with the highest EU. A feasible assignment with the highest EU is called *optimal solution.*

Given a BPP, the set I defined above has an upper bound given by the formula

$$UB = \sum_{\alpha \in L} \Big(\sum_{\beta \in J(\alpha)} u(c(\beta)) \Big) \Big(\prod_{a \in A(\alpha)} p(a) \Big) P_\alpha$$

with

- L the set of the terminal nodes of T
- $\forall \alpha \in L, J(\alpha) =$ the set of the nodes β of the path from the root to α;
- $\forall \alpha \in L, A(\alpha) =$ the set of the edges of the path from the root to α

which is a number depending only on the BPP and which corresponds to the EU of an hypothetical feasible assignment that assign all the nodes with a value different from $NULL$. Thus $\forall \psi$ feasible assignment of the BPP $EU(\psi) \leq UB$ and if we find a feasible assignment ψ such that $EU(\psi) = UB$ then ψ is a solution with the highest expected utility.

Recursive computation of $EU(\psi)$

Given a BPP and one of its feasible assignments ψ, $EU(\psi)$ can be calculated recursively. In fact we can define $v_\psi : N \to \mathbb{R}$ such that $\forall \alpha \in N$

1. if α is a leaf of T then

$$v_\psi(\alpha) = \begin{cases} 0 & \text{if } \psi(\alpha) = NULL \\ u_{c(\alpha)} & \text{else} \end{cases}$$

2. else

$$v_\psi(\alpha) = \begin{cases} \sum_{\beta \in C(\alpha)} v_\psi(\beta) p_\beta & \text{if } \psi(\alpha) = NULL \\ u_{c(\alpha)} + \sum_{\beta \in C(\alpha)} v_\psi(\beta) p_\beta & \text{else} \end{cases}$$

with $C(\alpha)$ the children of α and p_β the probability of the edge that connects β with its father α.

Then $EU(\psi) = v_\psi(\rho)$ with ρ the root of T.

Computation of $EU(\psi)$ by matrices

If we fix an ordering of the nodes $N = \{N_1, .., N_n\}$ and one of the terminal nodes $F = \{F_1, .., F_k\}$ of T, then there exists a vector $\vec{y}$, dependent only on the BPP and on the orderings of N and of F, such that $\forall\psi$ feasible assignment of the BPP, $EU(\psi) = \vec{y}^T\vec{b}$ with $\vec{b} = (b_1, .., b_n)$ and $\forall i = 1..n$

$$b_i = \begin{cases} 0 & \text{if } \psi(N_i) = NULL \\ 1 & \text{else} \end{cases}$$

Thus if we have $\vec{y}$, the calculation of $EU(\psi)$ consists on the simple computations of $\vec{b}$ and of the product $\vec{y}^T\vec{b}$. The vector $\vec{y}$ is given by the formula:

$$\vec{y} = F^T\vec{p}$$

where

1. $p = (p_1, .., p_k)$ with p_i the product of the probabilities of the edges in the path from the root to the terminal node F_i and of P_{F_i}
2. F the matrix of order $k \times n$ such that $\forall(i, j) \in \{1, .., k\} \times \{1, .., n\}$

$$F_{i,j} = \begin{cases} u_{c(N_j)} & \text{if } N_j \text{ is in the path from the root to } F_i \\ 0 & \text{else} \end{cases}$$

In other words, given a BPP C, there exists a function $G : N \to \mathbb{R}^+$ such that $\forall\psi$ solution of C it is

$$EU(\psi) = \sum_{\alpha\in N} G(\alpha)b_\psi(\alpha)$$

with

$$b_\psi(\alpha) = \begin{cases} 0 & \text{if } \psi(\alpha) = NULL \\ 1 & \text{else} \end{cases}$$

Moreover it is $UB = \sum_{\alpha\in N} G(\alpha)$, with UB the upper bound defined above. The function G is given by the formula

$$G(\alpha) = u(c(\alpha)) \cdot \sum_{\tau\in T(\alpha)} \Big(\prod_{a\in A(\tau)} p(a) \Big) P_\tau$$

with $T(\alpha) = \{\tau | \tau$ terminal node such that α is in the path[2] from the root to $\tau\}$ and $A(\tau) =$ the set of the edges of the path from the root to τ.

[2]Note that τ is in the path from the root to τ and if α is a terminal node, then it is $\alpha \in T(\alpha)$.

Note that function G is dependent only on the BPP. So given the solution ψ in order to compute $EU(\psi)$, once we have function G, we just have to compute the function b_ψ and the sum $\sum_{\alpha \in N} G(\alpha) b_\psi(\alpha)$. Function G can be calculated by the simple algorithm of Figure 3.1.

In spite of its notational complexity, this method is simple and it is the fastest way to compute $EU(\psi)$, once we have calculated G. It is particularly convenient when we have to compute the EU of several feasible assignments of the same BPP and it is the method used in the application developed in this thesis.

5.5 BPPs with one path trees

BPPs with one path trees and with all the edges associated with probability 1 deserve special attention.

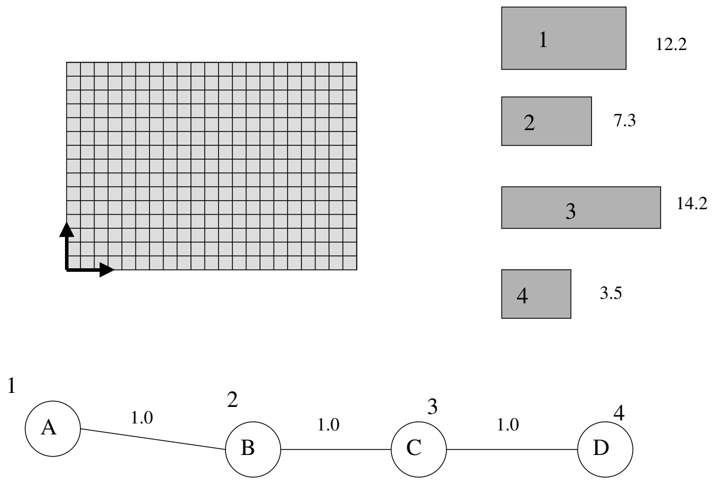


Figure 5.6: A BPP with a one path tree

A BPP with a one path tree (see figure 5.6) is produced by a boat manager that knows with certainty the set of containetrs to be loaded and must decide which of them he/she can load and where in order to maximize the sum of the utilities of the loaded conatainers .

In this case a feasible assignment is just a positioning of some of the containers inside of the hold and the EU of a feasible assignment is just the sum utilities of the loaded containers.

The BPP is then a generalization of the classical problem of placing rectangular objects on the points of a grid inside a rectangular container in order to maximize the sum of the utilities of placed objects.

Therefore, in order to solve a classical gridded packing problem, being it a particular case of the BPP, we can use an algorithm for the BPP as we have done in Section 9.3.1.

5.6 NP-hardness

In this section it will be demostrated that the optimization problem of finding a feasible assignment of a BPP with the highest expected utility is an NP-hard problem. It is then advisable to spend our energy in developing an approxiamtion algorithm rather than in developping one that find the exact best assignment (see [4, page 916]).

Lemma 1 *The optimization problem "given $\vec{w} \in \mathbb{N}^n$, $\vec{u} \in \mathbb{N}^n$ and $W \in \mathbb{N}$, find the $\vec{x} \in \{0,1\}^n$ such that $\vec{w}^T\vec{x} \leq W$ in a way that $\vec{u}^T\vec{x}$ is maximal" is NP-hard.* $\square$

The problem of Lemma 1 is the well known Optimization Knapsack Problem and the proof of its NP-hardness can be found in [2, page 65]. The NP-hardness of the problem of finding a feasible assignment with the highest EU of a BPP with integer precision, integer sizes of hold and containers and rational probabilities is a consequence of the NP-hardness of this problem.

Proposition 7 *The optimization problem "given the BPP*

$$(d_x, d_y, e, l_x, l_y, u, T, p, c)$$

such that $\forall i \in C : l_x(i), l_y(i), u(i) \in \mathbb{N}$ and $d_x, d_y, e \in \mathbb{N}$ and $\forall a \in E : p(a) \in \mathbb{Q}^+$, find one of its feasible assignments with the highest EU" is NP-hard. $\square$

Proof. Let D be the problem of this Proposition and C that of Lemma 1. We will prove that if we had a polinomial algorithm δ for D then we would have a polinomial algorithm γ for C.

In fact let δ be a polinomial algorithm for D and γ the algorithm that:

1. After receiving an input $\vec{w}$, $\vec{u}$ and W of problem C it builds the BPP

$$(d_x, d_y, e, l_x, l_y, u, T, p, c)$$

where

- $e = 1$
- $d_y = 1$
- $d_x = W$
- $l_x(i) = w_i$
- $l_y(i) = e$
- $u(i) = u_i$
- $T = (N, E, \rho)$ with $N = \{1, .., n\}$, $E = \{\{i, i+1\} : i = 1, .., n-1\}$ and $\rho = 1$
- $p(a) = 1, \forall a \in E$
- $c(i) = i$

2. It gives this BPP in input to algorithm δ which returns in output ψ.

3. Returns the vector $\vec{x}$ such that for $i = 1, .., n$

$$x_i = \begin{cases} 0 & \text{if } \psi(i) = NULL \\ 1 & \text{else} \end{cases}$$

Now γ is polinomial. In fact the time it takes to give the output is the sum of the times of the 3 steps: the construction of the BPP requires a polinmial time on the size of the inputs to γ; the BPP has a size which is polinomial on the size of the inputs; the assignment of the BPP by γ requires a polinomial time of the size of the BPP and so a polinomial time of the size of the inputs; the constuction of $\vec{x}$ from ψ requires a polinomial time on the size of the inputs. Therefore the sum of these times is again polinomial of the size of the inputs to γ.

Moreover γ solves C. In fact let's suppose that we have an instance $I = (\vec{w}, \vec{u}, W)$ of problem C and we give it in input to γ. Let $BPP(I)$ be the BPP built by γ on input I. Then $BPP(I)$ has a one path tree and corrisponds to the situation with a very narrow and long hold and very narrow and long containers (see figure 5.7). The width of the hold is e and the length is W; the width of container i is e and its lenght is w_i. The width of the containers is equal to that of the hold hence containers can only be placed on a row and not in parallel. The lines of the grid are far e one from the other and e is a submultiple of the lenghts of the containers so that if we place the containers inside the hold and we compress them tightly they will still move to a feasible position, with the lower left corner on a point of the grid. All the containers arrive with total certainty.

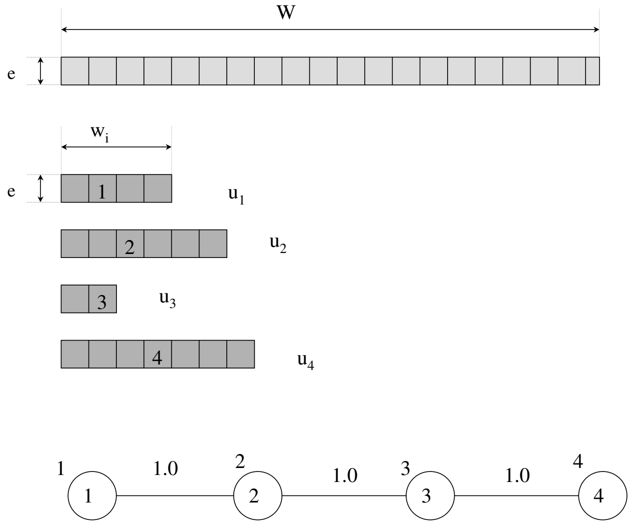


Figure 5.7: The BPP produced by γ

A feasible assignment of $BPP(I)$ gives us an acceptable set of objects – that is with a total weight $\leq W$ – for I with the same value. Viceversa, an acceptable subset of objects of I gives us a feasible assignment to $BPP(I)$ with the same value. As a matter of fact if we know that a certain set of objects can be placed on the bag, the corrispondent containers can enter the hold too, even if they must stay on the grid, because we can place them in a random position and then pack them together, for the new position is grid-feasible and their total lenght is surely less than that of the hold.

Therefore if we give $BPP(I)$ to δ, this algorithm gives us the best feasible assignment ψ of $BPP(I)$, from which we can build x a subset of objects feasible for I with the same value, that is the output of γ. If I had a better subset $\bar{x}$, there would exist also an assignment $\bar{\psi}$ of $BPP(I)$ better than ψ. But this is not possible. So the assignment given by γ is a feasible subset for the instance of problem C and it is the best. Then γ solves C.

Thus if we had a polinomial algorithm for D, we would have a polinomial algorithm for C and so, being C NP-hard, a polinomial algorithm for all the problems of class NP. Then D is NP-hard. $\square$

The problem of finding a feasible assignment with the highest EU of a given BPP is then at least as difficult as the one of the Proposition 7.

5.7 The BPP as a BCSP

Given a BPP we can easily build a BCSP with the same solutions and such that the solutions have the same EU. The BPP gives all the necessary information that define the idealized problem of the harbour and is exactly the input of the application described in Chapter 10. Many of the previous conclusions could be derived from the properties of the BCSPs. Actually the problem of the harbour has been modelled first as a BCSP and then as a BPP, which is simply another way to describe this particular BCSP.

Given the BPP

$$(d_x, d_y, e, l_x, l_y, u, T, p, c)$$

the BCSP with the same solutions of the BPP is simply

$$\Big((D_1, .., D_m, \mathcal{C}), \vec{u}, T, p, c\Big)$$

with $\forall i \in \{1, .., m\} D_i = D_x \times D_y \times \{0, 1\} = D$ and $\mathcal{C} =$ the set of all the $(I, \mathcal{I})$ such that:

- $|I| = 2$
- $\{i, j\} = I \subset \{1, .., m\}$
- $\mathcal{I} = \{f : I \to D|$ container i in position f_i is inside of the hold, container j in position f_j is inside of the hold, container i in position f_i does not overlap with container j in position $f_j\}$

5.8 The eBCSPs and the requirement of balance

In the previous section we have seen that the idealized problem of Section 5.3 can be modelled as a BCSP. If the boat manager of the ideal problem had to place the containers in the hold so as to keep the boat balanced – for example in a way that the total weight of the containers in the left part of the hold is almost the same of the total weight of the containers of the right part of the hold – the new problem would not be easily modellable as a BCSP.

As a matter of fact the requirement of balance is naturally modellable as a constraint concerning the loaded containers. But if for each subset of containers we added to the CSP the constraint allowing only the balanced disposition of containers, then we would refuse most of the balanced dispositions.

The new problem instead is naturally modellable as an eBCSP. In order to build the eBCSP representing the new ideal problem we can first build the BCSP corresponding to the problem without the requirement of balance; then we can add the $NULL$ value to the domains; finally for each terminal node, we can add a constraint concerning the variables of the path from the root to that node that accepts all the assignments f such that the set of containers associated by f to a value $\neq NULL$ are in a balanced disposition.

Chapter 6

The Genetic Algorithm

6.1 Overview

This Chapter describes a GA that searches for a feasible solution of a given BPP with the highest EU. This GA and its implementation is the purpose of the present dissertation.

6.2 Introduction

As stated in Chapter 5 the problem of finding a feasible solution of a given BPP with the highest EU is NP-hard and as such it is more convenient to solve it by an approximation algorithm. A GA is an approximation algorithm, as it starts from a set of feasible solutions and tries to improve them until the user decides that it must stop and it does not guarantee that the given solution is the best.

The following algorithm is not a classical GA as defined in [1]. As a matter of fact the chromosomes are not binary vectors, but solutions of a BPP. There is no encoding of individuals to strings of the binary alphabet and the genetic operators are clever and problem specific in the sense that they modify individuals by taking into account the information about the problem and the meaning of the objects that they modify. This algorithm is also a multicrossover and multimutation algorithm in the sense that different kinds of crossover and mutation operators are executed with different frequences in each iteration during the process of recombination and of mutation respectively. Moreover this GA uses a new feature called "Shake", which tries to avoid the premature convergence of the algorithm. For these reasons this algorithm could be classified as an Evolution Program ([1, page 10]), but because of its strong similarities with the bettere known GAs, it will be called GA.

The feasibility of the individuals is maintained in this GA by specific genetic operators which always produce feasible individuals, often by repairing unacceptable chromosomes obtained by rough low level operations. The design of effective specific genetic operators have required much time and effort.

The GA here presented depends on several parameters and its performance and behaviour varies strongly with them.

This algorithm is specific for BPPs, but its genetic operators are easily adaptable to be part of a GA that solves any BCSP.

6.3 Basic concepts and operations

6.3.1 The ordering of the positions

A solution of a BPP is a function which associates each node α of the tree of the BPP an element of the set $(D_x \times D_y \times \{0,1\}) \cup \{NULL\}$. The present GA assumes that the elements of this set are ordered by the one to one function

$$\phi : \{1,..,2n_xn_y+1\} \to (D_x \times D_y \times \{0,1\}) \cup \{NULL\}$$

such that

1. if $1 \le k \le 2n_xn_y$:

$$\phi(k) = \begin{cases} x = ((k\text{div}2-1)\text{mod}n_y)e \\ y = ((k\text{div}2-1)\text{div}n_y)e \\ z = k\text{mod}2+1 \end{cases}$$

2. if $k = 2n_xn_y+1$:

$$\phi(k) = NULL$$

whose inverse is

$$\chi : (D_x \times D_y \times \{0,1\}) \cup \{NULL\} \to \{1,..,2n_xn_y+1\}$$

such that

$$\chi(x,y,z) = 2 \cdot (\frac{x}{e}n_y + \frac{y}{e} + 1) - \begin{cases} 0 & \text{if } z = 1 \\ 1 & \text{else} \end{cases}$$

and

$$\chi(NULL) = 2n_xn_y + 1$$

This ordering is that obtained first by ordering the points of $D_x \times D_y$ by increasing ordinates and increasing abscissas (see figure 6.1) and then by ordering the elements of $(D_x \times D_y \times \{0,1\}) \cup \{NULL\}$ in the way that

1. $(x, y, z) < (\alpha, \beta, \gamma)$ if $(x, y) < (\alpha, \beta)$
2. $(x, y, 0) < (x, y, 1)$
3. $(x, y, z) < NULL$

So the positions are in this order:

$$(0,0,0),(0,0,1),(0,e,0),(0,e,1),(0,2e,0),(0,2e,1),..,(0,n'_y e,0),(0,n'_y e,1),$$

$$(e,0,0),(e,0,1),(e,e,0),(e,e,1),(e,2e,0),(e,2e,1),..,(e,n'_y e,0),(e,n'_y e,1),$$

$$...$$

$$(n'_x e,0,0),(n'_x e,0,1),(n'_x e,e,0),(n'_x e,e,1),,..,(n'_x e,n'_y e,0),(n'_x e,n'_y e,1),$$
$$NULL$$

with $n'_x = n_x - 1$ and $n'_y = n_y - 1$.

Other orderings could be used. Yet, the algorithm requires that $NULL$, according to the ordering, is the last element of the set.

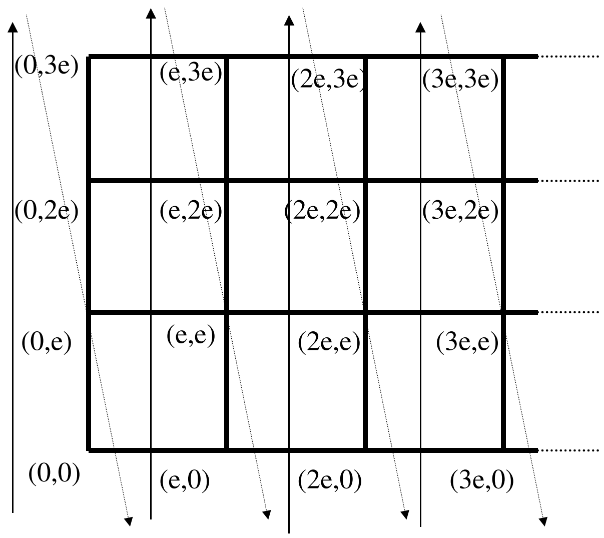


Figure 6.1: The ordering of the points of the grid.

6.3.2 The ordering of the nodes

The present GA orders the nodes of the tree of the input BPP in a depth first manner. More precisely, as soon as the BPP is received in input by the GA, the children of each node are ordered in some way; then all the nodes are ordered by the rule that: the father comes before all its children and a node comes before all its brothers that follows it in the ordering of the children of his father. This ordering is called the *basic* ordering of the nodes.

6.3.3 The Fill operation

On a feasible solution the GA often makes a Fill operation. This operation consists of trying to change the $NULL$ values of the nodes of the input solution to values $\neq NULL$, i.e. of trying to fill up the empty spaces of the hold with some unloaded containers.

When a feasible solution ψ is modified by a Fill operation, all its nodes are considered in the basic order and for each node α associated to $NULL$ by ψ, α is assignd the first value $\neq NULL$ of the order of section 6.3.1 such that the correspondent container $c(\alpha)$ in that position does not overlap with any container of nodes of the path from the root to α in its corresponding position.

The pseudo code for the Fill operation is given in figure 6.2.

The Fill operation optimizes the feasible solution as the EU of the filled solution is $\geq$ than than before the operation.

6.3.4 The repair operations

The GA here presented makes use of repair procedure in the generation of the initial population and after some genetic operators are applied. A repair function receives in input a solution and returns in output a feasible solution dependent on the input. It is used to make feasible an unfeasible solution.

The present GA makes use of two repair functions: the *Ordered*-repair and the *Random*-repair. These repair functions are designed in a way that the output feasible solution is similar to the input unfeasible one in order to take advantage of the evolutionary process. If the input solution is feasible it is returned in output unchanged.

The Ordered-repair

This procedure explores the nodes of the tree in the basic order and for each node α it considers the value given to α by the input solution ψ. If this

procedure Fill(ψ)
{ let $N = \{N_1, .., N_n\}$ be the basic ordering of the nodes;

for($i \leftarrow 1$ **to** n)
if($\psi(N_i) = NULL$ **and** there exists
a position $point \neq NULL$ such that container $c(N_i)$ in position
$point$ is inside of the hold and $\forall\beta \neq N_i$ node
of the path from the root to N_i: container $c(N_i)$
in position $point$ does not overlap with container $c(\beta)$
in position $\psi(\beta)$)
$\psi(N_i) \leftarrow$ the first such a position in the order ϕ;
}

Figure 6.2: The Fill procedure.

value satisfys all the constraints concerning the correspondent container – that is the container in this position is inside the hold and does not overlap with any container of the path from the root to α – then it is kept in the output solution; else the procedure considers one by one the next values in the order of section 6.3.1 until one that satisfies the constraints is found. This value is always found, being the $NULL$ value always acceptable and at the end of the ordering. Eventually the feasible solution so far obtained is filled.

In summary the Ordered-repair can be described in pseudo code as in figure 6.3 where $\phi_1, .., \phi_k$ is the ordering of the positions in the grid of section 6.3.1.

The output solution is feasible because a modification of a node value is made after all the values of the nodes of the path from the root to that node are made feasible, thanks to the basic ordering of this operation.

The *Ordered*-repair has the property that the modification process is done always in the same order. This implies that the output feasible solutions have the nodes of the lower levels of the tree rarely assignd with the value $NULL$.

The Random-repair

This repair function avoids the bias of the $NULL$ values towards the lower levels of the tree by exploring the nodes always in a random order. It guarantees that the value $NULL$ is present with the same probability in all

```
procedure Ordered-repair(ψ)
{   let {N_1, .., N_n} be the nodes of N in the basic order;

    for(i ← 1 to n)
    {   point ← ψ(N_i);
        while( container c(N_i) in the position point
        is not inside the hold or ∃K ≠ N_i a node of the path
        from the root to N_i such that c(K) in the position
        ψ(K) overlaps with c(N_i) in the position point )
            point ← φ(φ^{-1}(point) + 1);
        ψ(N_i) ← point;
    }
    Fill(ψ);
}
```

Figure 6.3: The Ordered-repair.

the nodes of the output solution.

This repair function creates an initially empty output solution – i.e. with all the nodes assignd to $NULL$ – that it modifies repeatedly. It randomly orders the nodes of the tree and in that order for each node α it executes the following modification: it considers the value assignd to α by the input solution and it make the output solution assign this value to α; if it is feasible in this solution, it is kept; else it is repeatedly changed to the next value until one that satisfy the constraints in the output solution is found. Eventually the feasible solution so far obtained is filled.

In summary the Random-repair can be described in pseudo code as in figure 6.4 where $\phi_1, .., \phi_k$ is the ordering of the positions in the grid of section 6.3.1.

6.3.5 The functions returning a random position

In this GA two functions that return a random position are used: the Random-position and the Random-position-NULL. The first one returns a random element of the set $D_x \times D_y \times \{0, 1\} \cup \{NULL\}$, each element with the same probability $\frac{1}{2n_x n_y+1}$. The second function returns a random element of the same set, $NULL$ with probability P_{NULL} and each of the other $2n_x n_y$

procedure Random-repair(ψ)
{ let $\{Q_1, .., Q_n\}$ be the nodes of N in a random order;
create the solution ψ' such that $\forall i : \psi'(Q_i) = NULL$;

for($i \leftarrow 1$ **to** n)
{ $\psi'(Q_i) \leftarrow \psi(Q_i)$
while(container $c(Q_i)$ in position $\psi'(Q_i)$
is not inside of the hold
or
$\exists\gamma$ node $\neq Q_i$ of the path
from the root to Q_i: such that $c(\gamma)$
in position $\psi'(\gamma)$ overlaps with $c(Q_i)$
in position $\psi'(N_i)$
or
$\exists\beta$ node $\neq Q_i$ of the subtree rooted at Q_i
such that $c(\beta)$ in position $\psi'(\beta)$ overlaps with $c(Q_i)$
in position $\psi'(Q_i)$)
$\psi'(Q_i) \leftarrow \phi\Big(\phi^{-1}(\psi'(Q_i)) + 1\Big)$;
}
Fill(ψ');
$\psi \leftarrow \psi'$;
}

Figure 6.4: The Random-repair.

elements with probability $\frac{1-P_{NULL}}{2n_x n_y}$.

The pseudocodes of functions Random-position and Random-position-NULL are presented in figures 6.5 and 6.6 respectively.

6.4 The main structure

The GA receives in input the BPP $(l_x, l_y, u, d_x, d_y, e, T, p, c)$ and the following parameters whose meaning will be explained in this and in the following sections:

- $numb_{iter}$, $pop_{size} \in \mathbb{N}_0$, $pop_{size} \geq 2$

function Random-position-NULL;
{ $r \leftarrow$ a random number in $\{1, .., 2n_x n_y + 1\}$;
  return $\phi(r)$;
}

Figure 6.5: The Random-position function.

function Random-position-NULL;
{ $r \leftarrow$ a random number of $]0, 1]$;
  if$(r \leq P_{NULL})$ **return** $NULL$;
  else
  { $r \leftarrow$ a random number in $\{1, .., 2n_x n_y\}$;
    return $\phi(r)$;
  }
}

Figure 6.6: The Random-position-NULL function.

- p_c, $p_{m_{A_i}}$, $p_{m_{B_i}}$, $p_{m_{C_i}}$, $p_{m_{D_i}}$, $p_{NULL} \in [0,1]$
- $shake_w$, $shake_d \in \mathbb{N}_0$, $shake_r \in]0,1]$
- $p_{m_{A_s}}$, $p_{m_{B_s}}$, $p_{m_{C_s}}$, $p_{m_{D_s}} \in [0,1]$ with $p_{m_{A_i}} < p_{m_{A_s}}$, $p_{m_{B_i}} < p_{m_{B_s}}$, $p_{m_{C_i}} < p_{m_{C_s}}$, $p_{m_{D_i}} < p_{m_{D_s}}$
- $a > 0$, $b \in \mathbb{N}_0$
- $scn_{type} \in \{A, B\}$, $selezione_{type}$
- f_{c_A}, f_{c_B}, f_{c_C}, $f_{c_D} \in [0,1]$ such that $f_{c_A} + f_{c_B} + f_{c_C} + f_{c_D} = 1$
- f_{m_A}, f_{m_B}, f_{m_C}, $f_{m_D} \in [0,1]$ such that $f_{m_A} + f_{m_B} + f_{m_C} + f_{m_D} = 1$

Initially the GA sets the mutation probabilities p_{m_A}, p_{m_B}, p_{m_C} and $p_{m_D}$to the initial values $p_{m_{A_i}}$, $p_{m_{B_i}}$, $p_{m_{C_i}}$ and $p_{m_{D_i}}$ respectively. Then it creates an initial ordered population of pop_{size} randomly generated feasible solutions $\vec{w} = (w_1, .., w_{pop_{size}})$ and the variable w_{best} is set to the w_i with the highest EU. The population $\vec{w}$ is modified repeatedly for $numb_{iter}$ times. A single modification of $\vec{w}$ is obtained by executing on it in this order a *recombination*, a *mutation* and a *selection* operation. The creation of the initial population and these operations will be presented in the sections 6.5, 6.6, 6.7 and 6.8 respectively. At the end of each iteration of the modification all the elements of $\vec{w}$ are evaluated and the w_i with the highest EU is found; if $EU(w_i) > EU(w_{best})$ then the present w_{best} is discarded and replaced with w_i. Also the best and the average value of the population of every iteration are stored and if in the last $shake_{window}$ iterations the average is greater than $shake_{ratio}$ times the best of the corresponding generation, then for $shake_{duration}$ iterations the mutation probabilities P_{m_A}, P_{m_B}, P_{m_C}, P_{m_D}, that influence the behaviour of the mutation operators, are increased and set to $P_{A_{m_s}}$, $P_{B_{m_s}}$, $P_{C_{m_s}}$ and $P_{D_{m_s}}$ respectively; after this number of iterations, the probabilities are set back to the initial values. After all the iterations are done, w_{best} is returned as output. Therefore w_{best} is the best feasible solution in the set of all the individuals obtained at the end of each iteration.

In summary the main structure of the GA can be described in pseudo code as in figure 6.7.

6.5 The creation of the initial feasible population

The initial population of feasible assignments is generated randomly. All the pop_{size} individuals are generated with the same method: an individual w_i

```
{   read BPP;
    read parameters;
    P_A ← P_{A_i}; P_B ← P_{B_i}; P_C ← P_{C_i}; P_D ← P_{D_i};
    w⃗ ← Initial-feasible-population;
    w_best ← the w_i with the highest EU;

    for (i ← 1 to pop_size)
    {   Mate(w⃗);
        Mutate(w⃗);
        Select(w⃗);

        w ← the w_l with the highest EU;
        if (EU(w) > EU(w_best)) w_best ← w;

        α_i ← the average EU of w⃗;
        β_i ← the best EU of w⃗;
        if (∀j ∈ {i − shake_w + 1, .., i}, j > 0 : α_j > β_j · shake_r )
        {   P_A ← P_{A_s}; P_B ← P_{B_s}; P_C ← P_{C_s}; P_D ← P_{D_s};
            h ← shake_d;
        }
        if (h > 0) h ← h − 1;
        else {P_A ← P_{A_i}; P_B ← P_{B_i}; P_C ← P_{C_i}; P_D ← P_{D_i};}

    }
    return w_best;
}
```

Figure 6.7: The main structure.

is generated by assigning each node of the tree with a random value of the set $(D_x \times D_y \times \{0,1\}) \cup \{NULL\}$ and by repairing this possibly unfeasible assignment by the Ordered-repair operator. The random value of the set of positions is given by the function Random-position-NULL.

The creation of the initial feasible population is described by the pseudocode of Figure 6.8.

function Initial-feasible-population;
{ **for**($i \leftarrow 1$ **to** pop_{size})
  { **for**($j \leftarrow 1$ **to** n) $w_i(N_j) \leftarrow$ Random-position-NULL;
    Ordered-repair(w_i);
  }
  return $\vec{w}$;
}

Figure 6.8: The creation of the initial feasible population.

6.6 The mating

In a GA the recombination process plays the role of the reproduction in Nature. It has the purpose of creating new individuals similar to some of those already present in the population, by mixing their genetic information. The new individuals are not too different from their parents and have features of boths. The recombination process causes the exploration of the neighbourhood of the point of the search space in which the GA has moved.

In this GA the recombination process is carried out by the procedure Mate that makes use of four genetic operators: the Upward Gentle Crossover (Crossover-A), the Downward Gentle Crossover (Crossover-B), the Ordered Brute Crossover (Crossover-C) and the Random Brute Crossover (Crossover-D). These operators receive in input a couple of feasible solutions and modify it producing a new couple of feasible solutions. Each crossover operator is associated with a frequency parameter which influences the frequency of its application. The Mate operator changes the population by choosing on the average $\frac{P_c}{2} \cdot pop_{size}$ couples from the population and by applying to each of them a crossover operator randomly chosen with a frequency dependent on its frequency parameter.

In other words in this GA during the recombination process on the average $\frac{P_c}{2} \cdot pop_{size}$ couples mate; each couple produces two children according to one of four different reproductive schemes randomly applied with a frequency dependent on their frequency parameters; each couple dies after generating the two children and the children replace their parents in the new population.

The pseudocode for the recombination process is shown in figure 6.9.

By varying the frequency parameters f_{c_A}, f_{c_B}, f_{c_C}, f_{c_D} we can vary the frequency of applications of each operator: on the average Crossover-X is applied $P_c \frac{pop_{size}}{2} f_{c_X}$ times in each application of operator Mate. If we set to 1 one of the f_{c_X} and all the others to 0, we obtain an traditional GA that uses only one crossover operator, the Crossover-X.

The individuals produced by these operators are a mixture of their parents and are reasonably similar to them. Also a couple of twins produces a couple of children identical to their parents, so the application of a crossover operator to a couple of twins has no effect in the variability of the population.

The crossover operators change the input feasible solutions v and u in the following way: they choose a node α of the tree and they swap in v and u the values of the nodes of the subtree rooted at α; the solutions v' and u' thus obtained are a raw mixture of v and u – as each of them has the nodes of a subtree with the values of the other and the rest of the nodes with the their values and they are possibly unfeasible – and are repaired.

The reparation method distinguishes the four crossovers and will be presented in the sections 6.6.1, 6.6.2, 6.6.3 and 6.6.4. The way α is chosen depends on parameter scn_{type} and is explained in subsection 6.6.5.

6.6.1 The Upward Gentle Crossover

This crossover operator – also called Crossover-A – repairs the possibly unfeasible solutions v' and u' by modifying the values of the nodes $\beta \neq \alpha$ of the path from the root to α. This modification is made by considering all the nodes of the path from the root to α, and for each of them by checking if in the correspondent position the correspondent container overlaps with any container of the nodes of the subtree rooted at α; if it overlaps with at least one of them, the correspondent node is set to $NULL$. The two solutions are then filled. See Figure 6.10.

The pseudocode is in figure 6.11.

The two solutions produced by this operator are then a mixture of the input solutions and are feasible. They are identical to the raw mixture of the parents a part from the nodes of the path from the root to α.

This operator is called upward because the reparation of the individuals obtained by just swapping the subtrees is obtained by modifying the ancestors

```
procedure Mate(w⃗);
{   I = ∅;
    for(i ← 1 to pop_size)
    {   r ← a random number ∈]0, 1];
        if(r ≤ P_c) I ← I ∪ {i};
    }
    if (|I| is odd) I ← I without its highest element;
    let I = {i_1, .., i_k} with i_1 < .. < i_k;

    if (I ≠ ∅)
        for(h ← 1 to k/2)
        {   r ← a random number in ]0, 1];
            if (r ≤ f_{c_A}) Crossover-A(w_{2h−1}, w_{2h});
            else if (r ≤ f_{c_A} + f_{c_B}) Crossover-B(w_{2h−1}, w_{2h});
                else if (r ≤ f_{c_A} + f_{c_B} + f_{c_C}) Crossover-C(w_{2h−1}, w_{2h});
                    else Crossover-D(w_{2h−1}, w_{2h});
        }
}
```

Figure 6.9: The Mate procedure.

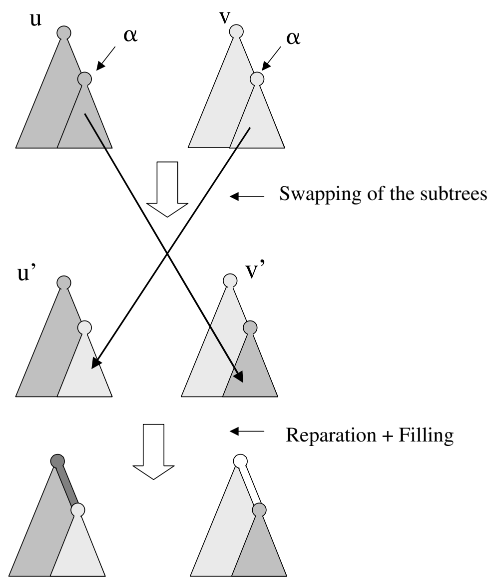


Figure 6.10: The Upward Gentle Crossover operations.

of node α and it is called gentle because it makes the effort of changing as few nodes as possible in the two solutions obtained by swapping the subtrees, thus respecting their appearance.

6.6.2 The Downward Gentle Crossover

This crossover operator – also called Crossover-B – repairs v' and u' by modifying the values of the nodes of the subtree rooted at α. The nodes of the subtree are considered in the basic order: for each of them, if the correspondent container overlaps with one of the containers of the nodes in the path from the root that node, then its value is set to $NULL$. The two solutions are then filled.

The pseudocode is in Figure 6.13. See Figure 6.12

Even with this operator the two produced solutions are a mixture of the input solutions and are feasible. Each solution is identical to the possibly unfeasible raw mixture of the parents a part from some nodes in the subtree rooted on the node in which is done the crossover.

This operator is called downward because the reparation of the individuals obtained by just swapping the subtrees is carried out by modifying the offsprings of node α and it is called gentle for the same reason of the previous operator.

procedure Crossover-A(u, v);
{ choose a node α;

  for each(β node of the subtree rooted at α)
    swap the values of u and v on the node β;

  for each(γ node $\neq \alpha$ of the path from the root to α)
    if($\exists\beta$ node of the subtree rooted at α such that $c(\gamma)$
      in position $u(\gamma)$ overlaps with $c(\beta)$ in position $u(\beta)$)
        $u(\gamma) \leftarrow NULL$;
  Fill(u);

  for each(γ node $\neq \alpha$ of the path from the root to α)
    if($\exists\beta$ node of the subtree rooted at α such that $c(\gamma)$
      in position $v(\gamma)$ overlaps with $c(\beta)$ in position $v(\beta)$)
        $v(\gamma) \leftarrow NULL$;
  Fill(v);

}

Figure 6.11: The Upward Gentle Crossover (Crossover-A).

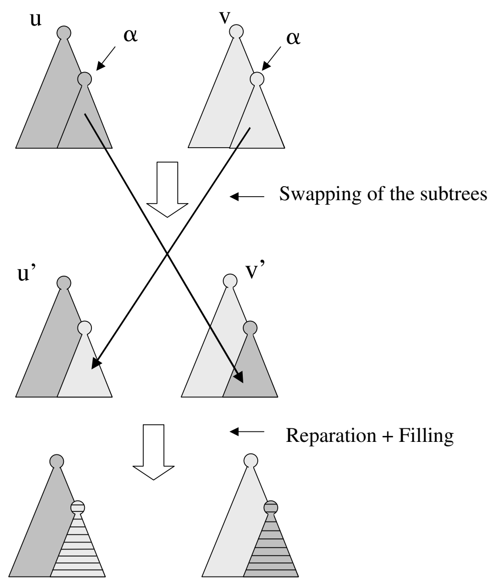


Figure 6.12: The Downward Gentle Crossover operations.

6.6.3 The Ordered Brutal Crossover

This operator – also called Crossover-C – repairs the solutions v' and u' by applying the Ordered-repair operator.

The pseudocode is in figure 6.14.

Note that the repairation procedure modifies the raw mixture of the parents only in the nodes of the subtree rooted at α.

This operator is called brutal because the reparation of v' and u' can involve the modification of all the nodes of the subtree tree and the resulting solutions can be not too much similar to those before the repairment. It is called ordered because it uses the Ordered-repair.

6.6.4 The Random Brutal Crossover

This operator – also called Crossover-D – repairs the solutions v' and u' by applying the Random-repair operator.

The pseudocode is in figure 6.15.

As with the previous operator, the nodes of the raw mixture of the parents modified by the repair operator are only those of the subtree rooted at α.

It is called brutal for the same reasons of the previous operator and it is called random because it uses the Random-repair procedure.

procedure Crossover-B(u, v);
{   choose a node α;

    for each(β node of the subtree rooted at α)
        swap the values of u and v on the node β;

    let $\{\beta_1, .., \beta_k\}$ the nodes of the subtree rooted at α
        in the basic order;

    for($i \leftarrow 1$ **to** k)
        if($\exists\gamma$ node $\neq \beta_i$ of the path from the root to β_i
            such that $c(\gamma)$ in position $u(\gamma)$ overlaps with $c(\beta_i)$
            in position $u(\beta_i)$)
                $u(\beta_i) \leftarrow NULL$;
    Fill(u);

    for($i \leftarrow 1$ **to** k)
        if($\exists\gamma$ node $\neq \beta_i$ of the path from the root to β_i
            such that $c(\gamma)$ in position $v(\gamma)$ overlaps with $c(\beta_i)$
            in position $v(\beta_i)$)
                $v(\beta_i) \leftarrow NULL$;
    Fill(v);

}

Figure 6.13: The Downward Gentle Crossover (Crossover-B).

procedure Crossover-C(u, v);
{ choose a node α;

  for each(β node of the subtree rooted at α)
    swap the values of u and v on the node β;

  Ordered-repair(u);
  Ordered-repair(v);
}

Figure 6.14: The Ordered Brutal Crossover (Crossover-C).

procedure Crossover-D(u, v);
{ choose a node α;

  for each(β node of the subtree rooted at α)
    swap the values of u and v on the node β;

  Random-repair(u);
  Random-repair(v);
}

Figure 6.15: The Random Brutal Crossover (Crossover-D).

6.6.5 The choice of the crossover node

A crossover made on a leaf results in just changing the values of that node in the two individuals and a crossover on the root leaves both the solutions unchanged. The leaves constitute a large part of the total number of nodes, especially if the the degree of each node is large. If this happens a large number of the crossovers are made on the leaves, thus not mixing very well the genetic information of the individuals. For this reason the choice of the crossover node is a delicate issue and different algorithms for choosing the crossover node may have different performances.

In this GA the crossover node can be chosen by one of three different algorithms. The crossover operator chooses the node α by an algorithm determined once and for all by parameter scn_{type}.

If $scn_{type} = A$, then the node is chosen by first choosing a level > 0 and then a node from that level, each node with the same probability. This means that the root has probability 0 of being chosen and a node α of level $l > 0$ has probability $\frac{1}{n_l p}$ with n_l the number of nodes of level l and p the number of levels > 0. This algorithm prevents the root from being chosen but let all the other nodes to be chosen, leaves included.

If $scn_{type} = B$, then the node is chosen by first randomly choosing a level > 0 with inner nodes and then randomly choosing an inner node from that level, each node with the same probability. This means that the root and the leaves have probability 0 of being chosen and an inner node α of level $l > 0$ has probability $\frac{1}{n_l p}$ with n_l the number of inner nodes of level l and p the number of levels > 0 with inner nodes. This algorithm prevents the root and the leaves from being chosen.

If $scn_{type} = C$, then the node is chosen randomly. All the nodes have the same probability $\frac{1}{n}$, root and leaves included.

6.7 The mutation

The mutation process has the purpose of introducing new genes in the population. It is fundamental in order to avoid the convergence of the algorithm to a local maximum. It moves the GA to new areas of the searching space.

In this GA the mutation process is carried out by procedure Mutate that makes use of four mutation operators called Ordered Gentle Mutation (Mutate-A), Random Gentle Mutation (Mutate-D), Ordered Brute Mutation (Mutate-B) and Random Brute Mutation (Mutate-C). These operators will be described in the sections 6.7.1, 6.7.2, 6.7.3 and 6.7.4 respectively.

The mutation operators receive in input a feasible solution and modify it more or less slightly thus creating a new feasible solution not too different from the original one. Each of them is associated with a frequency parameter that influences the frequency with which it is used and with a probability parameter that influences the strength of the modification. The mutation modifies the input population by applying on each individual one of the four mutation operators randomly chosen according to their frequency parameter.

The pseudocode of the procedure Mutate is given in figure 6.16.

procedure Mutate($\vec{w}$);
{ **for**($i \leftarrow 1$ **to** pop_{size})
  { $r \leftarrow$ a random number in $]0,1]$;
    if ($r \leq f_{m_A}$) Mutate-A(w_i);
    else if ($r \leq f_{m_A} + f_{m_B}$) Mutate-B(w_i);
      else if ($r \leq f_{m_A} + f_{m_B} + f_{m_C}$) Mutate-C($w_i$);
        else Mutate-D(w_i);
  }
}

Figure 6.16: The Mutate procedure.

By varying the frequency parameters f_{m_A}, f_{m_B}, f_{m_C}, f_{m_D} we can vary the frequency of applications of each operator: on the average Mutation-X is applied $pop_{size} f_{c_X}$ times in each application of operator Mutate. If we set to 1 the parameter f_{m_X} and all the others to 0, we obtain a traditional GA that uses only the mutation operator Mutate-X.

6.7.1 The Ordered Gentle Mutation

This operator – also called Mutate-A – receives in input a feasible solution, it sets to $NULL$ value some of its nodes and then it considers them in basic order; for each of these modified nodes α this operator assigns to α a random position and if this position is feasible for the path from the root to α and for the subtree rooted at α, it is kept and the operator goes to the next modified node; else the GA tries to find the next feasible value. The solution is then filled.

The pseudocode for operator Mutate-A is in figure 6.17.

Note that the input solution is modified and kept feasible by Mutate-A and that after the modification the input solution is changed on the average

procedure Mutate-A(ψ);
{ $I \leftarrow \emptyset$;
  for each($\alpha \in N$)
  { $r \leftarrow$ a random value in $]0, 1]$;
    if ($r \leq P_{m_A}$) add α to I;
  }

  for each($\alpha \in I$) $\psi(\alpha) \leftarrow NULL$;

  let $\{N_1, .., N_k\}$ be the elements of I (i.e. the modified
      nodes) in the basic order;

  if($I \neq \emptyset$)
    for ($i \leftarrow 1$ **to** k)
    { $\psi(N_i) \leftarrow$ Random-position;
      while$\Big($container $c(N_i)$ in position $\psi(N_i)$
      is not inside of the hold
        or
      $\exists \gamma$ node $\neq N_i$ of the path
      from the root to N_i: such that $c(\gamma)$
      in position $\psi(\gamma)$ overlaps with $c(N_i)$
      in position $\psi(N_i)$
          or
      $\exists \beta$ node $\neq N_i$ of the subtree rooted at N_i
      such that $c(\beta)$ in position $\psi(\beta)$ overlaps with $c(N_i)$
      in position $\psi(N_i)\Big)$
        $\psi(N_i) \leftarrow \phi\Big(\phi^{-1}(\psi(N_i)) + 1\Big)$;
    }

  Fill(ψ);

}

Figure 6.17: The Ordered Gentle Mutation (Mutate-A).

in $P_{m_A}n$ nodes. Also note that the line in which all the nodes of I are set to $NULL$ is need-less: it is present here just to emphasize the similarities with the random Gentle mutate operator.

This operator is called Gentle because it tries to keep as much as possible the original aspect of the solution. It is called ordered because the modified nodes are repaired in the basic order and it is called Gentle because it tries keep the original appearance of the input feasible solution.

6.7.2 The Random Gentle Mutation

This operator – also called Mutate-D – is very similar to the Ordered Gentle Mutation. The only difference is that the mutated nodes are considered in a random order.

The pseudocode for operator Mutate-D is in figure 6.18.

The input solution is modified and kept feasible by Mutate-D. After the modification the input solution is changed on the average in $P_{m_D}n$ nodes.

It is called random because the reparation of the modified nodes is done in a random order and it is called Gentle for the same reason of Mutate-A.

6.7.3 The Ordered Brutal Mutation

This operator – also called Mutate-B – is very simple: it chooses a set of nodes and it assigns them a random value by the Random-position-NULL; then it repairs the so obtained solution by the ordered repair operator.

The pseudocode for operator Mutate-B is in figure 6.19.

Note that the number of nodes in which the feasible output solution differs from the input one is not foreseeable on the average, because the Ordered-repair operator may change an unforeseeable number of nodes of the solution.

The name of this operator has been chosen with the same conventions of the names of the previous operators.

6.7.4 The Random Brutal Mutation

This operator – also called Mutate-C – is very similar to the operator ordered brutal mutation: it chooses a set of nodes and it assigns them a random value; then it repairs the so obtained solution by the random repair operator. The only difference is in the repair operator.

The pseudocode for operator Mutate-C is in figure 6.20.

Again the name of this operator has been chosen with the same conventions of the names of the other operators.

```
procedure Mutate-D(ψ);
{   I ← ∅;
    for each(α ∈ N)
    {   r ← a random value in ]0, 1];
        if (r ≤ P_{m_D}) add α to I;
    }

    for each(α ∈ I) ψ(α) ← NULL;

    let {N_1, .., N_k} be the elements of I (i.e. the modified
            nodes) in a random order;

    if(I ≠ ∅)
        for (i ← 1 to k)
        {   ψ(N_i) ← Random-position;
            while(container c(N_i) in position ψ(N_i)
            is not inside of the hold
                or
            ∃γ node ≠ N_i of the path
            from the root to N_i: such that c(γ)
            in position ψ(γ) overlaps with c(N_i)
            in position ψ(N_i)
                    or
            ∃β node ≠ N_i of the subtree rooted at N_i
            such that c(β) in position ψ(β) overlaps with c(N_i)
            in position ψ(N_i))
                ψ(N_i) ← φ(φ^{-1}(ψ(N_i)) + 1);
        }

    Fill(ψ);

}
```

Figure 6.18: The Random Gentle Mutation (Mutate-D).

procedure Mutate-B(ψ);
{ **for each**($\alpha \in N$)
  { $r \leftarrow$ a random value in $]0,1]$;
    if $(r \leq P_{m_B})$ $\psi(\alpha) \leftarrow$ Random-position-NULL;
  }

  Ordered-repair(ψ);

}

Figure 6.19: The Ordered Brutal Mutation (Mutate-B).

procedure Mutate-C(ψ);
{ **for each**($\alpha \in N$)
  { $r \leftarrow$ a random value in $]0,1]$;
    if $(r \leq P_{m_C})$ $\psi(\alpha) \leftarrow$ Random-position;
  }

  Random-repair(ψ);

}

Figure 6.20: The Random Brutal Mutation (Mutate-C).

6.8 The selection

In a GA the selection process plays the role of the natural selection in Nature. It has the purpose of letting the strongest individuals to survive and of killing the weakest and it is fundamental for the improvement of the population.

One of three selection mechanisms can be used in this GA and is determined by parameter $selection_{type}$: the *fixed* selection, the $f(\frac{val}{MAX})$-selection and the well known *proportional* selection corresponding to $selection_{type} = A, B$ and C respectively. These selection methods will be commented in the subsections 6.8.1, 6.8.2 and 6.8.3. However all of them have the same structure.

The selection process is carried out by procedure Select. Population $\vec{w}$ is modified by Select. Each individual w_i is associated with a number

$$p_i = f_X(i, EU(w_1), .., EU(w_{pop_{size}}))$$

with X the $selection_{type}$ and function $f_X \geq 0$ such that $F = \sum_{i=1}^{pop_{size}} p_i > 0$. Vector $\vec{q} = (q_0, .., q_{pop_{size}})$ is built such that $q_0 = 0$, and $q_i = q_{i-1} + \frac{p_i}{F}$, for $i \in \{1, .., pop_{size}\}$. Then a new temporary population $\vec{v}$ is built in the following way: for $i = 1, .., pop_{size}$ a random number r of the set $]0, 1]$ is generated and v_i is w_j with $j \in \{1, .., pop_{size}\}$ such that $r \in]q_{j-1}, q_j]$. Then $\vec{w}$ becomes $\vec{v}$.

In summary the selection mechanism of the GA can be described in pseudo code as in figure 6.21.

In this GA the selection is based on the idea of the roulette wheel, as in the first example of GA in [1, page 32]. As a matter of fact this process can be imagined as follows. Each individual w_i is associated with a value p_i dependent on the value $EU(w_i)$, the EU of all the individuals and the parameter $selection_{type}$. A roulette wheel is built with pop_{size} slots, each slot corresponding to an individual w_i and with an angle proportional to p_i. The population $\vec{w}$ is modified by the selection in the following way: a population $\vec{v}$ is built such that individual v_i is the individual of population $\vec{w}$ correspondent to the slot chosen by making the wheel spin; then $\vec{w}$ becomes $\vec{v}$. Therefore p_i is proportional to the probability of individual w_i surviving for the next generation.

Note that parameter $selection_{type}$ decides once and for all the selection mechanism of the iterations: if parameter $selection_{type} = X$ then in all the iterations the numbers p_i will be created by the function f_X.

Function f_A is defined as

$$\begin{aligned} f_A : \{1, .., pop_{size}\} \times \mathbb{R}^{+\,pop_{size}} &\rightarrow \mathbb{R}_0^+ \\ (i, b_1, .., b_{pop_{size}}) &\longmapsto g(rank_i) \end{aligned}$$

procedure Select($\vec{w}$)
{ $X \leftarrow selection_{type}$;
  for $(i \leftarrow 1$ **to** $pop_{size})$ $p_i \leftarrow f_X(i, EU(w_1), .., EU(w_{pop_{size}}))$;
  $F \leftarrow \sum_{i=1}^{pop_{size}} p_i$;
  $q_0 \leftarrow 0$;
  for $(i \leftarrow 1$ **to** $pop_{size})$ $q_i \leftarrow q_{i-1} + \frac{p_i}{F}$;
  for $(i \leftarrow 1$ **to** $pop_{size})$
  { $r \leftarrow$ a random number of the set $]0, 1]$;
    $j \leftarrow$ the only one $j \in \{1, .., pop_{size}\}$ such that $r \in]q_{j-1}, q_j]$;
    $v_i \leftarrow w_j$;
  }
  $\vec{w} \leftarrow \vec{v}$;
}

Figure 6.21: The selection.

with $\forall x \in \mathbb{R}^+, g(x) = \frac{1}{ax+1}$ and $a > 0$ and with $rank_i = j$ such that $b_i = \bar{b}_j$ where $\{\bar{b}_0, .., \bar{b}_k\} = \{b_l | = l, .., pop_{size}\}$ such that $\bar{b}_h > \bar{b}_{h+1}, \forall h$ (i.e. $rank_i$ is the index from 0 to k of b_i in the sequence $(\bar{b}_0, .., \bar{b}_k)$ increasing ordering of $\{b_l | = l, .., pop_{size}\}$).

Function f_B is defined as

$$f_B : \{1, .., pop_{size}\} \times \mathbb{R}^{+pop_{size}} \rightarrow \mathbb{R}^+$$
$$(i, b_1, .., b_{pop_{size}}) \longmapsto \begin{cases} 1 & \text{if all } b_i = 0 \\ f\left(\frac{b_i}{max\{b_l | l=1,..,pop_{size}\}}\right) & \text{else} \end{cases}$$

with $\forall x \in [0, 1], f(x) = x^b$.

Function f_C is defined as

$$f_C : \{1, .., pop_{size}\} \times \mathbb{R}^{+pop_{size}} \rightarrow \mathbb{R}^+$$
$$(i, b_1, .., b_{pop_{size}}) \longmapsto \begin{cases} 1 & \text{if all } b_i = 0 \\ b_i & \text{else} \end{cases}$$

Because p_i is proportional to the probability of w_i appearing in the next generation, p_i must be bigger for the best individuals. Thus function f must the have the property that $\forall X \in \{A, B, C\}$ and $\forall \vec{b} \in \mathbb{R}^{+pop_{size}}$, if $b_i < b_j$ then $f_X(i, \vec{b}) \leq f_X(j, \vec{b})$.

Note that whatever is the parameter $selection_{type}$, is $F > 0$.

Also note that the individuals of the population produced by the selection process are chosen from those of the population undergoing the selection and some of them may repeat. Until a certain extent the repetition of some individuals is not a bad event as those who repeat are those that have a bigger p_i and so a bigger EU. But in a population there must not be not too many twins. In such a population the positive effect of recombination disappear, because when a crossover operator is applied to a couple of identical individuals it produces children identical to their parents. The only cause for variability is the mutation and the GA risks to stick on a local maximum.

6.8.1 The Fixed Selection

If parameter $selection_{type} = A$ the GA will use the fixed selection. With this selection method the probability of an individual surviving to the next generation is proportional to $p_i = \frac{1}{a \cdot rank_i + 1}$ with $rank_i =$ the index j such that $EU(w_i) = \bar{b}_j$ if $\{EU(w_l) | l = 1, .., pop_{size}\} = \{\bar{b}_0, .., \bar{b}_k\}$, with $\bar{b}_0 > .. > \bar{b}_k$. The integer $rank_i$ is called rank of w_i. This means, for example, that if w_i has the highest EU in the population, its rank will be $j = 0$ and will be $p_i = g(0) = 1$; and if w_h will have the second best EU, then its rank will be $j = 1$ and will be $p_h = g(1) = \frac{1}{a+1}$; and so on.

The dependence of p_i from the rank of w_i is given by function $g(x) = \frac{1}{ax+1}$ with $a > 0$ (see figure 6.22). Note that whatever is parameter a, the lower

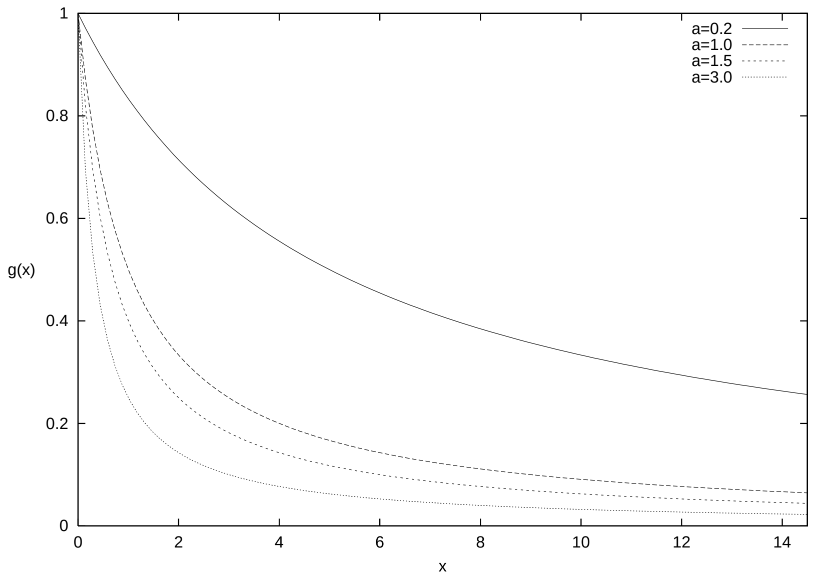


Figure 6.22: Function g with different values of parameter a

the $EU(w_i)$ the higher the rank of w_i, so the lower p_i and the lower the probability of survival for individual w_i.

By varying parameter a we can vary the severity of the selection. Informally the severity of a selection method measures its tendency to keep for the next generation the individuals with higher EU and to refuse those with lower EU. In this case the higher a, the higher the probability of survival for the individuals with high EU with respect to those with low EU. As a matter of fact $\forall j, i \in \mathbb{N}, j > i$ the function $r(a) = \frac{g(i)}{g(j)} = \frac{aj+1}{ai+1}$ is increasing and $\forall a \in \mathbb{R}_0^+, r(a) > 1$. So, given a population $\vec{w}$ and two individuals w_h and w_k such that $EU(w_h) > EU(w_k)$, we have $rank_h = i < rank_k = j$ and so $\frac{p_h}{p_k} = \frac{g(i)}{g(j)}$ is increasing with a and ≥ 1. So by increasing parameter a we increase the severity of the selection process. Note that $\forall j > 0, \frac{g(0)}{g(j)} = aj + 1$. This means that the probability of survival of the individuals with the higher EU can be made as much bigger than that of another individual with lower EU as we want just by increasing a.

In the choice of parameter a we must be careful not to make the selection too severe nor too allowing. If the selection is too severe, we risk to obtain a population with many twins stopping the evolution of the population (as explained in page 77) before the global maximum is found. If it is too allowing, the worst individuals may have almost the same chances to pass to the next generation as the best. Hence the population evolves just randomly, with no average improvement.

6.8.2 The $f(\frac{val}{MAX})$-selection

If parameter $selection_{type} = B$ the GA will use the $f(\frac{val}{MAX})$-selection. With this selection method if all the individuals have $EU = 0$ – i.e. they are all $NULL$ functions – they will all pass to the next generation and the new population will be the same. Else the probability of an individual w_i surviving to the next generation is proportional to $f(\frac{EU(w_i)}{MAX\{EU(w_j)\}})$ with $f(x) = x^b$.

The dependence of p_i on $\frac{EU(w_i)}{MAX\{EU(w_j)\}}$ is given by function $f(x) = x^b$ with $b \in \mathbb{N}_0$ (see figure 6.23). Note that whatever is parameter $b \in \mathbb{N}_0$, the higher the EU of an individual w_i, the higher its p_i and so its probability to survive in the next generation, because the closer is an individual to the MAX, the bigger its probability.

By varying parameter b we can vary the severity of the selection. The higher b, the higher the probability of survival for the individuals with high EU with respect to those with low EU. As a matter of fact given a population $\vec{w}$ and two individuals w_i and w_j with $EU(w_i) > EU(w_j)$ the ratio $r(b) =$

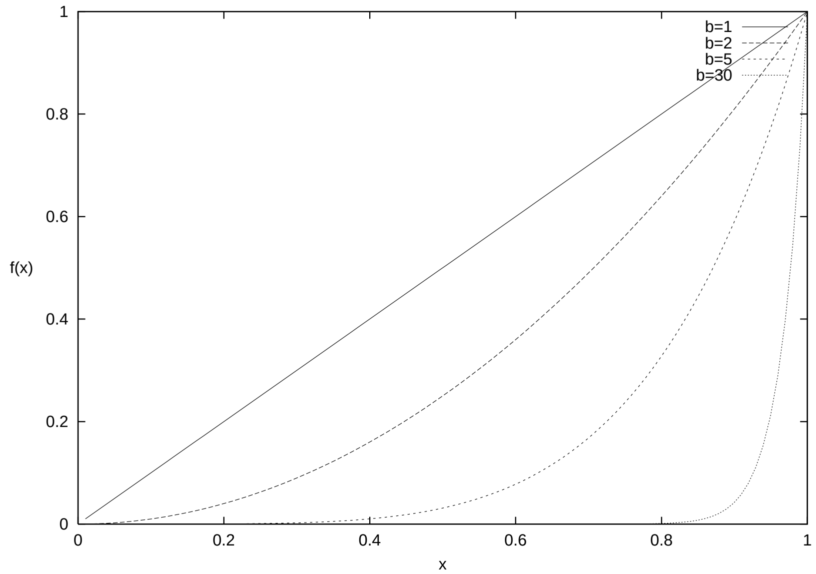


Figure 6.23: Function f with different values of parameter b

$\frac{p_i}{p_j} = \left(\frac{EU(w_i)}{EU(w_j)}\right)^b$ is increasing with b and > 1. So by increasing parameter b we increase the severity of the selection process. Also $\lim_{b \to +\infty} \frac{p_i}{p_j} = +\infty$. This means that the probability of survival of best individual can be made as much bigger than that of another individual with lower EU as we want just by increasing b.

6.8.3 The Proportional Selection

If parameter $selection_{type} = C$ the GA will use the *proportional* selection. With this selection method if all the individuals have $EU = 0$ – i.e. they are all $NULL$ functions – they will all pass to the next generation and the new population will be the same. Else the probability of an individual surviving to the next generation is proportional to its EU. With this method we cannot vary the severity of the selection. In some experiments this selection method have proved unable to let pass the best individuals to the next generation. With this method, when $\frac{max\{EU(w_i)\}}{min\{EU(w_i)\}} \simeq 1$ the probability of survival of the best individuals is almost the same of the worst.

Note that if we use parameter $selection_{type} = B$ and $b = 1$ we have a selection mechanism identical to the proportional one.

6.9 The Shake feature

The shake mechanism has been introduced in this GA with the purpose of avoiding the premature convergence of the GA and can be used with any GA. The shake starts and influence the evolutionary process when the individuals of the population in the last generations have more or less the same value, situation often due to the presence of twins or very similar individuals. In such a situation a traditional GA would produce with great probability even more twins and the population would not improve, as new individuals would be very improbabibly generated. When the shake starts, the mutation probabilities are increased for a certain number of generations thus increasing the probability of producing new individuals. Then the probabilities are set back to the original values. The effect of the shake action is the insertion in the population of new individuals from new areas of the search space. This individuals, mating with the old ones may produce better solutions, as they introduce new genetic information in the population.

In other words, if we imagine the population of a GA as a group of individuals moving more or less together and trying to climb the hills of an artificial landscape, the shake event occurs when the population has reached the top of a local hill and its effect is that of migrating the population in a new area of the landscape where the population starts climbing the local hill again. Yet the problem of preventing the population from moving to and fro in the same two hills has not been considered in this work.

Usually after a shake event has occurred there is an average worsening of the population as some of the good genes are lost, but after a certain number of generations the new genetic information, if good, can bear its fruits. The typical curve for the best and the average versus the generations is given in figure 6.24.

In this figure the shake happens after the average has been more than 0.995 of the best for 10 generations and it lasts 1 generation. During the shake, the mutation probabilities become five times bigger.

The shake feature should be used combined with a strong selection pressure and with a weak mutation.

Note that if we set $shake_r = 1$ the mutation probabilities will never be increased and the GA behaves as a traditional GA with no shake mechanism.

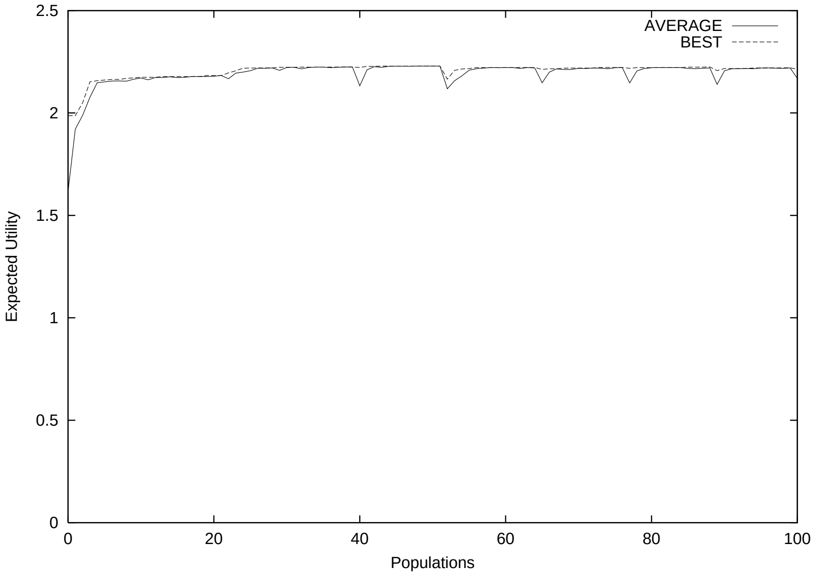


Figure 6.24: The average and the best curves with the shake feature.

Chapter 7

A GA for BCSPs

7.1 Overview

In this chapter a GA that tries to find a solution with the highest EU of a given BCSP is described.

7.2 Introduction

In Chapter 6 we have presented a GA that tries to find an optimal solution of a given BPP. This algorithm, by modifying slightly some of its genetic operators, can be easily extended and become a GA to find an optimal solution of a given BCSP. The GA of Chapter 6 is then a particular case of the algorithm described in this chapter.

7.3 Basic concepts and operations

The following definition will be often used in this chapter.

Definition 18 Given a BCSP C, one of its assignements ψ and one of its nodes α, we say that *the path of α is feasible in ψ* iff the assignement of variables I such that $\forall i \in I, f(i) = \psi(\beta)$ with $\beta \in B$ and $c(\beta) = i$ is feasible for P, with $B = \{\beta \in N \mid \psi(\beta) \neq NULL$ and β is of the path from the root to $\alpha\}$ and $I = \{c(\beta) \mid \beta \in B\}$. $\square$

The extended GA is based on conceps and basic operators similar to those of section 6.3 that will be summarized in this section.

As soon as the GA reads the input BCSP, $\forall i \in \{1, .., m\}$ it orders the elements of $D_i \cup \{NULL\}$ so that $NULL$ is the last element of this order.

We will call ϕ_i the ordering of $D_i \cup \{NULL\}$. Also it orders the nodes of the tree in the same basic order of section 6.3.2.

A fill operation is used even by the extended GA. Its purpose is that of reducing the presence of $NULL$ values in the solution so as to increase its EU.When the fill operator is applied to a solution ψ the nodes of the tree of the BCSP are visited in the basic order and for each node α if $\psi(\alpha) = NULL$ the GA tries to find the first element of $D_{c(\alpha)}$ in the order such that the path of α is feasible in ψ; if it does not exists, the $NULL$ value is left in $\psi(\alpha)$. The pseudocode for operator Fill can be found in Figure 7.1.

```
procedure Fill(ψ)
{   let N = {N_1, .., N_n} be the basic ordering of the nodes;

    for(i ← 1 to n)
        if(ψ(N_i) = NULL)
        {   j ← 1;
            ψ(N_i) ← φ_i(j);
            while(the path of N_i is not feasible in ψ)
            {   j ← j + 1;
                ψ(N_i) ← φ_i(j);
            }
        }
}
```

Figure 7.1: The Fill procedure for BCSPs.

The two reparation operators of section 6.3 are used in the extended GA, but they are slightly modified. When applied to the possibly unfeasible assignment ψ, the Ordered-repair operator visits the nodes of the tree in the basic order and for each node α, if the path of α is not feasible in ψ then the operator looks for a value $v \in D_{c(\alpha)} \cup \{NULL\}$ following $\psi(\alpha)$ in the order of this set such that if $\psi(\alpha) = v$, the path of α is feasible in ψ; then it fills ψ. The pseudocode of this operator is in Figure 7.2.

When the Random-repair is applied to the possibly infeasible assignement ψ, it creates an initially empty solution ψ'; it randomly orders the nodes of the tree and for each node α in that order it executes the following modifications: it assigns $\psi(\alpha)$ to $\psi'(\alpha)$ and it looks for a value $v \in D_{c(\alpha)} \cup \{NULL\}$ following $\psi'(\alpha)$ in the order of this set such that if $\psi'(\alpha) = v$, $\forall \beta$ node of the subtree rooted at α the path of β is feasible in ψ'; then it assigns this value to $\psi'(\alpha)$;

```
procedure Ordered-repair(ψ)
{   let {N_1, .., N_n} be the nodes of N in the basic order;

    for(i ← 1 to n)
        while(the path of N_i is not feasible for ψ)
            ψ(N_i) ← φ_i(φ_i^{-1}(ψ(N_i)) + 1);
    Fill(ψ);
}
```

Figure 7.2: The Ordered-repair for BCSPs.

at the end it fills ψ' and it replaces ψ with psi'. The pseudocode of Random-repair can be found in Figure 7.3. If ψ is feasible, these operators leave ψ unchanged.

The functions returning a random value of the domain of a variable, used in the mutation operators and in the procedure Initial-feasible-population, behave in the same way as the corrispondent ones of the GA for BPPs, but they must be slightly adapted to the more general situation of the BCSPs. The Random-position-NULL receives in input the variable i and it returns $NULL$ with probability P_{NULL} and an element of D_i with probability $1 - P_{NULL}$; in this case each element of D_i can be returned with probability $\frac{1-P_{NULL}}{|D_i|}$. The pseudocode is in Figure 7.4. The Random-position receives in input the variable i and it returns an element of $D_i \cup \{NULL\}$, each element with the same probability $|D_i| + 1$. The pseudocode is given in figure 7.5.

7.4 The main structure

The GA for BCSPs depends on the same parameters of the GA for BPPs and has the same main structure presented in Section 6.4. The procedures used by the main procedure, Mate, Mutate and Select of sections 6.6, 6.7 and 6.8 respectively and the 2 brute crossovers of sections 6.6.3 and 6.6.4 respectively are exactly the same. The only differences are in the Initial-feasible-population procedure, in the 2 gentle crossovers and in the mutation operators and in the conceps and operators spoken of in Section 7.3.

procedure Random-repair(ψ)
{ let $\{Q_1, .., Q_n\}$ be the nodes of N in a random order;

create the solution ψ' such that $\forall i, \psi'(Q_i) = NULL$;

for($i \leftarrow 1$ **to** n)
{ $\psi'(Q_i) \leftarrow \psi(Q_i)$
while$\Big($the path of Q_i is not feasible for $\psi'\Big)$
$\psi'(Q_i) \leftarrow \phi_i(\phi_i^{-1}(\psi'(Q_i)) + 1)$;
}
Fill(ψ');
$\psi \leftarrow \psi'$;
}

Figure 7.3: The Random-repair for BCSPs.

7.5 The changed Initial-feasible-population

The Initial-feasible-population is very similar to the correspondent procedure of the GA for BPPs. The only difference is in the use of the Random-position-NULL function that in this case requires the argument $c(\alpha)$. In Figure 7.6 the pseudocode of the new Initial-feasible-population is shown.

7.6 The changed crossovers

As stated in the previous section, the Gentle Upward and the Gentle Downward Crossover must be slightly modified.

The crossover operators swap the values of the nodes of the subtree rooted at some chosen node α in the two input solutions u and v. Then the two possibly unfeasible assignements u' and v' thus obtained are repaired. The difference between these two crossover is the reparation method. The Upward Gentle Crossover repairs u' and v' by changing the values of the nodes $\gamma \neq \alpha$ of the path from the root to α starting from the alpha's father and going towards the root as shown in Figure 7.7.

The Downward Gentle Crossover repairs u' and v' by changing the nodes of the subtree rooted at α in the way shown in Figure 7.8. Note that the fill function will only modify the values of the nodes $\gamma \neq \alpha$ of the path from the

```
function Random-position-NULL(i);
{   r ← a random number of ]0,1];
    if(r ≤ P_NULL) return NULL;
    else
    {   r ← a random number in {1,..,|D_i|};
        return φ_i(r);
    }
}
```

Figure 7.4: The Random-position-NULL function for BCSPs.

```
function Random-position(i);
{   r ← a random number of {1,..,|D_i| + 1};
    return φ_i(r);
}
```

Figure 7.5: The Random-position function for BCSPs.

```
function Initial-feasable-population;
    for(i ← 1 to pop_size)
    {   for each(α ∈ N) w_i(α) ← Random-position-NULL(c(α));
        Ordered-repair(w_i);
    }
    return w⃗;
}
```

Figure 7.6: The creation of the initial feasible population for BCSPs.

```
procedure Crossover-A(u, v);
{   choose a node α;

    for each(β node of the subtree rooted at α)
        swap the values of u and v on the node β;

    Let {γ_1, .., γ_k} be the nodes γ ≠ α of the path from the root to α
    such that ∀i, γ_{i+1} is the father of γ_i;

    Let u' be a solution identical tu u;

    for(i ← 1 to k) u'(γ_i) ← NULL;

    for(i ← 1 to k)
    {   u'(γ_i) ← u(γ_i);
        if(∃β node of the subtree rooted at α such that
            the path of β is not feasible in u')
                u'(γ_i) ← NULL;
    }

    Fill(u);

    Let v' be a solution identical tu v;

    for(i ← 1 to k) v'(γ_i) ← NULL;

    for(i ← 1 to k)
    {   v'(γ_i) ← v(γ_i);
        if(∃β node of the subtree rooted at α such that
            the path of β is not feasible in v')
                v'(γ_i) ← NULL;
    }

    Fill(v);
}
```

Figure 7.7: The Upward Gentle Crossover (Crossover-A) for BCSPs

root to α in the first case and the values of the nodes of the subtree rooted at α in the second case.

```
procedure Crossover-B(u, v);
{   choose a node α;

    for each(β node of the subtree rooted at α)
        swap the values of u and v on the node β;

    let {β_1, .., β_k} the nodes of the subtree rooted at α
        in the basic order;

    for(i ← 1 to k)
        if(the path of β_i is not feasible in u) u(β_i) ← NULL;
    Fill(u);

    for(i ← 1 to k)
        if(the path of β_i is not feasible in v) v(β_i) ← NULL;
    Fill(v);
}
```

Figure 7.8: The Downward Gentle Crossover (Crossover-B) for BCSPs.

7.7 The mutations

The mutations operators must be slightly changed too. The brute mutations are very similar to the corrispondet procedures of the GA for BPPs. The only difference is that the corrispondent random position function is called with the argument $c(\alpha)$. The pseudocodes for these mutation operators are given in figures 7.9 and 7.10.

The gentle mutation operators change the input solution in some nodes. After choosing which nodes to mutate and after mutating them – i.e. assigning them a $NULL$ value – these operands repair them in some order. The reparation of a mutated node α of the assignements ψ consists in assigning to $\psi(\alpha)$ a random value $v \in D_{c(\alpha)} \cup \{NULL\}$ and finding the first value $v' \in D_{c(\alpha)} \cup \{NULL\}$ following v in the ordering of this set such that when assigned to $\psi(\alpha)$, the assignement ψ is feasible. The only difference between

procedure Mutate-C(ψ);
{ **for each**($\alpha \in N$)
{ $r \leftarrow$ a random value in $]0,1]$;
if ($r \leq P_{m_C}$) $\psi(\alpha) \leftarrow$ Random-position-NULL($c(\alpha)$);
}

Ordered-repair(ψ);

}

Figure 7.9: The Ordered Brutal Mutation (Mutate-B) for BCSPs.

procedure Mutate-C(ψ);
{ **for each**($\alpha \in N$)
{ $r \leftarrow$ a random value in $]0,1]$;
if ($r \leq P_{m_C}$) $\psi(\alpha) \leftarrow$ Random-position($c(\alpha)$);
}

Random-repair(ψ);

}

Figure 7.10: The Random Brutal Mutation (Mutate-C) for BCSPs.

```
procedure Mutate-A(ψ);
{   I ← ∅;
    for each(α ∈ N)
    {   r ← a random value in ]0,1];
        if (r ≤ P_{m_A}) add α to I;
    }

    for each(α ∈ I) ψ(α) ← NULL;

    let {N_1,..,N_k} be the elements of I (i.e. the modified
            nodes) in the basic order;

    if(I ≠ ∅)
        for (i ← 1 to k)
        {   ψ(N_i) ← Random-position(c(N_i));
            while(∃β ≠ N_i, β node of the subtree rooted at α
                such that the path of β is not feasible in ψ)
                    ψ(N_i) ← φ_i(φ_i^{-1}(ψ(N_i)) + 1);
        }

    Fill(ψ);

}
```

Figure 7.11: The Ordered Gentle Mutation (Mutate-A) for BCSPs.

the Ordered Gentle Mutation and the Random Gentle mutation is the order in which the mutated nodes are repaired: the basic and a random order respectively. The pseudocode for the two perators is given in figures 7.11 and 7.12.

procedure Mutate-D(ψ);
{ $I \leftarrow \emptyset$;
    for each($\alpha \in N$)
    { $r \leftarrow$ a random value in $]0, 1]$;
        if ($r \leq P_{m_A}$) add α to I;
    }

    for each($\alpha \in I$) $\psi(\alpha) \leftarrow NULL$;

    let $\{N_1, .., N_k\}$ be the elements of I (i.e. the modified
            nodes) in a random order;

    if($I \neq \emptyset$)
        for ($i \leftarrow 1$ **to** k)
        { $\psi(N_i) \leftarrow$ Random-position($c(N_i)$);
            while$\Big(\exists \beta \neq N_i$, β node of the subtree rooted at α
                such that the path of β is not feasible in $\psi\Big)$
                    $\psi(N_i) \leftarrow \phi_i\Big(\phi_i^{-1}(\psi(N_i)) + 1\Big)$;
        }

    Fill(ψ);

}

Figure 7.12: The Random Gentle Mutation (Mutate-D) for BCSPs.

Chapter 8

The Parameters Tuning

8.1 Overview

This chapter presents the results of some experiments designed with the pourpose of finding some assignments of parameters that give the GA an average good performance.

8.2 Introduction

The GA described in Chapter 8 depends on several parameters which influence its performance and behaviour greatly and must be chosen before the algorithm is run. It is interesting and useful to know whether some assignments of parameters are better than others.

Yet finding out which is the best assignment is very difficult because of the large number of possibilities and because of the large variety of BPPs that the algorithm can solve. As a matter of fact there can be BPPs with trees of very different shapes and set of containers with completely different dimensions. An assignment of parameters can be good with a class of BPPs but bad with another. In order to make a complete study it would be necessary to fix a finite set of values for each parameter with a continuous domain and try all the possible assignments of parameter values on a large number of problems for several runs. Yet this systematic approach is impossible in the time at our disposal.

We have then decided to do experiments only on few assignments that seemed to be good. Not all the reasonable assignments have been tested. Often the results of some experiments gave interesting hints for further tests: the whole final set of experiments was not planned in advance, but is the result of an historical process where the new direction of the tests was influenced by the results of the past experiments. Even though this set of tests is not systematic at all and not the ideal one, it leaded us to a GA with better performance than the initial algorithm.

It is obvious that the performance of a GA improves as we increase the population size and the number of iterations. The aim of the experiments of this chapter is then that of finding good assignments of parameters with $numb_{iter} = 100$ and $pop_{size} = 100$. We hope that these assignments with different $numb_{iter}$ and pop_{size} are good as well.

It would be also interesting to know if for particular classes of problems the GA performs better with particular assignments of parameters, but this question has not been considered in this thesis.

8.3 The experiments

An experiment on a assignment of parameters consisted in running the GA with these parameters for three times on a BPP and on recording the best value at the end of each run.

The BPPs used in these tests have been created by the random problem generator described in Appendix A and can be found in Appendix B. They differ mostly on the shape of the trees, being some tall and slim and some fat and short, while the type of containers is almost the same. Of course they do not represent well the set of all possible problems, but it was not possible to do the experiments with a larger number of problems.

As far as the parameters are concerned, in general note that:

- if $shake_r = 1$ then the parameters $shake_d$, $shake_w$, $P_{m_{A_s}}$, $P_{m_{B_s}}$, $P_{m_{C_s}}$, $P_{m_{D_s}}$ do not influence the GA;

- if $select_{type} = A$ then parameter b does not influence the GA;

- if $select_{type} = B$ then parameter a does not influence the GA;

- $\forall X \in \{A, B, C, D\}$ such that $f_{m_X} = 0$ then $P_{m_{X_i}}$ and $P_{m_{X_s}}$ do not influence the GA.

For all the assignments of parameters tested in this chapter it is $pop_{size} = 100$, $numb_{iter} = 100$, $P_c = 0.5$, $P_{NULL} = 0.3$, $b = 30$.

8.4 The experiments on single genetic components

Initially some assignments of parameters which cause the GA to employ a single crossover and a single mutation have been tested. The porpose was that of examining the behaviour of the single genetic components with all the selection and the selection of crossover node types. At the time of these experiments mutation C and D and crossover D had not been implemented yet and the tests regarded only mutation A and B and crossover A,B and C. Two sets of experiments have been done.

8.4.1 The first set of experiments

The first set of experiments consisted in testing all the possible assignments such that:

- $\exists X \in \{A, B\}$ such that $f_{m_X} = 1$
- $\exists X \in \{A, B, C\}$ such that $f_{c_X} = 1$
- $a = 3$
- $shake_r = 1$
- $P_{m_{A_i}} = P_{m_{B_i}} = P_{m_{C_i}} = P_{m_{D_i}} = 0.01$

that is all the assignments of parameters with $a = 3$, $P_{m_{A_i}} = P_{m_{B_i}} = P_{m_{C_i}} = P_{m_{D_i}} = 0.01$ that cause the GA to use only[1] mutation A or B and only crossover A,B or C and not to use the shake feature.

One of these assignments is called $XYZK$ iff :

- $scn_{type} = X$
- $selection_{type} = Y$
- $f_{c_Z} = 1$
- $f_{m_K} = 1$

[1]Note that f_{c_A}, f_{c_B}, f_{c_C}, $f_{c_D} \in [0, 1]$ and $f_{c_A} + f_{c_B} + f_{c_C} + f_{c_D} = 1$ and that f_{m_A}, f_{m_B}, f_{m_C}, $f_{m_D} \in [0, 1]$ and $f_{m_A} + f_{m_B} + f_{m_C} + f_{m_D} = 1$, as stated in Section 6.4; so if $f_{c_X} = 1$, then $\forall Y \in \{A, B, C, D\} \setminus \{X\}, f_{c_Y} = 0$ and if $f_{m_X} = 1$, then $\forall Y \in \{A, B, C, D\} \setminus \{X\}, f_{m_Y} = 0$.

so the set of names of the assignments of parameters tested in the first set of experiments is $\{A, B, C\} \times \{A, B, C\} \times \{A, B, C\} \times \{A, B\}$.

The experiments consisted in running the GA with these assignments of parameters for three times on four problems, each time recording the EU of the best found solution. The details of the problems can be found in Appendix B.

Tables 8.1, 8.2, 8.3, 8.4 and 8.5 show the results of these experiments.

	rinput4	rinput6	rinput7	rinput8	average
AAAA	1.353445	1.946178	3.192496	1.568283	2.015100
AAAB	1.405830	1.960601	3.413845	1.655644	2.108980
AABA	1.299086	1.925874	3.330314	1.565640	2.030229
AABB	1.413915	1.960575	3.364486	1.659877	2.099714
AACA	1.355289	1.935350	3.142789	1.605473	2.009726
AACB	1.414226	1.960539	3.315673	1.640201	2.082660
ABAA	1.353445	1.903734	3.287117	1.595914	2.035052
ABAB	1.395574	1.958503	3.448740	1.635945	2.109691
ABBA	1.361976	1.953936	3.440931	1.590426	2.086817
ABBB	1.365164	1.959599	3.338532	1.643052	2.076586
ABCA	1.360614	1.911531	3.244828	1.579466	2.024110
ABCB	1.389594	1.959822	3.370170	1.662225	2.095453
ACAA	1.384514	1.953360	3.296594	1.574680	2.052287
ACAB	1.381627	1.959894	3.392883	1.622295	2.089175
ACBA	1.353445	1.959401	3.357063	1.619509	2.072354
ACBB	1.384140	1.955770	3.483635	1.636062	2.114902
ACCA	1.353445	1.934598	3.198563	1.608083	2.023672
ACCB	1.394372	1.949431	3.291218	1.627821	2.065710
BAAA	1.380267	1.922620	3.257862	1.592298	2.038262
BAAB	1.397197	1.960619	3.344885	1.669033	2.092934
BABA	1.369902	1.952951	3.382209	1.616292	2.080339
BABB	1.372443	1.960619	3.305681	1.664850	2.075898
BACA	1.360630	1.947118	3.314461	1.612455	2.058666
BACB	1.400480	1.958946	3.382207	1.651123	2.098189
BBAA	1.367969	1.920201	3.275867	1.592222	2.039065
BBAB	1.413350	1.959488	3.497765	1.668887	2.134873
BBBA	1.373820	1.943974	3.273976	1.626485	2.054564
BBBB	1.404543	1.959565	3.417102	1.671361	2.113143
BBCA	1.367475	1.912314	3.055494	1.616045	1.987832
BBCB	1.394547	1.959128	3.305560	1.632183	2.072855
BCAA	1.353443	1.925213	3.220211	1.580696	2.019891
BCAB	1.384581	1.953787	3.322590	1.596467	2.064357
BCBA	1.391306	1.939457	3.380465	1.599764	2.077748
BCBB	1.384018	1.954191	3.524883	1.627693	2.122696
BCCA	1.353448	1.953747	3.315125	1.620508	2.060707
BCCB	1.402303	1.948640	3.320430	1.622109	2.073371
CAAA	1.368899	1.945692	3.328179	1.569310	2.053020
CAAB	1.404296	1.960580	3.475705	1.658979	2.124890
CABA	1.353446	1.907229	3.352769	1.581197	2.048660
CABB	1.385539	1.960610	3.284869	1.661997	2.073254
CACA	1.360668	1.901510	3.229195	1.550107	2.010370
CACB	1.397265	1.960619	3.301210	1.640305	2.074850
CBAA	1.348774	1.927889	3.163013	1.567775	2.001863
CBAB	1.384148	1.958119	3.343545	1.631796	2.079402
CBBA	1.380508	1.945242	3.165825	1.575780	2.016839
CBBB	1.397036	1.958779	3.443887	1.634572	2.108568
CBCA	1.298788	1.909617	3.210899	1.588411	2.001929
CBCB	1.395646	1.959167	3.350774	1.628446	2.083508
CCAA	1.367972	1.922399	3.371209	1.577098	2.059670
CCAB	1.385945	1.957921	3.417102	1.592045	2.088253
CCBA	1.377985	1.935013	3.367743	1.599975	2.070179
CCBB	1.387981	1.945429	3.481339	1.648969	2.115929
CCCA	1.353443	1.928849	3.309996	1.570274	2.040641
CCCB	1.391612	1.939851	3.291218	1.609236	2.057979

Table 8.1: The averages of the first set of experiments.

Table 8.1 for each problem and assignment of parameters shows the average of the the best found values in the 3 runs of the GA with that assignment of parameters on that problem; the last column shows for each assignment the average of all the runs on all the problems of the GAs with that assignment of parameters.

	rinput4	rinput6	rinput7	rinput8	average
A	1.373317	1.947150	3.328327	1.616144	2.066234
B	1.381762	1.946254	3.327599	1.625582	2.070299
C	1.374442	1.940251	3.327138	1.604793	2.061656

Table 8.2: The selection of the crossover node types.

Table 8.2 for each type of selection of crossover node and for each problem shows the average af all the runs on that problem of all the assignments with that type of selection of crossover node; the last column shows for each type of selection of crossover node the average of all the runs on all the problems of all the assignments with that type of selection of crossover node.

	rinput4	rinput6	rinput7	rinput8	average
A	1.377379	1.946013	3.317713	1.620170	2.065319
B	1.375165	1.942256	3.313001	1.618944	2.062342
C	1.376977	1.945386	3.352348	1.607405	2.070529

Table 8.3: The selection types.

Table 8.3 for each type of selection and for each problem shows the average af all the runs on that problem of all the assignments with that type of selection; the last column shows for each type of selection the average of all the runs on all the problems of all the assignments with that type of selection.

	rinput4	rinput6	rinput7	rinput8	average
A	1.379515	1.944267	3.336089	1.608298	2.067042
B	1.375347	1.948790	3.371984	1.623528	2.079912
C	1.374658	1.940599	3.274989	1.614693	2.051235

Table 8.4: The crossover types.

Table 8.4 for each type of crossover and for each problem shows the average af all the runs on that problem of all the assignments with that type of crossover; the last column shows for each type of crossover the average of all the runs on all the problems of all the assignments with that crossover.

Table 8.5 for each type of mutation and for each problem shows the average af all the runs on that problem of all the assignments with that type of mutation; the last column shows for each type of mutation the average of all the runs on all the problems of all the assignments with that type of mutation.

The assignment corresponding to the row with the highest value in the last column of table 8.1 should be a good assignment for these problems. We also expected that the assignment with the type of selection of crossover node corresponding to the line of table 8.2 with the highest value in the last column, with the type of selection corresponding to the line of table 8.3 with the highest value in the last column, with the type of crossover corresponding

	rinput4	rinput6	rinput7	rinput8	average
A	1.359408	1.932037	3.276489	1.590525	2.039614
B	1.393606	1.957066	3.378886	1.640488	2.092512

Table 8.5: The mutation types.

to the line of table 8.4 with the highest value in the last column and with the type of mutation corresponding to the line of table 8.5 with the highest value in the last column was a good GA.

According to table 8.1 the first 10 best assignments are: BBAB, CAAB, BCBB, CCBB, ACBB, BBBB, ABAB, AAAB, CBBB, AABB, BACB, ABCB in this order. All of them use mutation B. Selection C and crossover B are in many of the best assignments. CAAB and BCBB have almost the same value, so BCBB can be considered the 2^{nd} best. Also 3 of the first 4 assignments end with CBB. The different types of selection of the crossover node obtain almost the same results and it seems that the type of the selection of the crossover node is not influent.

These assignments are also given by tables 8.2, 8.3, 8.4, 8.5. In fact in table 8.2, the highest average is obtained by selection of crossover node B, followed by A and C, but the numbers are very close, so that there seems to be not much difference between them; in table 8.3 the highest average is in row of selection C, followed by row of selection A and B; in table 8.4 the best average is obtained by crossover B followed by A and C; and in table 8.5 mutation B obtains better results than mutation A.

These experiments confirmed our idea that crossover B is good, but they also revealed the unexpected goodness of selection C and mutation B and the not bad performance of selection of the crossover node C. This result is apparently surprising because selection C and mutation B and the random selection of the crossover node have proved often inadequate to find a good solution in past experiments. Selection C is not much severe and often unable to select the best individuals for the next generation – as we can see from the figure 8.1, where assignment CCAA is used to solve rinput0 and where we can see the curves of the average EU and of the best EU of the population in the various generations – and mutation B tends to change radically an individual and risks to move the population to another region the searching space, loosing the good genes found so far. Being the leafs more numerous than inner nodes, selection of crossover node C tends to choose for crossover leaf nodes thus behaving like a mutation on the same node for two individuals.

The goodness of selection C and mutation B and the not bad performance of selection of the crossover node C can be understood.

We note that selection C, being not much severe, prevents the population from having a lot of twins – as it happens with the other selections – and

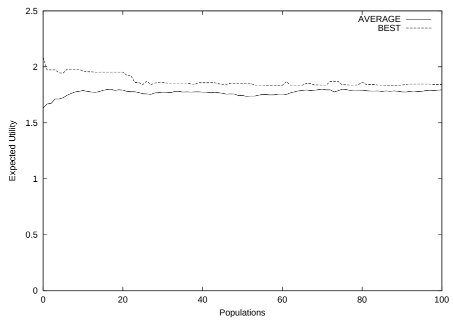


Figure 8.1: A bad behaviour of selection C.

keeps the population varied; mutation B varies the population often changing some individuals completely. On the other hand selection A is more severe and in each iteration can select the best individual many times reducing the number of different individuals in the population; mutation A changes an individual only slightly, in order to keep some of its genes; a GA with this two components tends to converge soon. These observations are confirmed by the graphs of figures 8.2 and 8.3. In the figure 8.2 we can see the graphs of the average EU and of the best EU of the population on all the generations of a run of BABA on the problem rinput0. We can see that the curve of the average gets very close to the curve of the best and from a certain generation onward there is too little variability and no improvement at all. Instead if we look at figure 8.3 where are displayed the graphs of a run of BCBB on the same problem, we can see that there is still some variablness even in the last generations and the final result is better.

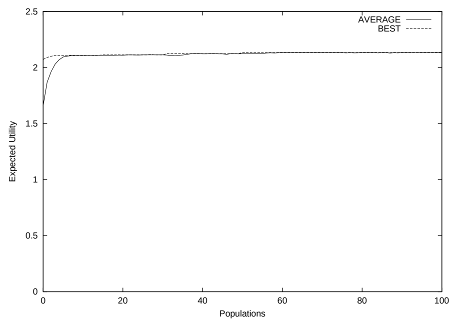


Figure 8.2: A run with the BABA assignment.

A GA with selection C and mutation B have a various population, while a GA with selection A and mutation A has population of very similar individuals. Selection of the crossover node C tends to behave like a mutation

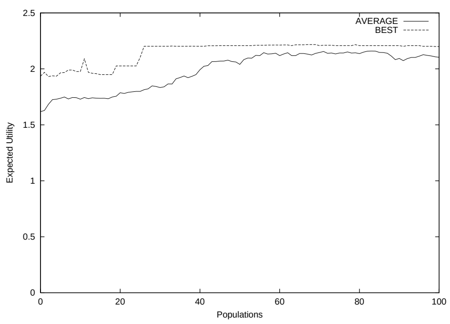


Figure 8.3: A run with the BCBB assignment.

rather than like a crossover. So in these tests the best results were obtained by the assignments that keep a good variability in the population.

Therefore mutation A with $P_m = 0.1$ could do better than mutation A with $P_m = 0.01$ and maybe it could compete with mutation B with $P_m = 0.01$. Also the best type of components, especially for selection and mutation, could be different with a bigger P_m. These experiments gave us the indication for new experiments and possible improvements.

8.4.2 The second set of experiments

We decided to compare the 5 best assignments according to table 8.1 (BBAB, CAAB, BCBB, CCBB, ACBB) with assignments BAAA, BABA, BBAA, BBBA with parameter $P_m = 0.1$ instead of $P_m = 0.01$.

The new assignments consist of all the assignments of parameters whose name is in $\{B\} \times \{A, B\} \times \{A, B\} \times \{A\}$ with $P_m = 0.1$. We chose selection of the crossover node B because it revealed to be the best in the previous experiments, mutation A because it is more sophisticated and should be more effective with the bigger P_m, selection A or B because they are more selective, but should work well with a strong mutation and crossover A or B because in the previous tests they behaved.

We run the GA with these assignments on six problems for three times. The detail of the two new problems are in Appendix B. We made the usual table of the averages and a table of the best that collects for each problem and assignment the best result in the 3 runs of that assignment on that problem.

From table 8.6 we can see that the best 3 results on the average are obtained by assignments with mutation A with $P_m = 0.1$. But the difference is small and we cannot say that the new assignments are much better than the old ones. What we can say is that the new ones are at least as good as the old ones and that mutation A, in order to be effective, needs a P_m bigger

	rinput0	rinput4	rinput5	rinput6	rinput7	rinput8	average
BBAB	2.195863	1.413350	1.870910	1.959488	3.497765	1.668887	2.101044
CAAB	2.228531	1.404296	1.876676	1.960580	3.475705	1.658979	2.100794
BCBB	2.200090	1.384018	1.851230	1.954191	3.524883	1.627693	2.090351
CCBB	2.167240	1.387981	1.814002	1.945429	3.481339	1.648969	2.074160
ACBB	2.167137	1.384140	1.839506	1.955770	3.483635	1.636062	2.077708
BAAA	2.183349	1.388026	1.874302	1.960619	3.565446	1.635412	2.101192
BABA	2.205129	1.375242	1.887914	1.923001	3.565853	1.653001	2.101690
BBAA	2.138591	1.375197	1.862915	1.935036	3.345424	1.638274	2.049240
BBBA	2.205103	1.375225	1.885490	1.930600	3.565535	1.650546	2.102083

Table 8.6: The averages of the second set of experiments.

	rinput0	rinput4	rinput5	rinput6	rinput7	rinput8
BBAB	2.227380	1.414572	1.888927	1.959997	3.566514	1.679662
CAAB	2.231921	1.414535	1.885318	1.960619	3.566514	1.679662
BCBB	2.212960	1.414087	1.863154	1.959798	3.559234	1.640943
CCBB	2.202562	1.401749	1.828145	1.957900	3.566514	1.679662
ACBB	2.202250	1.387112	1.858277	1.960437	3.507708	1.640943
BAAA	2.228652	1.413595	1.894548	1.960619	3.578114	1.674737
BABA	2.233956	1.375242	1.889488	1.960619	3.578114	1.675128
BBAA	2.205627	1.375242	1.890651	1.959239	3.429091	1.669891
BBBA	2.215443	1.375242	1.886204	1.960352	3.570834	1.671940

Table 8.7: The best results of the second set of experiments.

than 0.01.

In Figures 8.4, 8.5, 8.6, 8.7, we can see the curves of the average EU and the best EU in some runs of some assignments with mutation A and $P_m = 0.1$ and mutation B on the same problems.

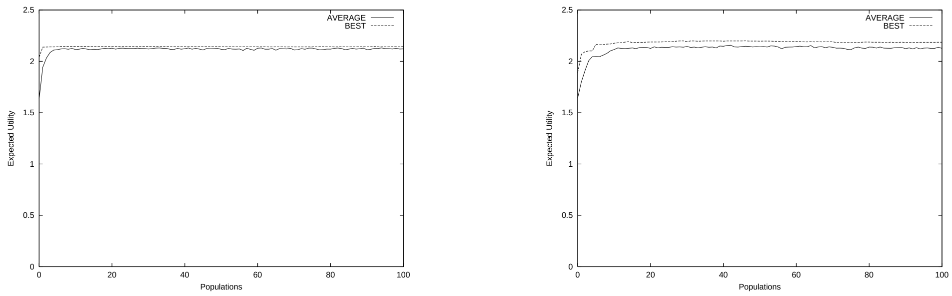


Figure 8.4: A run of CAAB and BABA on rinput0.

8.5 The experiments on mixed genetic components

Other experiments have been done on assignments of parameters that cause the GA to employ more than one crossover and mutation. The pourpose was that of investigating wether it is better to use one single genetic component or a mixture of them. In these experiments crossover D and mutation C and D have also been tested.

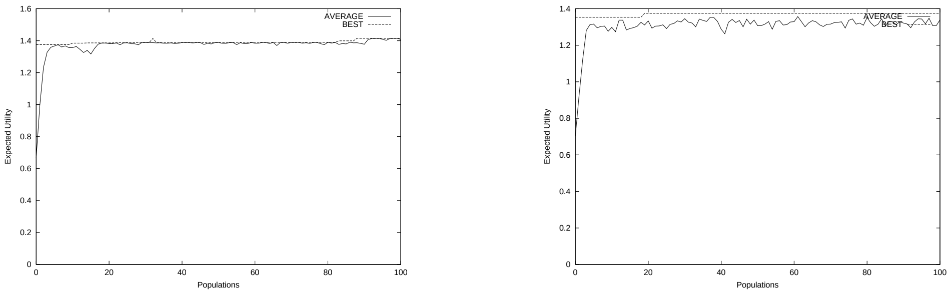


Figure 8.5: A run of CAAB and BABA on rinput4.

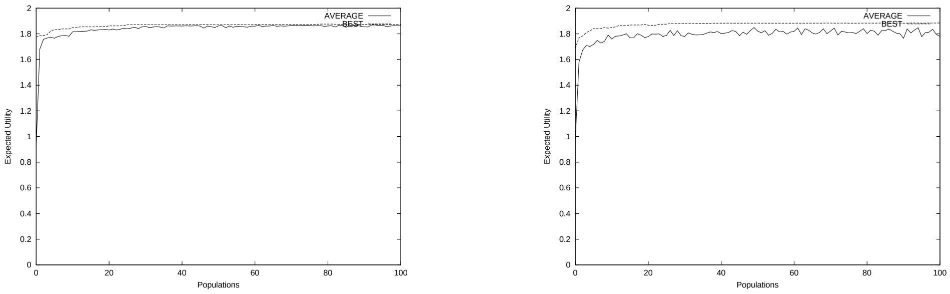


Figure 8.6: A run of BBAB and BBBA on rinput5.

The experiments of this section regard 10 different assignments of parameters. The problems we used are the same of Section 8.4.2, so the result of this section can be compared with those of that section. For all the tested assignments it is $scn_{type} = B$, $selection_{type} = A$, $shake_r = 1$ – so the shake feature is disabled – and $P_{m_{A_i}} = P_{m_{B_i}} = P_{m_{C_i}} = P_{m_{D_i}} = 0.1$. They differ only on the frequences of the crossovers and of the mutations and on the selection pressure. The tested assignments are:

- *mix1* assignment with $\vec{f_c} = (f_{c_A}, f_{c_B}, f_{c_C}, f_{c_D}) = (\frac{1}{3}, \frac{1}{3}, \frac{1}{3}, 0)$, $\vec{f_m} =$

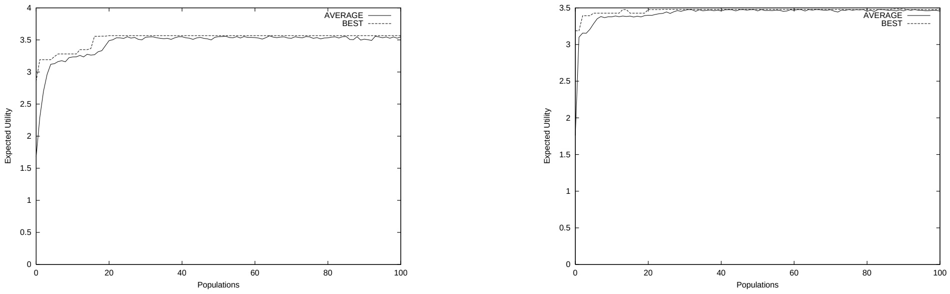
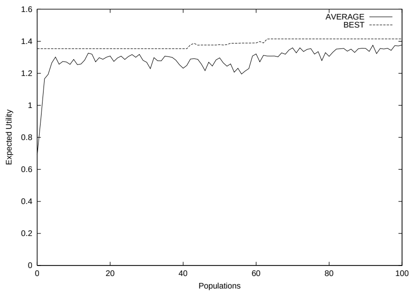


Figure 8.7: A run of CAAB and BBBA on rinput7.

$(f_{m_A}, f_{m_B}, f_{m_C}, f_{m_D}) = (\frac{1}{2}, \frac{1}{2}, 0, 0)$ and $a = 1$

- *mix3* with $\vec{f_c} = (\frac{1}{3}, \frac{1}{3}, \frac{1}{3}, 0)$, $\vec{f_m} = (\frac{1}{2}, \frac{1}{2}, 0, 0)$ and $a = 3$
- *mixAB1* with $\vec{f_c} = (\frac{1}{2}, \frac{1}{2}, 0, 0)$, $\vec{f_m} = (\frac{1}{2}, \frac{1}{2}, 0, 0)$ and $a = 1$
- *mixAB3* with $\vec{f_c} = (\frac{1}{2}, \frac{1}{2}, 0, 0)$, $\vec{f_m} = (\frac{1}{2}, \frac{1}{2}, 0, 0)$ and $a = 3$
- *mixABCD1* with $\vec{f_c} = (\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4})$, $\vec{f_m} = (\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4})$ and $a = 1$
- *mixABCD3* with $\vec{f_c} = (\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4})$, $\vec{f_m} = (\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4})$ and $a = 3$
- *mixABCD4* with $\vec{f_c} = (\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4})$, $\vec{f_m} = (\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4})$ and $a = 4$
- *mixABCD1n* with $\vec{f_c} = (\frac{3}{10}, \frac{3}{10}, \frac{3}{10}, \frac{1}{10})$, $\vec{f_m} = (\frac{3}{10}, \frac{3}{10}, \frac{1}{10}, \frac{3}{10})$ and $a = 1$
- *mixABCD3n* with $\vec{f_c} = (\frac{3}{10}, \frac{3}{10}, \frac{3}{10}, \frac{1}{10})$, $\vec{f_m} = (\frac{3}{10}, \frac{3}{10}, \frac{1}{10}, \frac{3}{10})$ and $a = 3$
- *mixABCD4n* with $\vec{f_c} = (\frac{3}{10}, \frac{3}{10}, \frac{3}{10}, \frac{1}{10})$, $\vec{f_m} = (\frac{3}{10}, \frac{3}{10}, \frac{1}{10}, \frac{3}{10})$ and $a = 4$

These assignments, here presented together, have in fact been tested separately. The first to be tested were those with parameter $a = 1$. Because in the graphs of their runs the curve of the average was jugged and far from the curve of the best, both in the bad and in the good performances – as can be seen in Figure 8.8 –, we decided to increase the selection pressure and test them with $a = 3$. Then, because of the better results with parameter $a = 3$, we tested the two best assignmets with $a = 4$.



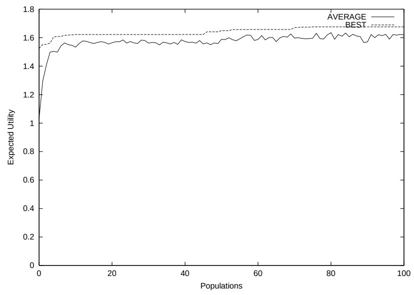


Figure 8.8: The curves of mixAB1 on a bad and a good performance.

The results of the tests are shown in tables 8.8 and 8.9.

As you can see from Table 8.8, apart from mix1, all the mixed assignments obtained better average results than the best assignmets for GAs with single components found so far. The best average is obtained by mixABCD3n closely followed by mixABCD3. They are the only assignments that reached the best average in 3 problems. However the averages differ only on the 3rd decimal position and the correspondent assignments cannot be considered much different.

	rinput0	rinput4	rinput5	rinput6	rinput7	rinput8	average
mix1	2.234217	1.375242	1.890409	1.960619	3.445047	1.676802	2.097056
mix3	2.231898	1.375242	1.895669	1.960619	3.578114	1.678445	2.119998
mixAB1	2.228764	1.401481	1.894872	1.960619	3.578114	1.677146	2.123499
mixAB3	2.230276	1.375242	1.895516	1.960619	3.577659	1.678445	2.119626
mixABCD1	2.233114	1.405355	1.895925	1.960619	3.578114	1.679662	2.125465
mixABCD3	2.229517	1.414601	1.895557	1.960619	3.578114	1.678445	2.126142
mixABCD4	2.231399	1.414601	1.894627	1.960619	3.574247	1.678445	2.125656
mixABCD1n	2.233438	1.414601	1.894502	1.960580	3.578114	1.666523	2.124626
mixABCD3n	2.233982	1.414601	1.894490	1.960619	3.574247	1.679662	2.126267
mixABCD4n	2.225278	1.414601	1.895609	1.960619	3.574247	1.677962	2.124719

Table 8.8: The averages of the mixed assignments.

Even Table 8.9 confirms that the mixed assignments are better than the ones with single component. The mixed assignments have reached or beated the records in all the problems. In particular mixABCD3n and mixABCD3.

	rinput0	rinput4	rinput5	rinput6	rinput7	rinput8
mix1	2.237790	1.375242	1.895891	1.960619	3.578114	1.679662
mix3	2.232425	1.375242	1.896098	1.960619	3.578114	1.679662
mixAB1	2.229349	1.414601	1.895458	1.960619	3.578114	1.679662
mixAB3	2.230954	1.375242	1.895516	1.960619	3.578114	1.679662
mixABCD1	2.235473	1.414601	1.896221	1.960619	3.578114	1.679662
mixABCD3	2.232677	1.414601	1.896270	1.960619	3.578114	1.679662
mixABCD4	2.232677	1.414601	1.895553	1.960619	3.578114	1.679662
mixABCD1n	2.235009	1.414601	1.895670	1.960619	3.578114	1.679662
mixABCD3n	2.235643	1.414601	1.895443	1.960619	3.578114	1.679662
mixABCD4n	2.230696	1.414601	1.896222	1.960619	3.578114	1.679662

Table 8.9: Best results of the mixed assignments.

8.6 The experiments on the shake feature

The last set of experiments regards 4 assignmets that cause the GA to use the shake feature. The GA has been run on the usual 6 test problems. For the tested assignments it is $scn_{type} = B$, $selection_{type} = A$, $shake_w = 10$, $shake_r < 1$, $\vec{f_c} = (\frac{3}{10}, \frac{3}{10}, \frac{3}{10}, \frac{1}{10})$ and $\vec{f_m} = (\frac{3}{10}, \frac{3}{10}, \frac{1}{10}, \frac{3}{10})$. The tested assignmets are:

- *shake1* with $\vec{P}_{m_i} = (P_{m_{A_i}}, P_{m_{B_i}}, P_{m_{C_i}}, P_{m_{D_i}}) = (\frac{5}{100}, \frac{5}{100}, \frac{5}{100}, \frac{5}{100})$, $a = 6$, $shake_r = \frac{995}{1000}$, $\vec{P}_{m_s} = (P_{m_{A_s}}, P_{m_{B_s}}, P_{m_{C_s}}, P_{m_{D_s}}) = (\frac{5}{10}, \frac{5}{10}, \frac{5}{10}, \frac{5}{10})$ and $shake_d = 1$

- *shake2* with $\vec{P}_{m_i} = (\frac{5}{100}, \frac{5}{100}, \frac{5}{100}, \frac{5}{100})$, $a = 6$, $shake_r = \frac{995}{1000}$, $\vec{P}_{m_s} = (\frac{5}{10}, \frac{5}{10}, \frac{5}{10}, \frac{5}{10})$ and $shake_d = 2$

- *bestshake1* with $\vec{P}_{m_i} = (\frac{1}{10}, \frac{1}{10}, \frac{1}{10}, \frac{1}{10})$, $a = 3$, $shake_r = \frac{99}{100}$, $\vec{P}_{m_s} = (\frac{7}{10}, \frac{7}{10}, \frac{7}{10}, \frac{7}{10})$ and $shake_d = 1$
- *bestshake2* with $\vec{P}_{m_i} = (\frac{1}{10}, \frac{1}{10}, \frac{1}{10}, \frac{1}{10})$, $a = 3$, $shake_r = \frac{99}{100}$, $\vec{P}_{m_s} = (\frac{7}{10}, \frac{7}{10}, \frac{7}{10}, \frac{7}{10})$ and $shake_d = 2$

Note that shake1 and shake2 differ only on parameter $shake_d = 1$ and 2 respectively; the same it happens for bestshake1 and bestshake2. Also note that assignments bestshake1 and bestashake2 are identical to mixABCD3n of Section 8.5 a part from the parameters that concern the shake feature and that shake1 and shake2 have a weaker mutation (determined by parameters $\vec{P}_{m_i}$) and a stronger selection pressure (determined by parameter a) as in these conditions – as stated in Section 6.9 – the shake feature should be more useful.

The results of these tests are shown in tables 8.10 and 8.11.

	rinput0	rinput4	rinput5	rinput6	rinput7	rinput8	average
shake1	2.228372	1.414601	1.895014	1.960619	3.572815	1.679662	2.125180
shake2	2.221539	1.401481	1.894434	1.960619	3.570381	1.679662	2.121353
bestshake1	2.226601	1.414601	1.895053	1.960604	3.578114	1.679662	2.125773
bestshake2	2.225410	1.414601	1.895488	1.960599	3.574247	1.679662	2.125001

Table 8.10: The averages of the experiments on the shake feature.

	rinput0	rinput4	rinput5	rinput6	rinput7	rinput8
shake1	2.230933	1.414601	1.896214	1.960619	3.578114	1.679662
shake2	2.231863	1.414601	1.895584	1.960619	3.578114	1.679662
bestshake1	2.229756	1.414601	1.896254	1.960619	3.578114	1.679662
bestshake2	2.226748	1.414601	1.895923	1.960619	3.578114	1.679662

Table 8.11: The best results of the experiments on the shake feature.

From these tables we can see that the best average result – obtained by bestshake1 – is the 3rd best ever, but the numbers are very similar and the 4 assignments tested here have almost the same performance. Also, according to these numbers, they have almost the same performance of the best assignments of Section 8.5.

8.7 Conclusions

The experiments of this chapter, even if not complete, lead us to assignment of parameters that are better than the ones we were using at the beginning, at least in solving the test problems[2].

[2]Note that the problem rinput6 has been exactly solved, as the EU of the output best solution is equal to the upper bound and that even if problems rinput4, rinput7 and rinput8 have not been solved with certainty, all the best algorithms give output solutions with the same EU, making us think that these problems too have been solved.

The best two assignmets found so far for GAs without shake feature are mixABCD3n and mixABCD3 of Section 8.5; while the best two assignmets found so far for GA with shake feature are bestshake1 and shake1 of Section 8.6. However, because of the small difference between the numbers other assignments could be good as well.

As we can see from tables 8.1, 8.8 and 8.10, the GAs that employ more than one genetic operator perform better. From these few experiments we can say that the shake feature does not improve much the performances of the GA, but we think it deserves to be studied more.

Eventually we have to remember that, because of the small number of test problems, these experiments could have selected good assignments for these problems and not so good in general. Yet, when facing a new problem, it is more reasonable to use one of the best assignments found in this chapter, rather than to use random parameters.

Chapter 9

The Evaluation of the GA

9.1 Overview

In this chapter some results useful to evaluate the quality of the GA developped in Chapter 6 are presented.

9.2 Introduction

A way to judge the goodness of an approximation algorithm is to compare the best value it finds with the real optimal value of the BPP or with the best value found by other approximation algorithms.

The first method is not always applicable in the case of our GA, as the best solution of BPPs is often unknown, for no exact algorithms have been implemented yet. The second method requires the existence of more than one algorithm to solve the same class of problems. So evaluating the quality of our GA has been quite difficult.

In the case of testing a GA by applying the second method, we can compare our best value with the best value found by the other algorithms in the same amount of time or, if the others are stochastic population based algorithms, after the same number of individuals are generated. The first technique is dipendent on the machine and on the implementation of the algorithms and it may happen that one machine is particularly good with one of the algorithms to be tested, because of the particular operations that it requires, for which the machine can be optimized, and another machine can be particularly good with another algorithm. Also the results of the first technique is strongly dependent on the level of optimization of the code. Because we want to judge the algorithm itself and not its implementation, the second technique is better for our purposes.

A very simple stochastic iterative algorithm is based on a random search. It consists in generating iteratively and randomly feasible individuals until a terminal condition occurs. We have implemented the algorithm *random-solver* which is slightly more cptious. Each random individual is produced by assigning each node of the tree a random and possibly unfeasible position for its container and then the individual is repaired by the ordered-repair function. This algorithm is called *random-solver* and its pseudocode can be found in Figure 9.1 where the function Initial-feasible-population of Section 6.5 is used. The implementation of this algorithm is described in Section A.2. This algorithm is slightly more than a simple random generation of solutions, as the solutions produced in such a way are a bit optimized after their generation because of the filling carried out at the end of the repair function.

{ $pop_{size} \leftarrow 1$;
  $P_{NULL} \leftarrow 0.3$;

  $\vec{w} \leftarrow$ Initial-feasible-population;
  $w_{best} \leftarrow w_1$;
  for($i \leftarrow 1, .., numb_{iter}$)
  { $\vec{w} \leftarrow$ Initial-feasible-population;
    if(EU(w_1)>EU(w_{best})) $w_{best} \leftarrow w_1$;
  }
  return w_1;
}

Figure 9.1: The random-solver algorithm.

In this chapter will be presented the results of one run of our GA – controlled by two of the best assignments of parameters found in Chapter 8, bestshake1 and best1 – on some problems with a known solution and on others with unknown solution. The results of one run of the random-solver on the same problems will also be given.

9.3 Problems with known optimal solution

9.3.1 Problems with linear trees

We knew the solution of some problems with linear trees and we tried to solve them by our GA. Of course, being a linear tree BPP a very particular problem, there are better algorithms to solve it. As a matter of fact a linear tree BPP is just a 2D cutting stock problem and in order to solve it we can use an algorithm specifically designed for this tighter class of problems. So we expected our algorithm not to perform particularly well.

Problems lin16 and lin73 are derived from two 2D strip packing problems with known optimal solution taken from [12]. They are described in sections B.8 and B.9 respectively. Because in these problems the utilities of the containers are their areas, these BPPs can be viewed as problems of placing a set of known containers on a known rectangular hold minimizing the hold waste space. Because we know that in both cases the hold can be completely filled, the optimal EU of both problems is the hold area, also corresponding to the UB.

	lin16	lin73
optimal EU	**400**	**5400**
best1	394	4686
bestshake1	400	4666
random	364	4542

Table 9.1: The best EUs and the optimal EUs.

We have run our GA with the assignments of parameters bestshake1 and best1 with $pop_{size} = 100$ and $numb_{iter} = 900$ on lin16 and with $pop_{size} = 50$ and $numb_{iter} = 200$ on lin73; we have run the random-solver with $numb_{iter} = 90000$ and $numb_{iter} = 10000$ on lin16 and lin73 respectively. The best EU of problem lin16 and lin73 found by our 3 algorithms and their optimal EU can be found in Table 9.1. The graphical representation of the best solutions found by the 3 algorithms are shown in figure 9.2, 9.3, 9.4, 9.5, 9.6, 9.7.

9.3.2 A modified problem

Problem art7 is described in section B.7. It is derived from problem rinput7 of section B.5 in the following way: we have added to rinput7 a new container so big that no other container can be placed on the hold along with it and with an utility that is $\geq$ than the sum of the utilities of the containers of every path from the root to a terminal node and we have associated each

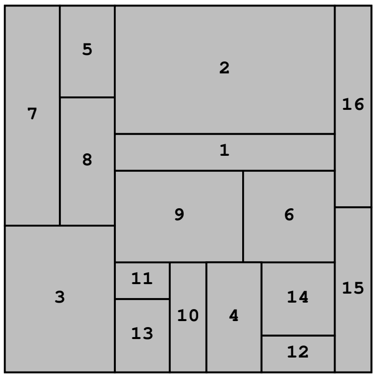


Figure 9.2: The hold of the best solution of problem lin16 found by bestshake1.

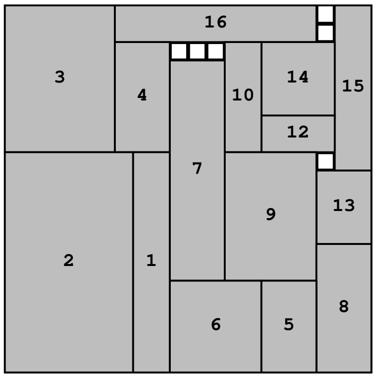


Figure 9.3: The hold of the best solution of problem lin16 found by best1.

terminal node[1] of the tree with this container. It is then obvious that the best solution is the one that assigns this container to all the terminal nodes and $NULL$ to all the other nodes and the optimal EU is the utility of this container.

We have run our GA on this problem with the parameters of bestshake1 and best1 and with $pop_{size} = 100$ and $numb_{iter} = 300$. We have also run random-solver on this problem with $numb_{iter} = 30000$. The best EU of problem art7 found by our 3 algorithms and its optimal EU can be found in Table 9.2. The curves of the average and of the best EU of these runs of bestshake1 and best1 on art7 are shown in figures 9.9 and 9.8 respectively.

[1]Note that the only terminal nodes of this BPP are its leaves.

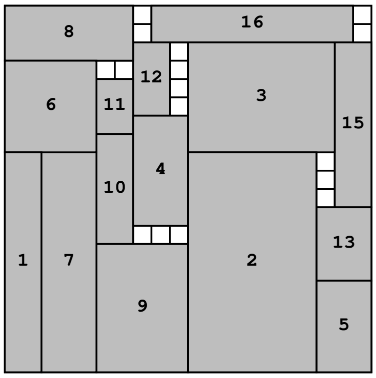


Figure 9.4: The hold of the best solution of problem lin16 found by random.

optimal EU	**20**
best1	20
bestshake1	20
random	11.561565

Table 9.2: The best EUs and the optimal EU for problem art7.

9.3.3 A problem with reached UB

Problem rinput6 of Section B.4 has been exactly solved by our GA in Chapter 8. As a matter of fact the best solution found by our GA has the same value of the UB of rinput6. So the optimal EU is exactly the UB. We have run again our GA with the parameters of bestshake1 and best1 and with $pop_{size} = 100$ and $numb_{iter} = 300$. We have also run random-solver on this problem with $numb_{iter} = 30000$. The best EU of problem rinput6 found by our 3 algorithms and its optimal EU can be found in Table 9.3.

optimal EU	**1.960619**
best1	1.960619
bestshake1	1.960619
random	1.905080

Table 9.3: The best EUs and the optimal EU for problem rinput6.

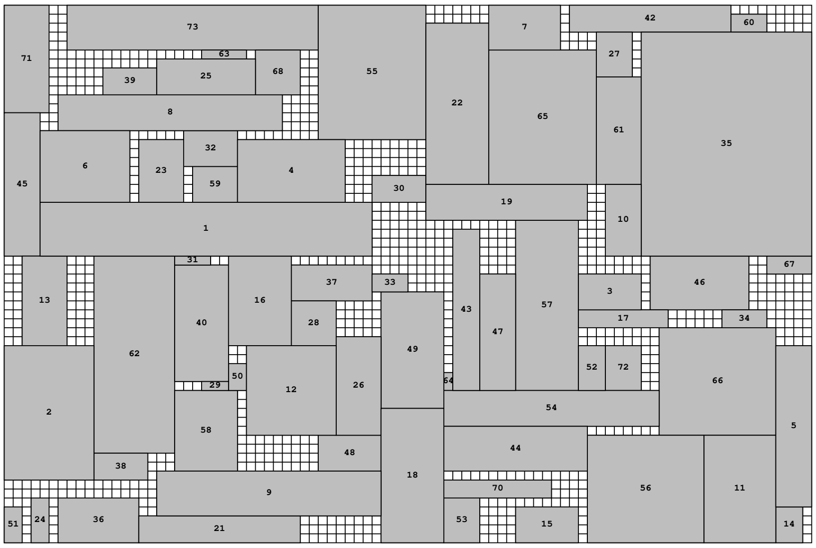


Figure 9.5: The hold of the best solution of problem lin73 found by bestshake1.

9.4 Problems with unknown optimal solution

Finally we have compared the best solutions of problems rinput0, rinput4, rinput5, rinput7 and rinput8 (see Appendix A) found by the random-solver using $numb_{iter} = 30000$ with those found by our GA using the parameters of bestshake1 and best1 and with $pop_{size} = 100$ and $numb_{iter} = 300$. The results of these runs are shown in Table 9.4.

	rinput0	rinput4	rinput5	rinput 7	rinput8
best1	2.235643	1.414601	1.894625	3.578114	1.679662
bestshake1	2.229265	1.414601	1.896203	3.578114	1.679662
random	2.147712	1.410169	1.844847	3.305993	1.640943

Table 9.4: The best EUs for the unknown solution problems.

9.5 Conclusions

As we can seen from the tables 9.1, 9.2, 9.3 and 9.4, our GA has always proved better than the random-solver generating the same number of individuals and sometimes it is able to find the optimal solution as it happens with bestshake1 on lin16 and best1 and bestshake1 on art7 and rinput6.

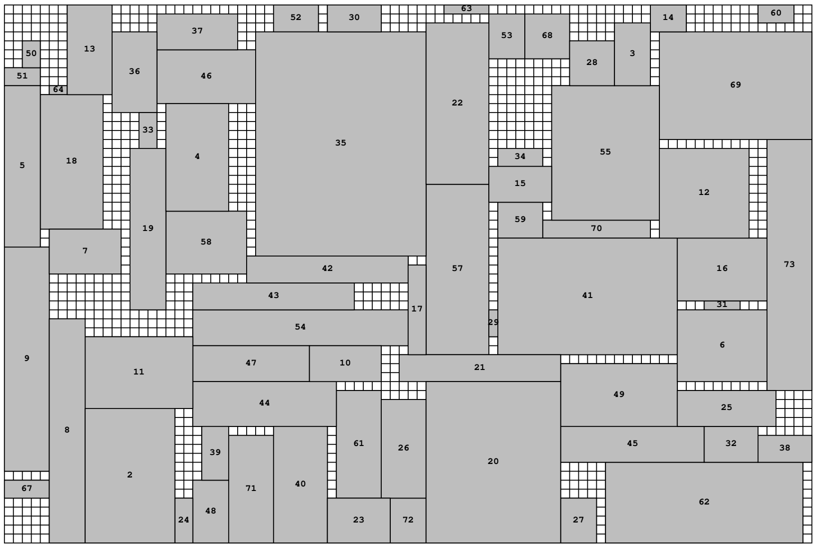


Figure 9.6: The hold of the best solution of problem lin73 found by best1.

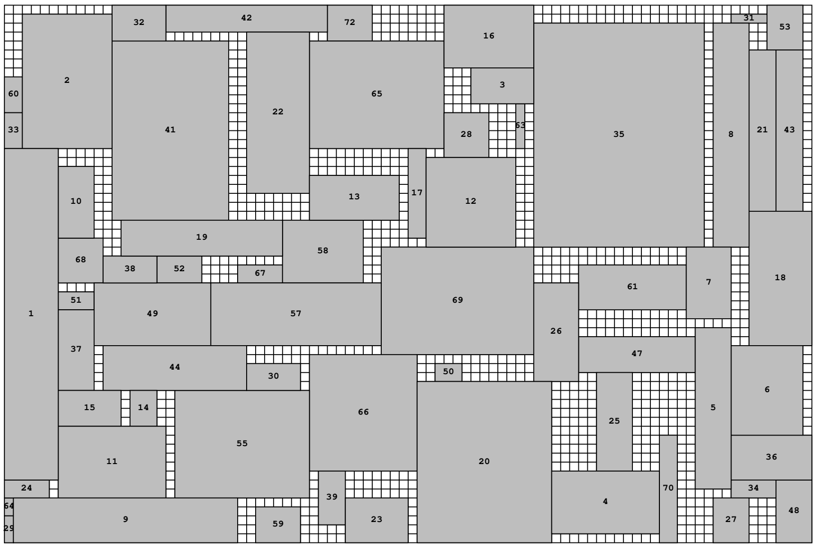


Figure 9.7: The hold of the best solution of problem lin73 found by random.

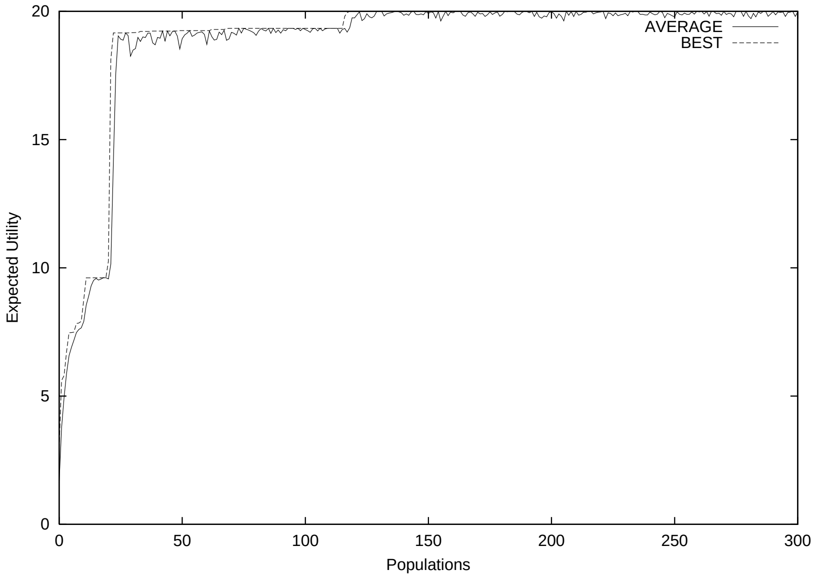


Figure 9.8: The curves of best1 on problem art7.

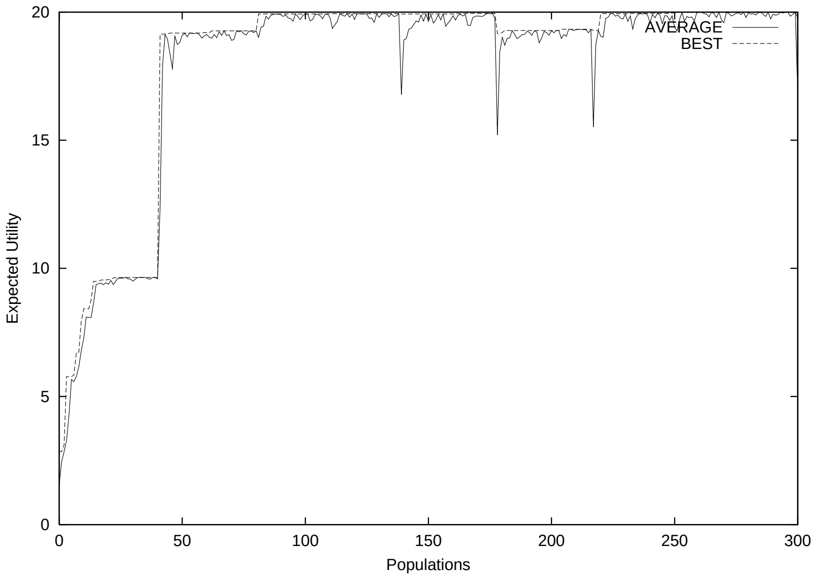


Figure 9.9: The curves of bestshake1 on problem art7.

Chapter 10

The Application

10.1 Overview

The GA described in Chapter 6 has been implemented in ANSI C. In this Chapter it is shown how to use this implementation.

10.2 Introduction

We have implemented the present GA in ANSI C and the resulting application is a command line program that can be run in any system with an ANSI C compiler. The program must be run with 6 arguments. The first argument is the name of an existing text file containing an assignement of all the parameters of the GA. The second is the name of an existing text file containing the description of a BPP. This two files will not be modified by the application. The other 4 arguments are names of files that will be created by the application.

When we run the program with these 6 arguments, the GA is run on the BPP described by the second argument with a behaviour determined by the assignemet of parameters of the first argument; at the end of the run, the solution of the BPP with the highest EU found by the GA will be recorded in the third file, the fourth and fifth files will contain the average and the best EU of the various generations respectively and the sixth file will keep the seed of the random number generator used for that run.

Therefore, given a BPP, in order to find one of its solutions with the highest EU, we must define all the parameters determining the behaviour of the GA and write them in a text file; hence we must write a text file describing the BPP and finally we must run the program with six filenames as arguments, being this two files the first two arguments. The best solution will be saved in the third argument, the forth and fifth will contain information about the average and best EU of the various generations and the sixth the seed of the random number generator.

10.3 The parameters file

The first argument is the name of a file that contains an assignment of the 27 parameters determining the behaviour of the GA and is called the *parameters file*. The parameters are presented in the sections 6.4 and following. The structure of the parameters file is given in Figure 10.1.

In this figure, $< int >$ is a sequence of digits representing a number $\in \mathbb{N}_0$; $< prob >$ is a sequence of digits representing a rational number $\in [0, 1]$; $< sep >$ is a sequence of one or more tab or space or CR characters; $< real >$ is a sequence of digits representing a rational number $\in \mathbb{R}_0^+$; $< scn >$ is A or B and $< select >$ is A,B or C. An example is in figure 10.2.

Each string preceding the symbol $=$ corresponds to a parameter. The value following each $=$ is assigned to the parameter corresponding to the string preceding $=$.

The application doesn't do any checking on the sintactical correctedness of the parameters file: the user must assure that the parameters file is sinctactically correct. Of course the numbers assigned to the parameters must respect the other conditions for the parameters that are:

- $f_{c_A} + f_{c_B} + f_{c_C} + f_{c_D} = 1$
- $f_{m_A} + f_{m_B} + f_{m_C} + f_{m_D} = 1$
- $pop_{size} \geq 2$
- $p_{m_{A_i}} < p_{m_{A_s}}$, $p_{m_{B_i}} < p_{m_{B_s}}$, $p_{m_{C_i}} < p_{m_{C_s}}$, $p_{m_{D_i}} < p_{m_{D_s}}$

10.4 The input file

The second argument is the name of a file that contains the description of the BPP whose the GA will try to find a solution with the highest EU and is called *input file*. The structure of the input file is given in Figure 10.3.

As in the previous figure, in this one $< int >$ is a sequence of digits representing a number $\in \mathbb{N}_0$; $< prob >$ is a sequence of digits representing a rational number $\in [0, 1]$; $< sep >$ is a sequence of one or more tab or space or CR caracters; $< real >$ is a sequence of digits representing a rational number $\in \mathbb{R}_0^+$; $< util >$ is a string of digits representing a number $\in \mathbb{R}^+$;

numb_iter=< *int* >< *sep* >
pop_size=< *int* >< *sep* >
Pc=< *prob* >< *sep* >
PmAi=< *prob* >< *sep* >
PmBi=< *prob* >< *sep* >
PmCi=< *prob* >< *sep* >
PmDi=< *prob* >< *sep* >
P_null=< *prob* >< *sep* >
shake_w=< *int* >< *sep* >
shake_d=< *int* >< *sep* >
shake_r=< *real* >< *sep* >
PmAs=< *prob* >< *sep* >
PmBs=< *prob* >< *sep* >
PmCs=< *prob* >< *sep* >
PmDs=< *prob* >< *sep* >
a=< *real* >< *sep* >
b=< *int* >< *sep* >
scn_type=< *scn* >< *sep* >
selection_type=< *select* >< *sep* >
fcA=< *prob* >< *sep* >
fcB=< *prob* >< *sep* >
fcC=< *prob* >< *sep* >
fcD=< *prob* >< *sep* >
fmA=< *prob* >< *sep* >
fmB=< *prob* >< *sep* >
fmC=< *prob* >< *sep* >
fmD=< *prob* >< *sep* >

Figure 10.1: The structure of the parameters file.

```
numb_iter=300
pop_size=100

Pc=0.5

PmAi=0.1
PmBi=0.1
PmCi=0.1
PmDi=0.1

P_null=0.3

shake_w=10
shake_d=1
shake_r=0.995

PmAs=0.7
PmBs=0.7
PmCs=0.7
PmDs=0.7


a=3
b=30

scn_type=B
selection_type=A


fcA=0.3
fcB=0.3
fcC=0.3
fcD=0.1

fmA=0.3
fmB=0.3
fmC=0.1
fmD=0.3
```

Figure 10.2: An example of parameters file.

$\texttt{e}=< real >< sep >$
$\texttt{dx}=< real >< sep >$
$\texttt{dy}=< real >< sep >$
$\texttt{m}=< int >< sep >$
$\{\texttt{u}< int >=< util >< sep >\}^m$
$\{\texttt{lx}< int >=< real >< sep >\ \texttt{ly}< int >=< real >< sep >\}^m$
$\texttt{n}=< int >< sep >$
$\{(< node >,< father >,< prob >,< var >)< sep >\}^n$

Figure 10.3: The structure of the input file.

n and m are the numbers following "n=" and "m="; $< node >$ is a string of digits representing a number $\in \{1, .., n\}$; $< father >$ is a string of digits representing a number $\in \{0, .., n\}$; $< var >$ is a string of digits representing a number $\in \{1, .., m\}$.

The BPP described by the input file is obtained in the following way: the first 3 assignments define the values for e, d_x, d_y respectively; the 4rth assignment define the number of containers; the next m assignments define the utilities of the containers from 1 to m; the next $2m$ assignments define the width and length of the containers from 1 to m; the next line define the number of nodes n of the tree; The following 4-tuples define the tree; the nodes of the tree are $N = \{1, .., n\}$ and to each of them corresponds one of the next n 4-tuple: the first element of each 4-tupla is that node; the second is the father of that node (if the first element is the root, the father must be 0 and the probability can be whaterver number and will be ignored); the third element is the probability of the edge between the first element and the second (if the first element is the root, this element can be whatever string $< prob >$ and will be ignored; for consistence with the rest, it can be set to 1); the last element is the container associated with that node.

The program doesn't control weather the input file is sintactically correct and defines a correct BPP. The user must take care of this task and give the program an input file both correct from the sintactical and from the semantical point of view.

A very simple example of input file is shown in Figure 10.4. It describes the BPP of figure 10.5.

```
e=1.0
dx=21.0
dy=15.0

m=5

u1=12.2
u2=3.4
u3=4.0
u4=7.3
u5=8.0

lx1=6.5 ly1=3.0
lx2=7.5 ly2=3.0
lx3=5.0 ly3=4.0
lx4=4.5 ly4=4.5
lx5=7.0 ly5=4.0

n=7

(1,0,1,1)
(2,1,0.7,4)
(3,1,0.3,2)
(4,2,0.1,3)
(5,2,0.3,2)
(6,2,0.6,2)
(7,3,1.0,5)
```

Figure 10.4: An example of input file.

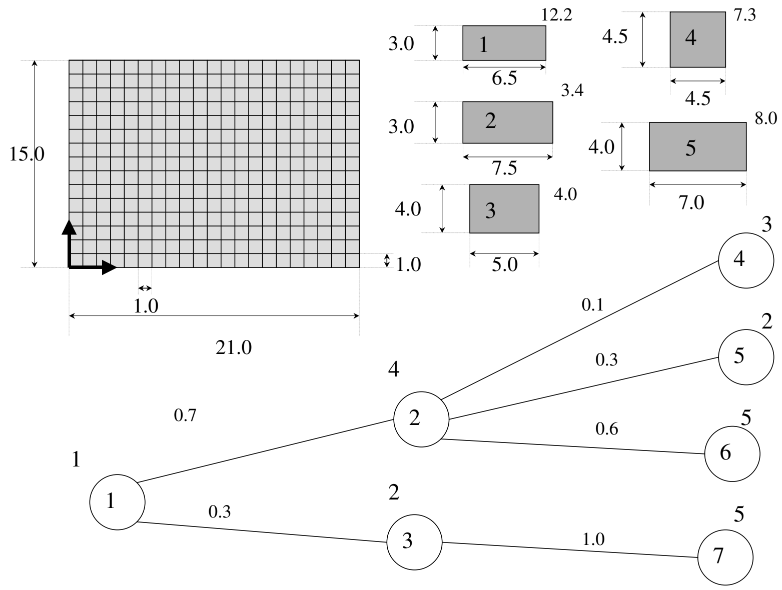


Figure 10.5: The BPP described by the input file of Figure 10.4.

10.5 The output file

The third argument is called *output file* and contains the solution ψ of the input BPP with the highest EU found by the GA. This text file gives the upper bound of the EU of the solutions of the input BPP, the EU of ψ and for each node α of the tree it gives $\psi(\alpha)$. An example output file is given in Figure 10.6, where it is shown the output file of a run of the application on the input file of Figure 10.3 with the parameters of Figure 10.1.

```
EU:23.152000
UB:23.152000

node1 x=0.000000;y=0.000000;z=0
node2 x=3.000000;y=4.000000;z=1
node3 x=0.000000;y=3.000000;z=0
node4 x=0.000000;y=9.000000;z=0
node5 x=16.000000;y=0.000000;z=1
node6 x=3.000000;y=9.000000;z=0
node7 x=0.000000;y=6.000000;z=0
```

Figure 10.6: An example of output file.

10.6 The average and best files

The fourth and fifth arguments are called *average* and *best file* respectively and give the average and the best EU of the population of each generation respectively. More precisely the average file describes the function that for each generation $i \in \{0, .., numb_{iter}\}$ associates the average EU of population of generation i; and the best file describes the function that for each generation $i \in \{0, .., numb_{iter}\}$ associates the best EU of population of generation i. These files are just a sequence of $numb_{iter} + 1$ numbers divided by CRs. The graphs of the functions they describe can be plotted by Gnuplot just by plotting the correspondent files.

10.7 The seed file

The sixth file is called *seed file* and contains the seed used to initialize the random number generator for that run. This file is important for the programmer when an error occurs. The seed can be used to repeat the run with the problem and find the cause of the error. The user will never use this file, but it is very important to keep this file especially of runs that didn't finish correctly.

10.8 The run time output

While running thr application prints some information on the screen: at the beginning it prints the average EU and the best EU of the initial population, the upper bound (UB) defined in page 43 and, if there exists a leaf such that the sum of the areas of the containers of the path from the root to that leaf is $>$ the area of the hold, it prints "upper bound unreachable" else "upper bound reachable"; then, after generating each generation, it prints the average EU and the best EU of that generation, the best EU found that far, the upper bound, the ratio between the best EU and the UB and the ratio between the average EU and the best EU of the last $shake_d$ generations.

Appendix A

The Support Applications

A.1 The random problem generator

The random problem generator is a program written in ANSI C that creates random BPPs to be used in experiments. It must be run with two arguments, names of files:

- the *input file*, an existing text file specifying some parameters that will influence the behaviour of the program: P_{term}, max_{depth}, max_c, m, max_n, max_d, min_d, max_l, min_l, max_e, min_e
- the *output file*, a text file created by the program with the same sintax of the one of Section 10.3, containing the description of a random BPP

When we run the program with these 2 arguments in this order, the program will generate a random BPP satisfying some constraints imposed by the parameters specified in the input file and will write it in the output file. The output BPP will have a tree with $n \leq max_n$, a depth $\leq max_{depth}$, for each node a number of children $\leq max_c$, m containers, $min_d \leq d_x \leq max_d$, $min_d \leq d_y \leq max_d$, for every container i $min_l \leq l_{x_i} \leq max_l$, $min_l \leq l_{y_i} \leq max_l$, $min_e \leq e \leq max_e$; $\forall i \in \{1, .., m\} : u_i \in]0, 1]$; if i_n is the number of inner nodes, then on the average there are $P_{term} \cdot i_n$ terminal inner nodes.

The structure of the input file is given in Figure A.1 where the symbols have the same meaning as in the Figure 10.1. The numbers following the symbol "=" are assigned respectively to P_{term}, m, max_c, max_{depth}, max_n, min_d, max_d, min_l, max_l, min_e, max_e. An example file is in figure A.2.

Of course it must be $max_{depth} \leq m$, $max_n \leq (m-1)!$, $max_c \leq m-1$ and $max_d > min_d > 0$, $max_l > min_l > 0$, $max_e > min_e > 0$ and $P_{term} \in [0, 1]$.

The output BPP is produced by the algorithm shown in Figure A.3.

P_term=$< prob >< sep >$
m=$< int >< sep >$
max_c=$< int >< sep >$
max_depth=$< int >< sep >$
max_n=$< int >< sep >$
min_d=$< real >< sep >$
max_d=$< real >< sep >$
min_l=$< real >< sep >$
max_l=$< real >< sep >$
min_e=$< real >< sep >$
max_e=$< real >< sep >$

Figure A.1: The structure of the input file of the random problem generator.

```
P_term=0.1
m=10
max_depth=8
max_n=1000
max_c=6
min_d=9.0
max_d=10.0
min_l=4.0
max_l=7.0
min_e=0.4
max_e=0.5
```

Figure A.2: An example of input file of the random problem generator.

{ read from the input file: max_n, max_{depth}, max_c, m, P_{term},
max_d, min_d, max_e, min_e, max_l, min_l;

the tree is created in the following way:

create the root node with a random container;
$n \leftarrow 1$;
for($i \leftarrow 1$ **to** $max_{depth} - 1$)
  for each(α node of depth i)
  { $\mu = min\{max_n - n, max_c, m - depth(\alpha)\}$;
    $k \leftarrow$ a random number in $\{0, 1, .., \mu\}$;
    if($k > 0$)
    { create k children of α;
      $n \leftarrow n + k$;
      $\forall\beta$ child of α: $v(\beta) \leftarrow$ a random number of $]0, 1]$;
      $r \leftarrow$ a random number in $]0, 1]$;
      if($r \leq P_{term}$) $v_{term} \leftarrow$ a random number in $]0, 1]$;
      else $v_{term} \leftarrow 0$;
      $\forall\beta$ child of $\alpha : p_\beta \leftarrow \frac{v(\beta)}{\sum_{\gamma \in C_\alpha} v(\gamma) + v_{term}}$
        with p_β the probability of the edge
        from α to β, and C_α the children of α;
      $\forall\beta$ child of α: choose a container between
        those not given to the nodes of the path
        from the root to α and to other children of α
        and give it to β;
    }
  }

the other elements of the BPP are created in the following way:

$\forall i \in \{1, .., m\} : u_i \leftarrow$ a real number in $]0, 1]$;
$\forall i \in \{1, .., m\} : l_{x_i} \leftarrow$ a real number in $[min_l, max_l]$;
$\forall i \in \{1, .., m\} : l_{y_i} \leftarrow$ a real number in $[min_l, max_l]$;
$d_x \leftarrow$ a real number in $[min_d, max_d]$;
$d_y \leftarrow$ a real number in $[min_d, max_d]$;
$e \leftarrow$ a real number in $[min_e, max_e]$;
}

Figure A.3: The random problem generator.

The program will also print on the display the number of nodes of each level.

A.2 The random-solver

The Random-solver algorithm presented in Section 9.2 has bee implemented in ANSI C. The implementation is a command line program that must be run with 3 arguments, names of files:

- the *iterations file*, an existing text file containing only an integer number
- the *input file*, an existing text file containing the description of a BPP, with the same sintax of the one in Section 10.3
- the *output file* a text file, created by the program, that contains the description of the best solution found by the program and with the same sintax of the one in Section 10.6

When we run the program with these 3 arguments in this order, the algorithm will be run on the BPP described in the input file for a number of iterations specified in the iterations file and the output of the algorithm will be written in the output file. While running, the program will print on the display the number of the current iteration, the EU of the corrispondent solution and the EU of the best solution found until then.

Appendix B

The Test Problems

B.1 rinput0

This problem has been randomly generated by the random problem generator with the parameters:

```
P_term=0
m=10
max_depth=8
max_n=1000
max_c=6
min_d=9.0
max_d=10.0
min_l=4.0
max_l=6.0
min_e=0.9
max_e=1
```

Then the dimensions of the deck, the utilities of the containers 2, 7 and 10 and the dimensions of all the containers have been changed.

The depth is 8 and the number of nodes in the levels $0,..,7$ are 1, 1, 5, 16, 33, 75, 119 respectively .

```
e=1
dx=12 dy=12
m=10
u1=0.879400
u2=0.3
u3=0.520081
u4=0.441070
u5=0.387028
u6=0.578156
u7=0.3
u8=0.555038
u9=0.522489
u10=0.25
```

lx1=4 ly1=5
lx2=4 ly2=4
lx3=4 ly3=6
lx4=6 ly4=6
lx5=5 ly5=6
lx6=5 ly6=5
lx7=4.5 ly7=5.5
lx8=4.5 ly8=6
lx9=4 ly9=6
lx10=6 ly10=6
n=251
(1,0,0,5) (2,1,1.000000,2) (3,2,1.000000,7) (4,3,0.307632,4)
(9,4,0.419167,9) (25,9,0.155749,8) (58,25,1.000000,6) (133,58,1.000000,1)
(26,9,0.399993,3) (59,26,0.319232,10) (134,59,1.000000,6) (60,26,0.329214,1)
(61,26,0.351554,8) (135,61,1.000000,1) (27,9,0.102655,6) (62,27,0.418978,10)
(136,62,0.512608,1) (137,62,0.152493,3) (138,62,0.334899,8) (63,27,0.581022,3)
(139,63,0.208845,1) (140,63,0.791155,8) (28,9,0.067031,10) (64,28,1.000000,3)
(141,64,0.248471,6) (142,64,0.751529,1) (29,9,0.274572,1) (10,4,0.314813,3)
(30,10,0.021485,8) (65,30,0.191291,10) (66,30,0.376486,9) (67,30,0.291959,1)
(143,67,1.000000,9) (68,30,0.140263,6) (31,10,0.054720,6) (32,10,0.438789,9)
(69,32,1.000000,6) (144,69,0.593631,1) (145,69,0.117735,8) (146,69,0.288633,10)
(33,10,0.485007,1) (11,4,0.266020,8) (34,11,1.000000,3) (70,34,0.135666,1)
(71,34,0.864334,10) (147,71,1.000000,6) (5,3,0.117060,3) (12,5,0.009935,4)
(13,5,0.132181,1) (14,5,0.338757,6) (15,5,0.311678,9) (35,15,0.395646,1)
(72,35,0.228570,10) (148,72,0.235887,8) (149,72,0.333206,6) (150,72,0.430907,4)
(73,35,0.701885,6) (74,35,0.069544,4) (151,74,0.302115,10) (152,74,0.245751,6)
(153,74,0.452134,8) (36,15,0.226040,8) (75,36,1.000000,10) (154,75,0.484739,6)
(155,75,0.250429,1) (156,75,0.264832,4) (37,15,0.198611,10) (76,37,0.335167,4)
(77,37,0.309238,6) (157,77,0.451798,8) (158,77,0.448022,4) (159,77,0.100180,1)
(78,37,0.090671,8) (79,37,0.264924,1) (160,79,0.333681,6) (161,79,0.328675,4)
(162,79,0.337643,8) (38,15,0.179703,4) (80,38,0.093453,1) (163,80,0.242955,10)
(164,80,0.757045,8) (81,38,0.312692,10) (165,81,0.302449,6) (166,81,0.697551,1)
(82,38,0.593855,6) (167,82,0.429515,8) (168,82,0.432217,1) (169,82,0.138268,10)
(16,5,0.207450,8) (39,16,0.288434,10) (83,39,0.073757,9) (170,83,0.410661,4)
(171,83,0.150200,1) (172,83,0.439139,6) (84,39,0.924349,4) (173,84,0.228803,1)
(174,84,0.771197,9) (85,39,0.001894,1) (175,85,0.668196,4) (176,85,0.327895,6)
(177,85,0.003909,9) (40,16,0.034273,1) (86,40,0.635942,9) (178,86,0.518803,4)
(179,86,0.481197,6) (87,40,0.116477,6) (180,87,1.000000,9) (88,40,0.247581,4)
(181,88,0.558790,10) (182,88,0.441210,9) (41,16,0.078266,9) (89,41,0.824884,4)
(90,41,0.175116,6) (183,90,0.163038,10) (184,90,0.503662,1) (185,90,0.333300,4)
(42,16,0.599028,6) (91,42,0.543464,9) (186,91,1.000000,1) (92,42,0.065065,4)
(93,42,0.251674,10) (187,93,0.627588,4) (188,93,0.200984,9) (189,93,0.171427,1)
(94,42,0.139797,1) (6,3,0.189528,9) (17,6,1.000000,1) (43,17,1.000000,10)
(95,43,0.351795,6) (190,95,1.000000,4) (96,43,0.372200,4) (191,96,0.275748,3)
(192,96,0.724252,8) (97,43,0.276005,3) (193,97,1.000000,6) (7,3,0.306821,10)
(18,7,0.099751,3) (44,18,0.407653,8) (98,44,0.529864,1) (194,98,1.000000,9)
(99,44,0.470136,4) (45,18,0.177191,1) (100,45,0.178143,6) (195,100,0.611458,4)
(196,100,0.388542,8) (101,45,0.435069,8) (197,101,0.258134,9) (198,101,0.149134,4)
(199,101,0.592733,6) (102,45,0.146064,9) (200,102,1.000000,8) (103,45,0.240724,4)
(201,103,0.941635,9) (202,103,0.058365,6) (46,18,0.330359,4) (47,18,0.084796,9)
(104,47,1.000000,1) (203,104,1.000000,6) (19,7,0.140484,8) (20,7,0.157712,9)
(21,7,0.148887,6) (48,21,1.000000,3) (105,48,0.356965,4) (204,105,1.000000,9)
(106,48,0.517569,8) (205,106,0.323187,9) (206,106,0.403910,1) (207,106,0.272903,4)
(107,48,0.125467,9) (208,107,1.000000,4) (22,7,0.246340,4) (49,22,0.293319,9)
(108,49,0.124218,8) (109,49,0.214525,1) (209,109,0.378899,8) (210,109,0.083922,3)
(211,109,0.537179,6) (110,49,0.324633,6) (212,110,1.000000,8) (111,49,0.336623,3)
(213,111,0.118216,6) (214,111,0.717910,8) (215,111,0.163874,1) (50,22,0.441840,6)
(112,50,1.000000,8) (216,112,0.419147,3) (217,112,0.257228,1) (218,112,0.323625,9)
(51,22,0.264840,8) (113,51,0.416180,9) (114,51,0.488321,3) (219,114,0.444530,1)
(220,114,0.555470,9) (115,51,0.095498,6) (221,115,0.266092,9) (222,115,0.733908,3)
(23,7,0.206827,1) (52,23,0.162666,9) (116,52,0.322524,3) (223,116,1.000000,4)
(117,52,0.677476,8) (224,117,0.816932,4) (225,117,0.183068,3) (53,23,0.240967,3)
(118,53,0.301657,4) (226,118,0.351704,8) (227,118,0.421123,9) (228,118,0.227174,6)
(119,53,0.281504,6) (229,119,0.155405,4) (230,119,0.672648,9) (231,119,0.171947,8)
(120,53,0.416839,9) (232,120,0.264719,6) (233,120,0.735281,4) (54,23,0.225703,4)
(121,54,0.292767,6) (122,54,0.198000,9) (234,122,0.953757,8) (235,122,0.025908,3)
(236,122,0.020335,6) (123,54,0.407716,8) (124,54,0.101517,3) (237,124,1.000000,6)
(55,23,0.195289,8) (125,55,1.000000,4) (56,23,0.175375,6) (126,56,0.298271,8)
(238,126,0.688829,9) (239,126,0.311171,4) (127,56,0.337734,9) (240,127,0.562923,3)
(241,127,0.437077,4) (128,56,0.109427,4) (242,128,0.057873,8) (243,128,0.525564,9)
(244,128,0.416563,3) (129,56,0.254568,3) (245,129,0.520532,8) (246,129,0.479468,9)
(8,3,0.078959,1) (24,8,1.000000,9) (57,24,1.000000,6) (130,57,0.391858,4)
(247,130,0.379463,8) (248,130,0.192214,3) (249,130,0.428324,10) (131,57,0.383642,8)
(250,131,0.972850,10) (251,131,0.027150,4) (132,57,0.224500,3)

B.2 rinput4

This problem has been generated by the random problem generator with the parameters file: This problem has been randomly generated by the random problem generator with the parameters:

```
P_term=0
m=10
max_depth=8
max_n=250
max_c=6
max_d=10.0
min_d=9.0
max_l=6.0
min_l=4.0
min_e=0.9
max_e=1
```

The depth is 5 and the number of nodes in the levels $0, .., 4$ are 1, 6, 17, 42, 123, 61 respectively .

e=0.918702
dx=9.126582 dy=9.671547
m=10
u1=0.952863
u2=0.759687
u3=0.446194
u4=0.242175
u5=0.244883
u6=0.178968
u7=0.218940
u8=0.143338
u9=0.941693
u10=0.613240
lx1=5.091558 ly1=4.942688
lx2=4.340651 ly2=4.344572
lx3=5.933731 ly3=5.165689
lx4=5.210119 ly4=4.355069
lx5=5.210560 ly5=4.727087
lx6=5.217512 ly6=5.366407
lx7=4.544291 ly7=4.267091
lx8=4.873959 ly8=4.759642
lx9=4.176219 ly9=5.305971
lx10=4.416357 ly10=5.559135
n=250
(1,0,0,9) (2,1,0.002380,3) (8,2,0.351808,8) (25,8,0.273552,6)
(67,25,0.283554,5) (190,67,0.972703,4) (191,67,0.027297,10) (68,25,0.091136,10)
(192,68,0.161187,4) (193,68,0.101448,1) (194,68,0.304021,5) (195,68,0.238892,7)
(196,68,0.194452,2) (69,25,0.145716,4) (197,69,0.348455,2) (198,69,0.249285,1)
(199,69,0.198101,10) (200,69,0.204158,7) (70,25,0.066341,1) (201,70,0.010732,10)
(202,70,0.173467,4) (203,70,0.307837,2) (204,70,0.333375,5) (205,70,0.174588,7)
(71,25,0.220569,7) (206,71,1.000000,2) (72,25,0.192685,2) (207,72,0.104400,10)
(208,72,0.409960,1) (209,72,0.171205,7) (210,72,0.152482,5) (211,72,0.161953,4)
(26,8,0.162262,7) (73,26,1.000000,2) (212,73,1.000000,10) (27,8,0.410838,10)
(74,27,0.031960,6) (213,74,0.631413,2) (214,74,0.368587,5) (75,27,0.477396,4)
(215,75,0.423734,5) (216,75,0.092440,2) (217,75,0.483826,1) (76,27,0.053662,1)
(218,76,0.213355,6) (219,76,0.143192,5) (220,76,0.220500,4) (221,76,0.324418,2)
(222,76,0.098536,7) (77,27,0.081072,7) (223,77,0.109694,5) (224,77,0.742332,1)
(225,77,0.147974,2) (78,27,0.355911,5) (226,78,0.358612,7) (227,78,0.631792,4)
(228,78,0.009596,2) (28,8,0.068150,4) (79,28,0.147078,10) (229,79,0.513077,2)
(230,79,0.359826,6) (231,79,0.127097,1) (80,28,0.186882,7) (232,80,0.017436,2)
(233,80,0.413250,1) (234,80,0.390995,10) (235,80,0.178319,5) (81,28,0.126331,5)
(236,81,0.156780,7) (237,81,0.224113,6) (238,81,0.313679,10) (239,81,0.224404,1)
(240,81,0.081024,2) (82,28,0.134719,2) (241,82,0.085541,10) (242,82,0.075621,7)

(243,82,0.316793,6) (244,82,0.522045,1) (83,28,0.182724,1) (245,83,0.006714,7)
(246,83,0.566031,5) (247,83,0.427255,10) (84,28,0.222266,6) (29,8,0.085197,2)
(85,29,1.000000,4) (248,85,0.639782,6) (249,85,0.360218,1) (9,2,0.648192,5)
(30,9,1.000000,7) (86,30,0.323411,8) (250,86,1.000000,6) (87,30,0.152046,6)
(88,30,0.071461,4) (89,30,0.247397,2) (90,30,0.174310,1) (91,30,0.031376,10)
(3,1,0.243703,6) (4,1,0.190924,10) (10,4,0.477023,4) (31,10,0.384253,1)
(92,31,0.013614,6) (93,31,0.429373,2) (94,31,0.557012,5) (32,10,0.119959,2)
(95,32,1.000000,5) (33,10,0.495788,5) (11,4,0.166124,7) (12,4,0.017388,6)
(34,12,0.359686,3) (96,34,0.654598,8) (97,34,0.345402,2) (35,12,0.293273,8)
(98,35,1.000000,2) (36,12,0.347041,2) (13,4,0.339465,8) (37,13,1.000000,2)
(99,37,0.321426,4) (100,37,0.645711,5) (101,37,0.032864,6) (5,1,0.115693,5)
(6,1,0.263350,2) (14,6,0.209780,1) (38,14,0.415268,3) (102,38,1.000000,8)
(39,14,0.315860,6) (40,14,0.268872,7) (103,40,0.302726,3) (104,40,0.311202,4)
(105,40,0.216233,8) (106,40,0.169839,10) (15,6,0.285803,4) (41,15,0.269791,1)
(107,41,0.703842,6) (108,41,0.296158,5) (42,15,0.005814,8) (109,42,1.000000,6)
(43,15,0.047864,10) (110,43,0.566766,8) (111,43,0.433234,3) (44,15,0.294383,5)
(112,44,0.578722,10) (113,44,0.421278,7) (45,15,0.198393,7) (114,45,0.014256,3)
(115,45,0.126886,8) (116,45,0.317068,10) (117,45,0.208490,1) (118,45,0.067875,5)
(119,45,0.265424,6) (46,15,0.183756,3) (120,46,0.215864,5) (121,46,0.120398,8)
(122,46,0.016963,1) (123,46,0.646775,7) (16,6,0.220712,3) (47,16,1.000000,7)
(124,47,0.348115,1) (125,47,0.263280,6) (126,47,0.388605,5) (17,6,0.127334,7)
(48,17,0.452265,4) (127,48,0.020719,6) (128,48,0.174115,1) (129,48,0.021940,10)
(130,48,0.238289,3) (131,48,0.165501,8) (132,48,0.379436,5) (49,17,0.004220,3)
(133,49,0.228233,8) (134,49,0.249114,4) (135,49,0.266822,1) (136,49,0.255830,6)
(50,17,0.388602,8) (137,50,0.402133,10) (138,50,0.597867,1) (51,17,0.067846,5)
(139,51,0.036646,6) (140,51,0.783527,4) (141,51,0.149303,1) (142,51,0.030524,3)
(52,17,0.087068,1) (143,52,0.253122,8) (144,52,0.233668,10) (145,52,0.286875,5)
(146,52,0.225340,4) (147,52,0.000995,6) (18,6,0.156371,8) (53,18,0.572320,7)
(148,53,0.212064,4) (149,53,0.257625,10) (150,53,0.530311,1) (54,18,0.427680,4)
(7,1,0.183950,8) (19,7,0.019343,1) (55,19,1.000000,5) (151,55,0.065213,6)
(152,55,0.108398,2) (153,55,0.303474,4) (154,55,0.080820,3) (155,55,0.442095,7)
(20,7,0.013363,7) (56,20,1.000000,2) (156,56,0.417220,5) (157,56,0.514818,6)
(158,56,0.067962,1) (21,7,0.417813,4) (57,21,0.142729,6) (159,57,0.206294,1)
(160,57,0.361732,3) (161,57,0.431975,5) (58,21,0.857271,10) (162,58,0.143495,2)
(163,58,0.575537,6) (164,58,0.280968,1) (22,7,0.163332,6) (23,7,0.165052,5)
(59,23,0.057834,1) (165,59,0.422618,7) (166,59,0.317568,3) (167,59,0.259814,6)
(60,23,0.184031,7) (61,23,0.189355,6) (168,61,0.645467,2) (169,61,0.354533,10)
(62,23,0.291542,3) (170,62,0.335402,1) (171,62,0.233186,7) (172,62,0.316868,4)
(173,62,0.114543,2) (63,23,0.186212,4) (174,63,0.192652,1) (175,63,0.202445,2)
(176,63,0.042960,10) (177,63,0.092728,7) (178,63,0.318025,6) (179,63,0.151191,3)
(64,23,0.091027,2) (180,64,0.097831,1) (181,64,0.117094,4) (182,64,0.447141,10)
(183,64,0.337935,3) (24,7,0.221096,10) (65,24,0.572410,6) (184,65,0.151725,1)
(185,65,0.171248,7) (186,65,0.184307,3) (187,65,0.223281,5) (188,65,0.183282,2)
(189,65,0.086157,4) (66,24,0.427591,4)

B.3 rinput5

This problem has been randomly generated by the random problem generator with the parameters:

```
P_term=0
m=10
max_depth=8
max_n=250
max_c=6
min_d=12
max_d=12
min_l=4
max_l=6
min_e=0.9
max_e=1
```

The depth is 7 and the number of nodes in the levels $0, .., 6$ are 1, 4, 8, 19, 48, 108, 61 respectively .

```
e=0.426953
dx=12.000000 dy=12.000000
m=10
u1=0.309596
u2=0.681586
u3=0.097839
u4=0.480864
u5=0.015402
u6=0.531325
u7=0.229494
u8=0.674735
u9=0.044884
u10=0.689140
lx1=4.258379 ly1=4.988028
lx2=6.046078 ly2=4.977947
lx3=5.890062 ly3=4.659769
lx4=4.075559 ly4=5.999486
lx5=5.068666 ly5=5.930395
lx6=6.569844 ly6=4.981225
lx7=5.267356 ly7=5.452986
lx8=5.066720 ly8=4.228218
lx9=6.597105 ly9=5.122993
lx10=5.188888 ly10=5.807456
n=249
(1,0,0,2) (2,1,0.028460,3) (6,2,1.000000,5) (14,6,0.058909,9)
(33,14,1.000000,4) (81,33,1.000000,6) (189,81,1.000000,1) (15,6,0.565391,7)
(34,15,0.405249,6) (82,34,0.583522,1) (190,82,0.178640,4) (191,82,0.821360,8)
(83,34,0.386001,9) (192,83,0.316287,1) (193,83,0.615647,4) (194,83,0.068066,8)
(84,34,0.030477,4) (195,84,1.000000,9) (35,15,0.119964,1) (85,35,0.639139,4)
(196,85,0.538989,6) (197,85,0.461011,8) (86,35,0.360861,6) (198,86,0.408364,8)
(199,86,0.337196,4) (200,86,0.254440,9) (36,15,0.023707,8) (87,36,1.000000,1)
(201,87,0.051766,6) (202,87,0.436797,4) (203,87,0.511437,9) (37,15,0.057062,4)
(88,37,1.000000,9) (204,88,0.618639,6) (205,88,0.381361,1) (38,15,0.394018,9)
(89,38,0.655113,8) (90,38,0.344887,1) (16,6,0.375700,8) (39,16,1.000000,9)
(91,39,0.388826,1) (206,91,1.000000,6) (92,39,0.000020,4) (207,92,0.460946,6)
(208,92,0.367153,1) (209,92,0.171900,7) (93,39,0.046858,7) (94,39,0.564296,6)
(210,94,0.711317,1) (211,94,0.288683,4) (3,1,0.181229,4) (7,3,0.037192,3)
(17,7,0.008504,6) (40,17,0.248041,7) (95,40,0.142679,8) (212,95,1.000000,5)
(96,40,0.315190,9) (97,40,0.191972,5) (213,97,0.492746,9) (214,97,0.507254,1)
(98,40,0.350159,1) (215,98,0.120362,5) (216,98,0.501829,9) (217,98,0.377809,8)
(41,17,0.359246,8) (99,41,0.491265,9) (218,99,0.021715,5) (219,99,0.978285,7)
(100,41,0.508735,7) (220,100,1.000000,1) (42,17,0.392713,9) (101,42,0.706835,8)
(221,101,0.807005,7) (222,101,0.192995,1) (102,42,0.097005,7) (223,102,1.000000,8)
(103,42,0.196159,1) (224,103,1.000000,8) (18,7,0.206857,9) (43,18,0.078597,1)
(104,43,0.188936,7) (105,43,0.278245,6) (225,105,1.000000,5) (106,43,0.116406,5)
(107,43,0.416413,8) (226,107,0.368078,7) (227,107,0.340413,5) (228,107,0.291509,6)
(44,18,0.040920,6) (45,18,0.497085,7) (108,45,0.641663,1) (109,45,0.358337,6)
(229,109,0.429149,1) (230,109,0.241360,8) (231,109,0.329490,5) (46,18,0.383398,5)
(110,46,0.410631,1) (232,110,0.848183,8) (233,110,0.151817,7) (111,46,0.589369,6)
(234,111,0.381005,1) (235,111,0.311619,8) (236,111,0.307376,7) (19,7,0.227641,7)
(47,19,0.263195,1) (112,47,1.000000,5) (237,112,0.405593,6) (238,112,0.184839,8)
(239,112,0.409568,9) (48,19,0.362464,9) (113,48,0.522062,1) (240,113,0.637202,6)
(241,113,0.362798,5) (114,48,0.477938,5) (242,114,1.000000,8) (49,19,0.163048,8)
(115,49,1.000000,6) (243,115,0.175448,9) (244,115,0.064712,1) (245,115,0.759840,5)
(50,19,0.211294,6) (116,50,0.391054,9) (246,116,1.000000,5) (117,50,0.608946,8)
(247,117,1.000000,9) (20,7,0.216369,1) (51,20,0.224368,9) (118,51,1.000000,8)
(248,118,0.483870,5) (249,118,0.516130,7) (52,20,0.381441,6) (119,52,0.903030,7)
(120,52,0.096970,9) (53,20,0.103659,8) (121,53,0.606471,7) (122,53,0.393529,9)
(54,20,0.290533,5) (123,54,0.087604,7) (124,54,0.083332,6) (125,54,0.506794,9)
(126,54,0.322270,8) (21,7,0.340630,8) (8,3,0.365122,9) (22,8,0.134202,3)
(23,8,0.216505,6) (55,23,0.232458,8) (56,23,0.192985,7) (57,23,0.164028,5)
(127,57,0.335907,8) (128,57,0.664093,3) (58,23,0.221326,3) (129,58,0.499380,8)
(130,58,0.115550,7) (131,58,0.210576,5) (132,58,0.174495,1) (59,23,0.189203,1)
(133,59,0.289482,7) (134,59,0.135498,8) (135,59,0.241503,3) (136,59,0.333517,5)
(24,8,0.126681,1) (60,24,0.508306,7) (61,24,0.491694,6) (137,61,0.517331,8)
(138,61,0.000940,5) (139,61,0.481729,3) (25,8,0.522612,7) (62,25,0.694207,8)
(140,62,0.328443,5) (141,62,0.419922,1) (142,62,0.251635,6) (63,25,0.082251,6)
(143,63,0.402777,5) (144,63,0.176316,8) (145,63,0.420908,3) (64,25,0.223543,5)
(146,64,0.511983,3) (147,64,0.266923,8) (148,64,0.207358,6) (149,64,0.013735,1)
(9,3,0.276837,8) (10,3,0.037037,7) (26,10,1.000000,8) (65,26,0.294315,5)
(150,65,0.405462,6) (151,65,0.229757,1) (152,65,0.364781,3) (66,26,0.175953,3)
(153,66,0.543490,6) (154,66,0.091702,5) (155,66,0.364808,1) (67,26,0.379698,6)
(156,67,0.297646,5) (157,67,0.364792,3) (158,67,0.229190,1) (159,67,0.108372,9)
(68,26,0.150034,9) (160,68,0.251041,3) (161,68,0.337247,5) (162,68,0.096760,6)
(163,68,0.314952,1) (11,3,0.283812,5) (27,11,1.000000,9) (69,27,0.455327,8)
(164,69,0.189171,3) (165,69,0.462408,6) (166,69,0.260764,1) (167,69,0.087657,7)
(70,27,0.544673,3) (4,1,0.693225,8) (12,4,0.366258,5) (28,12,0.283004,9)
(71,28,0.402722,1) (168,71,0.296643,3) (169,71,0.115354,7) (170,71,0.269312,4)
```

(171,71,0.318691,6) (72,28,0.190119,7) (172,72,1.000000,4) (73,28,0.407159,3)
(173,73,0.102167,4) (174,73,0.411126,6) (175,73,0.486707,7) (29,12,0.221945,4)
(74,29,0.448911,3) (176,74,1.000000,9) (75,29,0.551089,6) (30,12,0.495051,7)
(13,4,0.633742,6) (31,13,0.564195,7) (32,13,0.435805,9) (76,32,0.255419,7)
(177,76,1.000000,4) (77,32,0.084471,5) (178,77,0.359611,3) (179,77,0.161873,7)
(180,77,0.478516,4) (78,32,0.296013,3) (181,78,0.202623,7) (182,78,0.278415,5)
(183,78,0.243391,1) (184,78,0.275571,4) (79,32,0.004692,4) (185,79,0.239159,1)
(186,79,0.171408,3) (187,79,0.331311,7) (188,79,0.258121,5) (80,32,0.359405,1)

(5,1,0.097086,1)

B.4 rinput6

This problem has been randomly generated by the random problem generator with the parameters:

```
P_term=0
m=12
max_depth=10
max_n=250
max_c=12
min_d=12
max_d=12
min_l=4
max_l=6
min_e=0.9
max_e=1
```

The depth is 4 and the number of nodes in the levels $0, .., 3$ are 1, 9, 50, 190 respectively .

e=0.926691
dx=12.000000 dy=12.000000
m=12
u1=0.388017
u2=0.585707
u3=0.203572
u4=0.937831
u5=0.701077
u6=0.118898
u7=0.191186
u8=0.995888
u9=0.530815
u10=0.602019
u11=0.648486
u12=0.039909
lx1=4.335007 ly1=4.092933
lx2=5.176459 ly2=4.782847
lx3=5.179012 ly3=4.037170
lx4=5.159808 ly4=5.400181
lx5=4.159598 ly5=5.838077
lx6=4.920182 ly6=4.621942
lx7=5.555429 ly7=5.380552
lx8=4.241450 ly8=4.904070
lx9=4.828291 ly9=4.864192
lx10=4.448120 ly10=4.350715
lx11=4.981933 ly11=4.685722
lx12=5.757967 ly12=5.862182
n=250
(1,0,0,11) (2,1,0.090030,2) (11,2,0.186275,10) (61,11,0.018187,4)
(62,11,0.139105,6) (63,11,0.141525,1) (64,11,0.097382,5) (65,11,0.116188,12)

(66,11,0.135093,8) (67,11,0.146518,7) (68,11,0.041503,3) (69,11,0.164500,9)
(12,2,0.248533,7) (70,12,0.268166,12) (71,12,0.191306,8) (72,12,0.033227,9)
(73,12,0.246089,6) (74,12,0.224911,3) (75,12,0.036302,10) (13,2,0.059933,5)
(76,13,0.030901,7) (77,13,0.151589,8) (78,13,0.125832,12) (79,13,0.198898,4)
(80,13,0.108441,1) (81,13,0.099204,6) (82,13,0.020427,9) (83,13,0.107813,10)
(84,13,0.156894,3) (14,2,0.262961,4) (85,14,0.278737,6) (86,14,0.102994,1)
(87,14,0.272825,5) (88,14,0.165453,7) (89,14,0.179991,8) (15,2,0.122188,8)
(90,15,0.052396,9) (91,15,0.139887,12) (92,15,0.186545,1) (93,15,0.226553,7)
(94,15,0.085104,4) (95,15,0.006566,3) (96,15,0.033790,5) (97,15,0.113376,10)
(98,15,0.155783,6) (16,2,0.120110,3) (99,16,1.000000,6) (3,1,0.037325,8)
(17,3,0.147120,4) (100,17,0.434332,5) (101,17,0.278601,9) (102,17,0.287067,10)
(18,3,0.040295,6) (103,18,0.262423,7) (104,18,0.206417,3) (105,18,0.215391,5)
(106,18,0.102150,12) (107,18,0.085464,9) (108,18,0.128155,2) (19,3,0.076381,1)
(109,19,0.234247,9) (110,19,0.017329,3) (111,19,0.053996,10) (112,19,0.263388,7)
(113,19,0.052724,12) (114,19,0.042591,4) (115,19,0.232149,2) (116,19,0.103576,5)
(20,3,0.137370,3) (117,20,1.000000,5) (21,3,0.170269,5) (118,21,0.100901,10)
(119,21,0.168488,2) (120,21,0.052798,3) (121,21,0.199272,12) (122,21,0.088434,6)
(123,21,0.122647,1) (124,21,0.105985,4) (125,21,0.081721,9) (126,21,0.079754,7)
(22,3,0.043334,12) (127,22,0.170981,10) (128,22,0.105427,5) (129,22,0.105956,1)
(130,22,0.617636,4) (23,3,0.188942,9) (131,23,0.171602,1) (132,23,0.208844,6)
(133,23,0.160763,7) (134,23,0.048140,4) (135,23,0.159479,12) (136,23,0.218647,3)
(137,23,0.032525,10) (24,3,0.196288,10) (4,1,0.189728,3) (25,4,0.129647,12)
(138,25,0.169375,8) (139,25,0.306474,2) (140,25,0.524151,9) (26,4,0.108355,1)
(141,26,0.074677,9) (142,26,0.001560,12) (143,26,0.183865,10) (144,26,0.073046,5)
(145,26,0.111062,8) (146,26,0.190817,2) (147,26,0.166889,6) (148,26,0.060526,4)
(149,26,0.137557,7) (27,4,0.172188,7) (150,27,0.064765,10) (151,27,0.025597,8)
(152,27,0.142445,5) (153,27,0.083361,2) (154,27,0.094873,9) (155,27,0.169446,4)
(156,27,0.132219,6) (157,27,0.178433,1) (158,27,0.108860,12) (28,4,0.204810,5)
(159,28,0.421824,12) (160,28,0.047514,7) (161,28,0.116046,6) (162,28,0.414616,10)
(29,4,0.059577,6) (30,4,0.325423,2) (163,30,0.155916,4) (164,30,0.013632,1)
(165,30,0.056541,5) (166,30,0.185138,10) (167,30,0.176246,9) (168,30,0.151705,7)
(169,30,0.058460,8) (170,30,0.134701,6) (171,30,0.067661,12) (5,1,0.117862,10)
(31,5,0.037170,3) (172,31,0.106277,6) (173,31,0.106221,5) (174,31,0.107699,2)
(175,31,0.123798,7) (176,31,0.149014,1) (177,31,0.260592,8) (178,31,0.113121,9)
(179,31,0.033279,12) (32,5,0.182297,7) (33,5,0.212989,1) (180,33,0.166159,7)
(181,33,0.833841,3) (34,5,0.128933,9) (182,34,0.348004,1) (183,34,0.028667,5)
(184,34,0.139063,7) (185,34,0.037419,4) (186,34,0.298028,12) (187,34,0.148820,2)
(35,5,0.093701,8) (188,35,0.183160,3) (189,35,0.133117,12) (190,35,0.026330,6)
(191,35,0.059875,5) (192,35,0.228548,1) (193,35,0.071908,2) (194,35,0.149353,4)
(195,35,0.016895,7) (196,35,0.130815,9) (36,5,0.163917,4) (197,36,0.082748,5)
(198,36,0.028552,2) (199,36,0.216015,3) (200,36,0.227668,7) (201,36,0.100917,12)
(202,36,0.225739,9) (203,36,0.090027,6) (204,36,0.028334,1) (37,5,0.180993,2)
(205,37,0.138701,12) (206,37,0.158699,1) (207,37,0.161103,9) (208,37,0.063605,4)
(209,37,0.033161,8) (210,37,0.150684,6) (211,37,0.163995,5) (212,37,0.130053,3)
(6,1,0.193837,7) (38,6,0.246990,2) (213,38,0.168546,12) (214,38,0.159025,1)
(215,38,0.178867,10) (216,38,0.294801,4) (217,38,0.198761,3) (39,6,0.191255,8)
(218,39,0.195124,4) (219,39,0.086990,3) (220,39,0.147342,12) (221,39,0.142561,2)
(222,39,0.063899,5) (223,39,0.127792,10) (224,39,0.236292,1) (40,6,0.074545,10)
(225,40,0.340634,3) (226,40,0.283443,6) (227,40,0.015356,4) (228,40,0.360568,12)
(41,6,0.073483,3) (229,41,0.002178,1) (230,41,0.359585,8) (231,41,0.231563,9)
(232,41,0.406673,10) (42,6,0.002906,1) (233,42,0.052698,4) (234,42,0.113299,2)
(235,42,0.050737,10) (236,42,0.208334,9) (237,42,0.088130,12) (238,42,0.188072,6)
(239,42,0.008518,8) (240,42,0.044341,5) (241,42,0.245871,3) (43,6,0.194572,4)
(242,43,0.179186,5) (243,43,0.402963,8) (244,43,0.417851,3) (44,6,0.216250,9)
(245,44,1.000000,10) (7,1,0.125115,12) (45,7,1.000000,8) (246,45,0.135187,10)
(247,45,0.268168,6) (248,45,0.337087,1) (249,45,0.259557,5) (8,1,0.171412,5)
(46,8,0.130856,7) (250,46,1.000000,3) (47,8,0.043826,2) (48,8,0.044200,4)
(49,8,0.058389,3) (50,8,0.190028,1) (51,8,0.098372,6) (52,8,0.176408,9)
(53,8,0.196885,10) (54,8,0.061036,8) (9,1,0.049644,4) (55,9,0.451611,10)
(56,9,0.548389,12) (10,1,0.025046,6) (57,10,0.836454,12) (58,10,0.109694,8)

(59,10,0.010038,10) (60,10,0.043814,4)

B.5 rinput7

This problem has been randomly generated by the random problem generator with the parameters:

```
P_term=0
m=20
max_depth=20
max_n=250
```

```
max_c=2
min_d=12
max_d=12
min_l=4
max_l=6
min_e=0.9
max_e=1
```

The depth is 20 and the number of nodes in the levels $0, .., 19$ are 1, 1 1,2, 2, 3, 5, 6, 4, 5, 5, 4, 4, 5, 5, 3, 2, 3, 3, 3 respectively .

```
e=0.994018
dx=12.000000 dy=12.000000
m=20
u1=0.337024 u2=0.091553 u3=0.939355 u4=0.397234
u5=0.953905 u6=0.901818 u7=0.143803 u8=0.135479
u9=0.620797 u10=0.754579 u11=0.955412 u12=0.105485
u13=0.034497 u14=0.526333 u15=0.676833 u16=0.409236
u17=0.807043 u18=0.927795 u19=0.599367 u20=0.133161
lx1=5.508361 ly1=4.401666 lx2=5.195940 ly2=4.300145
lx3=4.631782 ly3=5.858026 lx4=5.619237 ly4=4.983671
lx5=4.402030 ly5=4.301000 lx6=4.506777 ly6=5.464806
lx7=5.052024 ly7=4.795615 lx8=5.521075 ly8=4.241503
lx9=4.537391 ly9=5.343146 lx10=4.298859 ly10=5.067587
lx11=4.179219 ly11=4.575948 lx12=4.853268 ly12=5.771889
lx13=5.036373 ly13=4.403670 lx14=4.915084 ly14=4.022907
lx15=5.709552 ly15=4.424937 lx16=5.617362 ly16=4.931715
lx17=5.420997 ly17=5.983738 lx18=5.708602 ly18=5.504813
lx19=5.979561 ly19=4.042204 lx20=5.221155 ly20=4.341063
n=67
(1,0,0,11) (2,1,1.000000,7) (3,2,1.000000,3) (4,3,0.560437,18)
(6,4,1.000000,12) (8,6,1.000000,8) (11,8,0.869068,4) (16,11,1.000000,17)
(12,8,0.130932,17) (17,12,0.398731,10) (18,12,0.601269,15) (22,18,1.000000,9)
(26,22,0.772080,20) (31,26,0.993237,14) (36,31,1.000000,16) (32,26,0.006763,2)
(27,22,0.227920,2) (33,27,1.000000,14) (37,33,0.634474,13) (40,37,0.562956,20)
(41,37,0.437044,19) (44,41,1.000000,20) (49,44,1.000000,5) (54,49,1.000000,4)
(57,54,1.000000,6) (59,57,1.000000,16) (62,59,1.000000,1) (65,62,1.000000,10)
(38,33,0.365526,1) (42,38,1.000000,6) (45,42,0.681309,10) (50,45,0.711515,20)
(51,45,0.288485,19) (46,42,0.318691,19) (5,3,0.439563,9) (7,5,1.000000,10)
(9,7,0.202128,5) (13,9,1.000000,16) (10,7,0.797872,17) (14,10,0.258987,2)
(19,14,0.530411,6) (20,14,0.469589,8) (23,20,0.158981,18) (28,23,0.310374,1)
(34,28,1.000000,16) (29,23,0.689626,6) (24,20,0.841019,12) (15,10,0.741013,1)
(21,15,1.000000,13) (25,21,1.000000,8) (30,25,1.000000,14) (35,30,1.000000,19)
(39,35,1.000000,12) (43,39,1.000000,18) (47,43,0.169870,4) (52,47,0.562343,20)
(53,47,0.437657,2) (55,53,0.071280,6) (58,55,1.000000,20) (60,58,0.479617,16)
(63,60,1.000000,5) (66,63,1.000000,15) (61,58,0.520383,5) (64,61,1.000000,15)
(67,64,1.000000,16) (56,53,0.928720,20) (48,43,0.830130,16)
```

B.6 rinput8

This problem has been randomly generated by the random problem generator with the parameters:

```
P_term=0
m=30
max_depth=30
max_n=250
max_c=3
min_d=12
```

```
max_d=12
min_l=4
max_l=6
min_e=0.9
max_e=1
```

The depth is 18 and the number of nodes in the levels $0, .., 17$ are 1, 3, 1, 2, 1, 1, 2, 2, 3, 5, 10, 13, 19, 27, 35, 40, 59, 26, respectively.

```
e=0.922248
dx=12.000000 dy=12.000000
m=30
u1=0.879693
u2=0.315857
u3=0.530273
u4=0.451764
u5=0.149363
u6=0.451858
u7=0.705635
u8=0.627565
u9=0.185777
u10=0.728908
u11=0.634328
u12=0.242377
u13=0.734580
u14=0.026793
u15=0.673433
u16=0.313410
u17=0.735025
u18=0.828534
u19=0.396488
u20=0.490870
u21=0.950437
u22=0.817923
u23=0.712021
u24=0.506416
u25=0.000558
u26=0.659692
u27=0.587871
u28=0.571155
u29=0.086542
u30=0.713065
lx1=4.730370 ly1=5.810840
lx2=4.815837 ly2=5.216098
lx3=4.053226 ly3=4.984405
lx4=5.800591 ly4=4.584427
lx5=5.399495 ly5=5.293280
lx6=4.329984 ly6=5.279953
lx7=4.538090 ly7=4.843230
lx8=4.110209 ly8=4.187018
lx9=5.064206 ly9=5.135907
lx10=4.222272 ly10=4.188765
lx11=4.667225 ly11=4.919134
lx12=4.426611 ly12=5.734972
lx13=5.058325 ly13=5.788198
lx14=4.118870 ly14=5.588789
lx15=5.022398 ly15=4.988284
lx16=5.321125 ly16=5.318268
lx17=4.224840 ly17=5.856359
lx18=5.636110 ly18=5.966568
lx19=4.891241 ly19=5.030774
lx20=5.262794 ly20=5.253046
lx21=4.720612 ly21=5.920270
lx22=5.989268 ly22=4.346882
lx23=4.474022 ly23=5.405207
lx24=5.943182 ly24=5.524077
lx25=5.996769 ly25=4.546475
lx26=5.343636 ly26=5.867599
lx27=5.970455 ly27=4.092440
lx28=5.440505 ly28=5.728549
lx29=5.097573 ly29=4.619791
lx30=5.890549 ly30=5.882585
n=250
(1,0,0,14) (2,1,0.093384,3) (3,1,0.491979,24) (5,3,1.000000,10)
(6,5,0.697731,18) (8,6,1.000000,28) (9,8,1.000000,19) (10,9,0.364994,6)
(12,10,1.000000,4) (14,12,1.000000,21) (17,14,0.254441,15) (22,17,0.498718,2)
(32,22,0.050292,3) (45,32,0.490063,1) (64,45,1.000000,8) (91,64,0.579144,7)
```

(126,91,1.000000,20) (166,126,0.673850,11) (167,126,0.326150,9) (225,167,0.790951,25)
(226,167,0.209049,27) (92,64,0.420857,9) (127,92,0.357230,30) (128,92,0.278980,12)
(168,128,0.202679,13) (227,168,0.336734,29) (228,168,0.323540,27) (229,168,0.339726,17)
(169,128,0.377536,25) (230,169,0.150748,11) (231,169,0.849252,23) (170,128,0.419786,23)
(232,170,1.000000,30) (129,92,0.363790,17) (46,32,0.380204,17) (65,46,1.000000,5)
(93,65,0.513368,11) (94,65,0.486632,7) (47,32,0.129733,30) (66,47,1.000000,5)
(33,22,0.949708,29) (48,33,0.742127,3) (67,48,0.146647,1) (95,67,0.200774,12)
(96,67,0.440013,25) (97,67,0.359213,7) (130,97,0.527392,5) (171,130,1.000000,22)
(233,171,1.000000,27) (131,97,0.472608,13) (172,131,0.514883,17) (173,131,0.485117,8)
(234,173,1.000000,30) (68,48,0.556717,26) (98,68,1.000000,22) (69,48,0.296636,5)
(99,69,0.347516,17) (100,69,0.652484,13) (132,100,0.290988,12) (174,132,1.000000,7)
(235,174,0.811927,17) (236,174,0.125897,26) (237,174,0.062176,9) (133,100,0.382075,26)
(175,133,0.828816,7) (176,133,0.171184,8) (238,176,0.068622,12) (239,176,0.931378,30)
(134,100,0.326938,23) (177,134,1.000000,30) (240,177,0.253654,11) (241,177,0.405996,7)
(242,177,0.340350,17) (49,33,0.257873,8) (70,49,1.000000,5) (101,70,0.480759,25)
(135,101,0.325875,3) (178,135,0.782163,20) (179,135,0.091388,13) (243,179,1.000000,30)
(180,135,0.126449,23) (244,180,0.168130,20) (245,180,0.380201,22) (246,180,0.451669,30)
(136,101,0.674126,23) (181,136,1.000000,12) (247,181,0.902440,22) (248,181,0.097560,3)
(102,70,0.157284,23) (137,102,1.000000,12) (182,137,0.452026,16) (249,182,0.866901,11)
(250,182,0.133099,1) (183,137,0.391277,11) (184,137,0.156697,7) (103,70,0.361957,16)
(138,103,0.304919,30) (185,138,1.000000,11) (139,103,0.166493,12) (186,139,0.501873,13)
(187,139,0.498127,17) (140,103,0.528588,3) (188,140,0.485691,7) (189,140,0.514309,11)
(23,17,0.390024,11) (34,23,0.005685,5) (50,34,1.000000,30) (71,50,0.248732,27)
(104,71,1.000000,20) (72,50,0.751268,22) (35,23,0.646130,27) (51,35,1.000000,2)
(73,51,0.183570,23) (105,73,1.000000,12) (141,105,0.408738,9) (190,141,0.994760,20)
(191,141,0.005240,3) (142,105,0.591262,22) (192,142,0.014253,29) (193,142,0.985747,17)
(74,51,0.469777,30) (106,74,1.000000,5) (143,106,1.000000,26) (75,51,0.346653,13)
(107,75,0.085255,26) (108,75,0.763905,29) (109,75,0.150840,9) (144,109,0.484165,23)
(194,144,0.322002,30) (195,144,0.616365,7) (196,144,0.061633,8) (145,109,0.220250,1)
(197,145,1.000000,20) (146,109,0.295585,30) (36,23,0.348185,23) (24,17,0.111258,9)
(37,24,0.503729,16) (38,24,0.496271,8) (52,38,0.553165,11) (53,38,0.360662,29)
(76,53,0.551169,12) (77,53,0.448831,25) (110,77,0.188842,16) (147,110,1.000000,22)
(198,147,1.000000,11) (111,77,0.811158,22) (148,111,0.216417,23) (199,148,0.335436,20)
(200,148,0.452625,1) (201,148,0.211939,30) (149,111,0.096843,11) (202,149,0.040542,20)
(203,149,0.359118,16) (204,149,0.600341,23) (150,111,0.686740,13) (205,150,0.166752,11)
(206,150,0.833248,23) (54,38,0.086173,26) (78,54,0.224232,22) (79,54,0.414985,23)
(112,79,1.000000,11) (151,112,0.542086,25) (207,151,0.344167,22) (208,151,0.498500,1)
(209,151,0.157333,7) (152,112,0.457914,12) (210,152,1.000000,25) (80,54,0.360783,3)
(18,14,0.478400,8) (25,18,1.000000,30) (19,14,0.267159,12) (26,19,0.370241,2)
(27,19,0.370666,27) (39,27,0.357167,23) (40,27,0.265163,26) (55,40,0.423954,15)
(81,55,0.723356,1) (82,55,0.276644,11) (56,40,0.082559,23) (83,56,1.000000,15)
(113,83,0.959875,5) (153,113,1.000000,22) (211,153,1.000000,17) (114,83,0.040125,22)
(57,40,0.493487,1) (84,57,0.665737,15) (115,84,1.000000,7) (85,57,0.334263,23)
(116,85,0.388816,25) (117,85,0.200723,5) (154,117,0.207102,7) (212,154,0.251594,11)
(213,154,0.064384,30) (214,154,0.684022,16) (155,117,0.149732,17) (215,155,1.000000,11)
(156,117,0.643165,9) (118,85,0.410461,30) (157,118,0.552643,22) (158,118,0.170290,25)
(159,118,0.277068,3) (216,159,0.738725,2) (217,159,0.261275,29) (41,27,0.377670,25)
(58,41,0.247583,3) (59,41,0.752417,7) (28,19,0.259093,9) (11,9,0.635006,7)
(13,11,1.000000,21) (15,13,0.653828,26) (20,15,1.000000,22) (29,20,0.503596,12)
(30,20,0.496404,5) (16,13,0.346172,15) (21,16,1.000000,11) (31,21,1.000000,20)
(42,31,0.287458,1) (60,42,1.000000,12) (86,60,0.638920,22) (119,86,0.083318,27)
(120,86,0.916682,17) (87,60,0.361080,13) (121,87,0.679043,4) (160,121,0.658539,6)
(161,121,0.341461,17) (218,161,0.729179,8) (219,161,0.270821,25) (122,87,0.320957,17)
(43,31,0.320278,27) (61,43,0.549175,1) (62,43,0.450825,25) (44,31,0.392264,29)
(63,44,1.000000,27) (88,63,0.436191,23) (123,88,0.433414,17) (162,123,0.245932,1)
(220,162,0.474582,6) (221,162,0.525418,30) (163,123,0.534087,6) (164,123,0.219981,5)
(222,164,0.719374,16) (223,164,0.269207,13) (224,164,0.011418,1) (124,88,0.566586,26)
(89,63,0.441438,17) (90,63,0.122371,30) (125,90,1.000000,1) (165,125,1.000000,6)

(7,5,0.302269,2) (4,1,0.414637,1)

B.7 art7

This problem is rinput7 with one more container This container is so big that no other container can be loaded on the deck together and is associated to all the leaves and only to them. Its utility is bigger that the sum of the utilities of all the other containers. The optimal solution is then the solution that associates this container to all the leaves and NULL to all the other nodes.

e=0.994018

```
dx=12.000000 dy=12.000000
m=21
u1=0.337024
u2=0.091553
u3=0.939355
u4=0.397234
u5=0.953905
u6=0.901818
u7=0.143803
u8=0.135479
u9=0.620797
u10=0.754579
u11=0.955412
u12=0.105485
u13=0.034497
u14=0.526333
u15=0.676833
u16=0.409236
u17=0.807043
u18=0.927795
u19=0.599367
u20=0.133161
u21=20
lx1=5.508361 ly1=4.401666
lx2=5.195940 ly2=4.300145
lx3=4.631782 ly3=5.858026
lx4=5.619237 ly4=4.983671
lx5=4.402030 ly5=4.301000
lx6=4.506777 ly6=5.464806
lx7=5.052024 ly7=4.795615
lx8=5.521075 ly8=4.241503
lx9=4.537391 ly9=5.343146
lx10=4.298859 ly10=5.067587
lx11=4.179219 ly11=4.575948
lx12=4.853268 ly12=5.771889
lx13=5.036373 ly13=4.403670
lx14=4.915084 ly14=4.022907
lx15=5.709552 ly15=4.424937
lx16=5.617362 ly16=4.931715
lx17=5.420997 ly17=5.983738
lx18=5.708602 ly18=5.504813
lx19=5.979561 ly19=4.042204
lx20=5.221155 ly20=4.341063
lx21=11 ly21=11
n=67
(1,0,0,11) (2,1,1.000000,7) (3,2,1.000000,3) (4,3,0.560437,18)
(6,4,1.000000,12) (8,6,1.000000,8) (11,8,0.869068,4) (16,11,1.000000,21)
(12,8,0.130932,17) (17,12,0.398731,21) (18,12,0.601269,15) (22,18,1.000000,9)
(26,22,0.772080,20) (31,26,0.993237,14) (36,31,1.000000,21) (32,26,0.006763,21)
(27,22,0.227920,2) (33,27,1.000000,14) (37,33,0.634474,13) (40,37,0.562956,21)
(41,37,0.437044,19) (44,41,1.000000,20) (49,44,1.000000,5) (54,49,1.000000,4)
(57,54,1.000000,6) (59,57,1.000000,16) (62,59,1.000000,1) (65,62,1.000000,21)
(38,33,0.365526,1) (42,38,1.000000,6) (45,42,0.681309,10) (50,45,0.711515,21)
(51,45,0.288485,21) (46,42,0.318691,21) (5,3,0.439563,9) (7,5,1.000000,10)
(9,7,0.202128,5) (13,9,1.000000,21) (10,7,0.797872,17) (14,10,0.258987,2)
(19,14,0.530411,21) (20,14,0.469589,8) (23,20,0.158981,18) (28,23,0.310374,1)
(34,28,1.000000,21) (29,23,0.689626,21) (24,20,0.841019,21) (15,10,0.741013,1)
(21,15,1.000000,13) (25,21,1.000000,8) (30,25,1.000000,14) (35,30,1.000000,19)
(39,35,1.000000,12) (43,39,1.000000,18) (47,43,0.169870,4) (52,47,0.562343,21)
(53,47,0.437657,2) (55,53,0.071280,6) (58,55,1.000000,20) (60,58,0.479617,16)
(63,60,1.000000,21) (66,63,1.000000,15) (61,58,0.520383,5) (64,61,1.000000,15)

(67,64,1.000000,21) (56,53,0.928720,21) (48,43,0.830130,21)
```

B.8 lin16

This problem has a linear tree with 16 nodes. All the containers can be loaded. The utility of a container is its area. The deck can be completely filled.

```
e=1
dx=20 dy=20
m=16
u1=24.0
u2=84.0
```

```
u3=48.0
u4=18.0
u5=15.0
u6=25.0
u7=36.0
u8=21.0
u9=35.0
u10=12.0
u11=6.0
u12=8.0
u13=12.0
u14=16.0
u15=18.0
u16=22.0
lx1=2 ly1=12
lx2=7 ly2=12
lx3=8 ly3=6
lx4=3 ly4=6
lx5=3 ly5=5
lx6=5 ly6=5
lx7=3 ly7=12
lx8=3 ly8=7
lx9=5 ly9=7
lx10=2 ly10=6
lx11=3 ly11=2
lx12=4 ly12=2
lx13=3 ly13=4
lx14=4 ly14=4
lx15=9 ly15=2
lx16=11 ly16=2
n=16
(1,0,0,1) (2,1,1,2) (3,2,1,3) (4,3,1,4) (5,4,1,5) (6,5,1,6) (7,6,1,7) (8,7,1,8) (9,8,1,9)
(10,9,1,10) (11,10,1,11) (12,11,1,12) (13,12,1,13) (14,13,1,14) (15,14,1,15) (16,15,1,16)
```

B.9 lin73

This problem has a linear tree with 73 nodes. All the containers can be loaded. The utility of a container is its area. The deck can be completely filled.

```
e=1
dx=90 dy=60
m=73
u1=222
u2=150
u3=28
u4=84
u5=72
u6=80
u7=40
u8=100
u9=125
u10=32
u11=96
u12=100
u13=50
u14=12
u15=28
u16=70
u17=20
u18=105
u19=72
u20=270
u21=54
u22=126
u23=35
u24=10
u25=44
u26=55
u27=20
u28=25
u29=3
u30=18
u31=4
u32=24
```

```
u33=8
u34=10
u35=475
u36=45
u37=36
u38=18
u39=18
u40=78
u41=260
u42=54
u43=54
u44=80
u45=64
u46=66
u47=52
u48=28
u49=91
u50=6
u51=8
u52=15
u53=20
u54=96
u55=180
u56=156
u57=133
u58=63
u59=20
u60=8
u61=60
u62=198
u63=5
u64=2
u65=180
u66=156
u67=10
u68=25
u69=204
u70=24
u71=60
u72=20
u73=140
lx1=6 ly1=37
lx2=10 ly2=15
lx3=4 ly3=7
lx4=12 ly4=7
lx5=4 ly5=18
lx6=10 ly6=8
lx7=5 ly7=8
lx8=4 ly8=25
lx9=5 ly9=25
lx10=4 ly10=8
lx11=12 ly11=8
lx12=10 ly12=10
lx13=5 ly13=10
lx14=3 ly14=4
lx15=7 ly15=4
lx16=7 ly16=10
lx17=2 ly17=10
lx18=7 ly18=15
lx19=4 ly19=18
lx20=15 ly20=18
lx21=3 ly21=18
lx22=7 ly22=18
lx23=7 ly23=5
lx24=2 ly24=5
lx25=4 ly25=11
lx26=5 ly26=11
lx27=4 ly27=5
lx28=5 ly28=5
lx29=1 ly29=3
lx30=6 ly30=3
lx31=1 ly31=4
lx32=6 ly32=4
lx33=4 ly33=2
lx34=5 ly34=2
lx35=19 ly35=25
lx36=5 ly36=9
lx37=4 ly37=9
lx38=3 ly38=6
lx39=3 ly39=6
lx40=6 ly40=13
lx41=20 ly41=13
lx42=3 ly42=18
```

lx43=3 ly43=18
lx44=5 ly44=16
lx45=4 ly45=16
lx46=6 ly46=11
lx47=13 ly47=4
lx48=7 ly48=4
lx49=13 ly49=7
lx50=3 ly50=2
lx51=4 ly51=2
lx52=3 ly52=5
lx53=4 ly53=5
lx54=4 ly54=24
lx55=15 ly55=12
lx56=13 ly56=12
lx57=19 ly57=7
lx58=9 ly58=7
lx59=5 ly59=4
lx60=2 ly60=4
lx61=12 ly61=5
lx62=9 ly62=22
lx63=5 ly63=1
lx64=2 ly64=1
lx65=15 ly65=12
lx66=13 ly66=12
lx67=2 ly67=5
lx68=5 ly68=5
lx69=12 ly69=17
lx70=2 ly70=12
lx71=5 ly71=12
lx72=4 ly72=5
lx73=28 ly73=5
n=73
(1,0,1,1) (2,1,1,2) (3,2,1,3) (4,3,1,4) (5,4,1,5) (6,5,1,6) (7,6,1,7) (8,7,1,8) (9,8,1,9) (10,9,1,10)
(11,10,1,11) (12,11,1,12) (13,12,1,13) (14,13,1,14) (15,14,1,15) (16,15,1,16) (17,16,1,17) (18,17,1,18) (19,18,1,19) (20,19,1,20)
(21,20,1,21) (22,21,1,22) (23,22,1,23) (24,23,1,24) (25,24,1,25) (26,25,1,26) (27,26,1,27) (28,27,1,28) (29,28,1,29) (30,29,1,30)
(31,30,1,31) (32,31,1,32) (33,32,1,33) (34,33,1,34) (35,34,1,35) (36,35,1,36) (37,36,1,37) (38,37,1,38) (39,38,1,39) (40,39,1,40)
(41,40,1,41) (42,41,1,42) (43,42,1,43) (44,43,1,44) (45,44,1,45) (46,45,1,46) (47,46,1,47) (48,47,1,48) (49,48,1,49) (50,49,1,50)
(51,50,1,51) (52,51,1,52) (53,52,1,53) (54,53,1,54) (55,54,1,55) (56,55,1,56) (57,56,1,57) (58,57,1,58) (59,58,1,59) (60,59,1,60)
(61,60,1,61) (62,61,1,62) (63,62,1,63) (64,63,1,64) (65,64,1,65) (66,65,1,66) (67,66,1,67) (68,67,1,68) (69,68,1,69) (70,69,1,70)

(71,70,1,71) (72,71,1,72) (73,72,1,73)

Bibliography

[1] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs.* Springer-Verlag, Berlin Heidelberg, 1992.

[2] M.R.Carey, D.S.Johnson. *Computers and Intractability.* W.H. Freeman and Company, 1979.

[3] G.Andreatta, F.Mason, G.Romanin Jacur. *Appunti di Ottimizzazione su Reti.* Edizioni Libreria Progetto, Padova, 1990.

[4] T.H.Cormen, C.E.Leiserson, R.L.Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[5] Thomas Bäck. *Evolutionary Algorithms in Theory and Practice.* Oxford University Press, New York, 1996.

[6] D.Corne,M.Dorigo, F.Glover. *New Ideas in Optimization.* Advanced Topics in Computer Science, McGraw Hill, London, 1999.

[7] Lawrence Davis. *Genetic Algorithms and Simulated Annealing.* Research Notes in Artificial Intelligence, London, 1987.

[8] K.Brown, D.W.Fowler. *Scheduling for an Uncertain Future with Branching Constraint Satisfaction Problems.* Computing Science Department, University of Aberdeen, Aberdeen (UK), December 23, 1999.

[9] B.M.Smith. *A Tutorial on Constraint Programming.* University of Leeds, Leeds, Report 95.14, April 1995.

[10] R.Barták. *Constraint Programming: in Pursuit of the Holy Grail.* Charles University, Faculty of Mathematics and Phisics, Depatment of Theoretical Computer Science, Praha.

[11] R.Barták. *On-line Guide to Constraint Programming.* Prague, 1998, http://kti.mff.cuni.cz/ bartak/constraints/.

[12] E.Hopper, B.C.H.Turton. *An Empirical Investigation of Meta Heuristic for a 2D Packing Problem.* European Journal of Operational Research, 2000.

[13] Martijn Dijksterhuis. *Vessel Loading Tool Constraint-based Vessel Loading.* Computing Science Department, University of Aberdeen, Aberdeen, 1997.

[14] John K. Ousterhout. *Tcl and Tk Tookit.* Addison-Wesley Professional Computing Series, Addison-Wesley, Reading, 1994.

[15] M.Harrison, M.McLennan. *Effective Tcl/Tk Programming: writing better programs with Tcl and Tk.* Addison-Wesley Professional Computing Series, Addison-Wesley, Reading, 1998.

[16] Adrian Zimmer. *Tcl/Tk for Programmers with solved exercises that work with Unix and Windows* . IEEE Computer Society, Los Alamitos, 1998.