POLITECNICO DI TORINO

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea

Genetic Local Search for Job Shop Scheduling Problem

Algoritmi Genetici e Ricerca Locale per
il Problema Job-Shop

Relatore:
prof. Roberto Tadei

Candidato:
Alberto Moraglio

OTTOBRE 2000

Abstract

Job Shop Scheduling Problem is a strongly NP-hard problem of combinatorial optimisation and one of the most well-known machine scheduling problem. Taboo Search is an effective local search algorithm for the job shop scheduling problem, but the quality of the best solution found depends on the initial solution used. To overcome this problem in this thesis we present a new approach that uses a population of Taboo Search runs in a Genetic Algorithm framework: GAs localise good areas of the solution space so that TS can start its search with promising initial solutions. The peculiarity of the Genetic Algorithm we propose consists in a natural representation, which covers all and only the feasible solution space and guarantees the transmission of meaningful characteristics. The results show that this method outperforms many others producing good quality solutions in less time.

Acknowledgments

The most part of this thesis has been done at the Technical University of Eindhoven in the department of computer science and mathematics, Eindhoven, The Netherlands.
In particular I would like to extend my thanks to Huub ten Eikelder who has supported me in this work, advising and helping me constantly with a great deal of paternal patience. I'm also grateful to Robin Schilham for being more than a colleague; always open to fulfil any doubt I had behind a cup of coffee.
I like to mention here my "virtual" officemate, Emile Aarts, for being so quiet all the time, and, above all, Jan Karel Lenstra who unwillingly has lent me a precious book "until the end of time", which unfortunately is still out of stock.

I wish to express my gratitude to my advisor at the Politecnico di Torino, Roberto Tadei, who has given to me the great opportunity to present this work in the beautiful city of Naples and to attend to the prestigious winter institute in Lac Noir.

With attention to my stay in The Netherlands, I retain fond memories of everyone I have met. I really can't mention all the friendly people, from The Netherlands and from everywhere else in Europe, that have contributed together to make my stay in Eindhoven such a great and unforgettable experience.

Finally, but above all, I wish to thank my dear parents and my whole family for their moral and economic support during my study.

Table of Contents

Part I – Introduzione *1*

Introduzione **2**

1 Sommario ... 3

2 Un Algoritmo Genetico per il Job Shop ... 5

2.1 Appello all'ammissibilità ... 5

2.2 Rappresentazione ... 7

2.3 Ricombinazione ... 10

3 Ricerca Genetica Locale ... 12

3.1 Schema dell'Algoritmo di Ricerca Genetica Locale ... 12

3.2 Infrastruttura Genetica ... 13

3.3 Algoritmo di Ricerca Loacle ... 14

4 Risultati Computazionali ... 15

4.1 Panoramica sui Parametri ... 15

4.2 GTS Vs TS ... 17

4.3 Un Confronto ad Ampio Spettro ... 18

4.4 GTS Vs SAGen ... 18

5 Conclusioni ... 19

Part II – Project Description *21*

1 Introduction **22**

1.1 Job Shop Scheduling Problem ... 22

1.2 Local Search and Taboo Search ... 22
1.3 Genetic Algorithms and Genetic Local Search ... 23
1.4 Applying Genetic Algorithms to Job Shop ... 24
1.5 Scope ... 24
1.6 Results ... 25
1.7 Reader's Guide ... 25

2 Job Shop Scheduling Problem **26**

2.1 Introduction ... 26
2.2 Complexity and Formal Definition ... 26
2.3 Disjunctive Graph ... 27
2.4 Schedules and Gantt Charts ... 30
2.5 Conventional Heuristics ... 33
 2.5.1 Dispatching Heuristics ... 33
 2.5.2 Shifting-Bottleneck Heuristics ... 35

3 Local Search and Taboo Search **37**

3.1 Introduction ... 37
3.2 Overview on Local Search Algorithms ... 37
 3.2.1 Iterative Improvement ... 38
 3.2.2 Threshold Algorithms ... 38
 3.2.3 Taboo Search ... 39
 3.2.4 Variable-Depth Search ... 39
 3.2.5 Genetic Algorithms ... 39
3.3 Focus on Taboo Search ... 40
3.4 Local Search for the Job Shop Problem ... 41

4 Genetic Algorithms and Evolutionary Computation **43**

4.1 Introduction ... 43
4.2 What are Genetic Algorithms? ... 44
4.3 Major Elements of Genetic Algorithms ... 45
 4.3.1 Initialisation ... 45

4.3.2 Selection 46
4.3.3 Recombination 48
4.3.4 Mutation 50
4.3.5 Reinsertion 50
4.3.6 Termination 51
4.4 Overview on Evolutionary Algorithms 51
4.4.1 Evolutionary Programming 52
4.4.2 Evolution Strategy 52
4.4.3 Classic Genetic Algorithms 52
4.4.4 Genetic Programming 52
4.4.5 Order-Based Genetic Algorithms 53
4.4.6 Parallel Genetic Algorithms 53
4.5 Genetic Algorithms as Meta-Heuristics 54
4.6 Encoding Problem 54
4.6.1 Coding Space and Solution Space 55
4.6.2 Feasibility and Legality 55
4.6.3 Mapping 56
4.7 Genetic Local Search 56
4.8 Genetic Algorithms and Scheduling Problems 57
4.9 Theory Behind Genetic Algorithms 59
4.9.1 Schema Theorem 60
4.9.2 Building Block Hypothesis 60

5 Genetic Algorithms and Job Shop 62

5.1 Introduction 62
5.2 Encoding Problems and Job Shop 62
5.2.1 Appeal to Feasibility 62
5.2.2 Causes of Infeasibility 63
5.2.3 Encoding and Constraints 64
5.2.4 Genetic Operators and Order Relationship 65
5.3 A New Genetic Representation for Job Shop 67
5.3.1 String Representation 67

5.3.2 Formal Definition of String and Coding/Decoding Theorems 70
5.3.3 Recombination Class 72
5.4 Recombination Operators 76
5.4.1 Multi-Step Complete Recombination Operator 78
5.4.2 Multi-Step Parallel Recombination Operator 79
5.4.3 Multi-Step Crossed Recombination Operator... 79
5.4.4 Multi-Step Merge and Split Operator.. 80
5.4.5 One-Step Crossed Insertion Recombination Operator 83
5.4.6 One-Step Insertion Recombination Operator... 83
5.5 The Genetic Local Search Algorithm 84
5.5.1 Genetic Local Search Template 84
5.5.2 Genetic Algorithm Framework 84
5.5.3 Local Optimiser.. 85

6 Computational Results 87

6.1 Introduction.. 87
6.2 Parameter Setting 88
6.2.1 Overview on Parameters.. 88
6.2.2 Determination of a Plausible Parameter Configuration................................ 90
6.3 Result of GTS 98
6.3.1 Quality and Time Experiment.. 99
6.3.2 Quality Experiment 100
6.4 Comparisons 101
6.4.1 GTS vs TS.. 101
6.4.2 A Wide Comparison 104
6.4.3 A Specific Comparison... 105
6.5 Summary.. 107

7 Conclusions 108

References 110

Part I
Introduzione

Introduzione

Il *Job Shop Scheduling Problem* è un problema di ottimizzazione combinatoria fortemente NP-hard ed è il più conosciuto e studiato dei problemi di schedulazione in ambiente multi-macchina. Il *Taboo Search* è un algoritmo di ricerca locale molto efficace quando applicato al Job Shop, tuttavia la qualità della soluzione migliore che questo metodo è in grado di trovare dipende molto dalla soluzione iniziale necessaria ad avviare il processo di ricerca. Al fine di porre rimedio a questa limitazione, nel presente documento si propone un approccio innovativo al problema che fa uso di una popolazione di soluzioni, ottimizzate dall'algoritmo di Taboo Search al momento della loro creazione, all'interno di un contesto algoritmico d'ispirazione genetica: l'*Algoritmo Genetico* localizza buone aree all'interno dello spazio delle soluzioni, spianando la strada alla procedura di Taboo Search che, in tal modo, può iniziare la ricerca utilizzando soluzioni iniziali promettenti. La particolarità dell'algoritmo genetico proposto, consiste in una rappresentazione naturale che copre completamente e solamente lo spazio delle soluzioni ammissibili e che garantisce la trasmissione di caratteristiche significative di soluzione in soluzione. Gli esperimenti computazionali mostrano che questo metodo risulta migliore di molti altri, trovando soluzioni di buona qualità in minor tempo.

1 Sommario

Il Job Shop Scheduling Problem (JSSP) è un problema di ottimizzazione combinatoria particolarmente arduo da risolvere. Poiché tale problema ha svariate applicazioni pratiche, è stato studiato da molti autori che hanno proposto un gran numero di algoritmi risolutivi. Solo casi molto particolari sono risolvibili attraverso algoritmi polinomiali, ma in generale il problema ricade nella classe di complessità NP-hard [9].

Il JSSP può essere sommariamente descritto come di seguito: siano dati un insieme di lavori ed un insieme di macchine. Ogni macchina può lavorare al massimo un lavoro alla volta. Ogni lavoro consiste in una sequenza di operazioni, ognuna delle quali deve essere lavorata su una data macchina senza interruzioni per un determinato periodo di tempo. Lo scopo è quello di trovare una schedulazione, cioè un ordinamento nel tempo delle operazioni su tutte le macchine, che abbia lunghezza minima.

Il Taboo Search (TS) è un algoritmo di ricerca locale progettato per trovare soluzioni quasi ottimali di problemi di ottimizzazione combinatoria [6]. La particolarità del TS è una memoria a breve termine utilizzata per tenere traccia delle soluzioni visitate più di recente le quali vengono considerate proibite (tabù per l'appunto), permettendo così al processo di ricerca di sfuggire dagli ottimi locali.

Il TS si è rivelato un algoritmo di ricerca locale molto efficace nell'affrontare il Job Shop Scheduling Problem [13, 16]. Tuttavia la soluzione migliore che il TS può trovare è influenzata da quella utilizzata per avviare il processo di ricerca. Come avviene per molte tecniche di ricerca locale, anche il TS inizia la sua ricerca partendo da una singola soluzione, la quale può instradare la ricerca verso un vicolo cieco. La presenza del meccanismo di tabù attenua tale possibilità, senza però eliminarla. Questo accade soprattutto quando il TS è applicato a problemi di ottimizzazione combinatoria particolarmente ostici quale il JSSP.

Gli Algoritmi Genetici (AG) sono metodi di ricerca stocastica globale che mimano l'evoluzione biologica naturale [7]. Gli AG operano su una popolazione di soluzioni potenziali applicando il principio di "sopravvivenza del migliore", producendo via via sempre migliori approssimazioni della soluzione ottima.

E' molto difficile applicare direttamente e con successo gli AG classici a problemi di ottimizzazione combinatoria. Per questo motivo, sono state create diverse

implementazioni non convenzionali che utilizzano gli algoritmi genetici come meta-euristica [3, 10]. Percorrendo questa nuova strada si è scoperto che, per un verso, gli algoritmi genetici sono uno strumento molto efficace di ricerca stocastica globale e che, per l'altro, permettono di essere combinati in maniera flessibile con euristiche legate al particolare dominio dello specifico problema rendendo molto più efficienti gli algoritmi risolutivi così concepiti.

Molti autori hanno proposto varianti di algoritmi di ricerca locale che fanno uso di idee provenienti dalla genetica delle popolazioni [1, 11, 19]. In generale, per via della complementarietà delle proprietà di ricerca degli algoritmi genetici e delle tecniche più convenzionali, gli algoritmi ibridi risultano spesso superare entrambi gli algoritmi che li compongono quando presi singolarmente.

Precedenti studi riguardanti problemi di ordinamento, come il problema del commesso viaggiatore (TSP), hanno provato che una rappresentazione naturale del problema è la chiave del successo dell'approccio genetico [17].

Il JSSP è sia un problema di ordinamento sia un problema fortemente vincolato. Entrambi questi aspetti, dunque, devono essere considerati al momento di individuare una rappresentazione naturale per tale problema in ambito genetico. Rappresentazioni genetiche per il JSSP possono essere trovate in [4, 8, 12].

Nella sezione 2 di questa introduzione, è stata proposta una rappresentazione che, appoggiata ad una particolare classe di operatori genetici, garantisce alla ricerca genetica di coprire completamente e solamente lo spazio delle soluzioni ammissibili e che non di meno garantisce la trasmissione di caratteristiche significative dalle soluzioni genitrici alle soluzioni figlie. Seguono poi, la definizione della classe degli operatori genetici accoppiata con la rappresentazione proposta e la scelta di un singolo operatore di ricombinazione che presenta interessanti proprietà.

Nella sezione 3 di questa introduzione, è presentato un algoritmo di ricerca genetica locale (GTS) che consiste in una particolare forma ibrida di algoritmo costituita da un algoritmo genetico di base con l'aggiunta che ogni nuovo individuo generato è sottoposto ad una fase di ottimizzazione attraverso l'algoritmo di Taboo Search prima di essere passato nuovamente al controllo dell'algoritmo genetico.

Nella sezione 4 di questa introduzione, gli esperimenti computazionali condotti mostrano che la combinazione dell'AG con il TS si comporta meglio del TS preso

singolarmente. Inoltre, in una comparazione tra GTS e un'ampia gamma di algoritmi risolutivi per il JSSP, effettuata su un parco di istanze molto note di piccola e media taglia, il GTS risulta molto ben posizionato. Infine, da una comparazione diretta tra GTS e un approccio simile, che però combina gli algoritmi genetici con Simulated Annealing (SAGen) [8], effettuata su un insieme di istanze grandi del problema, si evince che GTS sorpassa SAGen sia in termini di tempo computazionale impiegato sia in termini di qualità di soluzioni ottenute.

2 Un Algoritmo Genetico per il Job Shop

2.1 Appello all'ammissibilità

Al fine di applicare gli algoritmi genetici ad un problema particolare è necessario codificare una generica soluzione del problema in un cromosoma. La modalità di codifica di una soluzione risulta essere la chiave del successo dell'AG [3]. Gli AG classici fanno uso di una codifica binaria degli individui in stringhe di lunghezza fissa. Tale rappresentazione non si conforma in maniera naturale a problemi in cui l'ordine risulti essere un aspetto caratterizzante dello stesso, quali sono i casi del problema del commesso viaggiatore e del Job Shop, in quanto non sono state trovate modalità di codifica dirette ed efficienti che siano in grado di far corrispondere l'insieme comprendente tutte le soluzioni possibili a stringhe binarie [17].

La principale difficoltà nella scelta di una opportuna rappresentazione per problemi di ottimizzazione combinatoria fortemente vincolati, quale è il JSSP, è quella di far fronte all'insorgere dell'inammissibilità delle soluzioni prodotte durante il processo evolutivo. Questo problema è solitamente affrontato modificando gli operatori di ricombinazione, associando loro meccanismi di riparazione delle soluzioni o imputando penalità alle soluzioni inammissibili agendo sulla loro funzione di fitness o, ancora, scartando le soluzioni inammissibili quando create. L'utilizzo delle penalità o della strategia del rifiuto risultano inefficienti se applicate al JSSP perché lo spazio delle soluzioni ammissibili è molto più piccolo di quello delle soluzioni possibili, e quindi l'AG perde la maggior parte del suo tempo producendo e/o elaborando soluzioni inammissibili. Le tecniche di riparazione risultano essere una scelta migliore per molti problemi di ottimizzazione combinatoria perché sono facili da applicare e sorpassano in termini di

efficienza le strategie basate sul rifiuto e sulle penalità [14]. In ogni caso, laddove sia possibile, il metodo di gran lunga più efficiente e diretto rimane quello di incorporare in maniera trasparente i vincoli nella codifica degli individui. Così, un punto molto importante nella costruzione di un algoritmo genetico per il JSSP è quello di concepire insieme una rappresentazione opportuna delle soluzioni e degli operatori genetici specifici per il problema, in maniera tale che tutti i cromosomi, sia quelli generati durante la fase iniziale di formazione della popolazione sia quelli prodotti durante il processo evolutivo, producano soluzioni ammissibili. Questa è una fase cruciale che influenza in maniera decisiva tutti i passi successivi della costruzione di un algoritmo genetico.

In questo documento sono proposti una rappresentazione e una particolare classe di operatori di ricombinazione che, presi insieme, garantiscono che la ricerca genetica copra completamente e solamente lo spazio delle soluzioni ammissibili nonché la trasmissione delle caratteristiche significative alle soluzioni figlie.

Sulla base del tipo di rappresentazione delle soluzioni utilizzata, si possono verificare due distinte cause di inammissibilità:

1. soluzioni che non rispettano il preimposto ordine di precedenza nei job
2. soluzioni che presentano cicli

La prima causa di inammissibilità è legata all'esistenza dei vincoli di precedenza tra operazioni sui job. Se si considera una rappresentazione delle soluzioni che non imponga a priori un ordine fisso delle operazioni sui job ma piuttosto che permetta di disporre liberamente le operazioni sia sulle macchine sia sui job, allora si possono verificare dei conflitti tra l'ordine delle operazioni codificato in un generico cromosoma e quello prescritto dai job.

La seconda causa di inammissibilità s'incontra se si considera una rappresentazione delle soluzioni che ammetta di codificare anomalie di precedenza tra operazioni (cioè che ammetta soluzioni cicliche). In una soluzione ammissibile, infatti, due generiche operazioni possono essere esclusivamente parallele (non c'è alcun ordine imposto tra loro) o sequenziali (in questo caso, una precede l'altra). Non è possibile che una

operazione, in maniera congiunta, preceda e segua un'altra, direttamente o indirettamente (così formando un ciclo).

Al fine di evitare nel nostro algoritmo genetico entrambi i tipi di inammissibilità, saranno introdotti 1) una classe di operatori di ricombinazione che evitino il problema dei vincoli di precedenza sui job e 2) ed una rappresentazione che non risenta del problema riguardante le soluzioni cicliche. Più in dettaglio si vedrà che potranno essere rappresentate solo le soluzioni senza cicli. Le soluzioni così rappresentate però non rispettano necessariamente i vincoli di precedenza sui job. Al fine di gestire quest'altro tipo di inammissibilità, è sufficiente inizializzare il processo evolutivo con una popolazione di soluzioni che rispettino tutti i vincoli di precedenza sui job e, nel corso del processo evolutivo, applicare solo operatori di ricombinazione che lascino invariati i vincoli di precedenza.

2.2 Rappresentazione

Al fine di applicare la cornice genetica ad un determinato problema, è necessario definire un metodo di codifica per associare lo spazio di tutte le possibili soluzioni ad un insieme finito di cromosomi.

Di seguito sarà introdotta la rappresentazione che sarà utilizzata. Dapprima si mostrerà tramite un esempio, la relazione che sussiste tra 1) una istanza del problema, rappresentata dal suo grafo disgiuntivo, 2) una particolare soluzione per quell'istanza, rappresentata dal rispettivo grafo, e 3) la nostra codifica a stringa per quella soluzione. Dopodiché, saranno presentate le definizioni e i teoremi che assicurano la validità della rappresentazione proposta.

Problema, Soluzione e Codifica

Il problema del job shop può essere efficacemente rappresentato attraverso un grafo disgiuntivo [15]. Un grafo disgiuntivo $G = (N, A, E)$ è definito come di seguito: sia N l'insieme dei nodi che rappresentano le operazioni, sia A l'insieme degli archi che connettono operazioni consecutive su uno stesso job, e sia E l'insieme degli archi disgiuntivi che connettono le operazioni che devono essere lavorate su una stessa macchina. Un arco disgiuntivo può essere impostato nell'una o nell'altra orientazione delle due possibili. La costruzione di una soluzione consiste nell'orientare tutti gli archi

disgiuntivi in maniera tale da determinare una sequenza di operazioni su una stessa macchina. Una volta che è stata determinata una sequenza per una macchina, gli archi disgiuntivi che connettono operazioni che devono essere lavorate da quella macchina devono essere rimpiazzati con degli archi orientati, vale a dire archi congiuntivi. L'insieme E degli archi disgiuntivi può essere decomposto in clique (sottografi completi), uno per ogni macchina. Il tempo di elaborazione di ogni operazione può essere visto come un peso attaccato al nodo corrispondente. L'obiettivo del JSSP è quello di trovare un ordinamento delle operazioni su ogni macchina, cioè di orientare gli archi disgiuntivi in maniera tale che il grafo relativo alla soluzione che ne risulta sia aciclico (non ci sono conflitti di precedenza tra operazioni) e che la lunghezza del cammino pesato più lungo tra il nodo iniziale e quello terminale sia minimo. Questa lunghezza prende il nome di *makespan*.

La Figura [2.1] riporta il grafo disgiuntivo relativo ad una istanza del problema composta da 3 job e 4 macchine. La Figura [2.2] riporta il grafo che rappresenta una soluzione ammissibile per l'istanza data del problema. Questo è stato derivato dal grafo disgiuntivo sopra descritto dopo aver orientato tutti gli archi disgiuntivi e avendo avuto cura di non creare cicli. Nel grafo della soluzione, le frecce corrispondono ai vincoli di precedenza sui job o sulle macchine. Le linee tratteggiate indicano che due operazioni non presentano alcun ordine di precedenza (in linea di principio tali operazioni potrebbero essere lavorate in parallelo senza violare alcun vincolo di precedenza. In realtà il fatto che vengano effettivamente lavorate in parallelo dipende solo dal tempo di elaborazione delle operazioni). La sequenza delle operazioni sui job è unicamente funzione dell'istanza del problema e non della particolare soluzione. Al contrario, la sequenza delle operazioni sulle macchine dipende anche dalla particolare soluzione del problema dato.

Si considerino ora le cose da un punto di vista differente, enfatizzando la relazione d'ordine tra le operazioni. Il grafo disgiuntivo di Figura [2.1] rappresenta una particolare istanza del JSSP. Questo può essere interpretato come una relazione d'ordine parziale tra operazioni. Il grafo della soluzione mostrato in Figura [2.2] rappresenta una specifica soluzione dell'istanza data sopra. Anche quest'ultimo può essere interpretato come una relazione d'ordine parziale tra operazioni, anche se più vincolata quando confrontata con quella associata al grafo disgiuntivo. Ora si può immaginare di ottenere

una relazione d'ordine totale tra operazioni imponendo ulteriori vincoli di precedenza fino ad ottenere un'unica sequenza lineare di operazioni. Così operando otteniamo la stringa (la sequenza di operazioni) riportata al fondo di Figura [5.3], che altro non è che la codifica di una soluzione che si andrà ad usare.

Nella codifica a stringa è presente tutta l'informazione necessaria per ricostruire una soluzione vera e propria. Poiché si conosce a priori (dall'istanza data del problema) la macchina relativa ad ogni operazione, la sequenza delle operazioni di ogni macchina è facilmente estraibile dalla stringa. L'idea è quella di scandire la stringa da sinistra a destra, estraendo, per l'appunto, tutte le operazioni di una data macchina e di sequenziarle su di essa mantenendo lo stesso ordine. L'applicazione della procedura di decodifica appena descritta alla stringa di Figura [5.3] porta ad ottenere esattamente le stesse sequenze di operazioni sulle macchine che sono state estratte dal grafo della soluzione.

Una particolarità molto importante della rappresentazione a stringa è che essa non ammette di rappresentare soluzioni cicliche, quindi non è soggetta al secondo tipo di inammissibilità discusso alla sezione 2.1 di questa introduzione. Tuttavia è facilmente verificabile che una stringa codifichi sia le informazioni relative alla specifica soluzione che rappresenta (i vincoli di precedenza sulle macchine) sia le informazioni riguardo all'istanza del problema (i vincoli di precedenza sui job). Questo implica che una generica stringa può rappresentare una soluzione che non rispetti i vincoli di precedenza sui job, quindi è necessario far fronte opportunamente a questo tipo di inammissibilità, ovvero il primo tipo di cui si è discusso alla sezione 2.1 di questa introduzione.

Codifica/Decodifica delle Soluzioni

Nella Sezione [5.3.2] è riportata la parte di teoria che sta dietro la rappresentazione a stringa. Di seguito sono enunciati brevemente le definizioni ed i teoremi più importanti.

Definizione di Stringa Legale (Definizione 1)

Una stringa è legale se e solo se l'ordine di precedenza di ogni coppia di operazioni non risulta in conflitto con l'ordine di precedenza dato sui job.

Teorema di Codifica (Soluzione Ammissibile → Stringa Legale) (Teorema 1)
Ogni soluzione ammissibile può essere rappresentata attraverso una stringa legale. Può esistere più di una stringa legale corrispondente alla stessa soluzione ammissibile.

Teorema di Decodifica (Stringa Legale → Soluzione Ammissibile) (Teorema 2)
Ogni stringa legale corrisponde esattamente ad una e una sola soluzione ammissibile.

2.3 Ricombinazione

La parte di teoria relativa alla ricombinazione si trova nella Sezione [5.3.3]. Di seguito sono accennati i passi fondamentali.

Requisito di Ammissibilità per la Ricombinazione (Definizione 2)
Un generico operatore di ricombinazione per la rappresentazione a stringa è ammissibile, quando risulta che per ogni coppia di operazioni che in entrambe le stringhe genitrici abbiano lo stesso ordine di precedenza, tale coppia mantiene lo stesso ordine di precedenza anche nelle stringhe figlie ottenute tramite l'applicazione dell'operatore alle stringhe genitrici.

Teorema di Trasmissione della Legalità (Stringa Legale + Stringa Legale → Stringa Legale) (Teorema 3)
Ricombinando stringhe legali il requisito di ammissibilità per la ricombinazione, si ottengono ancora stringhe legali.

Definizione dell'Operatore Generale di Ricombinazione (Definizione 3)

Teorema di Validità dell'Operatore Generale di Ricombinazione (Teorema 4)
L'operatore generale di ricombinazione rispetta il requisito di ammissibilità.

L'operatore di ricombinazione proposto è molto generale. Esso presenta quattro gradi di libertà (i quattro puntatori) che possono essere guidati a piacere. Possono essere combinati in molte differenti configurazioni così da ottenere operatori di ricombinazione con caratteristiche molto differenti. Per esempio, è possibile pensare di

inibire due qualsiasi dei quattro puntatori lasciando libertà di movimento solo ai rimanenti due. Si può anche pensare di agire sulla sequenza casuale che guida i puntatori al fine di ottenere ricombinazioni più vicine al crossover uniforme piuttosto che a quello ad un taglio o viceversa, controllando così la capacità di distruzione della ricombinazione [7]. Si può ancora pensare di combinare due operatori di ricombinazione, che presentano caratteristiche interessanti ma complementari, durante il processo evolutivo, applicando una volta l'uno ed una volta l'altro, al fine di ottenere un effetto sinergico.

In effetti sono stati studiati e confrontati in pratica un insieme di operatori di ricombinazione selezionati seguendo per l'appunto le linee guida sopra menzionate. Questi sono descritti estensivamente in Sezione [5.4].

MSX è quello che si è rivelato essere il più efficace nel corso degli esperimenti computazionali. Val la pena menzionare che differenti operatori di ricombinazione possono influenzare notevolmente le prestazioni dell'algoritmo genetico, in special modo quando questo non sia appaiato con la ricerca locale. Nel nostro algoritmo genetico è stato utilizzato l'operatore di ricombinazione MSX. La Figura [5.10] illustra attraverso un esempio come MSX funzioni in pratica, la sua definizione dettagliata è riportata nella Sezione [5.4.4].

La caratteristica principale dell'operatore MSX è di produrre due stringhe figlie complementari combinando le caratteristiche di precedenza delle stringhe genitrici e contemporaneamente provando a minimizzare la perdita di diversità genetica nell'accoppiamento delle stesse. Più precisamente, data una generica coppia di operazioni avente ordine di precedenza differente nelle due stringhe genitrici, MSX tende per quanto possibile a trasmettere tale diversità alle stringhe figlie cosicché anche in queste ultime l'ordine di precedenza di quella coppia di operazioni risulti differente. E' importante notare che in generale tale requisito può risultare in contrasto con quello riguardante la non ciclicità delle soluzioni. Comunque, poiché la rappresentazione a stringa non ammette la codifica di soluzioni cicliche, risulta spesso impossibile ottenere una perfetta conservazione della diversità delle caratteristiche delle stringhe genitrici sottoposte a ricombinazione.

La preservazione della diversità attraverso il processo di ricombinazione è spiegabile intuitivamente notando che nella fase di unione le caratteristiche di precedenza delle

stringhe genitrici sono mischiate ma non distrutte. Poi, nella fase di divisione, le caratteristiche sono ripartite in due stringhe figlie e ancora non distrutte, così preservando le caratteristiche originarie ma combinate in modo differente.
Un dubbio pertinente che si può avere è se MSX rispetti o no il requisito di ammissibilità per ricombinazioni. Dopo tutto tale requisito è stato solo dimostrato per l'operatore generale di ricombinazione che di primo acchito non sembra imparentato con l'operatore MSX per via della sua peculiare procedura di ricombinazione suddivisa in due fasi. Comunque è possibile immaginare una definizione alternativa dell'operatore MSX che lo riporta ad essere assimilato alla classe degli operatori ammissibili. L'idea è di produrre le due stringhe figlie in sede separata, ognuna delle quali in una fase solamente, utilizzando però la stessa sequenza casuale due volte, una volta scandendo la sequenza casuale e le stringhe genitrici da sinistra a destra producendo la prima stringa figlia, una volta scandendole nel senso contrario ottenendo la seconda stringa figlia.

3 Ricerca Genetica Locale

3.1 Schema dell'Algoritmo di Ricerca Genetica Locale

Da un lato, i problemi di ottimizzazione combinatoria rientrano all'interno del campo d'azione degli algoritmi genetici. Gli algoritmi genetici, quando però confrontati con altre euristiche, non sembrano essere molto adatti a migliorare soluzioni che sono già di per se stesse molto vicine all'ottimo. Risulta quindi essenziale incorporare all'interno degli algoritmi genetici euristiche convenzionali, che utilizzano più da vicino la conoscenza specifica del problema affrontato, al fine di costruire un algoritmo più competitivo.
D'altra parte, in generale, la migliore soluzione che un algoritmo di ricerca locale è in grado di trovare dipende dalla soluzione iniziale utilizzata. Uno schema a partenze-multiple può risolvere questo problema. Come ulteriore raffinamento, l'efficacia dell'approccio iterativo a partenze-multiple può essere migliorato utilizzando l'informazione presente nelle soluzioni già ottenute per guidare la ricerca nelle iterazioni successive. Seguendo questa linea di pensiero, molti autori hanno proposto varianti di algoritmi di ricerca locale ispirandosi ad idee proprie della genetica delle popolazioni.

Un algoritmo di ricerca genetica locale [1] consiste in un algoritmo genetico di base con l'aggiunta di una fase di ottimizzazione eseguita da un algoritmo di ricerca locale applicata ad ogni nuovo individuo generato o nella fase iniziale di popolamento dell'algoritmo genetico o durante il processo evolutivo.

Un algoritmo di ricerca genetica locale è soggetto ad una interpretazione duale. Da un lato, può essere visto come un algoritmo genetico in cui la ricerca locale giochi il ruolo di un meccanismo di mutazione intelligente. D'altra parte, lo stesso algoritmo può essere inteso come un meccanismo strutturato a partenze-multiple per la ricerca locale in cui l'algoritmo genetico rivesta il ruolo della struttura portante.

Comunque sia, sforzandosi di avere una visione unitaria di questo approccio ibrido, si può dire che gli algoritmi genetici svolgono un'esplorazione globale all'interno della popolazione, mentre alla ricerca locale è affidato il compito di raffinare il più possibile i singoli cromosomi. Grazie alle proprietà di ricerca complementari degli algoritmi genetici e della ricerca locale, che vicendevolmente compensano l'uno le debolezze dell'altro, l'approccio ibrido supera spesso l'uno e l'altro quando applicati singolarmente. Lo schema dell'algoritmo di ricerca genetica locale si trova alla Sezione [5.5.1].

Si riempie ora lo schema della ricerca genetica locale con tutte le componenti necessarie per implementare un algoritmo vero e proprio per il problema del Job Shop. Prima si discuterà delle componenti principali dell'infrastruttura genetica dell'algoritmo, poi si focalizzerà l'attenzione sullo specifico algoritmo di ricerca locale utilizzato.

3.2 Infrastruttura Genetica

- POPOLAZIONE. La popolazione iniziale contiene un numero fisso di cromosomi che sono generati casualmente. Durante tutto il processo evolutivo la dimensione della popolazione rimane invariata.
- FUNZIONE DI FITNESS. Ogni cromosoma facente parte della popolazione riceve un valore di fitness. Questo valore influenza la probabilità del cromosoma di riprodursi. Nel nostro algoritmo il valore di fitness di un cromosoma corrisponde al makespan della soluzione in esso codificata.
- MODALITA' DI SELEZIONE. Vengono scelti un numero fisso di cromosomi che saranno sottoposti a ricombinazione. La selezione è fatta attraverso un semplice

meccanismo di classificazione. La popolazione è costantemente mantenuta ordinata secondo il valore di fitness (classifica). La probabilità di ogni cromosoma di essere selezionato dipende solo dalla sua posizione in classifica e non direttamente dal valore della sua fitness.

- MODALITA' DI REINSERIMENTO. L'insieme delle nuove soluzioni, create nella fase di riproduzione, è unito alla popolazione corrente. Successivamente la popolazione è riportata alla sua dimensione originaria eliminando i peggiori cromosomi presenti nella popolazione estesa.
- CRITERIO DI STOP. L'algoritmo termina dopo un numero prefissato di generazioni consecutive senza che si sia verificato alcun miglioramento della soluzione migliore della popolazione.
- RAPPRESENTAZIONE E RICOMBINAZIONE. Si utilizzano la rappresentazione a stringa e l'operatore di ricombinazione MSX presentati nella sezione 2. Si focalizzi l'attenzione ruolo rivestito da MSX nell'ambito dello schema GLS. Mentre MSX tende a preservare la diversità il più possibile, allo stesso tempo esso prova a mischiare molto le caratteristiche delle stringhe genitrici. La sequenza casuale in ingresso è libera di saltare da un genitore all'altro in ogni singolo passo, quindi si comporta come un crossover uniforme. Questi due aspetti della presente ricombinazione presi insieme risultano essere particolarmente utili nel contesto della ricerca genetica locale. Da una parte, MSX trasmette la diversità e quindi non spreca costose informazioni, in termini di tempo di computazione, presenti nelle stringhe genitrici ottenute attraverso la procedura di ricerca locale. D'altra parte, il ruolo richiesto all'algoritmo genetico quando accoppiato con la ricerca locale è quello di esplorare il più possibile lo spazio delle soluzioni. MSX adempie a questa esigenza mischiando il più possibile le informazioni presenti nelle stringhe genitrici comportandosi come un crossover uniforme.

3.3 Algoritmo di Ricerca Loacle

Qui di seguito è proposto un algoritmo di ricerca locale per il problema del Job Shop basato sul taboo search [18]. Questo è utilizzato nell'algoritmo di ricerca genetica locale nelle fasi di ottimizzazione ai punti 2 e 6. L'algoritmo di Taboo search è presentato alla Sezione [5.5.3]. L'algoritmo di taboo search utilizzato è basato su un algoritmo

proposto da Eikelder et al. Nel seguito sono riportati le principali componenti dell'algoritmo.

- RAPPRESENTAZIONE. Per applicare la ricerca locale al problema del Job Shop si è utilizzata la rappresentazione basata su grafo disgiuntivo. Una soluzione ammissibile si ottiene orientando gli spigoli in maniera tale da ottenere un ordine lineare delle operazioni su ogni macchina, e avendo il grafo della soluzione aciclico.
- VICINATO. E' utilizzata la struttura di vicinato proposta da Nowicki & Smutnicki [13]. Questa è essenzialmente basata sull'inversione dell'orientamento degli archi relativi alle macchine sul cammino più lungo. E' stato dimostrato che molti tipi di vicini possono essere omessi poiché non portano a soluzioni di costo minore. Per esempio non risulta utile invertire archi interni a blocchi di operazioni appartenenti al cammino più lungo.
- STRATEGIA DI RICERCA. Il tempo necessario per visitare un vicinato dipende dalla dimensione del vicinato stesso e dal costo computazionale in termini di tempo per accedere ai vicini. Poiché la dimensione di un vicinato nel nostro caso è piuttosto piccola si utilizzerà la strategia della salita più ripida che sebbene richieda la generazione e valutazione di ogni vicino, ne seleziona sempre il migliore.
- TABOO LIST. La nostra taboo list consiste in una coda FIFO di mosse di lunghezza fissa. La lunghezza della taboo list è data dalla media della dimensione del vicinato più un valore casuale.
- CRITERIO DI STOP. L'algoritmo termina dopo un prefissato numero di passi senza miglioramenti.

Poiché il nostro algoritmo fa uso combinato di algoritmi genetici e taboo search lo chiameremo GTS, acronimo di Genetic Taboo Search.

4 Risultati Computazionali

4.1 Panoramica sui Parametri

Di seguito, sono individuati e discussi i parametri più importanti del GTS, quelli che influenzano maggiormente le prestazioni dell'algoritmo, e la loro impostazione.

IMPEGNO COMPUTAZIONALE

Questo parametro permette un controllo qualitativo dell'impegno computazionale impiegato nella ricerca. Più in dettaglio, l'impegno computazionale è definito come il prodotto di due fattori; il primo è il numero di iterazioni consecutive senza miglioramenti (TS) dopo le quali ogni sessione di Taboo Search deve terminare; il secondo fattore è il numero di individui consecutivamente processati dall'algoritmo genetico senza miglioramenti (GA) dopo i quali GTS deve terminare. Poiché entrambi i criteri di stop sono adattivi alla complessità dello specifico problema trattato, la stessa impostazione del parametro di impegno computazionale può produrre risultati differenti quando applicato a istanze del problema differenti. Comunque, in prima approssimazione, esso consente di controllare l'impegno computazionale.
Si è trovato conveniente impostare differenti valori di tale parametro sulla base della dimensione come riportato in Tabella [6.1].

RAPPORTO TS/GA DI COMPOSIZIONE

Questo è un parametro molto importante che viene utilizzato per pesare il contributo relativo del TS e del AG. Conoscendo l'impegno computazionale (TS*GA) ed il rapporto di composizione TS/GA è possibile risalire ai criteri di fermata per il TS e il AG. Si è notato che più grande è il problema migliore è la prestazione dell'AG rispetto a quella del TS. Più in dettaglio si è assegnato un rapporto di composizione TS/GA di 10:1 per piccole e medie istanze e di 1:1 per quelle grandi. Vedi Tabella [6.1].

PARAMETRI DELL'AG

E' molto importante impostare opportunamente i parametri dell'AG al fine di garantire un buon flusso di informazioni tra l'AG e il TS durante l'intero processo evolutivo di ricerca, in modo tale da ottenere un'efficace cooperazione tra i due algoritmi. Si è scoperto che i seguenti parametri influenzano la qualità del flusso informativo e quindi è stata posta molta attenzione nel trovarne una buona impostazione:

- Dimensione della Popolazione. GTS è stato messo a punto focalizzando l'attenzione sulle relazioni significative tra i parametri anziché operare direttamente sui valori

assoluti. Prima si è provato a scoprire adeguati rapporti tra parametri rilevanti e solo in un secondo tempo si sono derivati indirettamente i loro valori assoluti. Seguendo questo approccio, la dimensione della popolazione è stata considerata in relazione diretta con il numero delle generazioni, scoprendo che un buon rapporto tra questi due parametri è 1:1. I valori assoluti trovati per il parametro dimensione della popolazione variano gradualmente da un minimo di 10 individui per piccole istanze fino ad un massimo di 50 individui per grandi istanze.

- Salto Generazionale. Questo parametro rappresenta il numero di discendenti da produrre ogni generazione attraverso la ricombinazione. Si è trovato che dimensione della popolazione / 2 è una buona impostazione del parametro.
- Pressione Selettiva. Questo parametro permette di controllare il livello di competizione all'interno della popolazione. Questo ifluenza il meccanismo di selezione basato sulla classificazione, rendendo la probabilità di selezione dei cromosomi più o meno dipendente dalla loro posizione nella classifica della popolazione sulla base del valore assegnato a questo parametro. L'intervallo dei valori validi per questo parametro varia da 0 (nessuna dipendenza) fino a 2 (dipendenza forte). Una pressione selettiva debole, fornisce agli individui "cattivi" quasi la stessa possibilità di riprodursi di quelli buoni, laddove una pressione selettiva forte favorisce molto di più la riproduzione dei soli individui "buoni". Nel nostro caso si è trovato che una pressione selettiva debole di 0.1 risulta appropriata. Questo fatto non dovrebbe risultare molto sorprendente perché nel nostro AG è stata utilizzata una modalità di reinserimento degli individui che di per se è già molto selettiva, quindi non è stato reso necessario rafforzare ulteriormente la pressione selettiva agendo su questo parametro.

4.2 GTS Vs TS

Nella Tabella [6.8] è riportato un confronto diretto tra l'algoritmo ibrido GTS e l'algoritmo di TS utilizzato al suo interno. Questa investigazione risulta cruciale perché in tal modo si può scoprire se l'algoritmo ibrido fornisce un reale contributo oppure se il contesto genetico risulta avere solo una funzione ornamentale piuttosto che un merito reale.

Al fine di effettuare un confronto equo tra GTS e TS sono stati fissati i parametri in maniera tale che entrambi gli algoritmi possano far uso, approssimativamente, dello stesso ammontare di tempo per la stessa istanza. Entrambi gli algoritmi sono stati testati su un insieme di istanze ben note di varie dimensioni. I dettagli delle istanze si possono trovare in Sezione [6.4.1]. Si può notare che su piccole istanze GTS e TS ottengono gli stessi buoni risultati negli stessi tempi. Al crescere delle dimensioni delle istanze GTS trova soluzioni migliori di quelle trovate dal TS. A prima vista TS sembra però risparmiare del tempo sulle istanze grandi. Questo è dovuto in buona sostanza all'adattività dei criteri di fermata. Al fine di prevenire questa prematura terminazione, è stata data la possibilità al TS di girare per più tempo, impostando opportunamente i parametri, in maniera tale da conferirgli la possibilità di scovare soluzioni migliori. Il TS non risulta comunque in grado di trovare soluzioni di qualità più elevata ed in buona sostanza spreca semplicemente il tempo addizionale fornitogli.

4.3 Un Confronto ad Ampio Spettro

E' stato fatto un confronto ad ampio spettro tra GTS e i migliori algoritmi risolutivi per il JSSP appartenenti ad una varietà di differenti approcci su un insieme di istanze molto note proposte da Vaessens. In Tabella [6.9] sono riportati i migliori risultati trovati per GTS e per gli altri approcci. In generale si nota che GTS si comporta molto bene. Di nuovo si vede che su grandi istanze GTS supera tutti gli altri approcci. I dettagli delle istanze e la lista completa degli algoritmi confrontati con GTS si trovano alla Sezione [6.4.2].

4.4 GTS Vs SAGen

Infine è stato fatto un confronto diretto tra il nostro algoritmo ibrido (basato sul taboo search) ed un altro algoritmo ibrido proposto di recente che combina algoritmi genetici e simulated annealing proposto da Kolonko [8].

Come di vede dalla Tabella [6.11], i due algoritmi sono stati confrontati su un insieme di istanze difficili, quasi tutte ancora aperte, fissando i criteri di fermata in maniera tale da privilegiare la migliore qualità rispetto al tempo impiegato. I dettagli delle istanze si trovano alla Sezione [6.4.3]. Come si può vedere, sia in termini di qualità delle

soluzioni trovate sia in termini di tempo impiegato, GTS batte di gran lunga SAGen e il più delle volte migliora i bound conosciuti per quelle istanze.

5 Conclusioni

Questo documento descrive un algoritmo ibrido (GTS) che combina Algoritmi Genetici e Taboo Search per risolvere il problema del Job-Shop. Gli elementi fondamentali del nostro Algoritmo Genetico sono una rappresentazione naturale delle soluzioni che ben si adatta allo specifico problema (la rappresentazione a stringa) ed una ricombinazione capace di trasmettere caratteristiche significative (la relazione si ordine comune) dai genitori ai figli. I problemi di ammissibilità riguardanti la presenza di cicli nelle soluzioni e il non rispetto dei vincoli dei job sono stati discussi e risolti in quell'ambito. Inoltre, è stato presentato l'operatore di ricombinazione MSX che tende a preservare la diversità delle soluzioni genitrici trasmettendo tale diversità alle soluzioni figlie. Al fine di combinare il nostro GA e un efficace algoritmo di TS è stato utilizzato lo schema della ricerca genetica locale. Esperimenti computazionali hanno mostrato che sulle istanze grandi la presenza della componente genetica è determinante. La miglior composizione per le istanze grandi è di 50 e 50 (secondo la definizione introdotta di rapporto di composizione) e quindi GTS deve essere considerato a pieno titolo come un vero algoritmo ibrido che combina efficacemente le differenti competenze dei due algoritmi. GTS non è da intendersi né come un GA modificato, né come un TS modificato. Inoltre, GTS è stato confrontato con una molteplicità di altri approcci e si è rivelato comportarsi molto bene in tale confronto. Nell'ultimo esperimento si è visto che gli algoritmi genetici risultano di gran lunga meglio combinati con il Taboo Search piuttosto che con il Simulated Annealing. Sia in termini di tempo richiesto sia in termini di qualità delle soluzioni trovate tra i due si è riscontrata la differenza di un ordine di grandezza.

Per concludere, parlando della filosofia sottostante all'approccio naturale della rappresentazione utilizzata, si nota come il punto cruciale è quello di vedere le soluzioni come relazioni d'ordine parziale tra operazioni. Non è importante che la relazione sia fatta di contributi di precedenza provenienti dai vincoli dati con l'istanza del problema

piuttosto che da quelli relativi alla particolare soluzione di quella data istanza. I vincoli sono visti uniformemente senza alcuna distinzione, tutti insieme formano la relazione d'ordine tra le operazioni.
Guardando le soluzioni come relazioni d'ordine, è naturale pensare alla ricombinazione come un modo per ricombinare relazioni d'ordine parziale trasmettendo alle soluzioni figlie la sottorelazione comune alle soluzioni genitrici. Questo sembra essere un requisito più che ragionevole quando si considerino le soluzioni sotto questo aspetto. Come apprezzato effetto collaterale di questo approccio, si ha che durante la trasmissione delle caratteristiche significative pure la proprietà di essere una soluzione ammissibile (una soluzione che rispetta tutti i vincoli sui job della data istanza del problema) è trasmessa dalle soluzioni genitrici alle soluzioni figlie senza porre ad essa una attenzione speciale. Essa è trattata uniformemente, come una generica caratteristica di una soluzione. Questo effetto collaterale positivo lascia pensare che questo sia il livello di astrazione corretto sotto il quale trattare il problema. Infine una ulteriore conseguenza di questo approccio è che la rappresentazione a stringa e la ricombinazione proposte non sono in nessun modo influenzati dalla particolare configurazione dei vincoli tipica del Job Shop e quindi possono naturalmente essere estesi a problemi di schedulazione più generali.

Part II

Project Description

Chapter 1

Introduction

1.1 Job Shop Scheduling Problem

Machine scheduling problems arise in diverse areas such as flexible manufacturing systems, production planning, computer design, logistics, communication, and so on. A common feature of many of these problems is that no efficient solution algorithm is known yet for solving them to optimality in polynomial time. The Job Shop Scheduling Problem (JSSP) is one of the best-known machine scheduling problems.

The form of the JSSP may be roughly sketched as follows: we are given a set of jobs and a set of machines. Each machine can handle at most one job at a time. Each job consists of a chain of operations, each of which needs to be processed during an uninterrupted time period of a given length on a given machine. The purpose is to find a schedule, that is an allocation of the operations to time intervals on the machines, which has minimum length.

1.2 Local Search and Taboo Search

Local search is based on what is perhaps the oldest optimisation method known, trial and error. In fact it is so simple that it is surprising just how well it works on a variety of difficult optimisation problems. The basic idea behind local search is to inspect the solutions that are close to a given solution, the so called neighbours, and to restart when a better solution – with respect to a predefined cost function – is obtained.

Taboo Search (TS) is a local search method designed to find a near-optimal solution of combinatorial optimisation problems [6]. The peculiarity of TS is a short term memory

used to keep track of recent solutions which are considered forbidden (taboo), thus allowing the search to escape from local optima.

TS has revealed to be an effective local search algorithm for the Job Shop Scheduling Problem [13, 16]. However, the best solution found by TS may depend on the initial solution used. This is essentially due to the fact that, like any other local search technique, TS starts its search from a single solution, which may lead the search to a dead-end despite the presence of the taboo mechanism which would prevent it. This happens especially when TS is applied to particularly hard optimisation problem like JSSP.

1.3 Genetic Algorithms and Genetic Local Search

Genetic Algorithms (GAs) are stochastic global search methods that mimic the natural biological evolution [7]. GAs operate on a population of potential solutions applying the principle of survival of the fittest to produce (hopefully) better and better approximations to a solution.

Simple GAs are difficult to apply directly and successfully into many difficult-to-solve optimisation problems. Various non-standard implementations have been created for particular problems in which genetic algorithms are used as meta-heuristics [3, 10]. In this new perspective, Genetic Algorithms are very effective at performing global search (in probability) and provide us a great flexibility to hybridise with domain-dependent heuristics to make an efficient implementation for a specific problem.

Several authors have proposed variants of local search algorithms, using ideas from population genetics [1, 11, 19]. Because of the complementary properties of genetic algorithms and conventional heuristics, the hybrid approach often outperforms either method operating alone.

A Genetic Local Search (GLS) algorithm is a particular kind of hybrid algorithm that consists of a basic Genetic Algorithm with the addition of a local search optimisation phase applied to every new individual created either in the initial population or during the evolutionary process.

1.4 Applying Genetic Algorithms to Job Shop

In order to apply GAs to a particular problem we have to encode a generic solution of the problem into a chromosome. How to encode a solution is a key-issue for the success of GAs. Canonical GAs use binary encoding of individuals on fixed-length strings. Such a representation is not naturally suited for ordering problems such as the Travelling Salesman Problem and the JSSP, because no direct and efficient way has been found to map all possible solutions into binary strings. Previous studies on ordering problems as the travelling salesman problem (TSP) have proven that a natural representation is the key-issue for the success of a GA approach [17].

The JSSP is mainly characterised by being a both highly constrained and ordering problem. Therefore, both aspects have to be considered in order to figure out a natural GAs representation. GA representations for JSSP can be found in [4, 8, 12].

The main difficulty in choosing a proper representation for highly constrained combinatorial optimisation problems such as JSSP is dealing with the infeasibility of the solutions produced during the evolutionary process.

1.5 Scope

The aim of this project is to investigate the combined use of Genetic Algorithms and Taboo Search in order to solve the Job Shop Scheduling Problem. To this end, it is necessary to develop an appropriate genetic representation especially suited for Job Shop.

In this thesis, we propose a representation that, coupled with a particular class of recombination operators, guarantees the genetic search to represent all and only the feasible solutions and that guarantees the transmission of meaningful characteristics to the offspring solutions. The definition of a class of recombination operators and the choice of an operator showing interesting properties follow.

As a scheme of hybridisation, we propose a genetic local search algorithm (GTS) consisting of a basic genetic algorithm with the addition of a taboo search optimisation phase applied to every new individual created.

1.6 Results

Computational experiments show that the combination of GAs and TS performs better than the TS alone. Moreover, a wide comparison of GTS with a variety of algorithms for JSSP on a set of well-known instances of small and medium sizes shows that GTS is very well positioned. Finally, a specific comparison of GTS with a similar approach combining Genetic Algorithms and Simulated Annealing (SAGen) [8] on a set of large size instances shows that GTS outperforms SAGen both on computational time and solutions quality.

1.7 Reader's Guide

The contents of the project have been arranged to be read chapter by chapter. However if the reader is already familiar with Job Shop, Local Search and Genetic Algorithms, he or she can skip chapter 2, chapter 3 and chapter 4, respectively, and go to chapter 5 directly. The contents of each chapter are as follows:

- *Chapter 1* gives a general introduction of the project.
- *Chapter 2* introduces the Job Shop Scheduling Problem, including a detailed example and common heuristics.
- *Chapter 3* gives an overview on Local Search especially focusing on Taboo Search.
- *Chapter 4* introduces Genetic Algorithms and other Evolutionary Techniques with special regard to Combinatorial Optimisation.
- *Chapter 5* is the core of this thesis. It proposes a new genetic representation for the Job Shop, a set of recombination operators and the Genetic Local Search algorithm.
- *Chapter 6* describes experiments with different parameters and provides interesting comparisons with other programs.
- *Chapter 7* provides conclusions and suggestions for further research.

Chapter 2

Job Shop Scheduling Problem

2.1 Introduction

In this chapter we are going to give a description of different aspects of the Job Shop Scheduling Problem. First, we consider its computational complexity and its formal definition. In the subsequent section a smart representation of the problem, the *Disjunctive Graph*, is introduced together with a working example that makes easier to become familiar with the problem. Afterward, the different kinds of schedule are discussed and *Gantt Charts*, schedule representations in time, introduced. Finally, two well-known conventional heuristics for Job Shop, *Priority Dispatching Rules* and *Shifting Bottleneck*, are reported.

2.2 Complexity and Formal Definition

Job Shop Scheduling Problem is considered as a particularly hard combinatorial optimisation problem. The difficulty of this problem may be illustrated by the fact that the optimal solution of an instance with 10 jobs and 10 machines, proposed by Fisher and Thompson, was not found until 20 years after the problem was introduced. In terms of computational complexity, only very special cases of the problem can be solved in polynomial time, but their immediate generalisations are *NP-hard* [9].

In the chapter 1 we have proposed an informal description of the JSSP, in the sequel we give its formal definition:

- Given are three finite sets, a set J of jobs, a set M of machines and a set O of operations.
- For each operation a there is a job $j(a)$ in J to which it belongs, a machine $m(a)$ in M on which it must be processed and a processing time $d(a)$.
- Furthermore for each operation a its successor in the job is given by $sj(a)$, except for the last operation in a job.
- The problem is to find start times $s(a)$ for each operation a such that the cost function $Min[s(a)+d(a) | a \text{ in } O]$ is minimal, subject to the conditions:

1. For all a, $sj(a)$ in O: $sj(a) >= s(a) + d(a)$
2. For all a, b in O with $a <> b$ and $m(a) = m(b)$: $s(b) >= s(a) + d(a)$ *or* $s(a) >= s(b) + d(b)$

2.3 Disjunctive Graph

The job shop scheduling problem can be represented with a disjunctive graph [15]. A disjunctive graph $G=(N, A, E)$ is defined as follows: N is the set of nodes representing all operations, A is the set of arcs connecting consecutive operations of the same job, and E is the set of disjunctive arcs connecting operations to be processed by the same machine. A disjunctive arc can be settled by either of its two possible orientations. The construction of a schedule will settle the orientations of all disjunctive arcs so as to determine the sequence of operations on the same machine. Once a sequence is determinate for a machine, the disjunctive arcs connecting operations to be processed by the machine will be replaced by the oriented precedence arrow, or conjunctive arc. The set of disjunctive arcs E can be decomposed into cliques, one for each machine. The processing time for each operation can be seen as a weight attached to the corresponding nodes. The JSSP is equivalent to find the order of the operations on each machine, that is, to settle the orientation of the disjunctive arcs such that the resulting solution graph is acyclic (there are no precedence conflicts between operations) and the length of the longest weighted path between the starting and terminal nodes is minimal. This length determines the *makespan*.

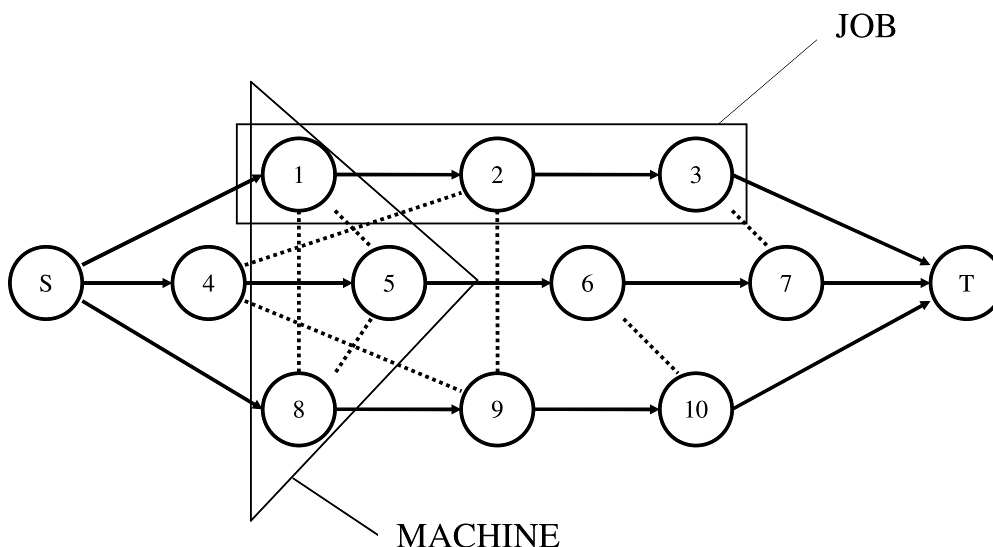


Figure 2.1 – Disjunctive Graph (Elements of *A* are indicated by arrows and elements of *E* are indicated by dashed lines.)

	Pos 1	*Pos 2*	*Pos 3*	*Pos 4*
Job 1	Op1(1, Mac1)	Op2(2, Mac2)	Op3(3, Mac3)	-
Job 2	Op4(3, Mac2)	Op5(1, Mac2)	Op6(2, Mac4)	Op7(2, Mac3)
Job 3	Op8(2, Mac1)	Op9(1, Mac2)	Op10(3, Mac4)	-

Table 2.1 – Example of a Three-Job Four-Machine Problem (for each operation, its position on the job, its machine and its processing time are given)

Figure 2.1 illustrates the disjunctive graph for a three-job four-machine instance. Its complete data are presented in Table 2.1 (for each operation, its position on the job, its machine and its processing time are given). The nodes of $N = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ correspond to operations. Nodes S and T are two special nodes, starting node (S) and terminal node (T), representing the beginning and the end of the schedule, respectively. The conjunctive arcs (arrows) of $A = \{(1, 2), (2, 3), (4, 5), (5, 6), (6, 7), (8, 9), (9, 10)\}$ correspond to precedence constraints on operations on same jobs. The

disjunctive arcs (dashed lines) of E_1 = {(1, 5), (1, 8), (5, 8)} concern operations to be performed on machine 1, disjunctive arcs E_2 = {(2, 4), (2, 9), (4, 9)} concern operations to be performed on machine 2, disjunctive arcs E_3 = {(3, 7)} concern operations to be performed on machine 3, and disjunctive arcs E_4 = {(6, 10)} concern operations to be performed on machine 4.

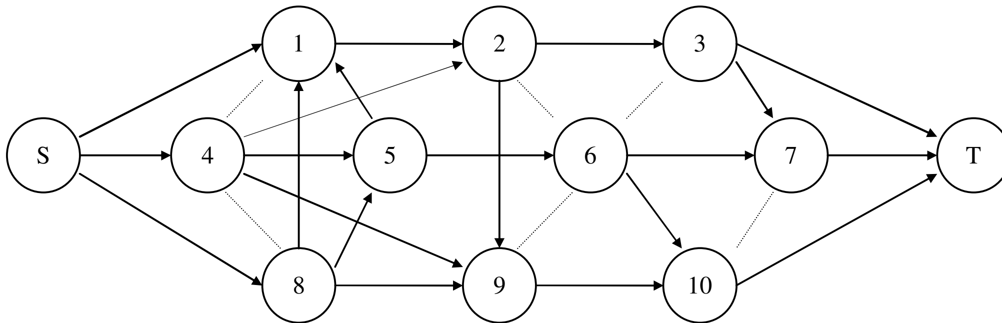


Figure 2.2 – Solution Graph (Sequential operations are connected by arrows, parallel operations are connected by dashed lines)

Figure 2.2 illustrates the solution graph representing a feasible solution to the given instance of the problem. It has been derived from the disjunctive graph described above by settling an orientation of all the disjunctive arcs having taken care to avoid the creation of cycles. In the solution graph, arrows correspond to precedence constraints among operations on jobs or machines. Dashed lines indicate that two operations don't have any precedence constraints (in principle they could be processed in parallel without violating any precedence constraints. In fact their actual parallel processing depends only on the processing time of operations). The sequences of operations on jobs depend only on the instance of the problem and not on the particular solution. In our example they are illustrated in Table 2.2.

	Pos 1	*Pos 2*	*Pos 3*	*Pos 4*
Job1	Op1	Op2	Op3	-
Job2	Op4	Op5	Op6	Op7
Job3	Op8	Op9	Op10	-

Table 2.2 – Sequences of operations on jobs

On the contrary, the sequences of operations on machines also depend on the particular solution to the given problem. In our example they are illustrated in Table 2.3.

	Pos 1	*Pos 2*	*Pos 3*
Mac1	Op8	Op5	Op1
Mac2	Op4	Op2	Op9
Mac3	Op3	Op7	-
Mac4	Op6	Op10	-

Table 2.3 – Sequences of operations on machines

2.4 Schedules and Gantt Charts

Once we have a feasible schedule, we can effectively represent it in time by Gantt charts. There are two kinds of Gantt charts: machine Gantt chart and job Gantt chart. Coming back to our example, Figure 2.3 shows the schedule corresponding to the solution graph in Figure 2.2 from the perspective of what time the various jobs are on each machine, while Figure 2.4 shows the same schedule from the perspective of when the operations of a job are processed. In both charts, the operations belonging to the critical path are marked with a star.

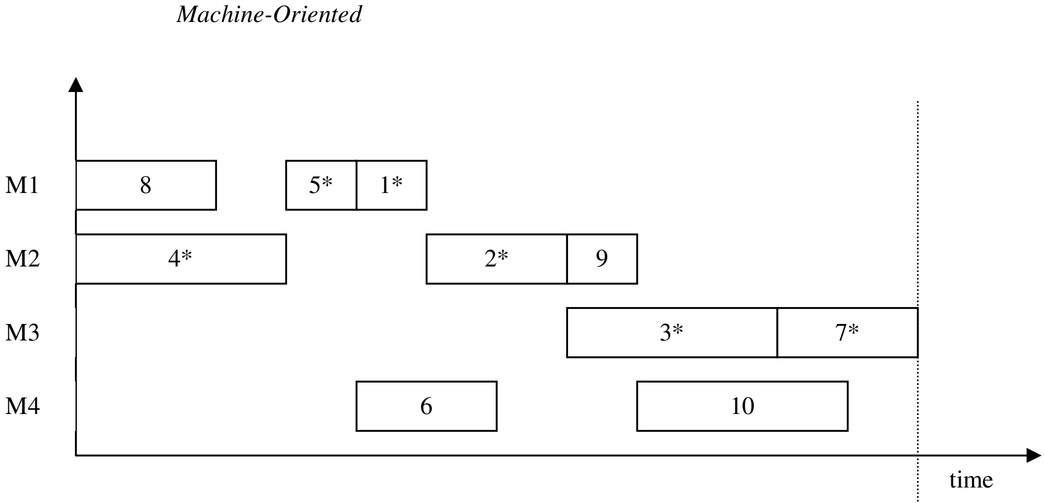


Figure 2.3 – Machine Gantt chart

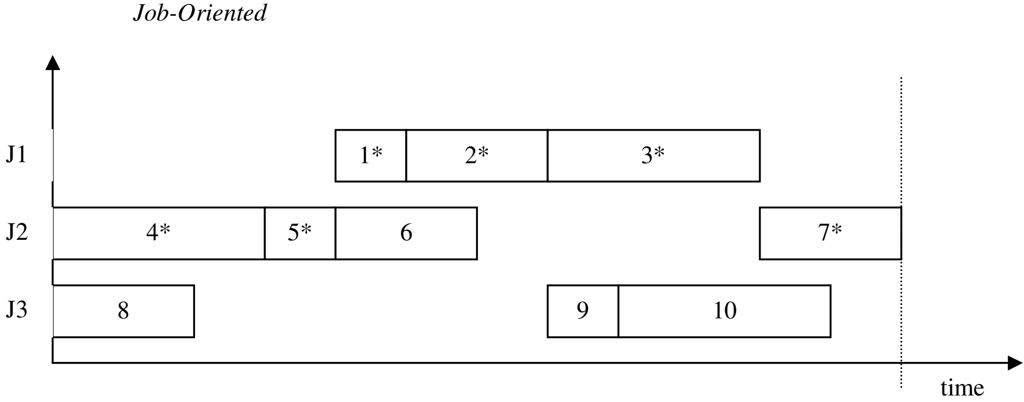


Figure 2.4 – Job Gantt chart

In principle, there are an infinite number of feasible schedules for a job shop problem, because superfluous idle time can be inserted between two operations. We may shift the operations to the left as compact as possible. A shift in a schedule is called *local left-shift* if some operations can be started earlier in time without altering the operation sequence. A shift is called *global left-shift* if some operations can be started earlier in time without delaying any other operation even though the shift has changed the

operation sequence. Based on these two concepts, three kinds of schedules can be distinguished as follows:

- *Semi-active* schedule. A schedule is semi-active if no-local left-shift exists.
- *Active* schedule. A schedule is active if no global left-shift exists.
- *Non-delay* schedule. A schedule is non-delay if no machine is kept idle at a time when it could begin processing some operations.

The relationship among active, semi-active and non-delay schedules is shown in the Venn diagram in Figure 2.5. Optimal schedule is within the set of active schedules. The non-delay schedules are smaller than active schedules, but there is no guarantee that the former will contain an optimum.

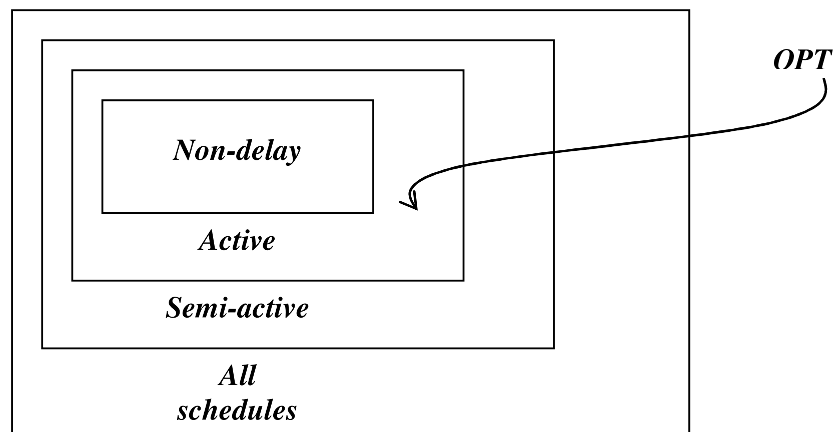


Figure 2.5 – Venn diagram of schedule relationship

2.5 Conventional Heuristics

2.5.1 Dispatching Heuristics

Job shop scheduling is a very important everyday practical problem. Since job shop scheduling is among the hardest combinatorial optimisation problems, it is therefore natural to look for approximation methods that produce an acceptable schedule in useful time. A simple heuristic is building a single complete solution by fixing one operation in the schedule at a time based on *priority dispatching rules*. There are many rules for choosing an operation from a specified subset to be scheduled next. This heuristic is fast and usually finds solutions that are not too difficult. In addition, this heuristic may be used repeatedly to build a more complicated multi-pass heuristic in order to obtain better schedules at some extra computational cost.

Priority rules are probably the most frequently applied heuristics for solving scheduling problems because of their ease of implementation and their low time complexity. The algorithms of Giffler & Thompson [9] can be considered as the common basis of all priority-rule-based heuristics. Giffler & Thompson have proposed two algorithms to generate schedules: *active schedule* and *non-delay schedule* generation procedures. A non-delay schedule has the property that no machine remains idle if a job is available for processing. An active schedule has the property that no operation can be started earlier without delaying another job. Active schedules form a much larger set and include non-delay schedules as a subset. The generation procedure of Giffler & Thompson is a tree-structured approach. The nodes in the tree correspond to partial schedules, the arcs represent the possible choices, and the leaves of the tree are the set of enumerated schedules. For a given partial schedule, the algorithm essentially identifies all processing conflicts (i.e. operations competing for the same machine), and an enumeration procedure is used to resolve these conflicts in all possible ways at each stage. By contrast, heuristics resolve these conflicts with priority dispatching rules, that is, they specify a priority rule for selecting one operation among those conflicting.

Generation procedures operate with a set of schedulable operations at each stage. Schedulable operations are unscheduled operations with immediately scheduled predecessors; this set can be simply determined from the precedence structure. The

number of stages of the procedure is equal to the number of operations $m \times n$. At each stage, one operation is selected to be added into the partial schedule. Conflicts among operations are solved by priority dispatching rules. For a given active partial schedule, the potential start time is determined by the completion time of the direct predecessor on job of the current operation and the latest completion time on the machine required by the current operation. The larger of these two quantities is therefore the potential start time of the current operation. The potential finishing time is the sum of the processing time of the current operation with its potential start time.

Priority Dispatching Heuristic (Active Schedule Generation)

Let PS_t be a partial schedule containing t scheduled operations; let S_t the set of schedulable operations at stage t, corresponding to a given PS_t; Let $s(i)$ and $c(i)$ the earliest time at which operation $i \in S_t$ could be respectively started and completed.

1. Set $t=0$ and begin with PS_t as the null partial schedule. Initially S_t includes all operations with no predecessors.
2. Determine the minimum completion time c^* in the set S_t of schedulable operations and the machine m^* on which c^* could be realized.
3. For each operation $i \in S_t$ that requires machine m^* and for which its earliest start time $s(i)$ is lesser than c^*, calculate a priority index according to a specific priority rule. Find the operations with the smallest index and add this operation to PS_t as early as possible, thus creating a new partial schedule PS_{t+1}.
4. For PS_{t+1}, update the data set as follows:
    a. Remove operation i from S_t.
    b. Form S_{t+1} by adding the direct successor of operation i to S_t.
    c. $t=t+1$
5. Go to step 2 until a complete schedule is generated.

With minor modifications the above algorithm can generate a non-delay schedule. The remaining problem is to identify an effective priority rule. Table 2.4 consists of some of the priority rules commonly used in practice.

Rule	*Description*
SPT	Shortest Processing Time
LPT	Longest Processing Time
MWR	Most Work Remaining
LWR	Least Work Remaining
MOR	Most Operations Remaining
LOR	Least Operations Remaining
EDD	Earliest Due Date
FCFS	First Come, First Served
RANDOM	Random

Table 2.4 – Commonly used Job Shop dispatch rules

2.5.2 Shifting-Bottleneck Heuristics

At the present time, the *shifting bottleneck heuristic* [9] is probably the most powerful procedure among all heuristics for the job shop scheduling problem. It sequences the machines one by one, successively, taking each time the machine identified as a bottleneck among the machines not yet sequenced. Every time after a new machine is sequenced, all previously established sequences are locally re-optimised. Both the bottleneck identification and the local re-optimisation procedures are based on repeatedly solving a certain one-machine scheduling problem that is a relaxation of the original problem. The method of solving the one-machine problems is not new in this context. The main contribution of this approach is the way to use the relaxation to decide upon the order in which the machines should be sequenced. This is based on the classic idea of giving priority to bottleneck machines. A brief statement of the shifting bottleneck procedure is as follows:

Shifting Bottleneck Heuristic

Let M' be the set of machines already sequenced (M' is empty at the start).

1. Identify a bottleneck machine m among the machines $k \in M \setminus M'$ and sequence it optimally. Set $M' = M'$ $\{m\}$.

2. Re-optimise the sequence of each critical machine $k \in M'$ in turn, while keeping the other sequences fixed. Afterward if $M' = M$ then stop, otherwise go to step 1.

Chapter 3

Local Search and Taboo Search

3.1 Introduction

In this chapter we are going to give a broad overview on local search algorithms, paying special attention to Taboo Search. Genetic Algorithms are mentioned in here, since they can be seen as a particular local search technique. However, since they are of central importance in this project, we prefer dedicate more room to Genetic Algorithms and other Evolutionary techniques in chapter 4. Moreover, local search representations and neighbourhood functions for Job Shop are discussed.

3.2 Overview on Local Search Algorithms

Many combinatorial optimisation problems have been proved NP-hard, and it is consequently believed that they cannot be solved in polynomial time. In practice this means that solving large instances of such problems to optimality (by *optimisation algorithms*) requires impracticable running times. To avoid this, one often resorts to *approximation algorithms* that try to find near-optimal solutions within acceptable running times.

From an application point of view, approximation algorithms range from tailored algorithms, designed for a specific problem type, to general algorithms, which can be applied to a broad range of problem types. For the latter, it is often more appropriate to speak of algorithmic templates (or *meta-heuristics*) since in many cases the problem specific details still need to be filled in to obtain an operational algorithm. Local search is an example of such an algorithmic template for combinatorial optimisation.

Local search employs the idea that a given solution may be improved by making small changes. Solutions are changed over and again, and better and better solutions are found. In the sequel, we present the most famous local search techniques.

3.2.1 Iterative Improvement

The basic local search algorithm to find a local optimum is called *iterative improvement*. Starting at some initial feasible solution, its *neighbourhood* is searched for a solution of lower cost. If such a solution is found, the algorithm is continued from there, otherwise a local optimum has been found.

The quality of the local optimum depends on the initial solution, on the neighbourhood function, and on the method of searching the neighbourhoods. An initial solution may be obtained by generating it randomly or by applying a heuristic rule. The choice of a good neighbourhood is often difficult. There is a clear trade-off between small and large neighbourhoods: if the number of neighbours is larger, the probability of finding a good neighbour may be higher, but looking for it takes more time. There are several alternatives for searching the neighbourhood: one may take the first neighbour found of lower cost (*first improvement*), or take the best neighbour in the entire neighbourhood (*best improvement*), or take the best of a sample of neighbours, provided it is an improving one.

Often, the problem remains. The local optima obtained may be of poor quality. Therefore, several variants of iterative improvement have been proposed. The main variants can be divided into threshold algorithms, taboo search algorithms, variable-depth search algorithms and genetic algorithms.

3.2.2 Threshold Algorithms

In *threshold algorithms*, a neighbour of a given solution becomes the new current solution if the cost difference between the current solution and its neighbour is below a certain threshold. There are three different kinds of threshold algorithms.

In classical *iterative improvement* the thresholds are 0, so that only true improvements are accepted. In *threshold accepting* the threshold are nonnegative. They are large in the beginning of the algorithm's execution and gradually decrease to become 0 in the end.

General rules to determine appropriate thresholds are lacking. In *simulated annealing* the thresholds are positive and stochastic. Their values depend on a control parameter called ‘temperature’, whose value gradually decreases in the course of the algorithm’s execution according to a ‘cooling schedule’, and on a uniform random variable. Each time a neighbour is compared with the current solution the random variable is drawn again. Under certain mild conditions simulated annealing is guaranteed to find an optimal solution asymptotically.

3.2.3 Taboo Search

In *taboo search* one selects from a subset of permissible neighbours of the current solution a solution of minimum cost. In basic taboo search a neighbour is permissible if it is not in the ‘taboo list’ or satisfies a certain ‘aspiration criterion’. The taboo list is recalculated at each iteration. It is often implicitly defined in terms of forbidden moves from the current solution to a neighbour. The aspiration criterion expresses possibilities to overrule the taboo status of a neighbour. We will consider this approach more in detail in the section 3.3.

3.2.4 Variable-Depth Search

In *variable-depth search* one starts from an initial solution and generates a sequence of subsequent neighbours by making relatively small changes. From this sequence one solution is selected to serve as the initial solution for a next sequence. Often, each time the sequence is extended, the set from which a neighbour is to be chosen is restricted by a sort of taboo list; this list may again be defined implicitly in terms of forbidden moves. Each time a new sequence is started, the list is emptied.

3.2.5 Genetic Algorithms

Genetic Algorithms are based on an extended notion of neighbourhood function. A hyper-neighbourhood function defines for each vector of variables, representing a solution, a set of neighbouring solutions, called hyper-neighbours. At each iteration a set of solutions, often called ‘population’, is given. From this population several subsets consisting of ‘parents’ are selected, and for each of such subset some hyper-neighbours,

called ‘offspring’, are determined by operations called ‘recombination’ and ‘mutation’. This set of hyper-neighbours and the current population are then combined and reduced to a new population by selecting a subset of solutions. We will consider extensively Genetic Algorithms and other Evolutionary techniques in chapter 4.

3.3 Focus on Taboo Search

The *taboo search* is a meta-heuristic approach designed to find a near-optimal solution of combinatorial optimisation problems. This method has been suggested by Glover et al. [6]. TS can be briefly sketched as follows. At the beginning a fundamental notion called *move* is defined. The move is a function which transforms a solution into another solution. For any solution, a subset of moves applicable to it is defined. This subset generates a subset of solutions called *neighbourhood*. TS starts from an *initial solution*. At each step the neighbourhood of a given solution is searched in order to find a neighbour (usually the best in the neighbourhood). This move, leading us to the best neighbour, is performed and then the newly obtained solution is set as the primal for the next step. In order to prevent cycling and to lead the search to “good” regions of the solution space the search history is kept in the memory and employed in the search. There are at least two classes of memory: a *short-term memory* for the very recent history and a *long-term memory* for distant history. Among many memory structures established in TS, a class of short-term memory called *taboo list* plays a basic role. This list does not permit to turn back to the solutions visited in the previous *maxt* steps, where *maxt* is a given number. In practice the list stores forbidden moves (or their attributes) or attributes of forbidden solutions rather than forbidden solutions. Quite often, the taboo list T is interpreted as a limited queue of length *maxt* containing forbidden moves; whenever a move from a solution to its neighbour is made we put the inverted move at the end of T and remove the first element from T if the queue is overloaded. Of course, it may happen that an interesting move is taboo. Nevertheless in order to perform such a move, an *aspiration function* must be defined. This function evaluates the profit in taking a forbidden move. If this profit is acceptable, then the taboo status of the move is dropped and the move can be performed. The *stopping rule* can be defined as an instance: 1) we found a solution which is close enough to the given

lower bound (in case of minimisation) of the goal function value; 2) we performed *maxiter* moves without improving the best solution obtained so far, where *maxiter* is a given number; and 3) the time limit ran out.

It should be noted that any implementation of TS is problem-oriented and needs particular definitions of structural elements such as the move, neighbourhood, memory structures, aspiration function, neighbourhood searching strategy, stopping rules, the initial solution, and values of several tuning parameters such as *maxt*, *maxiter*, and the level of aspiration. The final numerical characteristics of an algorithm (the performance, speed of convergence, and running time) depend both on structural elements and tuning parameters. A template of the taboo search procedure is presented in Section 5.5.3.

3.4 Local Search for the Job Shop Problem

A crucial ingredient of a local search algorithm is the definition of a neighbourhood function in combination with a solution representation. Quite a number of basic representations and neighbourhood functions have been introduced for job shop scheduling problem. For most threshold and taboo search algorithms, only left-justified or active schedules are represented. This is done by specifying the start times of the operations or, equivalently, the corresponding machine orderings of the operations. Other representations are used too, especially in combination with genetic algorithms.

Several neighbourhood functions have been proposed in the literature [20]. Most of these are not defined on a schedule itself but on the corresponding orientation (i.e. disjunctive graph). If this is changed into another feasible orientation, the corresponding schedule is a neighbour of the original schedule. In this way neighbours of a given schedule are always left-justified.

The following properties are helpful in obtaining reasonable neighbourhood functions:

1. Given a feasible orientation, reversing an oriented edge on a longest path in the corresponding digraph results again in a feasible orientation.
2. If reversing an oriented edge of a feasible orientation that is not on a longest path results in a feasible orientation, then the corresponding schedule is at least as long as the original schedule.

3. Given a feasible orientation, reversing an oriented edge between two internal operations of a *block* results in a feasible schedule at least as long as the original schedule. A block is a maximal sequence of size at least one, consisting of adjacent operations that are processed on the same machine and belong to a longest path. An operation of a block is *internal* if it is neither the first nor the last operation of that block.
4. Given a feasible orientation and reversing the oriented edge which connects the first two operations of the first block of the longest path where the second operation is internal, it results a feasible schedule at least as long as the original schedule. The same is true in case the two operations considered are the last two of the last block of a longest path where the first operation is internal.

In view of these properties, the simplest neighbourhood functions are based on the reversal of exactly one edge of a given orientation. So a neighbour may be obtained by interchanging two adjacent operations of a block, or more selectively, making the same interchanges except those involving two internal operations, or yet excluding also the interchange of the first two operations of the first block when the second is internal and the interchange of the last two operations of the last block when the first is internal.
Several neighbourhood functions may reverse more than one edge; other neighbourhood functions are not based on adjacent interchanges but rather on jumps of operations on the same machine; others may completely change one machine ordering or yet change several machine orderings at the same time.

Chapter 4

Genetic Algorithms and Evolutionary Computation

4.1 Introduction

In this chapter we are going to introduce Evolutionary Computation techniques starting from the most known, Genetic Algorithms, and emphasise their application to combinatorial optimisation problems. First, we discuss the classic genetic algorithms and their major elements. Afterward, we give a broad overview on other evolutionary computation techniques, parallel genetic algorithms are also mentioned, and discuss their peculiarities and differences. Subsequently, a different perspective on genetic algorithms is proposed, emphasising their role as a meta-heuristics. This naturally leads us to pay special attention to the their application to combinatorial optimisation problems. In particular, we will consider different aspects of non-conventional genetic representations and the opportunity to hybridise them with other search techniques. The genetic local search scheme is therefore introduced. Since in this project we apply genetic algorithms to scheduling problems, we will dedicate some space also to this topic in here. Finally, we provides some theoretical background that try to explain why genetic algorithms work.

4.2 What are Genetic Algorithms?

Genetic Algorithms (GAs) [7] are stochastic search methods that mimic the metaphor of natural biological evolution. GAs operate on a population of potential solutions applying the principle of survival of the fittest to produce better and better approximations to a solution. At each generation, a new set of approximations is created by the process of selecting individuals according to their level of fitness in the problem domain and breeding them together using operators borrowed from natural genetics. This process leads to the evolution of populations of individuals that are better suited to their environment than the individuals that they where created from, just as in natural adaptation. Figure 4.1 shows the structure of a simple genetic algorithm.

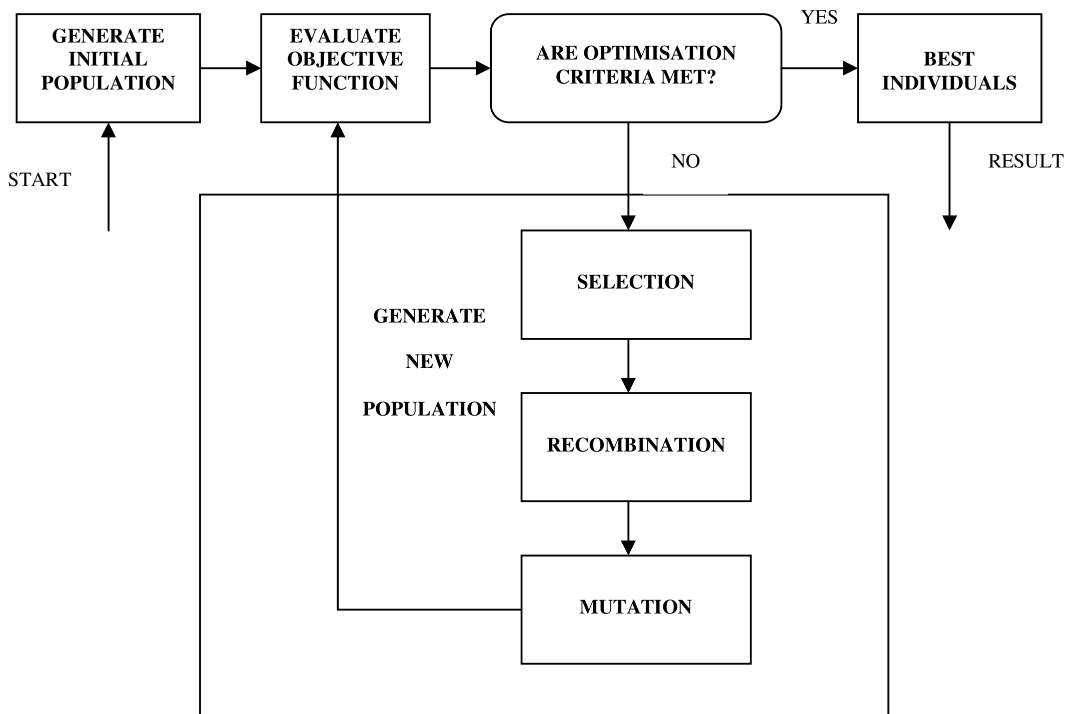


Figure 4.1 – Structure of a Genetic Algorithm

Genetic Algorithms, differing from conventional search techniques, start with an initial set of random solutions called *population*. Each individual in the population is called a

chromosome, representing a solution to the problem at hand. A chromosome is usually, but not necessarily, a binary bit string. The chromosomes *evolve* through successive iterations, called *generations*. During each generation, the chromosomes are *evaluated*, using some measure of *fitness*. To create the next generation, new chromosomes, called *offspring*, are formed by merging two chromosomes from current generation using a *crossover* operator or modifying a chromosome using a *mutation* operator. A new generation is formed by *selecting*, according to the fitness values, some of the parents and offspring and rejecting others so as to keep the population size constant. After several generations, the algorithm converges to the best chromosome, which hopefully represents the optimum or sub-optimal solution to the problem.
Genetic Algorithms differ substantially from more traditional search and optimisation methods. The most significant differences are:

- GAs search a population of points in parallel, not a single point.
- GAs do not require derivative information or other auxiliary knowledge; only the objective function and corresponding fitness levels influence the directions of search.
- GAs use probabilistic transition rules, not deterministic ones.
- GAs can provide a number of potential solutions to a given problem; the final choice is left to the user.

4.3 Major Elements of Genetic Algorithms

This section describes, one by one, the major elements of genetic algorithms.

4.3.1 Initialisation

There are many ways to initialise and encode the initial generation. The traditional Holland's encoding method is as a binary fixed-length string. At the initial stage, the system just randomly generates bits of each chromosome and encodes all the chromosomes as a population, then decodes and evaluates the chromosomes for using in reproduction and selection of the first generation. For a detailed discussion on representation see Section 4.6.

4.3.2 Selection

Selection determines which individuals are chosen for mating (recombination) and how many offspring each selected individual produces. The selection phase may be decomposed into two logically separated steps, *fitness assignment* and *sampling* respectively. Fitness assignment concerns the task of giving to an individual a value, which characterises its performance within the population. Sampling consists in the actual selection task where parents are selected according to their fitness for recombination.

Fitness assignment

The first step is fitness assignment. The most commonly used schemes are:

- Proportional fitness assignment
- Rank-based fitness assignment

Following the *proportional fitness* assignment, each individual in the population receives a reproduction probability depending on its own objective value and the objective value of all other individuals in the population. This fitness is used for the actual selection step afterwards.

In *rank-based fitness* assignment, the population is sorted according to the objective values. The fitness assigned to each individual depends only on its position in the individuals rank and not on the actual objective value.

There are many advantages in using rank-based fitness assignment. Rank-based fitness assignment overcomes the scaling problems of the proportional fitness assignment (stagnation in the case where the selective pressure is too small or premature convergence where selection has caused the search to narrow down too quickly). The reproductive range is limited, so that no individuals generate an excessive number of offspring. Ranking introduces a uniform scaling across the population and provides a simple and effective way of controlling selective pressure. Rank-based assignment behaves in a more robust manner than proportional fitness assignment.

Let us consider linear ranking more in detail. Let N be the number of individuals in the population, Pos the position of an individual in this population (least fit individual has $Pos=1$, the fittest individual $Pos=N$) and SP the selective pressure. The fitness value for an individual is calculated as:

$$Fitness(Pos) = 2 - SP + 2(SP - 1) \times \frac{Pos - 1}{N - 1}$$

Linear ranking allows values of selective pressure in [1.0, 2.0].

Sampling

The actual selection may be performed by means of one of the following algorithms:

- Roulette-wheel selection
- Stochastic universal sampling

The simplest selection scheme is *roulette-wheel selection*, also called stochastic sampling with replacement. This is a stochastic algorithm and involves the following technique:

The individuals are mapped to contiguous segments of a line, such that each individual's segment is equal in size to its fitness. A random number is generated and the individual whose segment spans the random number is selected. The process is repeated until the desired number of individuals is obtained (called mating population). This technique is analogous to a roulette wheel with each slice proportional in size to the fitness.

Stochastic universal sampling is fairer than the roulette-wheel selection; it ensures a selection of offspring, which is closer to what is deserved than roulette wheel selection. The individuals are mapped to contiguous segments of a line, such that each individual's segment is equal in size to its fitness exactly as in roulette-wheel selection. Here equally spaced pointers are placed over the line as many as there are individuals to be selected. Consider N the number of individuals to be selected, then the distance

between the pointers are *1/N* and the position of the first pointer is given by a randomly generated number in the range [*0, 1/N*].

4.3.3 Recombination

Recombination produces new individuals combining the information contained in the parents. The traditional representation of solutions (individuals) is by means binary string. Traditional recombination defined on binary string is known as crossover. We consider three different types of crossover operator:

- Single-point crossover
- Multi-point crossover
- Uniform crossover

Single-point Crossover

In single-point crossover one crossover position k in [1, L-1] where L is the length of an individual, is selected uniformly at random and the variables (bits) exchanged between the individuals about this point, then two new offspring are produced. An example of single-point crossover is shown in Figure 4.2.

Multi-point Crossover

For multi-point crossover, m crossover positions are chosen at random with no duplicates and sorted in ascending order. Then, the variables between successive crossover points are exchanged between the two parents to produce two new offspring. The section between the first variable and the first crossover point is not exchanged between individuals.

Uniform Crossover

Single-point crossover defines a cross-point where an individual can be split. Uniform crossover generalises this scheme to make every position a potential crossover point. A crossover mask of the same length as the individual structure is created at random and

the parity of the bits in the mask indicates which parent will supply the offspring with which bits.

For each variable the parent who contributes its variable to the offspring is chosen randomly with equal probability. The first offspring is produced by taking the bit from the first parent, if the corresponding mask bit is 1; from the second parent, if the corresponding mask bit is 0. The second offspring is created using the inverse of the mask.

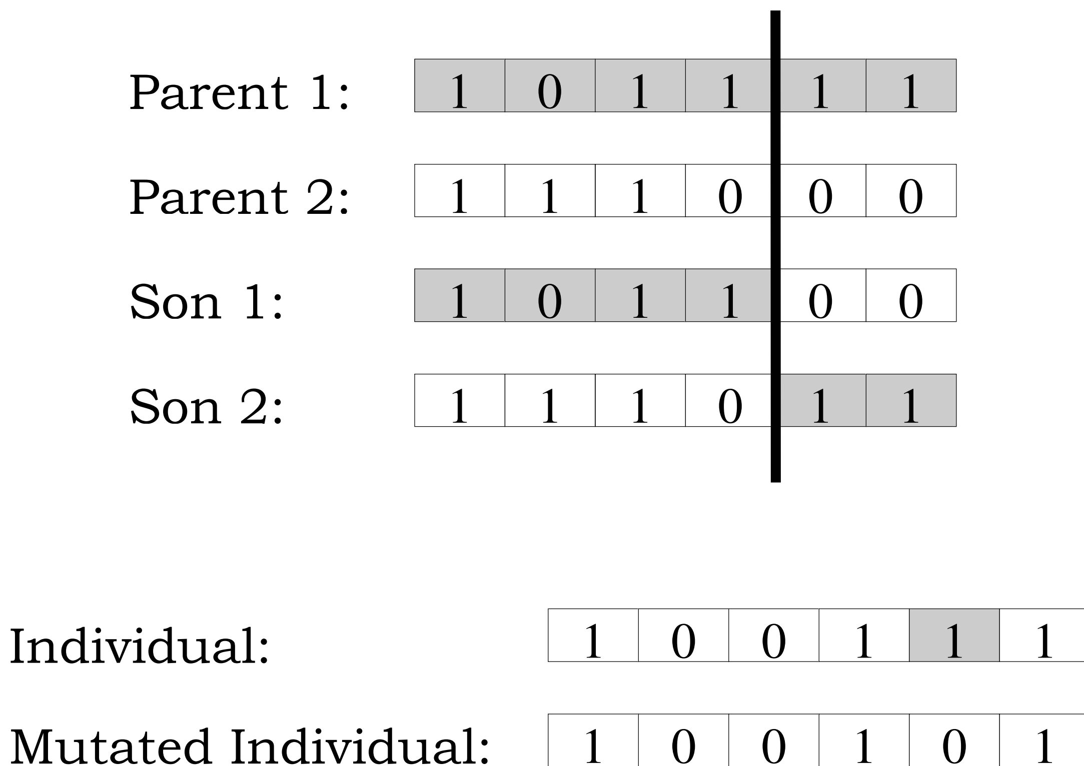


Figure 4.2 – Single-Point Crossover and Mutation for fixed –length binary string encoding

The idea behind multi-point crossover, uniform crossover, and indeed many of the variations on the crossover operator, is that parts of the chromosome representation that contribute to the most to the performance of a particular individual may not necessarily be contained in adjacent substrings. This helps to overcome the bias in single-point

crossover towards short substrings without requiring precise understanding of the significance of the individual bits in the individual representation. Further, the disruptive nature of multi-point and uniform crossover appears to encourage the exploration of the search space, rather than favouring the convergence to highly fit individuals early in the search, thus making the search more robust.

4.3.4 Mutation

After recombination every offspring undergoes mutation. Offspring are mutated by small perturbations with low probability. For binary valued individuals, mutation means flipping of variable values (see also Figure 4.2). For every individual the variable value to change is chosen uniform at random. The effect of the mutation depends on the actual coding.

4.3.5 Reinsertion

Once the offspring have been produced by selection, recombination and mutation of individuals from the old population, the fitness of the offspring may be determined. If less offspring are produced than the size of the original population, the offspring have to be reinserted into the old population. Similarly, if not all offspring are to be used at each generation or if more offspring are generated than needed a reinsertion scheme must be used to determine which individuals should be inserted into the new population.
Different schemes of reinsertion exist:

- Produce as many offspring as parents and replace all parents by the offspring (pure reinsertion).
- Produce less offspring than parents and replace parents uniformly at random (uniform reinsertion).
- Produce less offspring than parents and replace the worst parents (elitist reinsertion).
- Produce more offspring than needed for reinsertion and reinsert only the best offspring (fitness-based reinsertion).

Pure reinsertion is the simplest reinsertion scheme. Every individual lives one generation only. However, it is very likely, that very good individuals are replaced without producing better offspring and thus good information is lost.
The elitist reinsertion prevents this losing of information. At each generation, a given number of the least fit parents are replaced by the same number of offspring.
The fitness-based reinsertion scheme implements a truncation selection between offspring before inserting them into the population (i.e. before they can participate in the reproduction process). On the other hand the best individuals can live many generations. However, every generation some new individuals are inserted.
In the uniform reinsertion scheme, it is not checked whether the parents are replaced by better or worse offspring. Because parents may be replaced by offspring with a lower fitness, the average fitness of the population can decrease.

4.3.6 Termination

Because the GA is a stochastic search method, it is difficult to formally specify convergence criteria. As the fitness of a population may remain static for a number of generations before a superior individual is found, the application of conventional termination criteria becomes problematic. A common practice is to terminate the GA after a pre-specified number of generations and then test the quality of the best members of the population against the problem definition. If no acceptable solutions are found, the GA may be restarted or a fresh search initiated.

4.4 Overview on Evolutionary Algorithms

Evolutionary Algorithms is an umbrella term used to describe computer-based solving systems, which use computational models of natural evolution as key elements in their design. Genetic algorithms are the most well known among evolutionary algorithms, but other techniques exist as well. In the course of years, Genetic Algorithms have incorporated good features that in origin were peculiar of other techniques. In the following we give a broad overview on evolutionary algorithms.

4.4.1 Evolutionary Programming

Evolutionary Programming (EP), arising from the desire to generate machine intelligence, typically uses a representation tailored to the problem domain. For example, in a numerical optimisation vectors of real-valued numbers would be used and in combinatorial problems would employ ordered lists. Given a population size of N, all N individuals are selected as parents and a representation specific mutation operator is used to generate N offspring. The N offspring would then be evaluated and the next generation selected using a fitness-based probabilistic function on these 2N individuals. The mutation operator in EP is often adaptive and different adaptation rates may be used for each decision variable within an individual.

4.4.2 Evolution Strategy

Evolution Strategy (ES) employs real-valued representation for individuals, recombination, mutation and selection. Parents are randomly selected and recombination and mutation used to produce more than N offspring. Selection is then used to select either the N best offspring or the N best individuals from the parents and offspring to make up the next generation. Unlike EP, recombination is an important operator in ES.

4.4.3 Classic Genetic Algorithms

Genetic Algorithms (GA) traditionally use the more domain independent binary representation although other representations are now being employed. Selection of parents is probabilistic, based on a fitness function and N children are produced from the N parents using mutation and recombination operators. These offspring are then the new population. In GA recombination is considered to be the primary operator and mutation a background process.

4.4.4 Genetic Programming

Genetic Programming (GP) applies evolutionary search to the space of tree structures, which may be interpreted as computer programs in a language suitable to modification by mutation and recombination. The dominant approach to genetic programming uses (a

subset of) LISP programs as genotype space, but other programming languages including machine code are also used.

4.4.5 Order-Based Genetic Algorithms

Order-based genetic algorithms were proposed to search the space of permutation directly rather than using complex decoding functions for mapping binary strings to permutations and preserving feasible permutation under mutation and crossover. They apply specialised recombination (such as order crossover or partially matched crossover) and mutation operators (such as random exchanges of two elements of the permutation), which preserve permutations.

4.4.6 Parallel Genetic Algorithms

The GA may be parallelised in a number of ways. Indeed, there are numerous variations on parallel GA, many of which are very different from the original GA. Most of the major differences are encountered in the population structure and the method of selecting individuals for reproduction.

The motivation for exploring parallel GA is manifold. One may wish to improve speed and efficiency by employing a parallel computer, apply the GA to larger problems or try to follow the biological metaphor more closely by introducing structure and geographic location into a population. The benefits of using parallel GA, even when run on a sequential machine, can be more than just a speed up in the execution time.

Due to the structure of the population and the use of local selection rules, parallel GA offer an attractive mechanism for allowing diversity to exist within a population without affecting the convergence characteristics of the GA.

There are three main ways of exploiting parallelism in genetic algorithms. The first is the *Global Genetic Algorithm*, which treats the entire population as a single breeding unit and aims to exploit the algorithmic parallelism inherent in the GA. A second possibility is the *Migration Genetic Algorithm*, which divide the population into a number of subpopulations, each of which is treated as a separate breeding unit under the control of a conventional GA. To encourage the proliferation of good genetic material throughout the whole population, individuals migrate between the subpopulations from time to time. The third and last way is the *Diffusion Genetic Algorithm*, which treats

each individual as a separate breeding unit, the individuals it may mate with being selected from within a small local neighbourhood. The use of local selection and reproduction rules leads to a continuous diffusion of individuals over the population. It can be shown that this last model of parallel genetic algorithms is a subclass of cellular automata.

4.5 Genetic Algorithms as Meta-Heuristics

At first, genetic algorithms were created as a generic tool useful for many difficult-to-solve problems. Much of the early works of genetic algorithms used a universal internal representation involving fixed-length binary strings with binary genetic operators to operate in a domain-independent fashion at the level without any knowledge of the phenotypic interpretation of the strings. This universality was reflected in a strong emphasis on the design of robust adaptive systems with a broad range of applications.

However, simple GAs are difficult to apply directly and successfully into many difficult-to-solve optimisation problems. Various non-standard implementations have been created for particular problems in which genetic algorithms are used as *meta-heuristics* [3, 10]. In this new perspective, Genetic Algorithms are very effective at performing global search (in probability) and provide us a great flexibility to hybridise with domain-dependent heuristics to make an efficient implementation for a specific problem. Section 4.7 proposes one of the most common forms of hybrid genetic algorithms, *Genetic Local Search*. In a wider sense, genetic algorithms can also be seen as a framework, which has to be adapted to the problem at hand designing a suitable representation together with a set of genetic operators applicable to it. In Section 4.6, problems arising during the *design of the encoding* for a given problem are discussed.

4.6 Encoding Problem

How to encode a solution of the problem into a chromosome is a key issue for genetic algorithms. In Holland's work, encoding is carried out using binary strings. For many GA applications, especially for problems from industrial engineering world, the simple GA was difficult to apply directly because the binary string is not a natural coding. Various non-string encoding techniques have been created for particular problems, for

example real numbering coding for constrained optimisation problems and integer coding for combinatorial optimisation problems. Choosing an appropriate representation of candidate solutions to the problem at hand is the foundation for applying genetic algorithms to solve real world problems and it affects all the subsequent steps of genetic algorithms. For any application case, it is necessary to perform analysis carefully to ensure an appropriate representation of solutions together with meaningful and problem-specific genetic operators.

4.6.1 Coding Space and Solution Space

One of the basic features of genetic algorithms is that they work on coding space and solution space alternatively: genetic operations work on coding space (chromosomes), while evaluation and selection work on solution space. Natural selection is the link between chromosomes and the performance of their decoded solutions. For the non-string coding approach, three critical issues emerged concerning with the encoding and decoding between chromosomes and solutions (the mapping between phenotype and genotype):

- Feasibility of chromosomes
- Legality of chromosomes
- Uniqueness of mapping

4.6.2 Feasibility and Legality

Feasibility refers to the phenomenon of whether a solution decoded from a chromosome lies in the feasible region of a given problem. Legality refers to the phenomenon of whether a chromosome represents a solution to a given problem.

The infeasibility of chromosomes originates from the nature of the constrained optimisation problem. All methods, conventional ones or genetic algorithms, must handle the constraints. For many optimisation problems, the feasible region can be represented as a system of equalities or inequalities (linear or non-linear). For such cases, many efficient penalty methods have been proposed to handle infeasible chromosomes. In constrained optimisation problems, the optimum typically occurs at

the boundary between feasible and infeasible areas. The penalty approach will force the genetic search to approach the optimum from both feasible and infeasible regions.
The illegality of chromosomes originates from the nature of encoding techniques. For many combinatorial optimisation problems, problem-specific encodings are used and such encodings usually yield to illegal offspring by a simple one-cut-point crossover operation. Because an illegal chromosome cannot be decoded to a solution, it means that such chromosomes cannot be evaluated; thus the penalty approach is inapplicable to this situation. Repairing techniques are usually adopted to convert an illegal chromosome to a legal one. For many combinatorial optimisation problems, it is relatively easy to repair an infeasible or illegal chromosome; the repair strategy does indeed surpass other strategies such as rejecting strategy or penalising strategy.

4.6.3 Mapping

The mapping from chromosomes to solutions (decoding) may belong to one of the following three cases:

- 1-to-1 mapping
- n-to-1 mapping
- 1-to-n mapping

The 1-to-1 mapping is the best one among three cases. The 1-to-n mapping (a single solution corresponds to n chromosomes) is undesired because the coding space is bigger than the solution space and therefore the search may be slowed down. The n-to-1 mapping (more than one solution may correspond to the same chromosome) is undesired as well because a chromosome doesn’t refer to a precise solution, and therefore it can’t be easily evaluated. We need to consider these problems carefully when designing a new coding so as to build an effective genetic algorithm.

4.7 Genetic Local Search

On one hand, problems from combinatorial optimisation are well within the scope of genetic algorithms. Nevertheless, compared to conventional heuristics, genetic

algorithms are not well-suited for fine-tuning structures which are very close to optimal solutions. Therefore it is essential to incorporate conventional heuristics into genetic algorithms to construct a more competitive algorithm.

On the other hand, in general the best solution found by local search algorithm may depend on the initial solution used. However, a multi-start scheme may overcome this problem. As a further refinement, the effectiveness of multi-start iterative approach may be improved by using the information available from the solutions obtained in the individual cycles. Following this line, several authors have proposed variants of local search algorithms, using ideas from population genetics.

A Genetic Local Search (GLS) algorithm [1] consists of a basic Genetic Algorithm with the addition of a local search optimisation phase applied to every new individual created either in the initial population or during the evolutionary process.

We can give to the GLS algorithm a dual interpretation. On one hand, we can see it as a Genetic Algorithm where Local Search is intended as a smart mutation mechanism. On the other hand, we can see it as structured multi-start mechanism for Local Search where the Genetic Algorithm plays the role of the structure.

However, by seeing the hybrid approach as a whole, Genetic Algorithms are used to perform global exploration among population while Local Search is used to perform local exploitation around chromosomes. Because of the complementary properties of Genetic Algorithms and Local Search, which mutually compensate their points of weakness, the hybrid approach often outperforms either method operating alone. A Genetic Local Search outline is presented in Chapter 5.

4.8 Genetic Algorithms and Scheduling Problems

Scheduling problems are highly computationally complex problems, which often have highly domain-dependent features (i.e. they are strongly constrained). Genetic Algorithms have been successfully applied to ordering problems, as the travelling salesman problem (TSP), and to scheduling problems like Flow-Shop, Job-Shop, Open-Shop and more general machine scheduling problems.

During the past years a number of representations for job-shop scheduling problem have been proposed. In the following we list of the most common representations:

- Operation-Based Representation [3]
- Preference-List-Based Representation [4]
- Disjunctive-Graph-Based Representation [12]
- Gantt-Chart-Based Representation [8]

These representations can be classified into two basic encoding approaches, direct approach and indirect approach. In the direct approach, a schedule (the solution of JSSP) is encoded into a chromosome, and genetic algorithms are used to evolve those chromosomes to determine a better schedule. In the indirect approach, a sequence of dispatching rules for job assignment (i.e. a schedule) is encoded into a chromosome, and genetic algorithms are used to evolve those chromosomes to determine a better sequence of dispatching rules. A schedule then is constructed with the sequence of dispatching rules. Other ways to classify different representations for job-shop are considered in the following:

Lamarckian Property. The Lamarckian property for a coding technique concerns the issue whether a chromosome can pass on its goodness to future populations through a common genetic operation.

Complexity of Decoder. In principle, there are an infinite number of feasible schedules for a given job-shop scheduling problem. Generally, three kinds of schedules can be distinguished as follows: semiactive schedule, active schedule, and nondelay schedule. An optimal schedule is within the set of active schedules. The set of nondelay schedules is smaller than that of active schedules, but there is no guarantee that it will contain an optimum. Thus we hope that a schedule decoded from a chromosome would be an active one. All of the coding techniques proposed for the job-shop scheduling problem can generate an active schedule by use of the decoder. The degree of complexity of the decoder can be classified into the following four levels:

- No decoder. All burdens are put on genetic operators.
- Simple mapping relation.
- Simple heuristic.
- Complex heuristic.

The Property of Coding Space and Mapping. As we can see for any coding technique, through a decoding procedure, a chromosome always corresponds to a legal, feasible, and active schedule and the mapping relation is one-to-one. But the coding space of these methods can be classified into two classes: one contains only feasible solution space, whereas the other includes illegal solution space.
In addition, the spaces of some representations may correspond only to the partial space of whole solution space.

Memory Requirements. For an n-job m-machine problem, if we define the standard length for a chromosome as with n x m genes, the codings for the job-shop problem can be classified as follows:

- The coding length is less than standard length. Note that, in this case, the coding space corresponds to only part of the solution space, but not the whole solution space. Thus there is no guarantee for finding the optima with the coding techniques.
- The coding length is larger than standard length. The representation is highly redundant.
- The coding length is equal to standard length.

4.9 Theory Behind Genetic Algorithms

There is no accepted "general theory" which explains exactly why GAs have the properties they do. Nevertheless, several hypotheses have been put forward which can partially explain the success of GAs [7]. These can be used to help us implement good GA applications.

4.9.1 Schema Theorem

Holland's schema theorem was the first rigorous explanation of how GAs work. A schema is a pattern of gene values which may be represented (in a binary coding) by a string of characters in the alphabet {0, 1, #}. A particular chromosome is said to contain a particular schema if it matches that schemata, with the "#" symbol matching anything. So, for example, the chromosome "1010" contains, among others, the schemata "10##", "#0#0", "##1#" and "101#". The order of a schema is the number of symbols different from "#" which that schema contains (2, 2, 1, 3 respectively in the example). The defining length of a schema is the distance between the outermost symbols different from "#" (2, 3, 1, 3 respectively in the example).

The schema theorem explains the power of the GA in terms of how schemata are processed. Individuals in the population are given opportunities to reproduce and produce offspring. The number of such opportunities an individual receives is proportional to its fitness, hence the better individuals contribute more of their genes to the next generation. It is assumed that an individual's high fitness is due to the fact that it contains good schemata. By passing on more of these good schemata to the next generation, we increase the likelihood of finding even better solutions.

Holland showed that the optimum way to explore the search space is to allocate reproductive opportunities to individuals in proportion to their fitness relative to the rest of population. In this way, good schemata receive an exponentially increasing number of trials in successive generations. This is called the schema theorem. He also showed that, since each individual contains a great many different schemata, the number of schemata which are effectively being processed in each generation is in the order of n^3, where n is the population size. This property is known as implicit parallelism, and is one of the explanations for the good performance of GAs.

4.9.2 Building Block Hypothesis

According to Goldberg [7], the power of GA lies in its being able to find good building blocks. These are schemata of short defining length consisting of bits which work well together, and tend to lead to improved performance when incorporated into an individual. A successful coding scheme is one which encourages the formation of building blocks by ensuring that:

1. Related genes are close together on the chromosome
2. There is a little interaction between genes

Interaction between genes means that the contribution of a gene to the fitness depends on the value of other genes in the chromosome. In fact there is always some interaction between genes in multi-modal fitness functions. This is significant because multi-modal functions are the only sort of any real interest in GA research, since uni-modal functions can be solved more easily using simpler methods.

If these rules are observed, then a GA will be as effective as predicted by the schema theorem. Unfortunately, conditions (1) and (2) are not always easy to meet. Genes may be related in ways, which do not allow all closely related ones to be placed close together in a one-dimensional string. In many cases, the exact nature of the relationship between the genes may not be known to the programmer, so even if there are only simple relationships, it may still be impossible to arrange the coding to reflect this.

Condition (2) is a precondition for (1). If the contribution to overall fitness of each gene were independent of all other genes, then it would be possible to solve the problem by hill-climbing on each gene in turn. Clearly this is not possible in general. If we can ensure that each gene only interacts with a small number of other genes and these can be placed together on the chromosome, then condition (1) and (2) can be met. But if there is a lot of interaction between genes, then neither condition can be met. Clearly, we should try to design coding schemes to conform to Goldberg's recommendations, since this will ensure that the GA will work as well as possible.

Chapter 5

Genetic Algorithms and Job Shop

5.1 Introduction

In this chapter we are going to present the main contribution of this project, that is, a new genetic representation for job shop together with a suitable class of recombination operators. First, we consider the encoding problems that arise when we have to deal with job shop. Moreover, we try to make clear what is the effect of genetic operators when applied to permutation representation. The infeasibility problem is here of central importance, therefore it is identified and discussed. Subsequently, we propose the *string representation* that, coupled with a particular class of recombination operators, guarantees the genetic search to represent all and only the feasible solutions and that guarantees the transmission of meaningful characteristics to the offspring solutions. The definition of a suitable class of recombination operators follows. Next, a set of potential recombination operators, which show interesting characteristics, is introduced. Finally, we propose a genetic local search algorithm making use of the genetic framework introduced.

5.2 Encoding Problems and Job Shop

5.2.1 Appeal to Feasibility

In order to apply GAs to a particular problem we have to encode a generic solution of the problem into a chromosome. How to encode a solution is a key-issue for the success of GAs [3]. Canonical GAs use binary encoding of individuals on fixed-length strings. Such a representation is not naturally suited for ordering problems such as the

Travelling Salesman Problem and the JSSP, because no direct and efficient way has been found to map all possible solutions into binary strings [17].

The main difficulty in choosing a proper representation for highly constrained combinatorial optimisation problems such as JSSP is dealing with the infeasibility of the solutions produced during the evolutionary process. This problem is typically addressed by modifying the breeding operators, associating them with repair methods, or providing penalties on infeasible solutions in the fitness function, or yet discarding infeasible solutions when created. However, the use of penalty functions or a rejecting strategy is inefficient for JSSP because the space of feasible schedules is very small compared to the space of possible schedules, therefore the GA will waste most of its time producing and/or processing infeasible solutions. Repairing techniques are a better choice for many combinatorial optimisation problems since they are easy to apply and surpass strategies such as rejecting strategy and penalising strategy [14]. However, whereas it is possible, the most efficient and direct method remains to embed constraints in the coding of individuals. Thus, a very important issue in building a genetic algorithm for JSSP is to devise an appropriate representation of solutions together with a problem-specific genetic operator so that all chromosomes generated in either the initial phase or the evolutionary process will produce feasible schedules. This is a crucial phase that affects all the subsequent steps of GAs.

5.2.2 Causes of Infeasibility

Let us spend few words on the nature of infeasibility. On the basis of what kind of solution representation for the JSSP we consider, two different causes of infeasibility may occur:

- Schedules non-respecting all job precedence constraints
- Solutions with cycles

Because of the existence of the precedence constraints of operations on jobs, if we consider a solution representation which doesn't presuppose a fixed order of operations on jobs, but rather which can freely dispose operations both on machines and jobs, then mismatches between the order of operations encoded in a generic chromosome and the

prescribed order of operations on jobs may arise. Therefore this is a first cause of infeasibility.

In a schedule, two generic operations are allowed to be either processed in parallel (there is no precedence among them) or processed sequentially (in this case, one precedes the other one). What is not possible is that one operation both precedes and follows the other one. If we consider a representation of solutions which allow to encode precedence conflicts between operations like the one just mentioned (i.e. cycling solutions), then we encounter the second cause of infeasibility.

5.2.3 Encoding and Constraints

In the previous chapter we have already discuss about the problems which may arise when we try to define a mapping between solution space and coding space. These are mainly due to the constrained aspect of the problem at hand. Here we are interested in figure out the properties of the relationship among coding space and solution space, together with the related problems, for the particular case of JSSP.

In order to apply the GA framework we need to define an encoding method to map the search space of all possible solutions into a set of finite chromosomes. Because of the constraints, it is not always trivial to get a perfect mapping; we can meet the following problems related with the expressiveness of the encoding:

- Infeasible solutions can be represented
- Illegal solutions can be represented
- The optimum solution cannot be represented

An infeasible solution and an illegal solution are both non-valid. The former respects the constraints of the problem but it is a non-acceptable solution because of the nature of the problem. The latter doesn't respect the constraints. However, this distinction is quite vague and very depending on the particular problem applied. In the particular case of JSSP, we find convenient to define infeasible solutions those presenting cycles and illegal solutions those non-respecting the job constraints. In Figure 5.1 five cases of interest are illustrated.

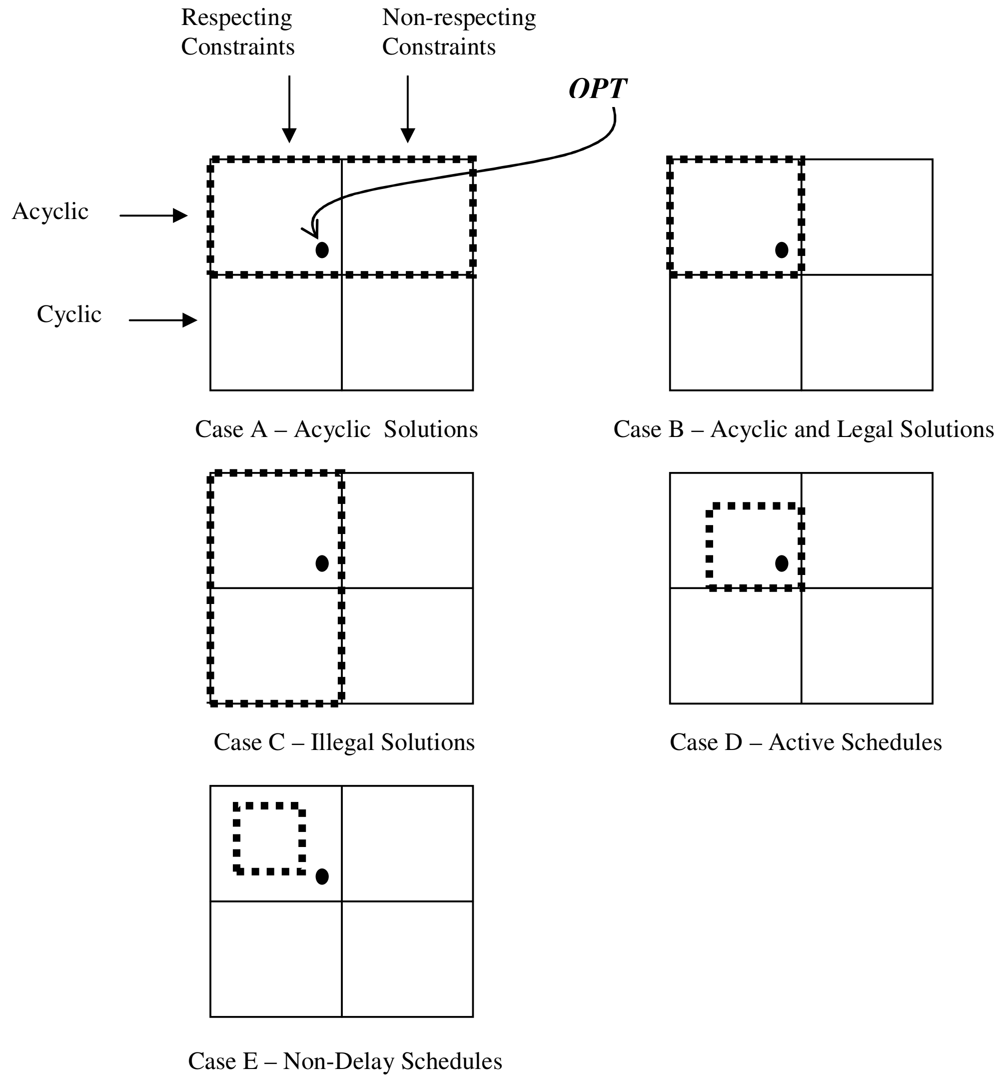


Figure 5.1 – Relationship among Representation Covering and Type of Solutions

5.2.4 Genetic Operators and Order Relationship

JSSP is a problem for which the order relationship among operations seems to be important. Here we want to clarify the effect of recombination operators under an ordering point of view. Let us consider two generic operations, *A* and *B*. Those operations are present in both of parents and in both of children produced by applying a recombination operator. On the basis of the order relationship among *A* and *B* in the parents and those ones in the children, we can distinguish four cases of interest, which are illustrated in Figure 5.2.

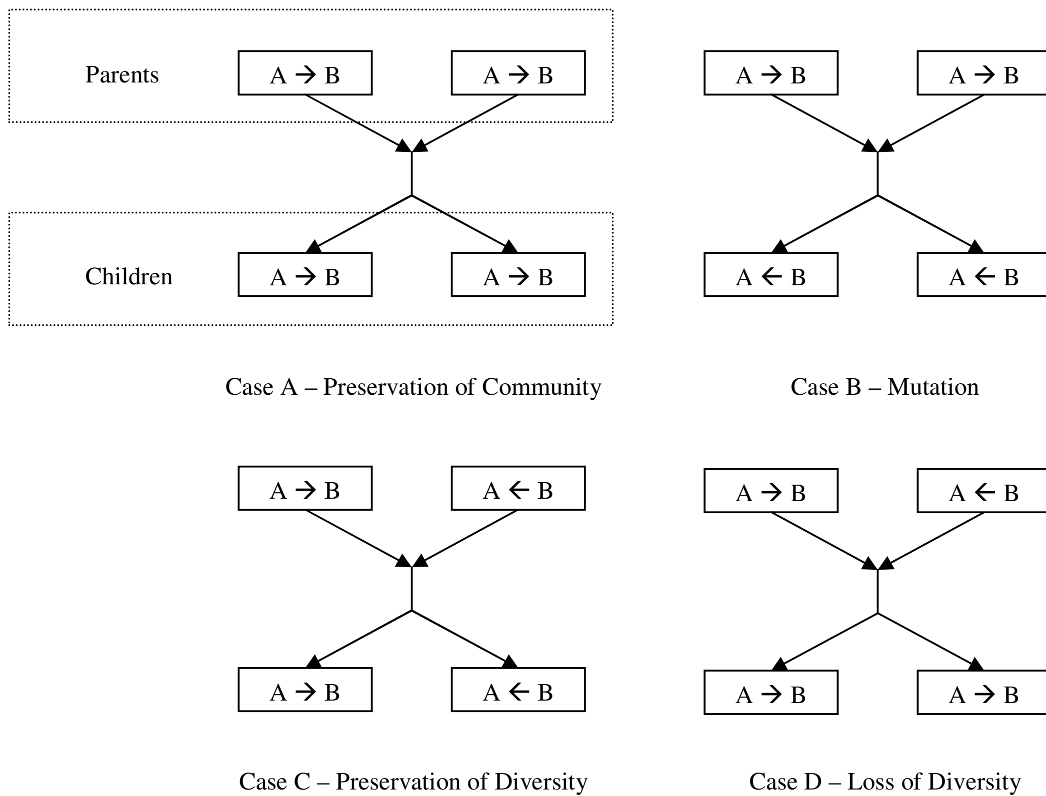


Figure 5.2 – Effects of Recombination on Order Relationship

5.3 A New Genetic Representation for Job Shop

In this section we propose a representation and a particular class of recombination operators that together guarantee the genetic search to cover all and only the space of feasible solutions and the transmission of meaningful characteristics to the offspring solutions.

In order to avoid both kinds of infeasibility presented above, we define a class of recombination operators that solves the problem with job constraints and a representation that solves the problem concerning cycling solutions. More in detail, we will see that only solutions without cycles can be represented, thus eliminating the cycling problem. Unfortunately the schedules so represented do not necessarily respect job precedence constraints. However, to manage this second kind of unfeasibility, it suffices initialising the evolutionary process with a population of schedules respecting all job precedence constraints and applying recombination operators that leave the job precedence constraints invariant.

5.3.1 String Representation

In order to apply the GA framework, we need to define an encoding method to map the search space of all possible solutions into a set of finite chromosomes.

In the sequel we introduce the representation we will use. First, we will show, by means of an example, the relationship among a problem instance, represented by its disjunctive graph, a particular solution for that instance, represented by its solution graph, and our string coding for that solution. Afterward, we will present definitions and theorems that assure the validity of the representation proposed.

Figure 5.3 illustrates the derivation of the string representation. At the top of the figure there is a disjunctive graph for a three-job four-machine instance, already presented in chapter two. Just below the solution graph representing a feasible solution to the given instance of the problem is reported. It has been derived from the disjunctive graph by settling an orientation of all the disjunctive arcs having taken care to avoid the creation of cycles. In the solution graph, arrows correspond to precedence constraints among operations on jobs or machines. Dashed lines indicate that two operations don't have any precedence constraints.

Let us see things emphasising the precedence order relationship among operations. The disjunctive graph represents a particular instance of JSSP. We can see it as a partial order relationship among operations. The solution graph shown represents a specific solution of the above JSSP instance. We can see it still as a partial order relationship among operations, even if more constrained when compared to the relationship associated with the disjunctive graph. We can now force a complete order by imposing further precedence constraints and eliminating all the redundant arcs so that obtaining a linear sequence of operations, the string shown at the bottom of Figure 5.3, which is the encoding of a solution we will use.

In the string is present all information we need to decode it into an actual schedule. Since we know a priori (from the problem instance) the machine which a given operation belongs to, the sequence of operations on each machine is easily determinable from the string. The idea is to scan the string from left to right, extract all the operations of a given machine and sequencing them keeping the same order. Considering again our example, if we apply the decoding procedure just described to the string at the bottom of Figure 5.3, it is easy to see that we obtain exactly the same sequences of operations on machines of the solution graph. The lower part of Figure 5.4 shows that the string representation contains all information we need to reconstruct the order of operations on each machine.

A peculiarity of the string representation is that it doesn't admit cyclic solutions. It is therefore not subject to the second kind of infeasibility we have discussed previously. However, we can notice that a string codifies both information about the solution it represents (precedence constraints on machines) and information about the instance of the problem. In fact, the upper part of Figure 5.4 shows that the string representation contains also all information about precedence constraints on jobs. This implies that a generic string may represent a solution, which does not respect the precedence constraints on jobs, therefore we still have to deal with this kind of infeasibility, that is the first kind discussed previously.

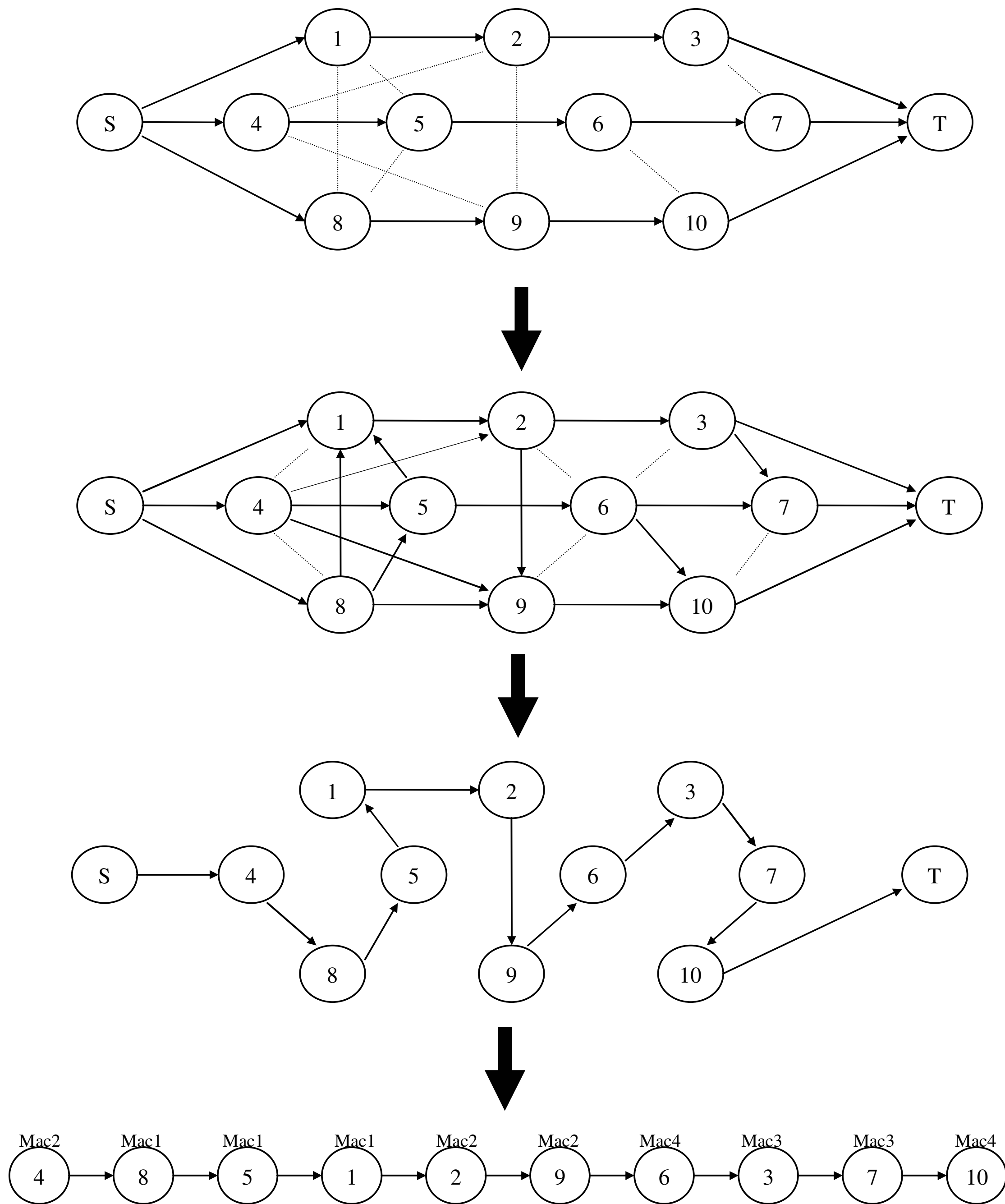


Figure 5.3 – Derivation of the String Representation (Complete precedence order among all operations)

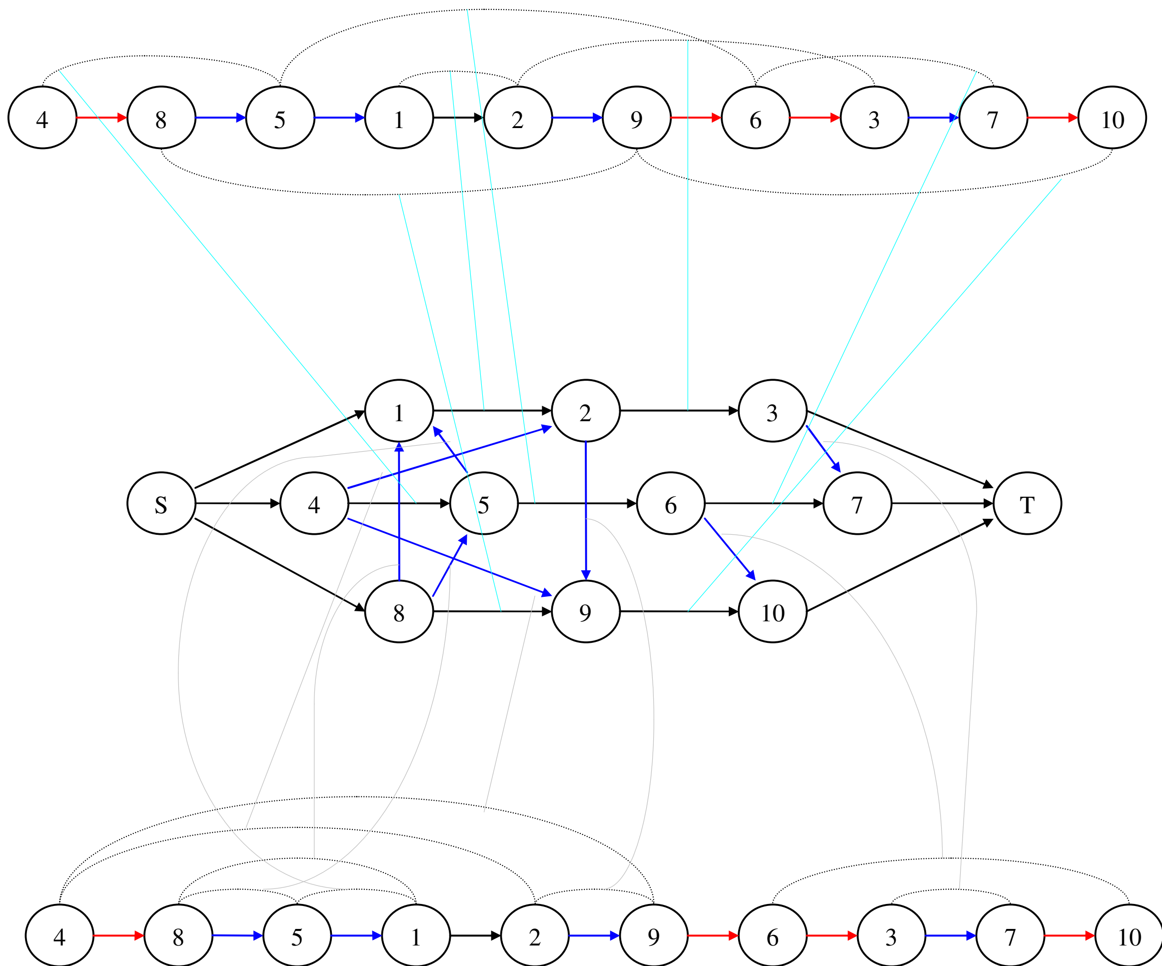


Figure 5.4 – Decoding Information from a String

5.3.2 Formal Definition of String and Coding/Decoding Theorems

In the following we give the formal definition of string representation. Then, in order to show that the string representation is a valid encoding for schedules, we formulate two theorems.

Definition 1. String Representation.

Let us consider three finite sets, a set J of jobs, a set M of machines and a set O of operations. For each operation a there is a job $j(a)$ in J to which it belongs, a machine $m(a)$ in M on which it must be processed and a processing time $d(a)$. Furthermore for each operation a its successor in the job is given by $sj(a)$, except for the last operation in a job. The representation of a solution is a string consisting of a permutation of all operations in O, i.e. an element of the set:

$StrRep = \{ s \in O^n \mid n = |O| \text{ and } \forall\, i, j \text{ with } 1 \leq i < j \leq n: s(i) \neq s(j) \}$

Now we can define legal strings. Formal for s in $StrRep$:

$Legal(s) \Leftrightarrow \forall\, a, sj(a) \in O: a \sim< sj(a)$

where $a \sim< b$ means: a occurs before b in the string s.

Theorem 1. (Feasible Solution → Legal String)

Every feasible solution can be represented by a legal string. More than one legal string corresponding to the same feasible solution may exist.

Proof.

Every feasible solution can be represented by an acyclic solution graph, say C.

Every acyclic solution graph S can be transformed in a legal string by means of the following construction procedure:

1. Set S as the current graph C
2. Calculate the transitive closure graph TC of the current graph C
3. WHILE the transitive closure TC doesn't define a total order in O DO
    4. Select two nodes in O still not linked by an arc in TC
    5. Link them by an arc obtaining a new acyclic graph that becomes the new current graph C
    6. Calculate the transitive closure graph TC of the current graph C
7. Convert the transitive closure graph TC in its corresponding string Str

The previous procedure:

- always produces a total order in *O* and never a cyclic graph. Therefore the conversion of *TC* in *Str* in step 7 is immediate
- is non-deterministic in step 4 and 5 and consequently it may produce different strings starting from the same acyclic solution graph *S*
- always produces a legal string since the initial solution graph *S* is still a sub-graph of the final transitive closure graph *TC*

■

Theorem 2. (Legal String → Feasible Solution)

Every legal string corresponds exactly to one feasible solution.

Proof.

A generic legal string *Str* can be interpreted as a complete order relationship among operations and consequently can be associated with an acyclic graph *TC*.

Let us consider the set of (directed) arcs *A* and the set of (undirected) edges *E* defined in the disjunctive graph. By eliminating from *TC* every arc not in *A* *E*, we obtain a new graph *S* representing the solution.

Moreover since arcs of the form [*a*, *sj(a)*] are present in the graph *TC* and these arcs are not removed in the elimination process, the resulting solution graph *S* has the correct job arcs, i.e. it corresponds to a feasible solution.

■

5.3.3 Recombination Class

In order to cope with the unfeasibility regarding job precedence constraints we propose the following requirement on the recombination operators allowed that guarantees both the respect of job constraints and the transmission of meaningful characteristics.

Definition 2. Feasibility Requirement for Recombination.

We say that a generic recombination operator for the string representation is feasible, if for every generic pair of operations *a* and *b* such as *a* ~<*b* in both parent strings then

also $a \sim< b$ must hold in the child strings produced by its application to the parent strings.

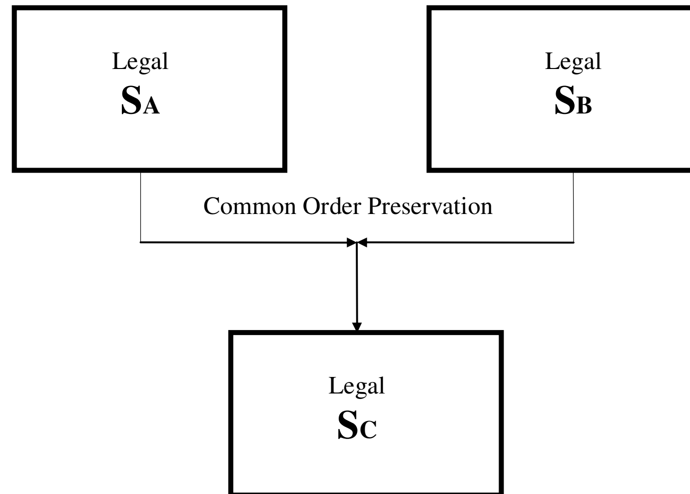


Figure 5.5 – Preservation of Legality through Recombination

Theorem 3. (Legal String + Legal String → Legal String)

By recombining legal strings following the feasibility requirement for recombination, we still obtain a legal string. See Figure 5.5.

Proof.

Let s and t be two legal parent strings. Let offspring u be obtained by a recombination that respects the feasibility requirement. By definition we have to show that $a \sim< sj(a)$ for all operations a in string u. Since s and t are legal strings, this property holds for s and t. From the feasibility requirement we immediately conclude that also $a \sim< sj(a)$ for all operations a in string u.

■

In the following we propose a recombination operator that respects the feasibility requirement.

Definition 3. Recombination Operator.
Let *SEQ* be a vector of n elements randomly chosen in the set {1, 2, 3, 4}, where $n=|O|$. We use it as input of the following procedure that produces from two parent strings a single offspring string:

1. Initialise the left scan pointers *PT1* and *PT2* at the beginning (on the left side) of parent strings *PAR1* and *PAR2*. Initialise the right scan pointers *PT3* and *PT4* at the end (on the right side) of the parent strings *PAR1* and *PAR2*. Let the result (son) be a string consisting of n blanks. Initialise the left write pointer *P1* at the beginning of the result string and initialise the right write pointer *P2* at the end of the result string. Set all operations as unmarked.
2. Consider the first number appearing in the sequence *SEQ*
3. Slide the corresponding scan pointer to the first unmarked operation (left scan pointers slide from left to right, right scan pointers slide from right to left)
4. If the pointer in step 3 was a left pointer, copy the current operation at the left write pointer and increase that pointer by 1. Otherwise copy the current operation at the right write pointer and decrease it by 1. Mark the current operation in the parents as already processed (see Figure 5.6 and Figure 5.7)
5. Take out the number at the beginning of *SEQ*
6. If *SEQ* is empty then stop otherwise go to step 2

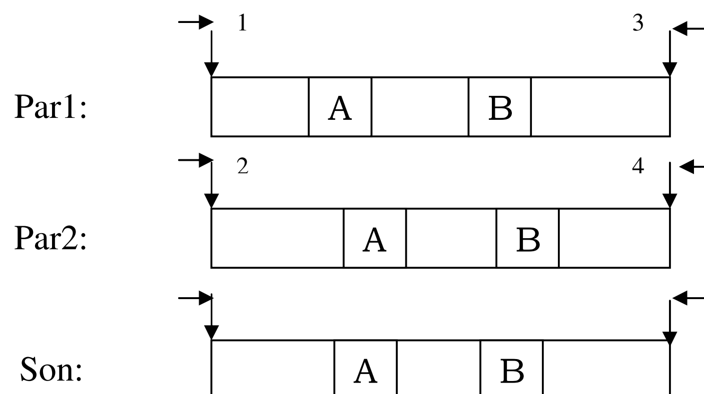


Figure 5.6 – Common order preserving recombination

Theorem 4. Validity of Recombination Operator.

The recombination operator defined above respects the feasibility requirement.

Proof.

Let us consider the two parent strings in Figure 4. To transmit a generic operation from a parent to the son, that operation must be reached by one of the four scan pointers (indicated in Figure 5.6 with numbers from 1 to 4). Therefore, to transmit operations *A* and *B* both must be reached by a pointer, one for each. A pair of pointers (*x*, *y*) defines a way to transmit operations *A* and *B* by means of the following procedure:

- First, the *x* pointer slides in its prescribed direction until it reaches operation *A* or operation *B* and transmits it to the son
- Then, the *y* pointer slides in its prescribed direction until it reaches the operation which among *A* and *B* is not yet assigned to the son

In Table 5.1 all possible pairs of pointers are grouped in four classes of equivalence following two lines of symmetry.

	Same Parent	*Different Parent*
Same Side	(1, 1) (2, 2) (3, 3) (4, 4)	(1, 2) (2, 1) (3, 4) (4, 3)
Different Side	(1, 3) (3, 1) (2, 4) (4, 2)	(1, 4) (4, 1) (2, 3) (3, 2)

Table 5.1 – Symmetries of the recombination

Let us consider only one pair for every class, since the other pair in the same class will produce the same result:

- Case (1, 1). The pointer 1 gets first *A* and puts it in the son. The same pointer then gets *B* and puts it in the son more to the right respect to *A*. This is because the pointer used in the son slides in the same direction as pointer 1. We obtain in this way *A*~<*B* in the son string.

- Case (1, 2). The pointer 1 gets *A* and puts it in the son by the left pointer. Later, the pointer 2 meets *A* and skips it, then it gets *B* and transmits it to the son using the left pointer and consequently we obtain *A~<B* in the son string.
- Case (1, 3). The pointer 1 gets *A* and the pointer 3 gets *B*. *A* is posed in the son more to the left respect to *B* because *A* is inserted using the left pointer in the son, *B* using the right pointer and the write pointers cannot cross each other. Then, we get *A~<B* in the son string.
- Case (1, 4). First, the pointer 1 gets *A* and put it in the son by the left write pointer. Then, the pointer 4 gets *B* and put it in the son by the right write pointer. As the write pointers cannot cross each others, then it must be *A~<B* in the son string.

5.4 Recombination Operators

The recombination operator proposed is very general indeed. It has four degrees of freedom (the four pointers) we can drive following our wishes. We can combine them in many different configurations so that obtaining recombination operators with very different behaviours. For example, we can think to inhibit a generic combination of two pointers letting free to move only the remaining two. We can think also to bias the random sequences which drive the pointers in order to obtain something more close to the uniform crossover rather than to the one-point crossover or vice versa, biasing in this way the destruction capability of the recombination [7].

In fact we will present and compare in practice a set of recombination operators selected following the guidelines mentioned above. In the following we show all the recombination operators we test in the computational experiments. However, we report a complete description of only two of them, the most general one, already described in the previous section, and the one that has revealed to be the most effective in our computational experiments. The recombination operators we propose are listed in the following:

1. Multi-step Complete
2. Multi-step Parallel
3. Multi-step Crossed
4. Multi-step Merge-Split (MSX)
5. One-step Crossed Insertion
6. One-step Insertion

The recombination operator at the top of the list is the most general. Its formal description is given in the Definition 3, its behaviour in practice is illustrated through an example in Figure 5.7.

In our genetic algorithm the Merge and Split recombination (MSX) operator has been used. In Section 5.4.4 is illustrated by an example how MSX works in practice, its detailed definition follows.

The other recombination operators are illustrated by an example in the following sections. Now, we propose two different classifications of the set of recombination operators in order to point out the characteristics they combine together.

Considering only one child at a time, we can distinguish two classes:

1. Recombination operators which *keep separated* (do not mix but merely put in sequence) the operations inherited from different parents in the children
2. Recombination operators which merge (interleave) operations inherited from one parent with those ones inherited from the other parent in the children.

Multi-step Complete, Multi-step Parallel and Multi-step Merge-Split belong to the class 2, the others belong to class 1. Those in class 1 are less destructive than those in class 2.

Considering both children at the same time, we can distinguish other two classes:

1. Recombination operators which *redistribute perfectly* the operations of the parents in the children (if a child receives a given operation from one parent, the other child receives the same operation from the other parent)

2. Recombination operators which don't redistribute perfectly the operations of parents in the children (the same parent may transmit the same operation to both of children)

Multi-step Merge-Split, One-step Insertion and One-step Crossed insertion belong to the class 1, the others belong to class 2. Those in class 1 preserve diversity better than those in class 2.

5.4.1 Multi-Step Complete Recombination Operator

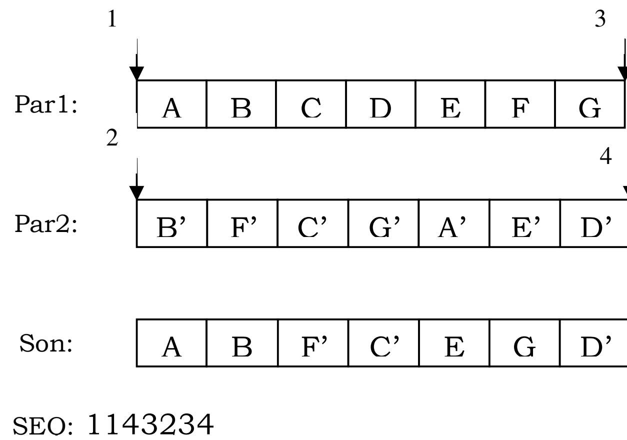
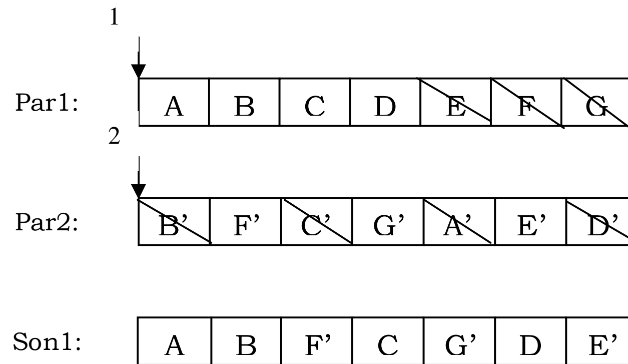


Figure 5.7 – Multi-Step Complete Recombination Operator

5.4.2 Multi-Step Parallel Recombination Operator



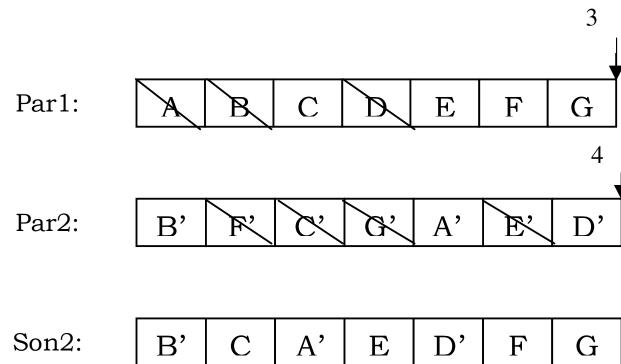
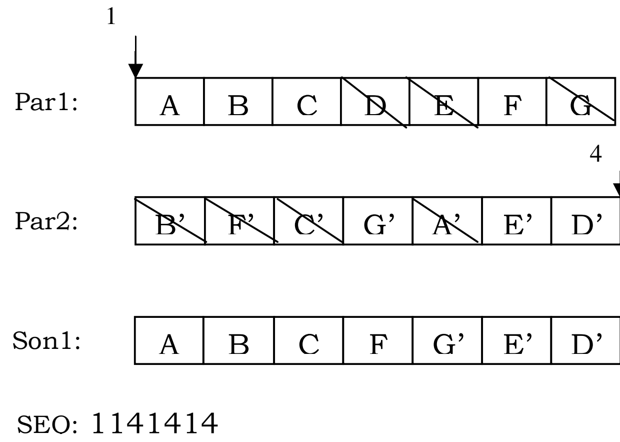


Figure 5.8 – Multi-Step Parallel Recombination Operator

5.4.3 Multi-Step Crossed Recombination Operator



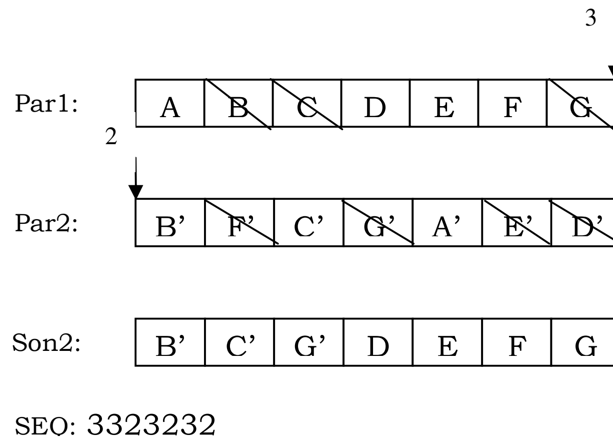


Figure 5.9 – Multi-Step Crossed Recombination Operator

5.4.4 Multi-Step Merge and Split Operator

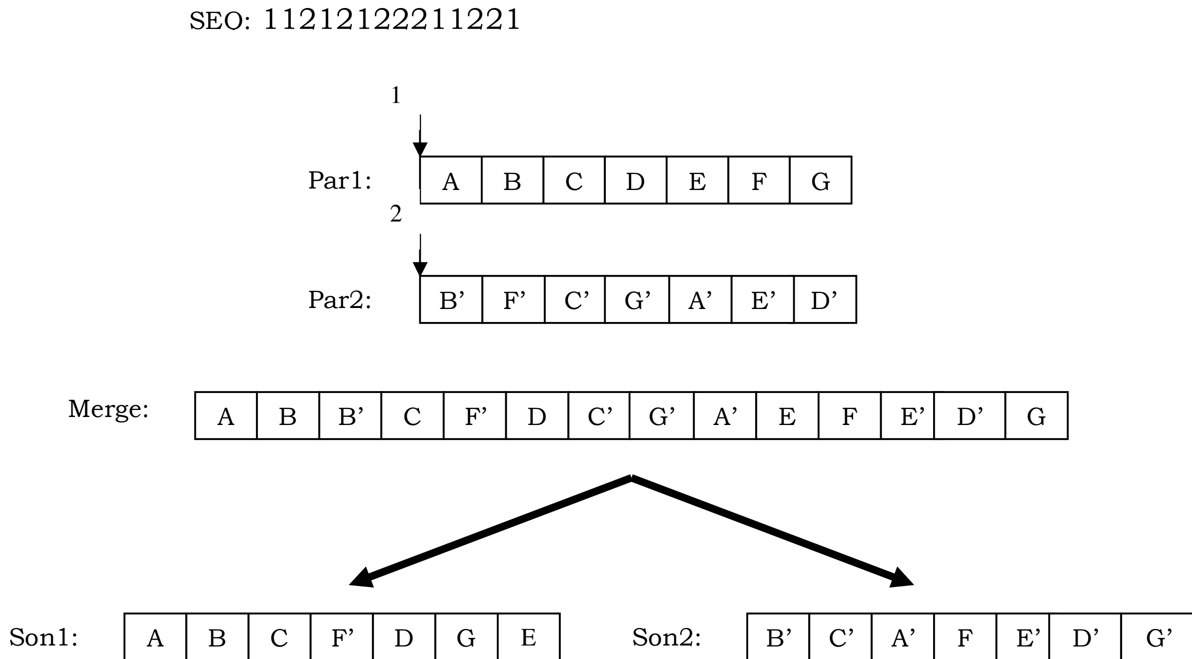


Figure 5.10 – Merge and Split Recombination (MSX)

Definition 4. Merge and Split Recombination (MSX).

Let *SEQ* be a vector of $2 \cdot n$ elements randomly chosen in the set {1, 2} such as both elements 1 and 2 occur *n* times each and where *n* is the length of strings. We use it as input of the following procedure that produces from two parent strings two offspring strings:

1. Initialise pointers *PT1* and *PT2* at the beginning (on the left side) of parent strings *PAR1* and *PAR2*. Set all operations as unmarked.
2. Consider the first number appearing in the sequence *SEQ*
3. Slide to the right the corresponding pointer to the first unmarked operation
4. Copy the current operation in the *Merge* string in the first position available to the left and mark that operation as already processed

5. Take out the number at the beginning of *SEQ*
6. If *SEQ* is empty then go to step 7 otherwise go to step 2
7. Scan the *Merge* string operation by operation from the left-most to the right-most. The first time an operation is met it is assigned to *Son1*, the second time the same operation is met it is assigned to *Son2* filling them from left to right.

The main peculiarity of MSX consists in getting two complementary sons by combining the precedence characteristics of their parents meanwhile trying to minimise the loss of diversity. More precisely, if the generic operations *a* and *b* have a different order in the parents, such as in parent one *a* precedes *b* and in parent two *b* precedes *a*, MSX tends as much as possible to transmit this diversity to the sons so that in one son *a* will precede *b* and in the other one *b* will precede *a*. It is important to notice that in general this requirement may contrast with the requirement regarding cycling solutions. Therefore, since the string representation doesn't allow to encode cycling solutions, it turns out to be often impossible to get a perfect preservation of parental characteristics through the recombination.

Intuitively, the preservation of diversity through the recombination is roughly explainable by noticing that in the merge phase the precedence characteristics of parents are mixed but not destroyed. Then, in the split phase, the characteristics are repartitioned in two sons and still not destroyed, so that obtaining the original characteristics preserved but combined in a different way.

A pertinent doubt one may have is whether MSX respects the feasibility requirement for recombination stated in Definition 2. After all we have only proven the feasibility for a class of recombination operators (Theorem 4) which seems not to include MSX because of its way of recombining strings in two phases. However we can imagine an alternative definition for MSX such that it results to match the form of the feasible class. The idea is to produce the two twin sons separately, each one in one phase, using the same random sequence twice, once scanning the input sequence and the parent strings from left to right producing one son, once scanning them in the other sense producing the other one.

5.4.5 One-Step Crossed Insertion Recombination Operator

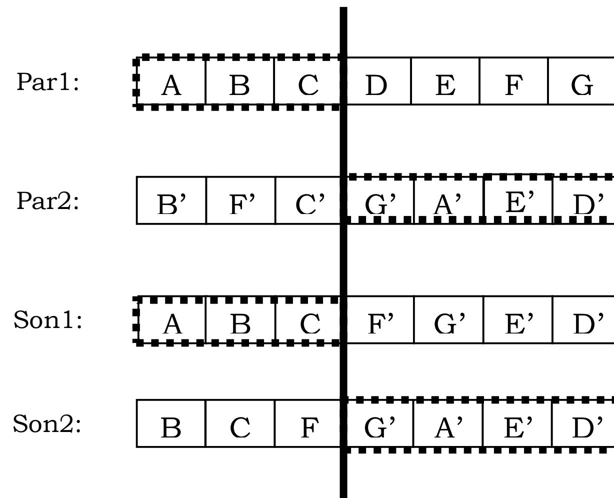


Figure 5.11 – One-Step Crossed Insertion Recombination operator

5.4.6 One-Step Insertion Recombination Operator

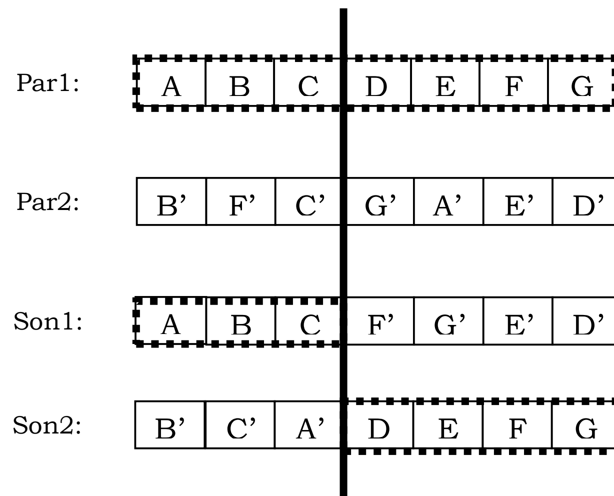


Figure 5.12 – One-Step Insertion Recombination operator

5.5 The Genetic Local Search Algorithm

We propose a genetic local search algorithm (GTS) consisting of a basic genetic algorithm with the addition of a taboo search optimisation phase applied to every new individual created.

5.5.1 Genetic Local Search Template

In the following a GLS outline is presented.

Genetic Local Search Template

1. Generate initial population
2. Execute for every individual an initial optimisation by applying local search
   3. Assign fitness to every individual
   4. Select individuals for recombination
   5. Apply the recombination operator producing a new generation of offspring
   6. Optimise every new offspring by applying local search
   7. Insert the offspring in the population and reduce it to the original size
8. Repeat the steps from 3 to 7 until a stop criterion is met

Let us now fill the Genetic Local Search template presented above with all the components we need to implement an actual algorithm for JSSP. First, we will discuss about the major components of the Genetic Algorithm framework, then we will focus our attention on the specific Local Search algorithm we have used.

5.5.2 Genetic Algorithm Framework

- POPULATION. The initial population contains a fixed number of chromosomes which are generated at random. During all the evolutionary process the population size remains constant.
- FITNESS FUNCTION. Every chromosome in the population receives a fitness value. It biases the probability of the chromosome to reproduce. In our case the fitness value of a chromosome is the makespan of its encoded solution.

- SELECTION SCHEME. A fixed number of chromosomes which will undergo recombination are selected. The selection is done via a simple ranking mechanism. The population is always kept sorted according to the fitness. The probability of each chromosome to be selected depends only on its position in the rank and not on the actual fitness value.
- REINSERTION SCHEME. The set of offspring is merged with the population. Then the population is reduced to its original size by eliminating the worst chromosomes.
- STOP CRITERION. The algorithm stops after a fixed numbers of consecutive generations without improvement of the best solution in the population.
- REPRESENTATION & RECOMBINATION. We use the string representation and the MSX recombination operator presented in Section 2. Let us now spend few words on the role played by MSX in the GLS framework, focusing again on its behaviour. Meanwhile MSX tends to preserve diversity as much as possible, it tries as well to mix parent characteristics a lot. The input sequence is randomly allowed to switch from one parent to the other in every step, therefore it behaves like a uniform crossover. These two aspects of the recombination taken together are particularly welcome in a genetic local search framework. On one hand, MSX transmits the diversity and therefore doesn't trash expensive information present in the parents gathered by local search, the most time-consuming GLS component. On the other hand, the role of the GA paired with local search is to explore as much as possible the solution space. MSX stresses it just shuffling the information present in the parents at most behaving like a uniform crossover.

5.5.3 Local Optimiser

The TS algorithm here proposed is an effective local search algorithm for JSSP. We use it in the above GLS algorithm as a local search optimisation phase in steps 2 and 6.

Taboo Search Algorithm

1. Current Solution := Initial solution
2. Best Solution := Initial Solution
3. Taboo List := Empty

4. Consider the neighbourhood of the current solution and select one of them not in the Taboo List following a Search Strategy
5. Insert the current solution in the Taboo List and, if it is full, make room taking out the solution ahead of the list
6. Update the best solution found so far
7. Make the selected neighbour as the new current solution

8. Repeat steps 4-7 until a Stop Criterion is met

More in detail, the Taboo Search we use is based on an algorithm proposed by Eikelder et al [18]. In the following we discuss the major components of the algorithm.

- REPRESENTATION. To apply local search to JSSP we use the disjunctive graph representation. A feasible solution is obtained by orienting the edges such that there is a linear ordering of the operations that have to be processed on one machine, and the resulting graph is acyclic.
- NEIGHBOURHOOD. We use the neighbourhood structure of Nowicki & Smutnicki [13]. It is based on reversing machine arcs on a longest path. However, they have shown that several types of neighbours can be omitted since they cannot have lower costs. For instance it is not useful to reverse internal arcs of a block of operations belonging to a longest path.
- SEARCHING STRATEGY. The time needed to search a neighbourhood depends on the size of the neighbourhood and on the time complexity of the computational cost of neighbours. Since the size of a neighbourhood is rather small we use the steepest neighbour search strategy that even if it requires evaluating every neighbour, it selects the best one.
- TABOO LIST. We use a taboo list consisting of a FIFO queue of moves of fixed length. The length of the taboo list is the average neighbourhood size plus a random value.
- STOP CRITERION. The algorithm stops after a fixed numbers of steps without improvement.

Because of the combined use of Genetic Algorithms and Taboo Search we will denote our algorithm with GTS, acronym of Genetic Taboo Search.

Chapter 6

Computational Results

6.1 Introduction

In this chapter we are going to show the results we have obtained applying in practice the hybrid algorithm introduced in Chapter 5. First, we try to figure out what are the right questions to pose in order to design a proper set of computational experiments. Afterward, we illustrate all the relevant parameters of the hybrid algorithm, describe the procedure to determine a plausible parameter setting and discuss the values so found. We pass then to actually test the performance of the algorithm. We propose two experiments: the first, the quality and time experiment, is meant to test our algorithm on a set of little and medium instances, trying to find out a good trade-off between quality of solutions found and time required; the second, the quality experiment, is intended to test our algorithm on a set of bigger instances, most of them still open, trying to improve the best bound known so far or approximate as much as possible the known optimum, no matter how much the time required is. Next, we propose three interesting comparisons: the hybrid algorithm versus its local search core, the hybrid algorithm in a wide comparison with a variety of good algorithms for Job Shop and finally a specific comparison of our algorithm, which is hybridised with taboo search, with a similar approach combining genetic algorithms and simulated annealing.

QUESTIONS

In this chapter we want to try to answer to the following questions:

1. What is the most successful composition of Genetic Algorithms and Taboo Search?
2. What is the architecture which guarantees a good flow of information between Genetic Algorithms and Taboo Search in our hybrid algorithm?
3. Which among the recombination operators proposed work better?
4. How much better does our hybrid algorithm perform compared to its basic components taken separately?
5. How are performances of our hybrid algorithm compared with those of other algorithms?
6. What kind of instances does our hybrid algorithm work better on?

The first three questions are investigated in the parameter setting section. Although these questions are of central importance, in our algorithm they take a simple form of parameters to be properly tuned.
The last 3 questions are investigated in the remaining part of the chapter dedicated to performance testing and comparisons.

6.2 Parameter Setting

In the sequel, we present and discuss the most important parameters of GTS (our hybrid algorithm), the ones that affect more the performance of the algorithm and their settings.

6.2.1 Overview on Parameters

COMPUTATIONAL EFFORT

This parameter permits a qualitative control of the computational search effort. More in detail, we define the computational effort as the product of two factors, where the first factor is the number of consecutive iterations without improvement (*TS*) after that each run of Taboo Search has to stop, and the second factor is the number of consecutive individuals processed by the GA without improvement (*GA*) after that GTS has to stop. Since both *TS* and *GA* stop criteria are adaptive to the complexity of the specific problem treated, the setting of the computational effort parameter produces different

effects applied on different instances. However, although roughly, it gives a way to control the computational effort.

TS/GA MIXING RATE

This is a very important parameter that is used to weigh the relative contribution of TS and GA. Knowing the Computational Effort (*TS*GA*) and the *TS/GA* ratio we then can determine the stop criteria for TS and GA.

GA PARAMETERS

It is very important to set the GA parameters properly in order to guarantee a good flow of information between GA and TS during all the evolutionary process so as to obtain an effective cooperation between them. We have found that the following parameters affect the quality of the flow of information and therefore we have paid great attention in finding a good setting:

- *Population Size*. We tuned GTS focusing on meaningful relationship among parameters rather than on their absolute values, trying first to find out good ratios among relevant parameters and only later deriving indirectly their absolute values. Following this approach, we have considered the Population Size being in direct relationship with the Number of Generations.

- *Generational Gap*. This parameter represents the number of offspring to produce every generation through recombination.

- *Selection Pressure*. This parameter permits to control the competition level in the population. It bias the ranking selection mechanism, making the selection probability of chromosomes more depending or less depending on their rank in the population on the basis of the parameter value. The range of the selection pressure varies from 0 (no dependency) to 2 (strong dependency). A weak selection pressure, therefore, gives a bad individual almost the same chance to

reproduce as a good individual, whereas a strong selection pressure strongly favours the reproduction of only good individuals.

RECOMBINATION TYPE

The recombination operators we test are listed in the following:

1. Multi-step Complete
2. Multi-step Parallel
3. Multi-step Crossed
4. One-step Insertion
5. Multi-step Merge-Split (MSX)
6. One-step Crossed Insertion
7. Multi-step Parallel + Multi-step Crossed

The operators from 1 to 6 are those introduced in the previous chapter and therefore don't require further comments. The recombination operator number 7 is not an actual new operator; rather, it consists of a combination of two recombination operators (number 2 and 3, respectively), which present complementary aspects, during the evolutionary process; the idea is applying once the one, once the other, in order to obtain a synergic effect.

6.2.2 Determination of a Plausible Parameter Configuration

The purpose of a parameter setting is finding a good trade-off among different aspects of the algorithm in such a way as a result of it we obtain a parameter configuration that allows the algorithm to get good solutions in a reasonable amount of time independently from the specific instance of the problem which is applied on.

Our algorithm has got many parameters and consequently to apply a fine-tuning would require a big amount of time. We follow then a different approach, very less time-consuming, that produces a reasonable parameter configuration. The idea is to set the parameters evaluating only few problem instances and to check the performance of our algorithm only on combinations of parameters that we think strongly related avoiding in

this way to check every possible combination but focusing our attention only on those interesting.

In the following we illustrate, step by step, the way we follow to find our parameter configuration.

- Selection of test instances
- Determination of the computational effort
- Determination of the TS/GA mixing rate
- Choice of the type of recombination
- Whether to apply a saving time stop criteria
- Determination of a good relation among Gen/Pop, Gap/Pop and Selection Pressure

SELECTION OF TEST INSTANCES

Later, in Section 6.3, we test our program on two sets of instances; the first one contains instances ranging from little to medium-big and the second set contains bigger instances. We select 3 instances belonging to the first set, one little, one medium and one big as our test-set to tune up the parameters. The instances we chose are:

- LA02 (10 jobs * 5 machines)
- LA25 (15 jobs * 10 machines)
- LA29 (20 jobs * 10 machines)

The LA02 instance has revealed too little to give a real contribution in the setting of parameters. Almost every parameter configuration brings the algorithm to find the optimum solution. Therefore we don't report it here.

DETERMINATION OF THE COMPUTATIONAL EFFORT

The first thing to do is to determine for every problem instance the quantity of search (i.e. time) we want to spend in order to solve it. Under-sizing this parameter will bring

the program to get sub-optimal solutions; over-sizing it will hopefully lead to find the best solution it can get, but at the cost of wasting time.

In order to properly set this parameter we have run more times the simply TS algorithm for increasing values of the *computational effort* parameter, ranging from a minimum value up to the first value which allows the algorithm to get a good quality solution. Following this method we have found for out test-set:

- LA02 (10000)
- LA25 (100000)
- LA29 (100000)

DETERMINATION OF THE TS/GA MIXING RATE

In order to determine a proper TS/GA mixing rate, we keep constant the computational effort (TS*GA) for any given instance we test, varying only the TS/GA composition (i.e. ratio). From one hand, this method is not very precise since the stop criteria for the TS and GA components are not constant, though it is acceptable for our purpose. On the other hand, the stop criteria have the good property to be adaptive to the complexity of the specific problem instance.

To be able to actually run the program we should provide values also to the other parameters (i.e. GA parameters), not only to the mixing rate and computational effort. The trick is to choose these values as neutral as possible, taking values positioned in the middle of their defining range. In Table 6.2 and Table 6.3 various TS/GA compositions are presented together with comparisons of the different types of recombination.

Further investigations following this line have brought us to set a different computational effort on the basis of the size of the instance treated as shown in Table 6.1. We have seen also that the bigger the problem is the better GA performs compared with TS. More in detail we have assigned a *TS/GA* ratio of 10:1 for little and medium size instances and 1:1 for large size instances (see Table 6.1).

Size of instances	***TS*GA***	***TS/GA***
Little instances (up to 150 operations)	10000	10:1
Medium instances (around 200 operations)	100000	10:1
Medium-large instances (around 300 operations)	500000	10:1
Large instances (around 500 operations)	1000000	1:1

Table 6.1 – Computational Effort and Mixing Rate

CHOICE OF THE TYPE OF RECOMBINATION

From Table 6.2 and Table 6.3, we see that the best recombination operator related to the chosen TS/GA ratio of 10:1 is the number 5, the MSX recombination operator.

STOP CRITERIA

During the course of early experiments, we have noticed that most of the times, when all the individuals in the current population reach the same makespan, the search hardly can make further improvement to the best solution found so far, even letting the algorithm run for much longer time. This suggests us to stop the algorithm as soon as this uniformity in the population is reached, even if the criterion based on the number of generation without improvement would continue the search. This adjunctive stop criterion has revealed very precious in saving time.

DETERMINATION OF THE GA PARAMETERS

Now we have to determine a good balance among GA parameters. In order to do it, first we fix the values of the parameters previously illustrated and then we test systematically all the combinations of three relevant values for each of the following terms:

- Number of Generation without Improvement / Population Size (Gen/Pop)
- Generational Gap Size / Population Size (Gap/Pop)
- Selective Pressure

Once we have got good values for those, we can easily determine the values to assign to Population Size, Number of Generation without Improvement (Gen) and Generational Gap Size. Table 6.4 shows the results for every combination of values considered.
In the following the settings found are discussed:

- *Population Size*. Following this approach, we have considered the Population Size being in direct relationship with the Number of Generations, obtaining that a good ratio is 1:1. The absolute values we have found for the Population Size parameter vary gradually from 10 individuals for small size instances up to 50 individuals for large size instances.

- *Generation Gap*. We have found Population Size / 2 to be a good setting for this parameter.

- *Selection Pressure*. We have found a weak selection pressure of 0.1 being appropriate for our algorithm. This should not be so surprising because in our GA we use a reinsertion scheme which is already very selective itself, thus making it not necessary to strengthen too much further the selection pressure through this parameter.

DISCUSSION ON THE SETTING FOUND

The main impression we have got meanwhile setting parameters is that the performances of the algorithm are not very dependent on a specific parameter setting, in other words, the architecture of the algorithm is robust. Therefore, even if the parameter setting so obtained has been produced by testing extensively only few instances, we conjecture that this setting will go well also for other instances. As we will see, this configuration produces satisfying results.

INSTANCE: LAW25 **DIMENSION: 15 jobs * 10 machines** **OPTIMUM: 977**							
Crossover	***Makespan***			***Iterations***			
	Best	*Average*	*Worst*	*Termination* *Average*	*Worst*	*Up to Makespan* *Average*	*Worst*
TS * GA = 100000 * 1							
-	977	981	984	156930	246380	-	-
TS * GA = 10000 * 10							
5	977	978	980	221082	331978	110930	265385
TS * GA = 1000 * 100							
1	977	980	984	165570	236775	102180	204998
2	977	979	982	178607	271260	86093	176715
3	977	979	984	167028	289113	101393	207150
4	977	980	984	155562	257041	92573	169202
5	977	979	988	167055	231271	96652	160939
6	977	980	984	135045	271767	85623	208311
7	977	978	984	186265	338639	107614	238930
TS * GA = 100 * 1000							
1	977	981	984	114332	178040	82970	149901
2	977	978	984	105182	136541	78613	116648
3	977	982	990	178597	319452	127106	288533
4	977	983	986	99860	158258	73906	124461
5	977	979	984	122221	219667	88712	161295
6	977	982	986	101708	135444	73189	97040
7	977	978	984	133105	178891	87781	138114
TS * GA = 10 * 10000							
2	978	985	992	86547	131416	61804	91943
TS * GA = 1 * 100000							
1	1219	1317	1389	-	-	-	-
2	1097	1110	1123	-	-	-	-
3	1146	1166	1220	-	-	-	-
4	1163	1202	1322	-	-	-	-
5	1080	1102	1130	-	-	-	-
6	1080	1167	1234	-	-	-	-
7	1089	1113	1136	-	-	-	-

Table 6.2 – Crossovers and Mixing Rates comparison on a medium size instance

INSTANCE: LAW29 **DIMENSION: 20 jobs * 10 machines** **OPTIMUM: 1142 - 1153**							
Crossover	***Makespan***			***Iterations***			
	Best	*Average*	*Worst*	*Termination* *Average*	*Worst*	*Up to Makespan* *Average*	*Worst*
TS * GA = 100000 * 1							
-	1167	1172	1176	204496	384934	-	-
TS * GA = 10000 * 10							
5	1167	1170	1177	295992	352754	201536	292609
TS * GA = 1000 * 100							
1	1164	1171	1175	305970	398333	229499	306309
2	1165	1169	1179	313345	452170	217820	414679
3	1164	1169	1176	329263	428066	247444	311194
4	1162	1170	1183	260764	334916	202696	283227
5	1164	1168	1174	303584	398809	235348	307897
6	1164	1170	1185	238538	370420	187000	330256
7	1164	1168	1180	315262	483902	235568	375211
TS * GA = 100 * 1000							
1	1167	1174	1184	199088	265840	168670	209345
2	1164	1171	1186	222270	375458	177974	272027
3	1167	1175	1184	216489	327733	174437	327733
4	1165	1176	1190	182118	245572	154242	221896
5	1167	1177	1186	163288	252929	131514	236874
6	1167	1177	1211	150157	218479	128711	199856
7	1163	1167	1177	213121	297958	174559	241925
TS * GA = 10 * 10000							
7	1164	1184	1201	143317	168048	128722	153557

Table 6.3 – Crossovers and Mixing Rates comparison on a medium-big size instance

INSTANCE: LAW29 **DIMENSION: 20 jobs * 10 machines** **OPTIMUM: 1142 - 1153**									
Parameters			***Makespan***			***Iterations***			
Gen/ Pop	Gap/ Pop	Sel Pres	*Best*	*Average*	*Worst*	*Termination*		*Up to Makespan*	
						Average	*Worst*	*Average*	*Worst*
1/1	1/10	1/10	1164	1172	1177	442593	598290	338232	506557
1/1	1/10	½	1163	1169	1174	464005	545349	370564	445646
1/1	1/10	9/10	1163	1169	1174	424787	534874	338066	450742
1/1	½	1/10	1157	1166	1174	375081	596267	301296	508339
1/1*	½	1/10	1157	1166	1174	405048	596267	307943	508339
1/1	½	½	1160	1167	1172	287037	421732	218760	352683
1/1	½	9/10	1163	1168	1174	271303	364578	225962	337994
1/1	9/10	1/10	1164	1169	1174	314614	481887	245344	371310
1/1	9/10	½	1164	1169	1178	274574	328368	221452	271953
1/1	9/10	9/10	1163	1169	1178	246853	349110	206293	299532
½	1/10	1/10	1165	1170	1179	437464	544087	330372	461014
½	1/10	½	1164	1171	1176	401468	482358	298445	373785
½	1/10	9/10	1160	1168	1176	416474	496671	316467	414207
½	½	1/10	1163	1168	1175	401425	508677	302145	419871
½	½	½	1163	1167	1171	413079	562796	314497	431981
½	½	9/10	1164	1170	1176	347672	464374	255857	399083
½	9/10	1/10	1164	1166	1174	413148	499425	340952	425640
½	9/10	½	1163	1166	1171	360866	455410	287535	379620
½	9/10	9/10	1164	1169	1177	335463	453518	267831	397910
2/1	1/10	1/10	1163	1168	1178	386650	563336	295754	466286
2/1	1/10	½	1164	1170	1176	316763	458172	229818	374987
2/1	1/10	9/10	1162	1169	1178	309395	418545	239207	348135
2/1	½	1/10	1167	1169	1173	242978	345128	190429	241957
2/1	½	½	1166	1170	1176	212491	303804	169513	256231
2/1	½	9/10	1161	1169	1180	189888	261951	156033	243929
2/1	9/10	1/10	1167	1172	1184	195663	268337	152062	195217
2/1	9/10	½	1167	1171	1180	154350	245035	123683	212370
2/1	9/10	9/10	1164	1171	1178	162014	205733	135073	185658

Table 6.4 – GA Parameter Setting

6.3 Result of GTS

We have tested GTS on two different sets of problem instances posing two different questions. The two sets differ in the average size of the instances involved and in their difficulty to be solved. More in detail, the first set contains little-medium instances and for most of them the optimal value is known; the second set contains medium-big instances and for most of them the optimal value in still unknown. We have tested GTS on both sets of instances substantially maintaining the parameter setting proposed earlier; however, in order to conduct two different experiments, we have adjusted consequently the computational effort parameter.

The first experiment consists in determining how good GTS is in relation to the quality of solutions it gets taking in consideration also the computational time spent to get them; we name it the quality and time experiment. In order to save time the idea is that of assigning a little value of computational effort for little instances, and as the size of instances is increasing assigning bigger values of this parameter.

The second experiment consists in determining how good GTS is, emphasizing the quality of solutions it gets and taking only in minor consideration the computational time consumed; we name it the quality experiment. In order to let GTS the possibility to get good quality solutions we assign a big value to the computational effort parameter for every instance. Since the size of instances involved in this second experiment is bigger than that in the first, we have felt to give a chance more to the GA part setting a TS/GA ratio to 1/1 against the previous 10/1. This is motivated by the believed conjecture that the bigger the problem is the better GA performs when compared with TS.

6.3.1 Quality and Time Experiment

In Table 6.5 the results of the Quality and Time Experiment are shown. The results refer to 10 runs. The values of the computational effort parameter are as follows:

- For the little instances la02 ... la25 the stop criterion is set to 10000
- For the medium instances la27 ... la40 the stop criterion is set to 100000
- For the big instances abz07 ... abz09 the stop criterion is set to 500000

Problem	***Makespan***				***Iterations***				***Time***	***Deviation***
					Termination		*Up to Makespan*			
	OPT	*Best*	*Average*	*Worst*	*Average*	*Worst*	*Average*	*Worst*		
10 jobs * 5 machines										
la02	655	655	655	655	40189	51158	22177	30695	0:51	0.00
10 jobs * 10 machines										
ft10	930	930	933	944	167797	266250	88755	164186	11:15	0.00
la19	842	842	842	843	104733	156187	22179	92733	6:48	0.00
15 jobs * 10 machines										
la21	1046	1047	1050	1055	228292	322453	156541	279007	19:28	0.09
la24	935	938	943	944	161029	216436	51800	112387	14:10	0.32
la25	977	977	978	984	167060	232609	75564	152942	15:11	0.00
20 jobs * 10 machines										
la27	1235	1235	1240	1244	305040	393452	213943	344058	42:50	0.00
la29	1142 / 1153	1157	1166	1174	375081	596267	301296	508339	52:40	1.31
15 jobs * 15 machines										
la36	1268	1268	1274	1278	200689	367788	108900	278463	30:44	0.00
la37	1397	1403	1410	1418	249471	347744	167014	270693	38:09	0.43
la38	1196	1201	1202	1208	188884	236997	123136	163134	29:45	0.42
la39	1233	1233	1239	1250	242348	384043	171993	329833	34:35	0.00
la40	1222	1226	1231	1234	207696	293530	111162	238411	32:00	0.33
20 jobs * 15 machines										
abz07	656	658	662	668	1268445	1750338	988141	1413208	4:54:00	0.30
abz08	645 / 669	670	672	679	1312418	1682689	952923	952923	4:12:58	3.87
abz09	669 / 679	682	687	688	1022050	1294914	731401	997224	3:28:23	1.94

Table 6.5 – Quality and Time Experiment

6.3.2 Quality Experiment

In Table 6.6 the results of the Quality Experiment are shown. The results refer to 3 runs. The value of the computational effort parameter is set to 1000000 for every instance.

Problem	***Makespan***				***Iterations***				***Time***	***Improvement***	***Deviation***
					Termination		*Up to Makespan*				
	OPT	*Best*	*Average*	*Worst*	*Average*	*Worst*	*Average*	*Worst*			
20 jobs * 10 machines											
swv01	1392 / 1418	1430	1430	1430	2853003	3712723	2254395	2996206	1:41:42	0.0	2.73
swv02	1475 / 1491	1481	1484	1487	2534814	3434302	1933733	2401064	1:32:34	62.5	0.41
swv03	1369 / 1398	1418	1425	1431	2436463	3120710	2222742	2935175	1:45:06	0.0	3.58
swv04	1450 / 1493	1482	1488	1500	3010062	3840404	2457313	3534180	1:55:44	25.6	2.21
swv05	1421 / 1448	1441	1447	1453	2427608	3227094	1908645	2365998	1:36:12	25.9	1.41
20 jobs * 15 machines											
swv06	1591 / 1718	1701	1710	1720	2748054	3196472	2552314	2988581	2:56:47	63.0	6.91
swv07	1446 / 1652	1625	1626	1629	3210262	3693342	2897878	3378399	3:39:43	13.1	12.38
swv08	1640 / 1798	1774	1781	1788	3125954	3450466	2890025	3159401	3:25:16	15.2	8.17
swv09	1604 / 1710	1675	1686	1697	2712672	3102098	2439335	2811873	3:03:20	33.0	4.43
swv10	1631 / 1794	1775	1780	1784	4080180	5197313	3798737	4805452	4:37:48	11.7	8.83
50 jobs * 10 machines											
swv11	2983 / 3047	3019	3025	3037	5340847	5806889	4750793	5426844	11:03:06	43.7	1.21
swv12	2972 / 3045	3040	3071	3091	6053560	8059171	5787063	7823797	13:21:26	6.8	2.29
swv13	3104 / 3173	3107	3116	3123	5959718	6853422	5574620	6464213	12:01:00	95.6	0.10
swv14	2968	2968	2971	2977	4983429	5707095	4343219	5149200	9:07:34	0.0	0.00
swv15	2885 / 3022	2918	2929	2938	6181398	8309943	5726330	7958352	13:58:39	75.9	1.14

Table 6.6 – Quality Experiment

6.4 Comparisons

6.4.1 GTS vs TS

GTS is a hybrid algorithm which combines GA with TS. It is important to understand whether such a combination is fruitful. The first question is therefore whether GTS performs better than its two components taken separately. From the literature is well-known that TS over-performs GA. Moreover, to effectuate a fair comparison between GTS and GA we should effectuate a fine-tuning of GA parameters and it would require a lot of time. Therefore, we prefer comparing GTS only with its TS core and conjecture that TS alone works better than GA alone.

In Table 6.7 a comparison among different mixing rates is presented. This shows clearly how a different composition of TS and GA may affect the performance of the hybrid algorithm.

In Table 6.8 a direct comparison between the hybrid algorithm GTS and its TS core is presented. This investigation is of crucial importance since we will find out whether the hybridisation is worthy or the genetic framework has just an ornamental function rather than a real merit.

In order to effectuate a fair comparison between GTS and TS we have set parameters in such a way both algorithms get approximately the same amount of time for the same instance. We have applied both algorithms to a set of well-known JSSP instances of various sizes. This set includes the benchmark set introduced by Vaessens, which comprises the hard-to-solve Lawrence instances, two of the easier ones and the famous Fisher & Thompson 10×10 instance. Moreover, we test the two algorithms also on three bigger instances, the abz-problems from the test library JSPlib (obtainable via ftp from mscmga.ms.ic.ac.uk). We report the results we have obtained from 10 runs on a Sparc station 5 (110Mhz). The CPU time is expressed in seconds. We can notice that on little instances GTS works as well as TS finding the same quality of solutions and using the same amount of time. As the size of instances increases the GTS works better than TS finding better quality solutions. At first glance TS seems saving time on large instances. This is substantially due to the adaptive stop criteria. In order to overcome this premature termination, we tried to compensate the time difference setting TS in such a

way it takes more time, therefore giving it the chance of getting better solutions. TS gets stuck anyway without improving the solution quality, thus wasting all the additional time we gave to it.

INSTANCE: LAW29 **DIMENSION: 20 jobs * 10 machines** **OPTIMUM: 1142 - 1153**				
Makespan			***Iterations***	
Best	*Average*	*Worst*	*Termination* *Average*	 *Worst*
TS * GA = 100000 * 1				
1167	1177	1229	179775	325064
TS * GA = 200000 * 1				
1167	1177	1229	298570	471723
TS * GA = 300000 * 1				
1167	1175	1229	565878	982318
TS * GA = 400000 * 1				
1167	1175	1229	665878	1082318
TS * GA = 500000 * 1				
1167	1175	1229	765878	1182318
TS * GA = 10000 * 10				
1167	1170	1177	295992	352754
TS * GA = 1000 * 100				
1164	1168	1174	303584	398809
TS * GA = 100 * 1000				
1167	1177	1186	163288	252929
TS * GA = 10 * 10000				
1164	1184	1201	143317	168048
TS * GA = 1 * 100000				
1352	1389	1420	-	-

Table 6.7 – Comparison among different mixing rates

Problem	OPT (UB)	BEST GTS	AVG GTS	BEST TS	AVG TS	AVG TIME GTS	AVG TIME TS
10 jobs * 5 machines = 50 operations							
la02	655	655	655	655	663	5	9
10 jobs * 10 machines = 100 operations							
ft10	930	930	933	930	933	67	65
la19	842	842	842	842	842	41	47
15 jobs * 10 machines = 150 operations							
la21	1046	1047	1050	1048	1063	117	122
la24	935	938	943	942	943	85	140
la25	977	977	978	977	978	91	190
20 jobs * 10 machines = 200 operations							
la27	1235	1235	1240	1255	1264	257	140
la29	(1153)	1157	1166	1167	1177	316	233
15 jobs * 15 machines = 225 operations							
la36	1268	1268	1274	1268	1276	184	197
la37	1397	1403	1410	1415	1420	229	208
la38	1196	1201	1202	1199	1204	178	275
la39	1233	1233	1239	1233	1247	207	220
la40	1222	1226	1231	1229	1232	192	211
20 jobs * 15 machines = 300 operations							
abz07	(656)	658	662	666	668	1764	975
abz08	(669)	670	672	680	681	1518	931
abz09	(679)	682	687	688	689	1250	1114

Table 6.8 – GTS Vs TS

6.4.2 A Wide Comparison

We have done a wide comparison on well-known instances among GTS and the best algorithms belonging to a variety of approaches proposed by Vaessens [20]. Table 6.9 gives the best costs found by GTS and other methods. In general we see that GTS behaves very well. Again we see that with big instances GTS outperforms all the other approaches. In the following we list the programs we have considered:

- RGLS-5 – Reiterated Guided Local Search by Balas and Vazacopoulos [2]
- TS-B – Taboo Search and Backtracking by Nowicki and Smutnicki [13]
- SA1 – Simulated Annealing by Van Laarhoven, Lenstra and Ulder [1]
- SB-GLS – Shifting Bottleneck and Guided Local Search by Balas and Vazacopoulos [2]
- GA-SB – Genetic Algorithms and Shifting Bottleneck by Dondorf and Pesch [5]

Problem	OPT	GTS	RGLS-5	TS-B	SA1	SB-GLS	GA-SB
la02	655	655	655	655	-	666	-
ft10	930	930	930	930	-	930	-
la19	842	842	842	842	-	852	848
la21	1046	1047	1046	1047	1053	1048	1074
la24	935	938	935	939	935	941	957
la25	977	977	977	977	983	993	1007
la36	1268	1268	1268	1268	-	1268	1317
la37	1397	1403	1397	1407	-	1397	1446
la38	1196	1201	1196	1196	1208	1208	1241
la39	1233	1233	1233	1233	-	1249	1277
la40	1222	1226	1224	1229	1225	1242	1252
la27	1235	1235	1235	1236	1249	1243	1269
la29	1142 / 1153	1157	1164	1160	1185	1182	1210

Table 6.9 – Wide Comparison

6.4.3 A Specific Comparison

Finally we have done a specific comparison between our hybrid algorithm (Taboo Search Based) and another recent hybrid genetic algorithm based on Simulated Annealing proposed by Kolonko [8].

As we can see in Table 6.10, we have compared these two algorithms on a set of small and medium instances, la and abz instances, setting the stop criteria trying to set a proper trade-off between quality and time. We report the results of 10 runs. We can see that considering the quality of the solutions found, GTS behaves good as SAGen, but it is 18 times faster.

As we can see in Table 6.11, we have compared these two algorithms on the set of difficult swv instances from JSPlib, almost all still open, setting the stop criteria preferring quality against time. We report the results of 3 runs for both algorithms, on a Sparc station 5 (110Mhz) for GTS and on Pentium 120/166Mhz for SAGen. The time is expressed in seconds. As we can see both on the quality and time GTS strongly outperforms SAGen and most of the times GTS breaks the known bound for those instances.

Problem	OPT	BEST GTS	AVG GTS	BEST SAGen	AVG SAGen	AVG TIME GTS	AVG TIME SAGen
10 jobs * 10 machines = 100 operations							
la19	842	842	842	842	844	41	35
15 jobs * 10 machines = 150 operations							
la21	1046	1047	1050	1047	1051	117	549
la24	935	938	943	938	940	85	570
la25	977	977	978	977	979	91	644
20 jobs * 10 machines = 200 operations							
la27	1235	1235	1240	1236	1244	257	3651
la29	1130 / 1153	1157	1166	1167	1169	316	4494
15 jobs * 15 machines = 225 operations							
la36	1268	1268	1274	1268	1270	184	4655
la37	1397	1403	1410	1401	1413	229	4144
la38	1196	1201	1202	1201	1202	178	5049
la40	1222	1226	1231	1226	1229	192	4544
20 jobs * 15 machines = 300 operations							
abz07	656	658	662	658	660	1764	28487
abz08	645 / 669	670	672	670	670	1518	28195
abz09	669 / 679	682	687	683	686	1250	26202

Table 6.10 – GTS Vs SAGen on small and medium instances

Problem	OPT LB / UB	BEST GTS	AVG GTS	BEST SAGen	AVG SAGen	AVG TIME GTS	AVG TIME SAGen
20 jobs * 10 machines = 200 operations							
swv01	1392 / 1418	1430	1430	1427	1428	2034	47828
swv02	1475 / 1491	1481	1484	1487	1490	1851	43089
swv03	1369 / 1398	1418	1425	1422	1428	2102	40684
swv04	1450 / 1493	1482	1488	1487	1490	2315	44257
swv05	1421 / 1448	1441	1447	1449	1453	1924	40045
20 jobs * 15 machines = 300 operations							
swv06	1591 / 1718	1701	1710	1697	1703	3536	112647
swv07	1446 / 1652	1625	1626	1627	1630	4394	97504
swv08	1640 / 1798	1774	1781	1773	1776	4105	56781
swv09	1604 / 1710	1675	1686	1665	1682	3667	24474
swv10	1631 / 1794	1775	1780	1791	1794	5556	44467
50 jobs * 10 machines = 500 operations							
swv11	2983 / 3047	3019	3025	3075	3081	13262	117454
swv12	2972 / 3045	3040	3071	3108	3115	16029	124549
swv13	3104 / 3173	3107	3116	3177	3178	14420	92756
swv14	2968	2968	2971	3010	3013	10951	104088
swv15	2885 / 3022	2918	2929	3004	3004	16773	161365

Table 6.11 – GTS Vs SAGen on big instances

6.5 Summary

We have started this chapter with some key-questions, now we want to summarise the answers we have found.

1. *What is the most successful composition of Genetic Algorithms and Taboo Search?*
   COMPOSITION TS/GA. The combination of GA and TS is more effective than GA and TS taken separately. The best composition of them seems to be half and half.

2. *What is the architecture which guarantees a good flow of information between Genetic Algorithms and Taboo Search in our hybrid algorithm?*
   ARCHITECTURE OF THE GA. The architecture of GA that guarantees a good flow of information is squared-shape, half population generational gap and a little level of competition.

3. *Which among the recombination operators proposed work better?*
   TYPE OF RECOMBINATION. There is no such a big difference in terms of performance among the recombination operators proposed. However, in our computational experiments we have used the MSX operator.

4. *How much better does our hybrid algorithm perform compared to its basic components taken separately?*
   GTS VS TS. GTS works better than TS on big instances.

5. *How are performances of our hybrid algorithm compared with those of other algorithms?*
   PERFORMANCES. The performances of GTS compared to those of other algorithms are good.

6. *What kind of instances does our hybrid algorithm work better on?*
   INSTANCES. GTS seems to work better on big instances with more jobs than machines.

Chapter 7

Conclusions

This paper describes an hybrid algorithm (GTS) combining Genetic Algorithms and Taboo Search for the JSSP. The ingredients of our GA are a natural representation of solutions (the string representation) and a recombination capable of transmitting meaningful characteristics (the common order relationship) from parents to children. The problems of feasibility regarding cycling and job constraints have been discussed and solved in that framework. Moreover, the MSX recombination operator that tends to preserve the diversity of the parent schedules in the offspring schedules has been presented. The Genetic Local Search scheme has been used to hybridise our GA with an effective TS algorithm for JSSP. Computational experiments have shown that on large size instances the GA counterpart makes indeed the difference. The best mix of TS and GA for those instances is half and half (following our mix definition) and therefore GTS has to be considered as a real hybrid, neither a modified TS nor a modified GA. GTS has been compared with a variety of other approaches and it has revealed to perform very well in the comparison. The last experiment has shown that GAs are far more profitably hybridised with Taboo Search than with Simulated Annealing. As a matter of fact both on time required and solution quality a difference of one order of magnitude has been found.

Let us spend some more words on the philosophy underlying the natural approach we use. The crucial point is that we see a schedule as a partial order relationship among operations. It is not important that the relationship is made of contributes from precedence constraints given with the problem instance and those ones given with the

particular solution to that problem. We see all the constraints uniformly without any distinction, all forming the relationship among operations.

By seeing schedules like relationships, it is natural to think about recombination as a way to recombine partial order relationships transmitting to son schedules the common sub-relationship of parent schedules. This seems a natural requirement as we are considering schedules at this level. As a welcome side effect of this approach, we obtain that in the transmission of meaningful characteristics to sons even the feasibility property (intended as the job precedence constraints of the problem instance) is transmitted from parents to sons without paying special attention to it. We treat it uniformly as a generic characteristic of a schedule. This positive side effect leaves us thinking we are approaching the problem at the right level of abstraction without being misled by the syntactical details of the representation used. Finally a further consequence of the way we approach the problem is that the string representation and the recombination proposed do not depend on a particular configuration of the constraints and therefore they can be naturally extended to more general scheduling problems.

References

[1] E.H.L. Aarts, P. J. M. van Laarhoven, J. K. Lenstra, N. L. J. Ulder – A computational study of local search algorithms for job shop scheduling – In: *ORSA Journal on Computing* Vol. 6, No. 2, Spring 1994.

[2] E. Balas and A. Vazacopoulos – Guided Local Search with Shifting Bottleneck for Job Shop Scheduling – In: *Management Science Research Report* #MSRR-609, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, Pennsylvania.

[3] L.D. Davis – *Handbook of Genetic Algorithms* – Van Nostrand Reinhold, 1991.

[4] F. Della Croce, R. Tadei, G. Volta – A Genetic Algorithm for the Job Shop Problem – In: *Computers and Operations Research* Vol. 22, No. 1, 1995, pp. 15-24.

[5] U. Dorndorf and E. Pesch – Evolution Based Learning in a Job Shop Scheduling Environment – *Computer and Operations Research* 22, 1995, pp. 25-40.

[6] F. Glover, C. McMilan and B. Novick – Tabu Search, Part I – *ORSA J. Computing*, 1, 3, 1989, pp. 190-206.

[7] D.E. Goldberg – *Genetic Algorithms in Search, Optimisation and Machine Learning* – Addison Wesley Publishing Company, January 1989.

[8] M. Kolonko - Some new results on simulated annealing applied to the job shop scheduling problem – In: *European Journal of Operational Research* Vol. 113, No. 1, 1999, pp. 123-136.

[9] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, D.B. Shmoys. – Sequencing and scheduling: Algorithms and complexity. – In: S.C. Graves, A.H.G. Rinnoy Kan and P. Zipkin, editors, *Handbooks in Operations Research and Management Science 4*, North-Holland, 1993.

[10] Z. Michalewicz – *Genetic Algorithms + Data Structures = Evolution Programs* – Springer-Verlag, AI Series, New York, 1996.

[11] H. Mühlenbein, M. Gorges-Schleuter and O. Krämer – Evolution Algorithms in Combinatorial Optimisation – In: *Parallel Computing* 7, 1988, pp. 65-85.

[12] R. Nakano, T. Yamada – Conventional genetic algorithm for job shop problems – *Proceedings of 4th ICGA*, 1991, pp. 474-479.

[13] E. Nowicky and C. Smutnicki - A fast taboo search algorithm for the job shop problem – In: *Management Science* Vol. 42, 6, June 1996.

[14] J.T. Richardson, M.R. Palmer, G.E. Liepins, M.R. Hilliard – Some guidelines for genetic algorithms with penalty functions – In: J.D. Shaffer editor, *Proceedings of the third international conference on genetic algorithms*, Morgan Kaufmann, 1989, pp. 191-197.

[15] B. Roy, B. Sussmann – Les problèmes d'ordonnancement avec contraints disjonctives – Note DS 9 bis, SEMA, 1964, Paris, France.

[16] E. Taillard – Parallel taboo search techniques for the job shop scheduling problem – *ORSA J. Computation*, 6, 1994, pp. 108-177.

[17] A.Y.C. Tang and K.S. Leung – A Modified Edge Recombination Operator for the Travelling Salesman Problem – In: H.-P. Schwefel and Manner editors, *Parallel Problem Solving from Nature III*, Springer-Verlag, 1994, pp. 180-188.

[18] H.M.M. ten Eikelder, B.J.M. Aarts, M.G.A. Verhoeven, E.H.L. Aarts – Sequential and Parallel Local Search Algorithms for Job Shop Scheduling – In: S. Voss, S. Martello, I.H. Osman and C. Roucairol, editors: *Meta-Heuristics, Advances and Trends in Local Search Paradigms for Optimization*, Kluwer, 1999, pp. 359-371.

[19] N.L.J. Ulder, E.H.L. Aarts, H-J. Bandelt, P.J.M. van Laarhoven and E. Pesch – Genetic Local Search for the Travelling Salesman Problem – In: *Lecture Notes in Computer Science* 496, Springler, 1990, Berlin, pp. 109-116.

[20] R.J.M. Vaessens, E.H.L. Aarts and J.K. Lenstra - Job shop scheduling by local search – In: *INFORMS Journal on Computing* Vol. 8, No. 3, Summer 1996, pp. 302-317.