Susana Isabel de Matos Fernandes

Técnicas heurísticas para o problema *Job Shop Scheduling*

Departamento de Estatística e Investigação Operacional
Faculdade de Ciências da Universidade de Lisboa
Dezembro de 2002

Susana Isabel de Matos Fernandes

Técnicas heurísticas para o problema *Job Shop Scheduling*

Tese submetida à Faculdade de Ciências da Universidade de Lisboa
para obtenção do grau de Mestre em Investigação Operacional
sob orientação do Professor Doutor João Pedro Pedroso

Departamento de Estatística e Investigação Operacional
Faculdade de Ciências da Universidade de Lisboa
Dezembro de 2002

Resumo

Nesta dissertação serão desenvolvidas heurísticas para o problema de *job shop scheduling*. O problema de *job shop scheduling* consiste em sequenciar um conjunto de trabalhos num grupo de máquinas, minimizando uma determinada medida de performance. Nesta tese a medida a minimizar é o comprimento do caminho mais longo no grafo disjuntivo que representa o problema, ou seja, o *makespan*.

Para cada trabalho existe uma ordem pré-estabelecida de passagem nas máquinas. Designa-se por operação o processamento de um trabalho numa máquina. Cada máquina não pode processar mais que uma operação simultaneamente. Cada operação tem um duração constante e predefinida e, uma vez iniciado, o seu processamento não pode ser interrompido. Sequenciar os trabalhos significa atribuir intervalos de tempo nas máquinas às operações.

As heurísticas desenvolvidas incluem componentes como GRASP (construção de soluções admissíveis e procura local), procura tabu, selecção de soluções elite e religação dessas soluções (*path relinking*).

A estratégia GRASP utiliza na fase construtiva um algoritmo semi-guloso baseado no conceito de máquina limitante (*bottleneck*), que vai introduzindo iterativamente as operações de uma única máquina de cada vez na solução. Para tal são resolvidos de forma exacta problemas de uma única máquina, cuja solução determina um limite inferior para o problema. A cada passo da construção aplica-se procura local à solução parcialmente construída, de forma a prosseguir o processo construtivo com óptimos locais da solução parcial. A vizinhança da procura local é definida através de movimentos sobre pares de operações críticas na mesma máquina, consecutivas ou não, desde que esses pares respeitem determinadas condições de admissibilidade.

O procedimento de procura tabu é executado sobre soluções parciais e/ou completas resultantes da fase construtiva. A estrutura de vizinhança subjacente à procura tabu tem como suporte trocas de operações críticas consecutivas numa mesma máquina. O ciclo fase construtiva – procura tabu é repetido um determinado número de vezes, gerando várias soluções distintas, que poderão ser incorporadas numa lista de soluções de boa qualidade e suficientemente diferentes. Define-se uma medida de distância entre duas soluções baseada na sequência de operações em cada máquina.

O procedimento de religação de soluções é executado sobre pares de soluções contidas

na lista de soluções elite. Uma solução é chamada ponto de partida e outra ponto de chegada, e o caminho é construído através de trocas nas operações de cada máquina na solução de partida, com vista a aproximá-la da solução de chegada. A melhor solução visitada nos caminhos construídos pode não ser um óptimo local na vizinhança definida para a procura local. Assim, após determinar a melhor solução no caminho executa-se uma procura local.

Palavras chave: *job shop scheduling*, GRASP, procura tabu, religação de soluções.

Abstract

In this thesis we will develop heuristic procedures for the job shop scheduling problem. The job shop scheduling problem consists of scheduling a set of jobs on a set of machines, minimizing a performance measure, in here, the makespan.

There is a fixed sequence of machines for each job. The processing of a job on a machine is called an operation which has a constant duration and must not be interrupted. Each machine can process at most one operation at a time and operations of the same job can not be processed simultaneously. Scheduling jobs means assigning time slots on the machines to the operations.

The heuristic methods for the job shop problem includes phases of construction and local search (GRASP), tabu search, managing a list of elite solutions and performing path relinking between solutions in that list.

On the GRASP strategy the construction of a solution is done by sequencing one identified bottleneck machine at a time in a semi-greedy algorithm. The one-machine problems solved exactly at the beginning give a lower bound,which might be used to stop the overall procedure. Each time a new machine is included in the solution, a local search algorithm is executed, based on the concept of pairs of critical operations on the same machine – consecutive or not –.

Tabu search is applied on partial or/and on complete solutions. The neighborhood structure of tabu search uses moves over pairs of consecutive critical operations on the same machine. Several solutions are obtained repeating the construction phase followed by the tabu search procedure. Solutions visited in the tabu search procedure may be candidates to enter a list of good and diversified solutions. A measure of distance between two solutions is defined using the sequence of operations in each machine.

Solutions in the elite list will be the starting and ending points of the path relinking procedure. The solutions found in the path might not be local optima, so local search is applied to the best solution found in the path.

Key words: job shop scheduling, GRASP, tabu search, path relinking.

Agradecimentos

Agradeço ao Professor Doutor João Pedro Pedroso, o acolhimento, apoio e incentivo indispensáveis à elaboração desta tese.

Ao Departamento de Estatística e Investigação Operacional da Faculdade de Ciências da Universidade de Lisboa, ao seu corpo docente, na pessoa da Professora Doutora Maria Eugénia Captivo, agradeço o profissionalismo e excelência científica que caracterizam o Mestrado em Investigação Operacional.

Ao Departamento de Ciência de Computadores da Faculdade de Ciências da Universidade do Porto agradeço a disponibilização de máquinas imprescindíveis à realização dos testes computacionais.

Aos meus colegas do Departamento de Matemática da Faculdade de Ciências e Tecnologia da Universidade do Algarve, na pessoa do Dr. Celestino Coelho, um muito obrigado por todo o apoio.

O amor incondicional de familiares e amigos não se agradece, retribui-se...

Conteúdo

Capítulo 1

Introdução

Nesta dissertação são desenvolvidos métodos para resolução do problema *job shop scheduling*. Uma das estratégias utilizadas é o GRASP (*greedy randomized adaptive search procedure*) assente num procedimento de construção de soluções admissíveis do tipo semi-guloso, conduzido pelo conceito de máquina limitante (*bottleneck machine*) e num procedimento de procura local. Desenvolve-se também um método de procura tabu, e outro de religação de soluções elite (*path relinking*).

O documento tem a seguinte estrutura: neste capítulo faz-se uma apresentação do problema *job shop scheduling*. O segundo capítulo faz o levantamento dos métodos mais significativos desenvolvidos para o resolver, com relevância para o trabalho aqui realizado. Apresenta também sugestões de trabalho descritas na literatura e assim esclarece as motivações subjacentes a esta dissertação. O terceiro capítulo descreve o algoritmo desenvolvido, contendo considerações sobre as opções tomadas nas diversas componentes. O quarto capítulo relata a experiência computacional levada a cabo para avaliar as estratégias desenvolvidas. O documento termina com o quinto capítulo, onde são apresentadas algumas conclusões e apontadas algumas linhas orientadoras para trabalho futuro.

1.1 O problema Job Shop Scheduling

O problema *job shop scheduling*, que terá surgido no início da década de 50 [JM99] considera um conjunto de tarefas ou trabalhos a realizar num conjunto de processadores ou máquinas distintas. Cada tarefa é composta por um conjunto ordenado de operações. Cada operação é processada por uma das máquinas durante um período de tempo predefinido. A máquina que processa cada operação assim como a ordem pela qual se executa o processamento das operações de cada trabalho são fixos, conhecidos à priori, e independentes de trabalho para trabalho.

Este tipo de problema procura encontrar uma sequência de operações para cada

máquina, respeitando certas restrições e minimizando um determinado objectivo. Admite-se que o tempo de processamento de cada operação é determinístico e ininterrupto, duas operações consecutivas do mesmo trabalho são processadas em máquinas diferentes, cada máquina não pode operar mais que uma operação ao mesmo tempo, e máquinas diferentes não podem processar o mesmo trabalho simultaneamente. Como objectivo vamos adoptar o de minimizar o tempo de conclusão de todos os trabalhos — *makespan*.

Formalmente, seja $O = \{0, 1, \ldots, o+1\}$ o conjunto de operações a processar sendo 0 e $o+1$ operações fictícias que representam os instantes de início e fim de processamento de todos os trabalhos, respectivamente. Considere-se M o conjunto de máquinas, A o conjunto de pares de operações consecutivas em cada um dos trabalhos e E_k o conjunto de todos os pares possíveis de operações processadas na mesma máquina, $k \in M$. Defina-se $p_i > 0$ como a duração fixa do processamento da operação i e t_i o seu instante de início. Temos então a seguinte formulação em programação disjuntiva para o problema *job shop scheduling*, largamente divulgada na literatura:

(P)

$$
\begin{array}{lllll}
 & \min t_{o+1} & & & \\
\text{s.a.} & t_j - t_i \geq p_i & (i,j) \in A & & (1) \\
 & t_i \geq 0 & i \in O & & (2) \\
 & t_j - t_i \geq p_i \vee t_i - t_j \geq p_j & (i,j) \in E_k,\ k \in M & & (3)
\end{array}
$$

As desigualdades (1) expressam as restrições de precedências em cada trabalho e a não simultaneidade de operações do mesmo trabalho, dado que $p_i > 0$. As restrições (3), denominadas restrições de capacidade, expressam a não sobreposição de operações nas máquinas.

Chamamos sequenciamento a qualquer solução admissível do problema formulado. Uma representação comum do problema é através de um grafo disjuntivo $G = (O, A, E)$ [RS64]. O é o conjunto de nós, correspondendo cada um a uma operação; A o conjunto de arcos entre nós referentes a operações do mesmo trabalho, orientados de forma a representar as relações de precedência; e E o conjunto de todas as arestas entre nós de operações a realizar na mesma máquina. As arestas funcionam na realidade como dois arcos de sentidos opostos. Para cada nó j de $O \backslash \{0, o+1\}$ existe em A um arco convergente para j e um arco que diverge de j; dito de outra forma, existem e são únicos os nós i e l tais que os arcos (i,j) e (j,l) estão em A. Ao nó i chamamos antecessor de j no trabalho — $at\,[j]$, e analogamente ao nó l chamamos sucessor de j no trabalho — $st\,[j]$. O comprimento de cada arco (i,j) de A é p_i e o comprimento de cada aresta (i,j) de E será ou p_i ou p_j consoante a orientação que lhe seja atribuída. A cada máquina k corresponde um subconjunto O_k de nós e um subconjunto de arestas E_k. Cada grafo $C_k = (O_k, E_k)$ forma um "clique"disjuntivo de G.

Consideremos por exemplo a instância constituída por 4 trabalhos e 3 máquinas apresentada na Tabela 1.1.

	Trabalho 1			Trabalho 2			Trabalho 3			Trabalho 4		
Operações	1	2	3	4	5	6	7	8	9	10	11	12
Máquinas	1	2	3	1	3	2	1	3	2	1	2	3
Tempo	1	1	2	4	2	2	1	1	1	4	2	1

Tabela 1.1: Exemplo de uma instância do problema *job shop scheduling*

A sua representação num grafo disjuntivo será a apresentada na Figura 1.1.

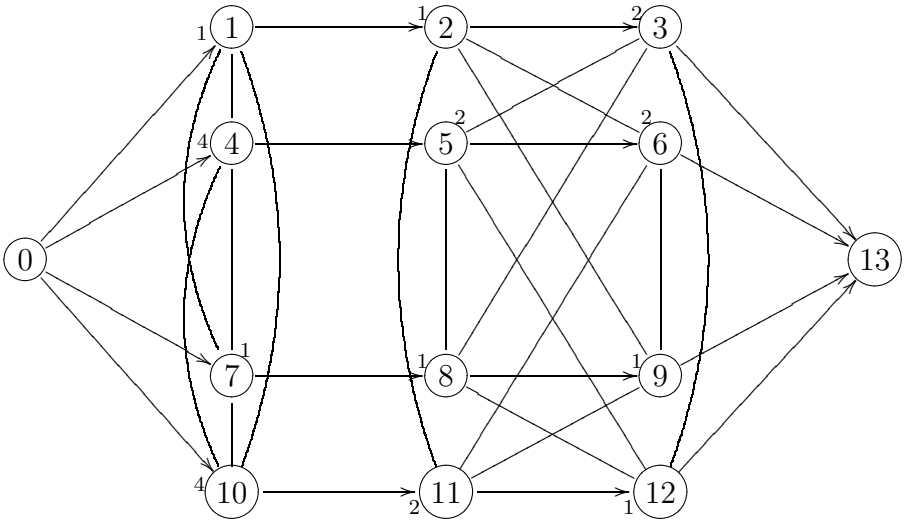


Figura 1.1: Grafo correspondente à instância da Tabela 1.1

Encontrar uma solução para o problema corresponde a orientar todas as arestas de E, ou de outro modo, substituir o par de arcos de sentidos opostos entre duas operações da mesma máquina por um só, tal que o resultado seja um grafo orientado acíclico — $D_S = (O, A \cup S)$. O grafo $D = (O, A)$ corresponde ao grafo orientado obtido de G depois de eliminadas todas as arestas. S é o conjunto de arcos (arestas orientadas) entre duas operações da mesma máquina. Como estes arcos resultam da substituição de dois arcos de sentidos opostos por um só, diz-se que S é uma selecção.

O grafo da Figura 1.2 apresenta uma solução para a instância exemplificada, com valor de *makespan* 13, correspondente ao comprimento do caminho crítico.

A solução óptima do problema *job shop scheduling* será aquela cujo grafo orientado D_S tenha o menor caminho mais longo entre o nó início e o nó fim.

Dada uma solução e as respectivas sequências de operações para cada uma das máquinas; à operação realizada na mesma máquina e imediatamente antes de j chamamos antecessor de j na máquina — $am\,[j]$ e, da mesma forma, a operação realizada imediatamente após j na mesma máquina é designada sucessor de j na máquina — $sm\,[j]$.

Da pesquisa bibliográfica executada ressalta o facto de que, até ao momento o pro-

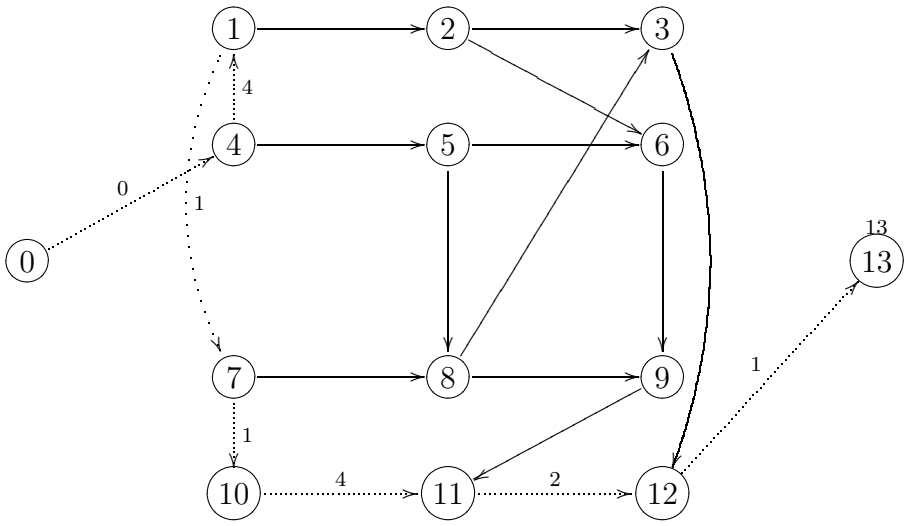


Figura 1.2: Solução para a instância da Tabela 1.1 com *makespan* = 13

cedimento que apresenta de uma forma robusta melhores resultados na resolução do problema considerado, quer em termos de qualidade das soluções quer em termos de tempos de execução, é uma heurística de Balas e Vazacopoulos [BV98]. Assim o próximo capítulo está organizado da seguinte forma: São descritos por ordem cronológica os principais trabalhos sobre os quais assenta a dita heurística concluindo com a exposição da mesma. Segue-se uma breve comparação com alguns trabalhos desenvolvidos paralelamente, a indicação de trabalhos posteriores e continua-se avançando sugestões encontradas na literatura para novos procedimentos. No capítulo seguinte faz-se a exposição das heurísticas propostas nesta tese.

Capítulo 2

Abordagens heurísticas

O problema *job shop scheduling* é um problema de optimização combinatória que pertence à classe dos NP-difíceis [GJ79] pelo que a procura de heurísticas para encontrar boas soluções tem sido muito intensa.

No início proliferaram heurísticas construtivas baseadas em regras de sequenciação, sendo mais eficientes aquelas que trabalhavam com sequenciamentos activos, das quais o algoritmo de Giffler e Thompson em 1960 [GT60] é apenas um exemplo. Entende-se por sequenciamento activo [Fre82] um em que nenhuma operação pode ser iniciada mais cedo sem que isso atrase outra operação ou viole alguma restrição de capacidade. Sucederam-se as heurísticas de procura local. Estas partem de uma solução admissível e definem uma estrutura de vizinhança. Entende-se por vizinhança de uma solução admissível o conjunto de todas as soluções admissíveis obtidas ao realizar sobre aquela uma determinada transformação, usualmente denominada movimento. Depois de definida a vizinhança, as heurísticas de procura local vão saltando de solução em solução vizinha, sempre no sentido de melhoria do valor da função objectivo. Estas heurísticas têm como principal desvantagem o facto de facilmente ficarem estagnadas em óptimos locais. Surgem então as meta-heurísticas para ultrapassar este obstáculo, que se caracterizam por permitirem a execução de movimentos conducentes a soluções com pior valor da função objectivo. De entre elas destaca-se a procura tabu que será utilizada neste trabalho.

A procura tabu foi proposta e desenvolvida por Fred Glover [Glo86], [Glo89], [Glo90], [GL97]. Dada uma solução admissível para o problema e definida uma vizinhança e respectivos movimentos, a procura tabu é um processo iterativo que a cada passo averigua toda a vizinhança da solução corrente e escolhe para nova solução a melhor vizinha, mesmo que esta seja pior em termos da medida de performance em consideração. Para prevenir o regresso a soluções já visitadas é mantida uma lista, denominada *tabu*, que guarda atributos ou das últimas soluções visitadas ou dos movimentos que a elas conduziram. O ideal seria guardar na lista tabu todas as soluções visitadas no processo, mas isso seria incomportável quer em termos de espaço ocupado, quer em termos de esforço necessário para identificar se uma solução vizinha seria ou não tabu. Ao

guardar apenas alguns atributos referentes a uma solução ou à forma de chegar a ela, pode acontecer que eles sejam comuns a mais soluções, e assim estamos a dar o estatuto de tabu não só a uma solução mas a uma classe de soluções. Para ultrapassar esta questão Glover propôs a utilização de um critério de aspiração, que permite escolher uma solução vizinha tabu caso a sua qualidade seja considerada suficientemente boa. Deste modo, a solução escolhida a cada iteração é a melhor vizinha que não é tabu, ou sendo tabu verifica o critério de aspiração.

Estas e outras abordagens heurísticas encontram-se compiladas no artigo "*Deterministic job-shop scheduling: Past, present and future*" [JM99], que contém entre outras coisas, um levantamento exaustivo da evolução quer da definição do problema *job shop scheduling*, quer das técnicas que lhe têm sido aplicadas. Vaessens, Aarts e Lenstra em *Job Shop Scheduling by Local Search* [VAL96] fazem uma análise comparativa de meta-heurísticas desenvolvidas para a resolução do problema.

Na base de muitas das meta-heurísticas que actualmente apresentam melhores resultados para o problema *job shop scheduling* está um algoritmo exacto de Jacques Carlier [Car82] para a resolução do problema de sequenciamento de vários trabalhos numa única máquina, que passamos a descrever sumariamente.

2.1 Algoritmo exacto para o problema de uma única máquina

O algoritmo de Carlier para resolver problemas de uma única máquina considera que a cada trabalho i estão associados um instante de entrada no sistema (a_i), um tempo de processamento (p_i) e uma cauda (q_i). Por cauda do trabalho entende-se o tempo que o mesmo permanece no sistema após ter terminado o seu processamento na máquina. Um trabalho só está concluído depois de abandonar o sistema.

Carlier desenvolveu um algoritmo de procura em árvore com ramificação e limitação (*branch and bound*) para minimizar o tempo de conclusão de todos os trabalhos. Para cada nó da árvore obtém-se um limite superior construindo uma solução com o algoritmo de Schrage [Sch70]. Este algoritmo sequencia os trabalhos que vão ficando disponíveis, dando prioridade aos que possuem caudas maiores. Em caso de empate dá-se preferência a tempos de processamento mais avultados.

Para exemplificar o funcionamento do algoritmo de Schrage consideremos a instância com 4 trabalhos apresentada na Tabela 2.1 e representada no grafo da Figura 2.1, onde o nó 0 corresponde ao início e o nó 5 à conclusão de todos os trabalhos.

Sejam t o tempo corrente, t_i o instante de início de processamento do trabalho i, U o conjunto de trabalhos já sequenciados (vazio no início), $\overline{U}$ o conjunto de trabalhos por sequenciar (inicializado com todos os trabalhos) e S o conjunto de trabalhos disponíveis em determinado instante t. A Tabela 2.2 esquematiza a aplicação do

	J_1	J_2	J_3	J_4
a_i	2	4	1	6
p_i	2	2	1	1
q_i	0	2	1	0

Tabela 2.1: Instância com uma máquina e quatro trabalhos

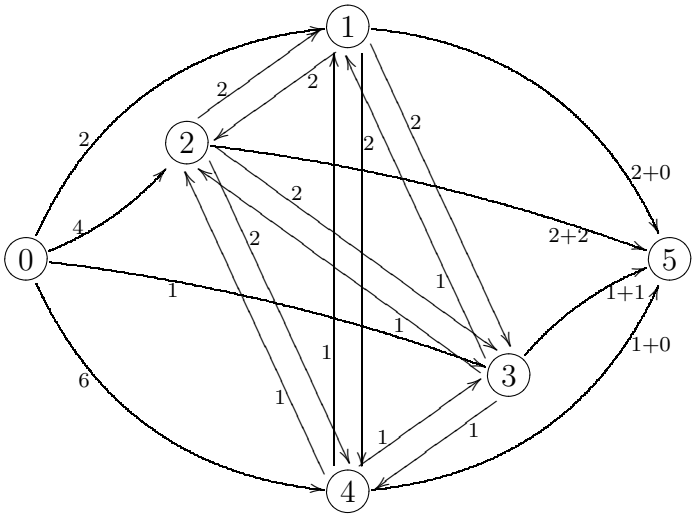


Figura 2.1: Grafo correspondente à instância apresentada na Tabela 2.1.

algoritmo Schrage.

A solução encontrada é a de processar os trabalhos pela sequência $J_3 \rightarrow J_1 \rightarrow J_2 \rightarrow J_4$. O caminho crítico máximo (aquele que contém o maior número de trabalhos) é $J_3 \rightarrow J_1 \rightarrow J_2$, com comprimento igual a 8. O grafo correspondente a esta solução é apresentado na Figura 2.2, onde se evidencia o caminho crítico máximo.

Na árvore de procura de algoritmos de ramificação e limitação (*branch and bound*) são definidos um limite superior, como o valor da melhor das soluções calculadas em cada nó, e um limite inferior, que assume o valor do máximo dos limites inferiores dos nós da árvore.

O limite inferior correspondente a cada nó da árvore de procura do algoritmo de Carlier deriva de um caminho crítico da solução encontrada com o algoritmo Schrage, calculado da seguinte forma. Quando existem vários caminhos críticos escolhe-se o que contém maior número de trabalhos (caminho crítico máximo). Seja ele constituído pelos trabalhos $\{i_1 \longrightarrow i_2 \longrightarrow \cdots i_p\}$. Se para todos os caminhos críticos existentes, sempre que um trabalho i antecede um trabalho j a cauda de i é superior ou igual à de j ($q_{i_1} \geq q_{i_2} \geq \cdots \geq q_{i_p}$), então a solução é óptima. Caso contrário, procura-se no caminho crítico o trabalho i_c sequenciado mais tarde com cauda inferior à do último trabalho no caminho crítico ($q_{i_c} < q_{i_p}$ e $q_{i_j} \geq q_{i_p} \, \forall j \in \{c+1, \ldots, p\}$). Considere-se o conjunto J formado por todos os trabalhos críticos do caminho desde $c+1$ até ao fim,

Inicialização:	$U = \{\}$	$\overline{U} = \{J_1, J_2, J_3, J_4\}$
$t = \min\{a_i : J_i \in \overline{U}\} = a_3 = 1$ $t_3 = t = 1$	$S = \{J_3\}$ $U = \{J_3\}$	$\max\{q_i : J_i \in S\} = q_3 = 1$ $\overline{U} = \{J_1, J_2, J_4\}$
$t = \max(t_3 + p_3, \min\{a_i : J_i \in \overline{U}\}) = 2$ $t_1 = t = 2$	$S = \{J_1\}$ $U = \{J_3, J_1\}$	$\max\{q_i : J_i \in S\} = q_1$ $\overline{U} = \{J_2, J_4\}$
$t = \max(t_1 + p_1, \min\{a_i : J_i \in \overline{U}\}) = 4$ $t_2 = t = 4$	$S = \{J_2\}$ $U = \{J_3, J_1, J_2\}$	$\max\{q_i : J_i \in S\} = q_2$ $\overline{U} = \{J_4\}$
$t = \max(t_2 + p_2, \min\{a_i : J_i \in \overline{U}\}) = 6$ $t_4 = t = 6$	$S = \{J_4\}$ $U = \{J_3, J_1, J_2, J_4\}$	$\max\{q_i : J_i \in S\} = q_4$ $\overline{U} = \{\}$

Tabela 2.2: Exemplo de aplicação do algoritmo Schrage à instância apresentada na Tabela 2.1

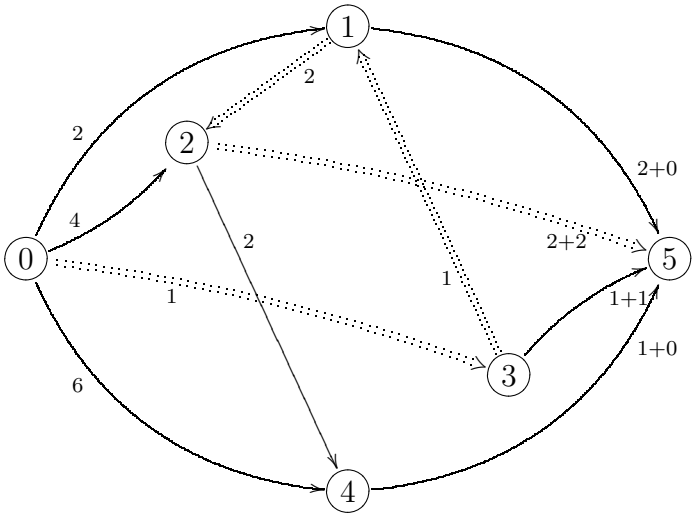


Figura 2.2: Grafo correspondente à solução encontrada pelo algoritmo Schrage, para a instância apresentada na Tabela 2.1.

$J = \{i_{c+1}, \ldots, i_p\}$. O limite inferior da solução encontrada $h(J)$ é dado pela soma de três componentes referentes aos trabalhos em J:

1. o mínimo dos instantes de disponibilidade;
2. a soma dos tempos de processamento de todos os trabalhos;
3. o mínimo das caudas.

$$h(J) = \min_{r \in J}\{a_r\} + \sum_{r \in J} p_r + \min_{r \in J}\{q_r\}.$$

Para o exemplo apresentado temos que J_2 é o último trabalho no caminho crítico mas q_2 não é a menor das caudas: $q_2 \neq \min\{q_3, q_1, q_2\}$. Então o último trabalho no caminho crítico com cauda inferior a q_2 é $i_c = J_1$; sendo $J = \{J_2\}$ o limite inferior da solução tem o valor $a_2 + p_2 + q_2 = 4 + 2 + 2 = 8$. O limite inferior é igual ao valor da solução, logo esta é óptima.

Carlier prova que a distância ao óptimo do comprimento do sequenciamento obtido com o algoritmo Schrage é não superior ao tempo de processamento do trabalho i_c, e ainda que, numa solução óptima, i_c é processado ou antes de todos os trabalhos em J, ou depois deles estarem todos processados.

Os filhos de cada nó da árvore de procura são pois gerados forçando o trabalho i_c a ser processado ou antes ou depois de todos os trabalhos em J, como ilustrado na Figura 2.3. Para obrigar o trabalho i_c a ser processado antes de todos os trabalhos

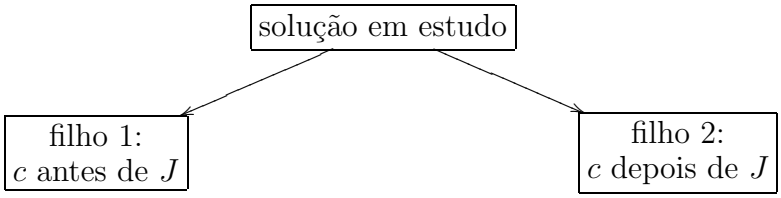


Figura 2.3: Ramificação da árvore de procura

em J, altera-se a cauda de i_c fazendo $q_c = \max\{q_c, \sum_{r \in J} p_r + \min_{r \in J} q_r\}$. Para forçar o trabalho i_c a passar pela máquina depois de todos os trabalhos em J, altera-se o instante de disponibilidade do trabalho i_c para $a_c = \max\{a_c, \min_{r \in J} a_r + \sum_{r \in J} p_r\}$. Estas alterações ficam fixas para o nó e todos os seus descendentes.

Seja $\underline{l}$ o limite inferior do nó pai de um nó da árvore. O limite inferior desse nó é $max(\underline{l}, h(J), h(J \cup \{i_c\}))$. Só se acrescenta um novo nó à árvore se o seu limite inferior for mais pequeno que o limite superior obtido até então. O limite superior da árvore só é actualizado para valores menores.

2.2 Algoritmo de reoptimização local baseada no conceito de máquina limitante

Adams, Balas e Zawack [ABZ88] utilizando a representação do problema *job shop scheduling* num grafo constroem uma heurística de procura local resolvendo subproblemas de uma única máquina com o algoritmo de Carlier.

Considere-se um problema em que um subconjunto M_0 de máquinas está já sequenciado, isto é, para as máquinas em M_0, cada par de arcos de sentidos inversos que unem duas operações dessa máquina foi substituído por apenas um dos arcos, indicando assim a ordem pela qual a máquina processa essas operações. Para todas as máquinas que não pertencem a M_0, excepto uma, seja ela k, os pares de arcos são completamente eliminados. Resolver este problema, corresponde a resolver o problema de vários trabalhos numa única máquina, em que cada trabalho tem associados um instante de disponibilidade e um tempo de permanência no sistema após processamento. O instante de disponibilidade de cada trabalho é dado pelo comprimento do caminho mais longo entre a origem e o nó da operação da máquina considerada. A cauda é definida pelo comprimento do caminho mais longo entre o nó da respectiva operação e o nó fim, retirando-lhe o tempo de processamento da operação. Resolvendo um problema destes para cada uma das máquinas, define-se a máquina limitante (*bottleneck*) como aquela a que corresponde o subproblema com maior valor óptimo. Cada solução de cada um destes subproblemas de uma única máquina fornece um limite inferior para o problema global, sendo o valor óptimo do subproblema da máquina limitante, o limite inferior mais forte.

Exemplificando com a instância introduzida na secção 1.1, os problemas de uma única máquina, para cada uma das três máquinas serão os apresentados na Tabela 2.3.

M_1	O_1	O_4	O_7	O_{10}	M_2	O_2	O_6	O_9	O_{11}	M_3	O_3	O_5	O_8	O_{12}
a_i	0	0	0	0	a_i	1	6	2	4	a_i	2	4	1	6
p_i	1	4	1	4	p_i	1	2	1	2	p_i	2	2	1	1
q_i	3	4	2	3	q_i	2	0	0	1	q_i	0	2	1	0

Tabela 2.3: Problemas de uma única máquina — máquinas 1, 2 e 3

O algoritmo de Carlier constrói para o problema da máquina 1 a solução óptima $O_4 \rightarrow O_{10} \rightarrow O_1 \rightarrow O_7$ com valor 12; para a máquina 2 $O_2 \rightarrow O_9 \rightarrow O_{11} \rightarrow O_6$, com valor 8 e para a máquina 3 $O_8 \rightarrow O_3 \rightarrow O_5 \rightarrow O_{12}$, também com valor 8. O maior *makespan* é o da máquina 1 logo esta é a máquina limitante e será a primeira a incluir na solução. O grafo correspondente a esta solução parcial está representado na Figura 2.4.

Os novos problemas de uma única máquina derivados desta solução parcial, para as máquinas 2 e 3, estão na Tabela 2.4

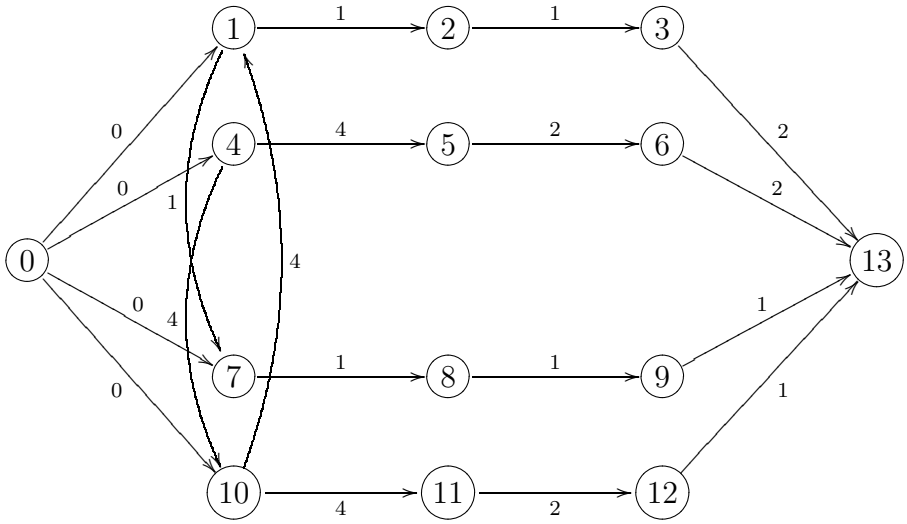


Figura 2.4: Solução parcial — máquina 1

M_2	O_2	O_6	O_9	O_{11}
a_i	9	6	11	8
p_i	1	2	1	2
q_i	2	0	0	1

M_3	O_3	O_5	O_8	O_{12}
a_i	10	4	10	10
p_i	2	2	1	1
q_i	0	2	1	0

Tabela 2.4: Problemas de uma única máquina — máquinas 2 e 3

O algoritmo de Carlier constrói para o problema da máquina 2 a solução $O_6 \rightarrow O_2 \rightarrow O_{11} \rightarrow O_9$, com valor 13 e para a máquina 3 $O_5 \rightarrow O_8 \rightarrow O_3 \rightarrow O_{12}$, com valor 14. O maior *makespan* é o da máquina 3 logo esta é a nova máquina limitante e será a próxima a incluir na solução. O grafo correspondente a esta segunda solução parcial está representado na Figura 2.5.

A Tabela 2.5 apresenta o problema de uma única máquina para a máquina 2, derivado desta solução parcial.

M_2	O_2	O_6	O_9	O_{11}
a_i	9	6	11	8
p_i	1	2	1	2
q_i	3	0	0	1

Tabela 2.5: Problema de uma única máquina — máquina 2

A solução obtida pelo algoritmo de Carlier é $O_6 \rightarrow O_2 \rightarrow O_{11} \rightarrow O_9$, com *makespan* 13.

A solução completa construída está representada no grafo da Figura 2.6 onde se evidencia o caminho crítico $O_4 \rightarrow O_{10} \rightarrow O_1 \rightarrow O_7 \rightarrow O_8 \rightarrow O_3 \rightarrow O_{12}$, cujo

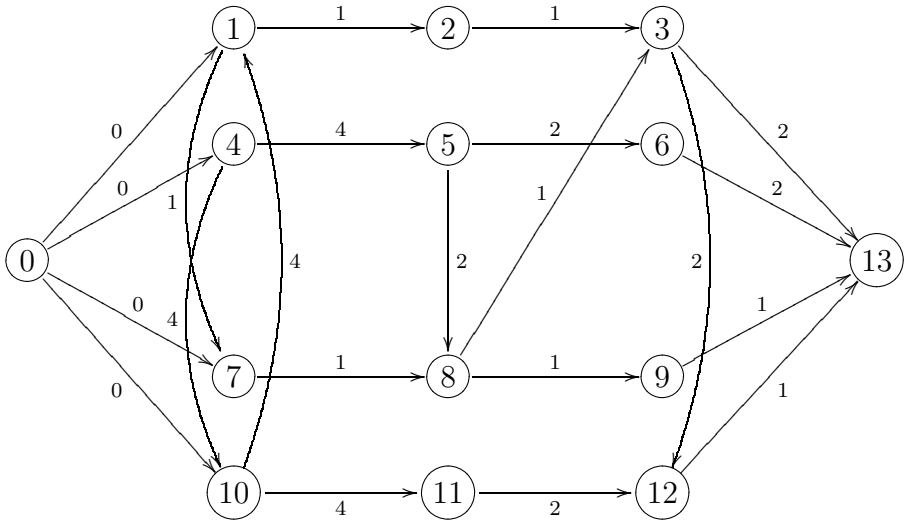


Figura 2.5: Solução parcial — máquinas 1 e 3

comprimento é 14.

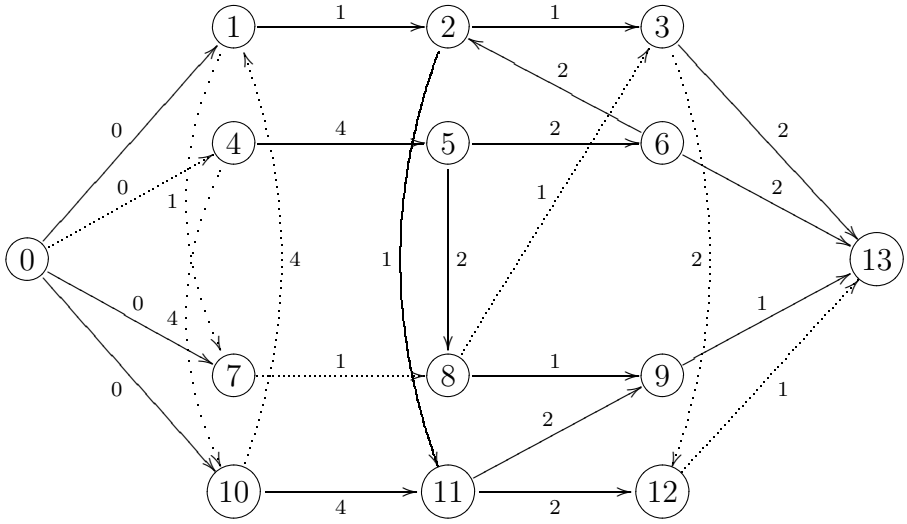


Figura 2.6: Grafo da solução completa

A heurística de Adams, Balas e Zawack [ABZ88], que tem a designação de *shifting bottleneck* (SB) é um algoritmo de construção com optimização local. Começando com uma única máquina vão-se introduzindo máquinas em M_0 da forma descrita. Para todas as máquinas já sequenciadas, de cada vez que se introduz uma nova máquina, volta-se a optimizar o seu subproblema, considerando fixas as sequências de todas as outras máquinas em M_0. Enquanto M_0 não incluir todas as máquinas este processo de optimização máquina a máquina repete-se. Quando já todas as máquinas foram sequenciadas, o processo repete-se passando por todas as máquinas até que nenhuma sofra qualquer alteração. Desta forma a solução encontrada é um óptimo local sobre a vizinhança definida como o conjunto de todas as soluções obtidas de outra optimizando independentemente a sequência de operações numa única máquina. No início as

máquinas estão por ordem lexicográfica mas de cada vez que se completa um ciclo são consideradas por ordem decrescente do valor óptimo do subproblema correspondente. Depois de completamente reoptimizado um subconjunto M_0 de máquinas, o algoritmo retira ainda as últimas máquinas não críticas e repete o processo de optimização das restantes.[1] Por fim reintroduzem-se uma a uma as máquinas retiradas.

Mais tarde Balas e Vazacopoulos [BV98] criam uma meta-heurística que consiste numa procura local conduzida, baseada no *shifting bottleneck* que alcança os melhores resultados, em termos gerais, produzidos até à altura para o problema de *job shop scheduling* [JM99].

2.3 Algoritmo de procura local conduzida, baseada no conceito de máquina limitante

A meta heurística de Balas e Vazacopoulos denominada "*guided local search with shifting bottleneck*" assenta em 3 pilares:

- Uma heurística de procura local - *shifting bottleneck*
- O conceito de árvores de vizinhança - para fugir de óptimos locais.
- A estrutura de vizinhança definida sobre o conceito de par crítico.

Comecemos por apresentar o que os autores definem como par crítico. Considere-se uma solução de uma instância do problema representada num grafo e um seu caminho crítico. Sejam i e j duas operações do caminho crítico realizadas na mesma máquina, tal que todas as operações no caminho crítico entre i e j são também processadas pela mesma máquina. Represente $L(i,j)$ o comprimento do caminho mais longo entre i e j. Para que i e j possam formar um par crítico é necessário que verifiquem pelo menos uma das duas condições seguintes:

1- A operação $st\,[j]$ também está no caminho crítico e o comprimento do caminho mais longo do nó j ao nó $o+1$ (fim), não é mais pequeno que o comprimento do caminho mais longo de $st\,[i]$ até ao fim ($L(j,o+1) \geq L(st[i],o+1)$).

2- O caminho crítico inclui também $at\,[i]$ e o comprimento do caminho mais longo do início até i, incluindo o tempo de execução de i, não é inferior ao comprimento do caminho mais longo desde o nó de início até $at\,[j]$, incluindo o tempo de processamento desta operação ($L(0,i)+p_i \geq L(0,at[j])+p_{at[j]}$).

Por exemplo observemos a solução final apresentada na secção anterior para a instância exemplificada, com caminho crítico $O_4 \rightarrow O_{10} \rightarrow O_1 \rightarrow O_7 \rightarrow O_8 \rightarrow O_3 \rightarrow O_{12}$ e

[1] Por máquina não crítica entende-se qualquer máquina que não possua um único arco no caminho crítico.

consideremos o par de operações (O_{10}, O_7). O_{10} e O_7 são operações críticas realizadas na mesma máquina, assim como todas as operações que se encontram entre elas no caminho crítico, neste caso a operação O_1. A operação sucessora de O_7 no trabalho é O_8, que também está no caminho crítico. O comprimento do caminho mais longo entre O_7 e o nó final é 5. A operação sucessora de O_{10} no trabalho é O_{11} e o caminho mais longo de O_{11} ao nó final tem comprimento 3. Assim (O_{10}, O_7) formam um par crítico que verifica a primeira condição.

Analogamente as operações (O_8, O_3) formam um par crítico que verifica a segunda condição pois ambas são críticas e realizam-se na mesma máquina; $O_7 = at[O_8]$ também é crítica; o comprimento do caminho mais longo do nó inicial até ao fim da execução de O_8 é 11; e o caminho mais longo do início até à conclusão de $O_2 = at[O_3]$ tem comprimento 10.

A vizinhança é definida por trocas realizadas sobre um par crítico. Se as operações u e v formam um par crítico e a condição que se verifica é a condição 1, altera-se a ordem de processamento de forma a que a operação u passe a ser executada imediatamente após a operação v. No exemplo do par (O_{10}, O_7), a sequência na máquina seria alterada para $O_4 \rightarrow O_1 \rightarrow O_7 \rightarrow O_{10}$, dando a origem a uma nova solução com caminho crítico $O_4 \rightarrow O_1 \rightarrow O_7 \rightarrow O_{10} \rightarrow O_{11} \rightarrow O_{12}$ de comprimento 13.

Se u e v formam um par crítico e a condição verificada é a condição 2, o movimento a realizar é processar v imediatamente antes de u. No exemplo do par (O_8, O_3) a sequência na máquina passaria a ser $O_5 \rightarrow O_3 \rightarrow O_8 \rightarrow O_{12}$ e a solução correspondente tem *makespan* igual a 14.

A definição aparentemente complicada desta estrutura de vizinhança é bem fundamentada em diversos resultados desenvolvidos para o problema *job shop scheduling*, como veremos de seguida.

Dada uma solução para uma instância do problema e considerando a sua representação num grafo, se a inversão de um arco que não está no caminho crítico não produz um ciclo, isto é, gera uma solução admissível, o caminho crítico da nova solução não pode ser mais pequeno que o da solução anterior [LAL88]. Assim para conseguir reduzir o *makespan* os arcos a inverter terão de pertencer ao caminho crítico e Van Laarhoven, Aartes e Lenstra [LAL88] mostram que a inversão de um arco crítico nunca conduzirá a uma solução não admissível. Mas Matsuo et al [MSS88] verificam que a alteração do sentido de um arco crítico só poderá realmente reduzir o *makespan* se o caminho crítico incluir ou $at\,[i]$ ou $st\,[j]$. Dell'Amico e Trubian [DT93] mostram que a vizinhança assim definida não é uma vizinhança completa, o que significa que não conseguimos garantir ser possível atingir o óptimo global num número finito de movimentos dentro desta vizinhança. Até agora estamos a falar de operações i e j consecutivas no caminho crítico; para duas operações críticas u e v não consecutivas, Potts [Pot80] mostra que só é possível reduzir o comprimento do caminho mais longo do nó início ao nó fim ao trocar a ordem de processamento das duas operações, se o caminho crítico incluir ou $at\,[u]$ ou $st\,[v]$, o que justifica a escolha efectuada atrás nas condicões 1- e 2-.

Inverter a ordem de processamento de duas operações u e v não adjacentes implica a alteração do sentido de mais de um arco; o número de arcos a inverter depende do número de operações existentes entre u e v. Neste sentido diz-se que esta é uma vizinhança de profundidade variável, uma vez que o número de arcos a inverter em cada movimento não é constante. Como se inverte mais que um arco a proposição de Van Laarhoven, Aartes e Lenstra já não é aplicável. Assim para garantir que não se criam ciclos ao passar a realizar u imediatamente após v, é suficiente que o caminho mais longo desde v até ao fim seja pelo menos tão comprido quanto o caminho mais longo desde $st[u]$ até ao fim. Analogamente para que ao passar a processar v imediatamente antes de u ainda se garanta a admissibilidade, basta verificar que o caminho mais longo desde o início até u, incluindo o tempo de processamento de u, não é mais curto que o caminho mais longo do início até $at[v]$, somando o tempo de execução de $at[v]$. Este resultado é devido a Balas e Vazacopoulos [BV98]. Note-se que as condições impostas são suficientes mas não necessárias para garantir a admissibilidade. Pode-se estimar o valor da solução obtida por execução de um movimento sobre um par crítico u, v recalculando apenas as datas de disponibilidade e as caudas das operações no caminho crítico entre u e v. A estimativa assim calculada só é válida se o caminho crítico não se alterar.

As árvores de vizinhança, outra componente da meta-heurística, são geridas por um procedimento de procura local conduzida (GLS - *guided local search*) utilizando a estrutura de vizinhança que acabámos de descrever.

A cada nó da árvore de vizinhança corresponde uma solução e cada aresta da árvore une um nó pai a um nó filho, obtido a partir do pai por execução de um movimento dentro da vizinhança considerada. Enquanto que no *shifting bottleneck* a vizinhança é constituída pelo conjunto de todas as soluções obtidas de outra, reoptimizando de forma condicionada a sequência de operações numa única máquina, no *guided local search* a vizinhança é definida por uma troca num par crítico. Esta troca pode conduzir a uma solução com maior *makespan*, ou seja criar uma solução pior. Isto pode acontecer se, por exemplo, ao passar a processar a operação u depois da operação v, o início de processamento de v não consegue ser antecipado tanto quanto o tempo de processamento de u. No caso da troca produzir uma solução pior o algoritmo analisa se o aumento de *makespan* ocorreu só antes ou só depois do arco invertido, e em caso afirmativo, no ramo da árvore que se inicia neste nó, a vizinhança passa a estar restringida a trocas em pares críticos que estão ou antes ou depois do arco (v, u), respectivamente.

A dimensão da árvore de vizinhança é condicionada por uma função que limita o número de filhos de cada nó, dependendo do nível da árvore em que este se encontra. Além disso quando num dado nó da árvore de vizinhança se realiza uma troca num par crítico formado pelas operações u e v, o arco (v, u) fica fixo para todos os seus descendentes. Existe ainda um terceiro factor que condiciona a dimensão das árvores que é um limite ao número máximo de níveis admitidos em cada árvore.

A junção dos dois procedimentos de procura local é feita da seguinte forma: A meta-heurística executa o procedimento *shifting bottleneck* mas, depois de remover os pares de arcos de sentidos opostos correspondentes a máquinas não sequenciadas, substitui o ciclo de reoptimização máquina a máquina pela procura local conduzida. Isto é, nesta fase geram-se 2θ árvores de vizinhança onde $\theta = \lceil \log_2 (m_0 \cdot n) \rceil$, com m_0 o número de máquinas cuja sequência está fixa e n o número de trabalhos. A raiz da primeira árvore de vizinhança é a solução obtida com o *shifting bottleneck*. Vai-se construindo a árvore e armazenando as melhores soluções encontradas a uma distância superior a $\log_2 (m_0 \cdot n)$ da raiz. Caso este nível não seja atingido guarda-se o melhor nó. Se a árvore produziu uma solução melhor que a sua raiz, essa solução é a raiz da próxima árvore, caso contrário escolhe-se aleatoriamente uma das soluções armazenadas para ser a raiz da árvore seguinte.

O melhor sequenciamento, em termos de *makespan*, obtido das 2θ árvores de vizinhança é então utilizado como ponto de partida para a continuação do *shifting bottleneck*.

Os autores elaboram ainda mais duas versões deste procedimento. Uma em que depois de todas as máquinas estarem sequenciadas, ou seja quando $M_0 = M$, retiram os arcos correspondentes à sequência de cada máquina, uma de cada vez, e aplicam a procura local conduzida com árvores, ao problema das restantes $m - 1$ máquinas. De seguida introduz-se a máquina removida na melhor solução para o problema de $m-1$ máquinas, e volta-se a executar o processo com θ árvores ao problema completo.

A outra versão pega na melhor solução obtida e apaga os arcos correspondentes a $\lfloor \sqrt{m} \rfloor$ máquinas escolhidas por um processo aleatório enviesado que dá preferência a máquinas sequenciadas no início. A solução parcial restante será a raiz da primeira de θ árvores de vizinhança. Segue-se a introdução das máquinas retiradas pelo processo descrito e depois da solução voltar a estar completa aplica-se a versão anterior. Esta última versão tem como objectivo conduzir a procura para uma zona do espaço de soluções admissíveis mais afastada da solução corrente, ao que usualmente se chama diversificação da procura.

Jain e Meeran [JM99] salientam que embora Balas e Vazacopoulos tenham elaborado estas várias versões da fase de reoptimização, não indicam nenhuma estratégia de como tal deve ser feito, isto é, para resolver uma dada instância do problema torna-se necessário realizar todas as versões e só então escolher a que deu melhores resultados.

2.4 Comparação com outros trabalhos

O conceito de par crítico definido por Balas e Vazacopoulos [BV98] e a respectiva vizinhança não são completamente novos. Vários autores haviam já descrito de outra forma estruturas de vizinhança muito semelhantes. Nomeadamente Dell'Amico e Trubian [DT93] definem-na assim: Designando como bloco uma sequência maximal de

operações consecutivas no caminho crítico realizadas na mesma máquina, considerar as trocas de qualquer operação dentro do bloco com a primeira ou com a última operação do bloco. Ora decorre da definição de bloco que, o antecessor no trabalho da primeira operação do bloco está obrigatoriamente no caminho crítico, o mesmo acontecendo ao sucessor no trabalho da última operação do bloco; o que é o mesmo que o exigido por Balas e Vazacopoulos. Além disso a forma de prevenção da criação de ciclos é muito semelhante, baseada na comparação dos comprimentos de caminhos mais longos. Dell'Amico e Trubian mostram que a estrutura de vizinhança definida desta forma verifica a propriedade de conectividade, isto é, é uma vizinhança completa. Tal como Balas e Vazacopoulos, também Dell'Amico e Trubian combinam várias estruturas de vizinhanças no seu processo de procura local, e aqui começam as diferenças. No trabalho dos autores italianos, que alcançou também muito bons resultados, a outra vizinhança baseia-se na troca de operações adjacentes no caminho crítico que verificam a propriedade introduzida por Matsuo et al [MSS88] e a meta-heurística que gere as procuras locais é a procura tabu.

Como a estrutura de vizinhança baseada no conceito de par crítico do *guided local search* é a mesma da definida a partir dos blocos e o trabalho de Balas e Vazacopoulos apresenta uma melhoria razoável sobre o de Dell'Amico e Trubian, somos levados a pensar que os factores preponderantes no sucesso da meta-heurística mais recente serão a procura local conduzida através das árvores de vizinhança e a presença do *shifting bottleneck*. As árvores de vizinhança permitem guardar informação sobre todo o processo de pesquisa na vizinhança, o que não acontece na procura tabu.

Outro procedimento que em termos gerais apresenta resultados de qualidade comparável à dos obtidos com o *guided local search* é um método da procura tabu desenvolvido por Nowicki e Smutnicki [NS96], assente numa estrutura de vizinhança cujo movimento é o de trocar a ordem de processamento das duas primeiras e das duas últimas operações de cada bloco, de um qualquer caminho crítico de uma dada solução. Esta vizinhança é muito pequena, não é completa mas se uma solução tiver a vizinhança vazia então é solução óptima. O procedimento é muito rápido, devido à sua pequena vizinhança, e somos levados a concluir que o seu sucesso se deve maioritariamente ao grande cuidado na implementação da procura tabu, não esquecendo componentes como a oscilação e a memória a longo prazo com recuperação de soluções de elite, muitas vezes ignorados por simplicidade de implementação.

O procedimento guarda uma lista das melhores soluções visitadas, com informação sobre os seus movimentos vizinhos não realizados e utiliza-a para reiniciar a procura tabu - *back jump tracking*.

2.5 Sugestões para novos procedimentos

A comparação dos vários métodos aplicados ao problema *job shop scheduling* é feita com base num grupo de instâncias geradas aleatoriamente por vários autores. Umas

instâncias são facilmente resolvidas por praticamente todos os processos enquanto que outras continuam por resolver; significando resolver uma instância, encontrar uma solução que se prove ser óptima. Entende-se pois a tentativa de caracterizar instâncias mais difíceis e instâncias menos difíceis. Jain e Meeran [JM99] apresentam uma compilação das várias ideias desenvolvidas sobre este assunto. Assim é de aceitação geral que instâncias com muitas máquinas (mais de 10) e aproximadamente quadradas, isto é, com um quociente pequeno entre o número de trabalhos e o número de máquinas, são instâncias difíceis. Quando as máquinas estão divididas em dois grupos, ou seja, os trabalhos passam todos primeiro pelas máquinas de um grupo e só depois passam para as do outro grupo, o resultado também é uma instância difícil, e neste caso nem precisa de estar em causa um grande número de máquinas. Vários autores têm constatado também que, para instâncias difíceis, existe uma considerável diferença entre o valor do limite inferior fornecido pelo subproblema de uma única máquina e o valor óptimo do problema global; e que a solução óptima do subproblema difere muito da solução desse subproblema na solução óptima global. Jain e Meeran referem uma classificação de instâncias difíceis em duas categorias, devida a Martin [Mar96]. Numa primeira categoria estão as instâncias para as quais se obtém com relativa facilidade bons limites inferiores, mas atingir um bom limite superior é complicado. Na segunda categoria de instâncias difíceis, ficam aquelas em que um bom limite superior é relativamente acessível, mas os limites inferiores alcançados são muito fracos.

Ao comparar a performance de vários métodos num grupo alargado de instâncias ditas difíceis do problema em causa, Jain e Meeran constatam que a meta-heurística de Balas e Vazacopoulos apresenta mais dificuldades ao lidar com instâncias da segunda categoria enquanto que a procura tabu de Nowicki e Smutnicki apresenta resultados menos bons com as da primeira categoria. Posto isto os autores sugerem que uma boa estratégia seria criar um procedimento que incorporasse características destas duas meta-heurísticas, juntando também algumas componentes de um algoritmo genético de procura local desenvolvido por Yamada e Nakano [YN96a], [YN96b].

Mais especificamente Jain e Meeran sugerem que uma boa estratégia seria construída usando uma estrutura de vizinhança baseada na definição de blocos, com uma procura local de profundidade variável, combinada com uma técnica de recuperação de soluções de elite. Propõem ainda que se incorpore a criação de caminhos entre duas soluções de elite, para desta forma conseguir uma exploração mais alargada do espaço de soluções admissíveis.

Glover [FGM00] propôs uma técnica denominada *path relinking* (religação de soluções) como uma estratégia de intensificação, que consiste precisamente em explorar trajectórias que unam soluções consideradas elite, produzidas pela procura tabu.

2.6 Trabalhos posteriores

No ano de 2000, Pezzella e Merelli publicam um artigo onde apresentam um algoritmo que combina procura tabu com o *shifting bottleneck* [PM00], que não consegue suplantar os resultados obtidos com o *guided local search* de Balas e Vazacopoulos. As estruturas de vizinhança utilizadas por Pezzella e Merelli são definidas sobre o conceito de blocos, que têm dado provas de bom funcionamento. A versão de procura tabu implementada é simples, no sentido em que se limita à memória de curto prazo (lista tabu) e aos critérios de aspiração, não incorporando componentes de memória a longo prazo ou oscilação. No entanto e apesar da dita simplicidade, em muitas das instâncias do "núcleo duro" testadas por todos os que trabalham em *job shop scheduling*, o algoritmo consegue alcançar resultados tão bons quanto os melhores conhecidos até à data.

Em 2001, Aliex, Binato e Resende implementam um algoritmo GRASP com religação de soluções para resolver o problema de *job shop scheduling* [ABR01], sem que consigam alcançar grandes resultados em termos de qualidade das soluções.

GRASP [FR95] significa *greedy randomized adaptive search procedure* e é um processo iterativo em que cada iteração é composta por duas fases; uma fase de construção baseada num algoritmo estilo guloso (*greedy*) com componente aleatória e uma fase de procura local.

O processo de religação de soluções é executado entre soluções consideravelmente diferentes, sendo a dissemelhança medida pelo número de operações em ordens diferentes, nas sequências de operações construídas para cada uma das máquinas. O caminho entre duas soluções é percorrido incorporando atributos de uma solução guia numa solução inicial. Em cada passo do caminho são avaliados todos os movimentos e executa-se o que conduz à melhor solução. As melhores soluções encontradas, quer no processo construtivo, quer no processo de religação de soluções, vão sendo guardadas numa lista de soluções boas e diversificadas. O processo só pára quando a religação de soluções não consegue introduzir novas soluções na lista. Por fim executa-se procura local sobre as boas soluções encontradas.

A qualidade dos resultados obtidos fica um pouco aquém dos alcançados com os outros procedimentos descritos, talvez porque a fase construtiva do GRASP seja pouco ambiciosa, ficando-se pelas heurísticas baseadas em regras de sequenciação.

Capítulo 3

Algoritmos propostos

A proposta de trabalho para esta dissertação é a de, acatando as sugestões anteriormente descritas, aplicar uma meta-heurística tipo procura tabu à resolução do problema *job shop scheduling*, tendo como base o processo construtivo do *shifting bottleneck* e uma procura local sobre várias estruturas de vizinhança (algumas definidas sobre blocos de operações), não esquecendo a incorporação de componentes como a oscilação e a memória a longo prazo. Pretende-se ainda considerar a incorporação de alguns aspectos de religação de soluções para desenhar caminhos entre boas soluções. Assim foram desenvolvidos os seguintes algoritmos:

Alg1- Construção GRASP com procura local.

Alg2- Alg1 seguido de procura tabu.

Alg3- Construção GRASP com procura tabu incorporada.

Alg4- Alg1 seguido de procura tabu com retrocesso.

Alg5- Aplicação do processo de religação de soluções às melhores soluções obtidas com o Alg2.

3.1 Algoritmo de construção com procura local

```
GRASP(runs)
(1)   M := {1, ..., m}
(2)   for r = 1 to runs
(3)       x := {}
(4)       K := M
(5)       while K ≠ {}
(6)           foreach k ∈ K
(7)               x_k := CARLIER(k)
(8)           k* := SEMIGULOSO(K)
(9)           x := x ∪ x_k*
(10)          f(x) := TAILLARD(x)
(11)          K := K\{k*}
(12)          if |K| < |M| − 1
(13)              x := PROCURALOCAL(x, M\K)
(14)      if x* não inicializado or f(x) < f*
(15)          x* := x
(16)          f* := f(x)
(17)  return x*
```

runs =número máximo de corridas; m =número de máquinas.

O algoritmo constrói soluções admissíveis utilizando a ideia de sequenciar as operações correspondentes a uma única máquina de cada vez. Para isso recorre-se ao conceito de máquina limitante, isto é, a máquina a que corresponde o maior valor de *makespan*, como descrito na heurística *shifting bottleneck*. Este conceito serve de base para um procedimento GRASP.
Um algoritmo GRASP caracteriza-se por construir uma solução admissível incorporando um elemento de cada vez. Cada elemento é avaliado por uma função heurística; constrói-se uma lista restrita de candidatos a entrar na solução (RCL), com base no valor dessa função, e escolhe-se aleatoriamente um dos candidatos desta lista.

```
SEMIGULOSO(K)
(1)   α := RANDOM(0, 1)
(2)   f̄ := max{f(x_k), k ∈ K}
(3)   f := min{f(x_k), k ∈ K}
(4)   RCL = {}
(5)   foreach k ∈ K
(6)       if f(x_k) ≥ f̄ − α(f̄ − f)
(7)           RCL := RCL ∪ {k}
(8)   return RANDOMCHOICE(RCL)
```

Aqui cada elemento é uma máquina e a função heurística o respectivo *makespan*. Para construir a lista restrita de candidatos, identificam-se o maior e o menor *makespans* dos problemas de uma única máquina resolvidos com o algoritmo de Carlier, e insere-se uma máquina na lista se $f(x_k) \geq \overline{f} - \alpha \times (\overline{f} - \underline{f})$, onde $f(x_k)$ é o *makespan* da solução da máquina considerada, $\overline{f}$ o maior dos *makespans*, $\underline{f}$ o menor dos *makespans* e α um valor aleatório entre 0 e 1. A máquina a incorporar na solução, k^*, é escolhida aleatoriamente da lista de candidatos.
Inserir uma máquina na solução significa incluir no grafo disjuntivo que representa o problema, arcos que indicam a sequência de processamento das operações da máquina em causa. Temos então que calcular o valor da nova solução parcial, ou seja, determinar o comprimento do caminho mais longo entre o nó inicial e o nó final do novo grafo. Para tal utilizamos o algoritmo descrito por Taillard [Tai94].

```
TAILLARD(x)
(1)   Q := {operações com at[i] e am[i] indefinidos}
(2)   E := {}
(3)   while Q ≠ {}
(4)       escolher i ∈ Q
(5)       E := E ∪ {i}
(6)       Q := Q\{i}
(7)       r_i := max(r_{am[i]} + p_{am[i]}, r_{at[i]} + p_{at[i]})
(8)       if am[st[i]] não existe or am[st[i]] ∈ E
(9)           Q := Q ∪ {st[i]}
(10)      if at[sm[i]] não existe or at[sm[i]] ∈ E
(11)          Q := Q ∪ {sm[i]}
(12)  Q := {operações com st[i] e sm[i] indefinidos}
(13)  E := {}
(14)  while Q ≠ {}
(15)      escolher i ∈ Q
(16)      E := E ∪ {i}
(17)      Q := Q\{i}
(18)      q_i := max(q_{sm[i]} + p_{sm[i]}, q_{st[i]} + p_{st[i]})
(19)      if sm[at[i]] não existe or sm[at[i]] ∈ E
(20)          Q := Q ∪ {at[i]}
(21)      if st[am[i]] não existe or st[am[i]] ∈ E
(22)          Q := Q ∪ {am[i]}
(23)  return max_i(r_i + p_i + q_i)
```

Acreditando que boas soluções parciais conduzem a boas soluções completas, de cada vez que se insere uma nova máquina na solução executa-se uma procura local. Assim o processo continua com base numa solução parcial que é um óptimo local.

3.1.1 Procura local

```
PROCURALOCAL(x, M_0)
(1)   s := VIZINHO(x, M_0)
(2)   while s ≠ x
(3)       x := s
(4)       s := VIZINHO(x, M_0)
(5)   return s

VIZINHO(x, M_0)
(1)   foreach s ∈ N(x, M_0)
(2)       if f(s) < f(x)
(3)           return s
(4)   return x
```

A estrutura de vizinhança utilizada para a procura local baseia-se no conceito de pares críticos introduzido por Balas e Vazacopoulos.
Dada uma solução corrente, x, o processo inspecciona a vizinhança, $N(x, M_0)$, até encontrar o primeiro vizinho cujo *makespan* é inferior ao da solução corrente. Para avaliar a qualidade de um vizinho, não se recalcula todo o caminho crítico, mas apenas os caminhos mais longos que passam por cada uma das operações que, na solução corrente, estão entre as operações que constituem o par crítico u, v. O algoritmo apresentado por Balas e Vazacopoulos [BV98] para este propósito é muito semelhante ao descrito por Taillard [Tai94] para todo o grafo, tendo apenas que considerar o facto de os antecessores e/ou sucessores das operações em torno de u e v, incluindo estas, terem sofrido alterações.
Pode acontecer que, uma vez reduzido o comprimento do caminho mais longo que passa por u e v, um caminho, que não inclua o arco $v \rightarrow u$, passe a ser o crítico

na solução. Neste caso a estimativa calculada não corresponde ao valor do *makespan* da solução vizinha. Quando isto acontece, o que foi considerado melhor vizinho não actualiza a solução corrente e passa-se ao vizinho seguinte.

3.2 Algoritmo de construção seguido de procura tabu

```
GraspTabu(runs, I)
(1)   for r = 1 to runs
(2)       x := Grasp(1)
(3)       x := ProcuraTabu(I, x, M)
(4)       if x* não inicializado or f(x) < f*
(5)           x* := x
(6)           f* := f(x)
(7)   return x*
```

I =número máximo de iterações da procura tabu.

O objectivo de construir várias soluções usando um processo estilo GRASP é o de fornecer soluções iniciais diferentes à procura tabu e assim diversificar a procura.

3.2.1 Procura tabu

```
ProcuraTabu(I, x, M)
(1)   x+ := x
(2)   Ltabu := {}
(3)   for i = 1 to I
(4)       τ := número máximo de movimentos possíveis
(5)       t := Random(1, τ)
(6)       x := MelhorVizinho(x, f(x+), Ltabu)
(7)       actualizar Ltabu
(8)       if f(x) < f(x+)
(9)           x+ := Intensificação(x, M)
(10)  return x+
```

t =dimensão da lista tabu; $Ltabu$ =lista tabu.

A procura tabu assenta numa estrutura de vizinhança em que, um vizinho de uma solução é obtido trocando a ordem de quaisquer duas operações consecutivas no caminho crítico, processadas pela mesma máquina. Esta vizinhança tem sido utilizada por muitos autores. Van Laarhoven, Aarts e Lenstra [LAL88] mostram que esta vizinhança só tem soluções admissíveis e é completa, ou seja, consegue atingir-se a solução óptima com um número finito de movimentos.

```
MelhorVizinho(x, f+, Ltabu)
(1)   foreach y ∈ N(x)
(2)       φ := Estimativa(y)
(3)       if φ < f+ or φ* não inicializado or (y não tabu and φ < φ*)
(4)           x := y
(5)           φ* := φ
(6)   return x
```

A lista tabu tem uma posição para cada operação. Dado um arco (i, j) guarda-se na lista tabu, na posição referente a j, o número da iteração em que se executou a troca.

O número máximo de movimentos realizáveis (τ) é calculado para cada solução somando o número de operações críticas consecutivas de cada máquina. Mais concretamente, por cada bloco de cardinalidade β de operações críticas consecutivas da mesma máquina, τ é incrementado em $\beta-1$ unidades. A dimensão da lista tabu é determinada aleatoriamente para cada solução corrente, podendo assumir valores de 1 a τ.
Um movimento é tabu se implica a inversão de um arco (i, j) em que a operação i foi trocada à menos de τ iterações, isto é, se a soma do valor guardado na lista tabu, na posição referente a i, com τ, é superior ao número da iteração corrente.
A estrutura da lista tabu descrita, em conjunto com a definição da regra que classifica um movimento como tabu, fazem com que não se proíba apenas o movimento inverso ao realizado mas toda uma classe de movimentos que partilham o mesmo atributo, ou seja, todos os movimentos em que a operação i tenha sido trocada para trás da operação que a antecedia na máquina, há "relativamente pouco tempo". Justifica-se assim a definição de um critério de aspiração que permite realizar movimentos tabu que conduzem a uma solução com *makespan* inferior ao da melhor solução visitada em todo o processo.
Como Glover salientou [Glo89], o facto da classe de movimentos permanecer tabu durante uma determinada quantidade de iterações, assume que a probabilidade de voltar a uma solução visitada é inversamente proporcional à distância entre soluções, medida em número de movimentos realizados.
De cada vez que a melhor solução visitada é actualizada, executa-se um procedimento de intensificação com o intuito de fazer uma procura mais minuciosa na proximidade de boas soluções. A forma escolhida para o fazer foi a de executar o procedimento de procura local definido na fase construtiva do algoritmo.

```
IntensificaçãoProcuraLocal(x, M)
(1)   s := ProcuraLocal(x, M)
(2)   return s
```

3.3 Algoritmo de construção com procura tabu incorporada

```
TabuNoGrasp(runs, I)
(1)   M := {1, ..., m}
(2)   for r = 1 to runs
(3)       x := {}
(4)       K := M
(5)       while K ≠ {}
(6)           foreach k ∈ K
(7)               x_k := Carlier(k)
(8)           k* := SemiGuloso(K)
(9)           x := x ∪ x_k*
(10)          f(x) := Taillard(x)
(11)          K := K\{k*}
(12)          if K ≠ M
(13)              x := ProcuraTabu(I, x, M\K)
(14)      if x* não inicializado or f(x) < f*
(15)          x* := x
(16)          f* := f(x)
(17)  return x*
```

Aceitando a hipótese de que boas soluções parciais conduzem a boas soluções completas, embora conscientes do acréscimo da complexidade algorítmica, pretende-se, com este algoritmo, averiguar se haverá vantagem em aplicar procura tabu às soluções parciais que vão sendo construídas.

3.4 Algoritmo de construção seguido de procura tabu com retrocesso

```
GraspTabuBt(I)
(1)     x := Grasp(1)
(2)     x* := x
(3)     BTlista := {x}
(4)     BTviz := {None}
(5)     while BTlista ≠ {}
(6)         x :=último elemento de BTlist
(7)         viz :=último elemento de BTviz
(8)         BTlista := BTlista\{x}
(9)         BTviz := BTviz\{viz}
(10)        x* := ProcuraTabuBT(I, x, viz)
(11)    return x*
```

BTlista =lista dos melhores óptimos locais que vão sendo encontrados na procura tabu; *BTviz* =lista do movimento executado pela procura tabu para prosseguir a partir do elemento correspondente em *BTlista*.

```
  ProcuraTabuBT(I, x, viz)
(1)     flag :=false
(2)     x+ := x
(3)     Ltabu := {}
(4)     for i = 1 to I
(5)         τ := número máximo de movimentos possíveis
(6)         t := Random(1, τ)
(7)         x := MelhorVizinhoBT(x, f(x+), Ltabu, viz)
(8)         actualizar Ltabu
(9)         if f(x) < f(x+)
(10)            x+ := Intensificação(x, M)
(11)            if f(x+) < f(x*)
(12)                x* := x+
(13)                BTlista := BTlista ∪ {x+}
(14)                flag :=true
(15)    return x*

MelhorVizinhoBT(x, f+, Ltabu, viz)
(1)     foreach y ∈ N(x)\{viz}
(2)         φ = Estimativa(y)
(3)         if φ < f+ or φ* não inicializado or (y não tabu and φ < φ*)
(4)             y* := y
(5)             φ* := φ
(6)     if flag :=true
(7)         BTviz := BTviz ∪ {y*}
(8)         flag :=false
(9)     return x
```

O algoritmo de procura tabu com retrocesso utiliza a ideia, apresentada no artigo de Nowicki e Smutnicki [NS96], de reiniciar a procura a partir das melhores soluções já visitadas. No algoritmo de procura tabu com retrocesso apresentado nesta dissertação, a procura é reiniciada partindo dos melhores óptimos locais que vão sendo encontrados. Para garantir que o caminho percorrido pela procura a partir de uma solução já visitada (x^+) não se repete, é proibida a execução do movimento que anteriormente foi utilizado para sair da referida solução x^+. A inclusão do retrocesso no algoritmo de procura tabu

tem como objectivo diversificar a procura, utilizando informação sobre o caminho já percorrido; o que não acontece quando reiniciamos a procura tabu a partir de diferentes soluções obtidas com o GRASP.

3.5 Algoritmo de construção seguido de procura tabu e religação de soluções elite

```
GraspTabuPr(runs, I)
(1)     L = {}
(2)     for r = 1 to runs
(3)         x := Grasp(1)
(4)         x := ProcuraTabu(I, x, M)
(5)         if x* não inicializado or f(x) < f*
(6)             x* := x
(7)             f* := f(x)
(8)         L := GerirElite(L, maxL, x)
(9)     x* := ReligarElite(L)
(10)    return x*
```

L =lista de soluções elite.

O algoritmo GRASPTABUPR gere uma lista de soluções elite, onde vão sendo guardadas as melhores soluções encontradas em cada iteração da fase construtiva com a procura tabu. A lista é mantida com o objectivo de posteriormente religar pares de soluções nela guardadas. Assim pretende-se que as soluções da lista tenham boa qualidade e alguma diversidade. Como a religação de soluções pode ter um comportamento bastante pesado computacionalmente na procura do melhor caminho entre duas soluções, a lista de soluções elite é mantida com uma dimensão relativamente reduzida ($maxL$).

3.5.1 Lista de soluções elite

Uma vez que se pretende guardar soluções diferentes e gerir a lista de soluções elite com base na diversidade torna-se necessário definir uma medida de dissemelhança entre duas soluções. Cada solução é caracterizada pela sequência de operações processadas nas máquinas. Considerando duas soluções, conta-se o número de operações cujo antecessor na máquina é diferente nas duas soluções e assim definimos a distância entre elas (Δ).
Dadas duas soluções S_1 e S_2, para cada operação i seja $\delta(i) = 1$ se $am[i]_{S1} \neq am[i]_{S2}$ e 0 em caso contrário; então $\sum_{i \in O} \delta(i) = \Delta$ define a distância entre S_1 e S_2.
Aiex, Binato e Resende [ABR01] definem outra distância que conta o número de operações em ordem diferente na sequência de processamento de cada máquina. Parece-nos que a distância por nós definida traduz melhor a noção intuitiva de dissemelhança entre duas soluções. Por exemplo sejam a, b, c, d e e as operações processadas por uma determinada máquina. Suponhamos que numa solução a sequência de processamento é $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e$ e noutra $b \rightarrow c \rightarrow d \rightarrow e \rightarrow a$. A distância definida por Aiex, Binato e Resende diz que esta máquina contribui com 5 unidades para a diferença entre as duas soluções, uma vez que todas as operações estão em ordens diferentes;

enquanto que a distância por nós definida assume o valor 2, a e b têm antecessores diferentes.
A forma adoptada de gestão da lista de soluções elite é muito semelhante à descrita por Aiex, Binato e Resende [ABR01].

```
   GerirElite(L, maxL, x)
(1)    d_min := ∞
(2)    if L = ∅
(3)        L := {x}
(4)        f_ = f‾ = f(x)
(5)        return L
(6)    if 0 < |L| < maxL
(7)        L := L ∪ {x}
(8)        f_ = min(f(x), f_)
(9)        f‾ = max(f(x), f‾)
(10)       Distância(x, L)
(11)       return L
(12)   if |L| == maxL
(13)       if f(x) < f_:
(14)           L := L ∪ {x}
(15)           f_ = f(x)
(16)           y ∈ L : f(y) = f‾
(17)           L := L\{y}
(18)           f‾ := max{f(y), y ∈ L}
(19)           Distância(x, L)
(20)           return L
(21)       if f(x) ∈ [f_, f‾]
(22)           maxd := Distância1(x, L)
(23)           if maxd > d_min
(24)               L := L ∪ {x}
(25)               y ∈ L : f(y) = f‾
(26)               L := L\{y}
(27)               f‾ := max{f(y), y ∈ L}
(28)               Distância2(x, L)
(29)               return L
(30)   return L
```

De cada vez que uma nova solução é candidata a entrar na lista, calcula-se a distância entre ela e cada uma das soluções guardadas. Se a maior distância entre a nova solução e as da lista, for superior à menor das maiores distâncias entre duas soluções da lista, então considera-se que a nova solução acrescenta diversidade ao conjunto e, desde que o seu *makespan* não seja maior que o pior *makespan* na lista, a solução é incorporada. Quando a lista já atingiu a sua dimensão máxima é necessário remover uma solução. A remoção é orientada por critérios de qualidade, saindo a solução com maior *makespan* presente na lista.

```
   Distância(x, L)
(1)    maxd := 0
(2)    dmax(x) := 0
(3)    foreach y ∈ L
(4)        Δ(x, y)
(5)        if Δ(x, y) > d_max(y)
(6)            d_max(y) := Δ(x, y)
(7)            max(y) := x
(8)        if Δ(x, y) > d_max(x)
(9)            d_max(x) := Δ(x, y)
(10)           max(x) := y
(11)   d_min := min{d_max(y)}
```

d_{min} =menor das maiores distâncias entre soluções elite; $\underline{f}$ =menor dos *makespans* das soluções elite; $\overline{f}$ =maior dos *makespans* das soluções elite; $maxd$ =maior das distâncias entre a solução candidata e todas as soluções elite; $max(y)$ =solução elite mais distante da solução elite y; $d_{max}(y)$ =maior das distâncias entre uma solução elite y e todas as restantes.

```
DistÂncia1(x, L)
(1)   maxd := 0
(2)   foreach y ∈ L
(3)       Δ(x, y)
(4)       if Δ(x, y) > maxd
(5)           maxd := Δ(x, y)
(6)   return maxd

DistÂncia2(x, L)
(1)   foreach y ∈ L
(2)       if Δ(x, y) > d_max(y)
(3)           d_max(y) := Δ(x, y)
(4)           max(y) := x
(5)       if Δ(x, y) > d_max(x)
(6)           d_max(x) := Δ(x, y)
(7)           max(x) := y
(8)   d_min := min{d_max(y)}
```

3.5.2 Religação de soluções

```
ReligarElite(L, x*)
(1)   LRS := {}
(2)   foreach x ∈ L
(3)       LRS := Religar(x)
(4)   foreach x ∈ LRS
(5)       if f(x) < f(x*)
(6)           x* := x
(7)       L := GerirElite(L, maxL, x)
(8)       LRS := LRS\{x}
(9)       if x ∈ L
(10)          LRS := Religar(x)
(11)  return x*
```

LRS :=lista das melhores soluções encontradas no processo de religação de soluções elite.

A religação de soluções consiste em, dadas duas soluções admissíveis, considerar uma como ponto de partida e outra como ponto de chegada e construir um caminho entre elas, incorporando sucessivamente características da solução de chegada na solução de partida. O objectivo da religação de soluções é o de pesquisar o espaço em torno de boas soluções, que não terá sido visitado pelos procedimentos de procura. A construção do caminho é feita utilizando movimentos que poderão conduzir a soluções não admissíveis, mas como o processo termina sempre na solução guia, há a garantia de retorno à admissibilidade. A esperança de que os caminhos construídos sejam profícuos reside no facto de estarmos a ligar soluções diferentes de boa qualidade.

Se a solução devolvida pelo processo de religação de soluções for suficientemente boa e diferente para entrar na lista de soluções elite, ela é incluída na lista. Enquanto forem sendo incluídas novas soluções na lista de elites, é executado a religação de soluções entre cada nova solução e a solução da lista que lhe é mais diferente.

```
RELIGAR(x)
(1)   S_1 := x
(2)   S_2 := max(x)
(3)   x^+ := CAMINHOS(S_1, S_2)
(4)   if (x^+ admissível)
(5)       x^+ := PROCURALOCAL(x^+, M)
(6)       LRS := LRS ∪ {x^+}
(7)   x^+ := CAMINHOS(S_2, S_1)
(8)   if (x^+ admissível)
(9)       x^+ := PROCURALOCAL(x^+, M)
(10)      LRS := LRS ∪ {x^+}
(11)  return LRS

CAMINHOS(A, B)
(1)   γ := γ(A, B)
(2)   S_c^+ := None, f(S_c^+) := ∞
(3)   S_c := A
(4)   while γ > 2
(5)       y := None, f(y) := ∞
(6)       foreach operação i ∈ S_c : pos(i)_{S_c} ≠ pos(i)_B
(7)           j : pos(j)_{S_c} == pos(i)_B
(8)           S'_c := S_c com pos(i)_{S'_c} := pos(j)_{S_c} e pos(j)_{S'_c} := pos(i)_{S_c}
(9)           if f(S'_c) < f(y)
(10)              y := S'_c
(11)      S_c := y
(12)      Reduzir γ
(13)      if f(S_c) < f(S_c^+)
(14)          S_c^+ := S_c, f(S_c^+) := f(S_c)
(15)  return S_c^+
```

$\gamma(A, B)$ =número de operações que nas soluções A e B se encontram em ordens diferentes na sequência de processamento; $pos(i)_{S_1}$ = posição da operação i na sequência de processamento da solução S_1.

A construção do caminho entre duas soluções adoptada foi a descrita por Aiex, Binato e Resende [ABR01], que consiste em colocar uma de cada vez, na solução inicial, as operações na ordem de sequenciamento em que estão na solução guia. A cada passo são avaliados todos os movimentos possíveis, utilizando para tal o algoritmo de Taillard anteriormente descrito, e escolhe-se o melhor deles para ser realizado. A soluções inadmissíveis associa-se um valor muito grande de forma a enviesar o caminho para a admissibilidade. O procedimento guarda a melhor solução visitada no caminho. Esta solução poderá não ser um óptimo local pelo que se aplica procura local à melhor solução encontrada no processo de religação de soluções. O caminho de uma solução S_1 para uma solução S_2, tal como é construído, é diferente do que vai da solução S_2 para a solução S_1, por isso são ambos realizados.
Segue-se um exemplo fictício da construção de um caminho entre duas soluções:

Seja **(a,e,b,c,d)** a sequência de operações numa determinada máquina na solução inicial e **(a,b,d,e,c)** a sequência da mesma máquina na solução guia.

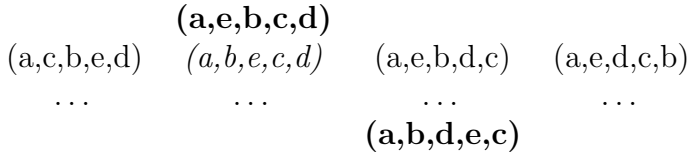


Figura 3.1: Primeira série de movimentos

A operação a está na mesma posição nas duas soluções. O primeiro dos quatro movimentos possíveis corresponde a trocar a operação e, que na solução inicial está na segunda posição e na solução guia na quarta, com a operação c, pois é esta que se encontra na quarta posição na solução inicial.
Suponhamos que o segundo movimento é aquele a que corresponde a melhor das quatro soluções avaliadas. A solução com a sequência *(a,b,e,c,d)* será considerada a nova solução inicial.

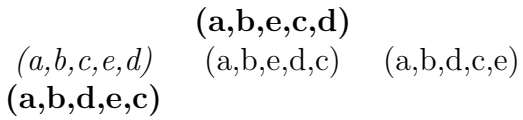


Figura 3.2: Segunda série de movimentos

Na segunda fase existem apenas três movimentos realizáveis, suponhamos que o primeiro é o melhor. Entre *(a,b,c,e,d)* e a solução guia existem apenas duas operações trocadas na sequência, logo o caminho termina.

As opções aqui tomadas no desenho do processo de religação de soluções tiveram em atenção o estudo realizado por Ghamlouche, Crainic e Gendreau [GCG02], sobre estratégias de construção do conjunto de soluções (visitadas por uma procura tabu) que seriam objecto de religação, e formas de escolher, do referido conjunto, as soluções inicial e guia para realizar os caminhos.
Os autores testaram 6 estratégias de construção do conjunto alvo versus 6 formas de escolher os pares de soluções para construir cada caminho.

As 6 estratégias analisadas foram:

E_1 Guardar as soluções que vão actualizando a melhor solução encontrada na procura.

E_2 Guardar os melhores óptimos locais encontrados.

E_3 Guardar os melhores óptimos locais que vão actualizando a melhor solução.

E_4 Guardar uma solução se ela actualiza a melhor solução encontrada até então ou se ela é melhor que a pior solução já guardada e aumenta a diversidade do conjunto.

E_5 Construir o conjunto com a estratégia E_1 e acrescentar-lhe soluções que maximizem a medida de dissemelhança do conjunto.

E_6 Utilizar a estratégia E_5 e acrescentar-lhe soluções que minimizem a dissemelhança do conjunto.

As 6 formas estudadas para escolher as soluções inicial e guia foram:

F_1 A solução inicial é a melhor do conjunto, a solução guia a pior do conjunto.

F_2 A solução guia é a melhor do conjunto, a solução inicial a segunda melhor.

F_3 A solução guia é a melhor do conjunto, a solução inicial aquela do conjunto mais diferente da escolhida para guia.

F_4 Escolher aleatoriamente as soluções inicial e guia.

F_5 Escolher para soluções inicial e guia as duas soluções mais diferentes do conjunto.

F_6 A solução inicial é a pior do conjunto, a solução guia a melhor.

Os autores concluíram que a combinação de estratégia de construção com a escolha de soluções a ligar que apresentava melhores resultados era E_3 com F_5, sendo a segunda melhor opção o par (E_3, F_3).

Capítulo 4

Experiência computacional

Acatando as sugestões presentes no artigo *Guidelines for Designing and reporting on Computational Experiments with Heuristic Methods* de Barr, Golden, Kelly, Stewart e Resende [BGK+95], levámos a cabo uma experiência computacional com o objectivo de avaliar o comportamento dos algoritmos construídos, utilizando instâncias conhecidas para avaliar a rapidez com que são encontradas boas soluções, evidenciando a melhor solução alcançada e o tempo até a atingir. Pretendemos mostrar a contribuição de cada componente dos algoritmos para a performance geral, e na medida do possível, comparar os resultados obtidos com os alcançados por heurísticas documentadas.
Os testes computacionais foram realizados num AMD Athlon a 1.4 GHZ com 512 Mb de memória RAM. A implementação dos algoritmos descritos foi realizada utilizando a linguagem de programação Python 2.2 e os testes foram efectuados em ambiente Linux.
A eficiência dos algoritmos desenvolvidos foi testada com base na sua performance em 13 instâncias consideradas de difícil resolução, encontradas na Livraria de Investigação Operacional (http://mscmga.ms.ic.ac.uk). Uma designada FT10 proposta por Fisher e Thompson (1963), doze devidas a Lawrence (1984) designadas LA.
A Tabela 4.1 apresenta para cada instância, a respectiva designação, as suas dimensões em termos de número de trabalhos por número de máquinas e o valor óptimo, ou caso este não seja conhecido, os melhores limites inferior e superior conhecidos.
Os algoritmos desenvolvidos têm como característica construir uma solução admissível que vai sendo melhorada posteriormente. Para avaliar o seu comportamento com um mínimo de rigor estatístico foi aplicada a seguinte estatística, que passamos a descrever recorrendo a um exemplo fictício.
Suponhamos que se guarda sucessivamente para determinado procedimento a qualidade da melhor solução obtida até então e o respectivo tempo decorrido, assim como o tempo total de cada corrida; e que se fazem 5 corridas independentes do referido procedimento. Sejam os resultados obtidos os que se encontram na Tabela 4.2.
Quer-se determinar o tempo médio que é necessário esperar até alcançar uma solução com determinado valor. Para esse efeito são calculadas médias dos tempos decorridos para cada um dos valores obtidos na experiência das 5 corridas independentes, da

Instâncias	**(N×M)**	**Opt. (LI,LS)**
FT10	(10×10)	930
LA02	(10×05)	655
LA19	(10×10)	842
LA21	(15×10)	1046
LA24	(15×10)	935
LA25	(15×10)	977
LA27	(20×10)	1235
LA29	(20×10)	(1142,1153)
LA36	(15×15)	1268
LA37	(15×15)	1397
LA38	(15×15)	1196
LA39	(15×15)	1233
LA40	(15×15)	1222

Tabela 4.1: Instâncias

Corrida 1		Corrida 2		Corrida 3		Corrida 4		Corrida 5	
valor	tempo	valor	tempo	valor	tempo	valor	tempo	valor	tempo
50	12330	50	12330	50	14890	48	12330	50	13450
48	13450	48	14890	47	15850	47	15670	48	16600
47	14890	47	15670		17000	32	16900		17500
32	15670	32	16600				17500		
25	17200		17500						
	17500								

Tabela 4.2: Valores de *makespan* e tempos de 5 corridas independentes

seguinte forma:

$$tm_{50} = \frac{12330 + 12330 + 14890 + 12330 + 13450}{5} = 13066$$

Na corrida 4 o primeiro valor menor ou igual a 50 que se atinge é 48 depois de 12330 milésimos de segundo; por isso 12330 é a contribuição da corrida 4 para a média de tempos do valor 50.

$$tm_{48} = \frac{13450 + 14890 + 15850 + 12330 + 16600}{5} = 14624$$

Na corrida 3 o primeiro valor não superior a 48 é atingido após 15850 milésimos de segundo.

$$tm_{47} = \frac{14890 + 15670 + 15850 + 15670 + 17500}{4} = 19895$$

A corrida 5 termina após 17500 milésimos de segundo sem que seja alcançado um valor inferior ou igual a 47. Assim acrescenta-se a parcela 17500 à média, sem que se divida pela quinta corrida.

$$tm_{32} = \frac{15670 + 16600 + 17000 + 16900 + 17500}{3} = 27890$$

O valor 32 não é atingido nas corridas 3 e 5. Na média são incluídas parcelas correspondentes aos tempos máximos destas corridas, fazendo a divisão apenas por 3; o número de vezes que o valor foi atingido.

$$tm_{25} = \frac{17200 + 17500 + 17000 + 17500 + 17500}{1} = 86700$$

Finalmente, como o valor 25 só é atingido numa corrida, utilizam-se os tempos máximos de todas as outras.

Com os tempos médios de espera calculados para cada valor atingido na experiência podem traçar-se gráficos de qualidade versus tempo, para mais facilmente analisar o comportamento do procedimento e tirar conclusões.

Se na construção da estatística fossem usadas apenas as 4 primeiras corridas, os resultados seriam ligeiramente diferentes. A Tabela 4.3 lista os tempos médios de espera por um determinado valor, na primeira coluna, calculados utilizando as 5 corridas, na segunda coluna apenas as primeiras 4. Como se verifica na tabela, a estatística construída é razoavelmente sensível ao número de corridas usadas nos cálculos, nomeadamente nos cálculos para os valores menos frequentes. Torna-se pois necessário determinar com cuidado qual o número de corridas a executar. Para que os resultados tenham algum significado estatístico, exige-se que o número de corridas presentes na amostra utilizada para calcular a estatística seja pelo menos 20, mas se existirem parâmetros a estimar no algoritmo este valor terá de ser aumentado. O número máximo de corridas será condicionado pelo tempo disponível para toda a experiência.

	temp. med. 5 cor.	temp. med. 4 cor.
50	13066	12970
48	14624	14130
47	19895	15520
32	27890	22057
25	86700	69200

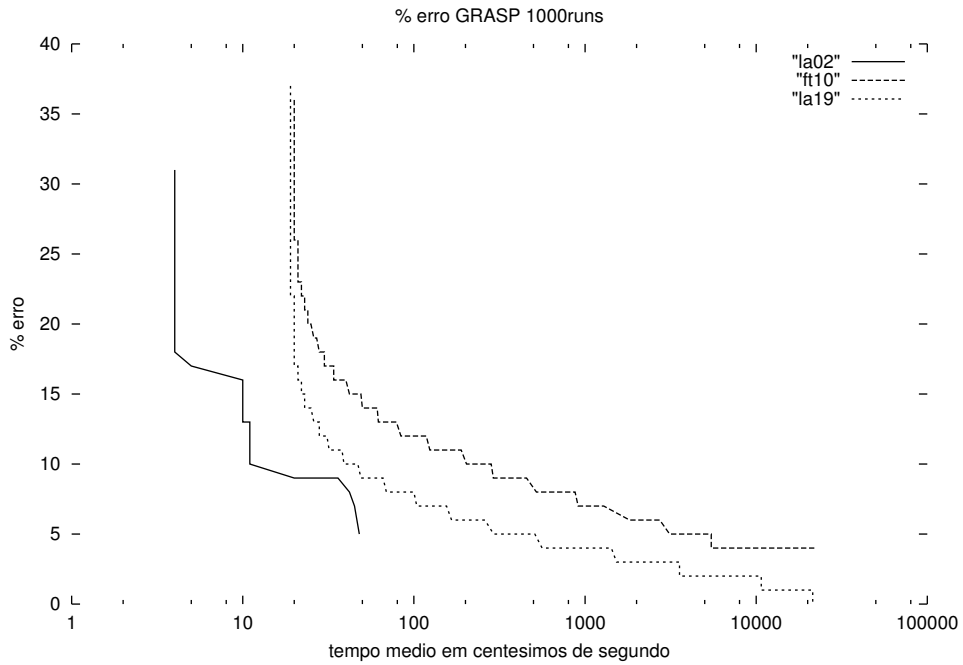
Tabela 4.3: Tempos médios de espera calculados com 5 e 4 corridas independentes

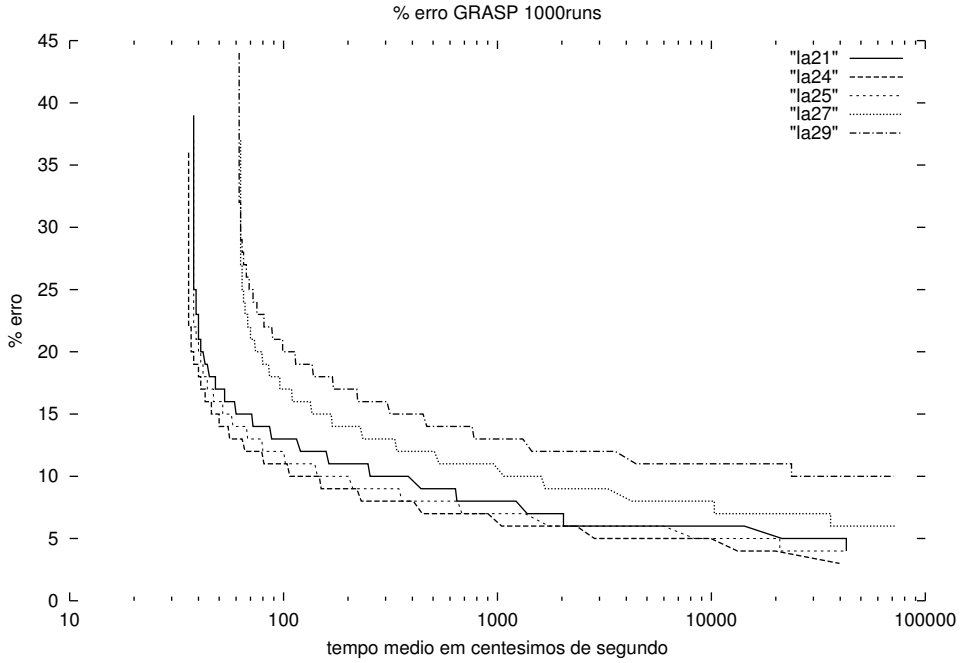
4.1 Fase construtiva

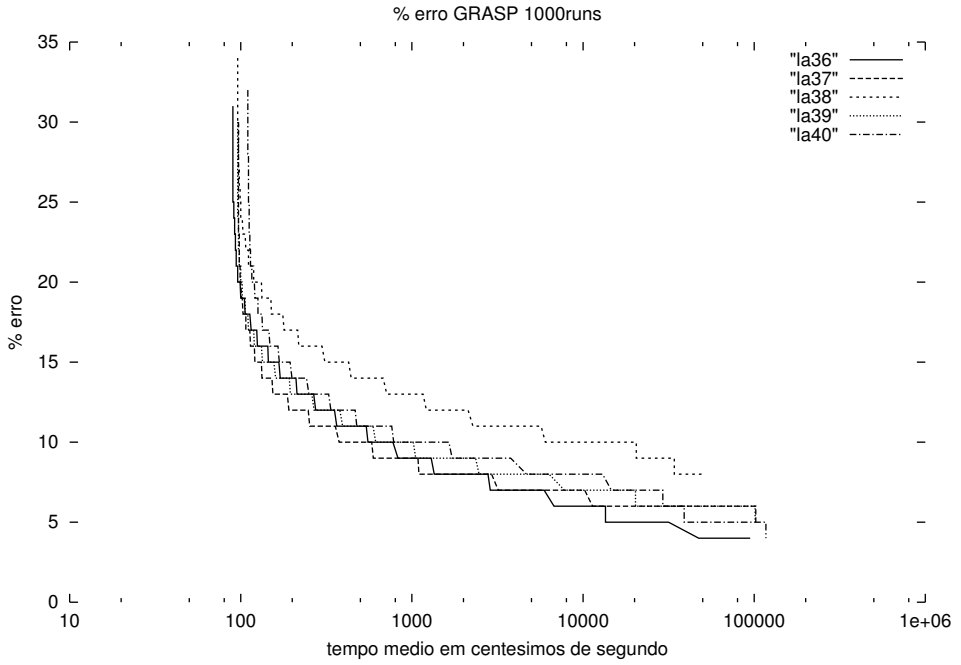
Comecemos por avaliar o comportamento do procedimento construtivo GRASP. Foram feitas 1000 corridas independentes e calculados os tempos médios de espera para cada valor de *makespan*, da forma anteriormente descrita.
Com o objectivo de comparar a qualidade das soluções obtidas na fase construtiva de todas as instâncias, foram construídos gráficos que relacionam a percentagem de erro com o tempo médio de espera. A percentagem de erro é calculada utilizando a fórmula

$$\%erro = 100 \times \frac{makespan\ obtido - valor\ óptimo}{valor\ óptimo}$$

Para a instância *la29* não se conhece o valor óptimo, sendo substituído na fórmula pelo melhor limite inferior.
Nos gráficos utiliza-se uma escala logarítmica para os valores dos tempos médios para ser mais fácil a visualização dos mesmos.
A leitura de um gráfico com as 13 instâncias seria um pouco difícil, pelo que se dividiram as instâncias em 3 grupos, de acordo com as suas dimensões (trabalhos×máquinas).





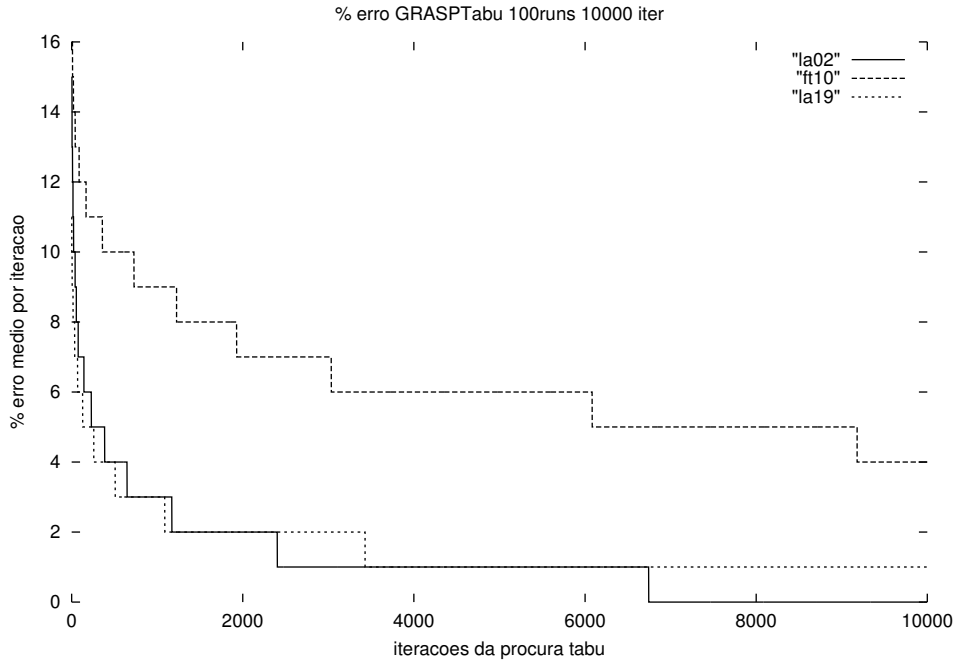


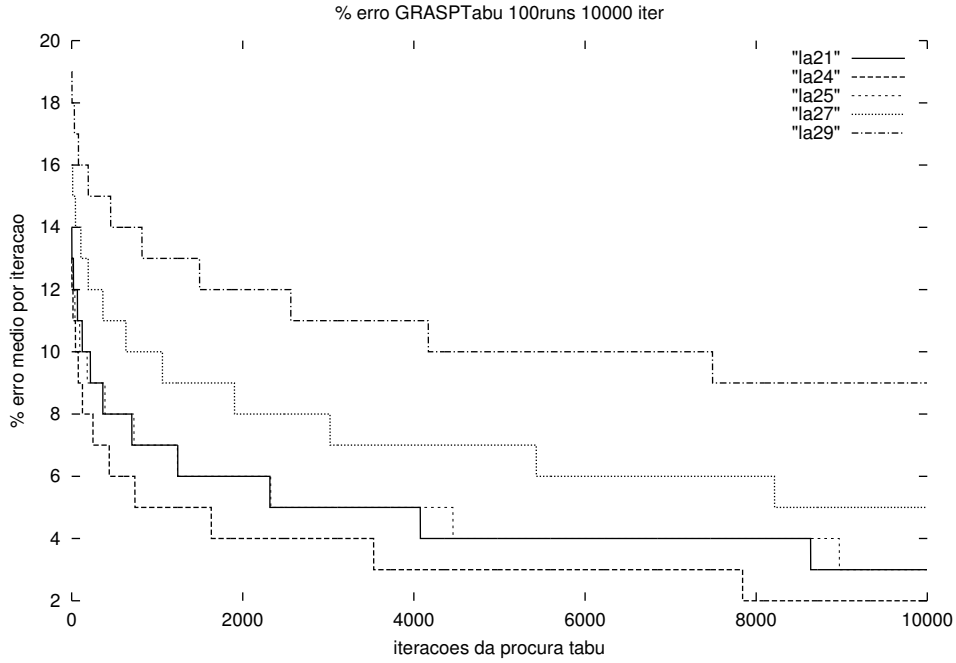
Ao observar os gráficos verificamos que o algoritmo GRASP tem um comportamento muito semelhante em todas as instâncias testadas, exceptuando talvez a instância $la02$. Esta instância é a menor de todas, em termos de número de trabalhos e máquinas. Analisando os gráficos podemos concluir que a qualidade das soluções encontradas não é muito sensível à dimensão das instâncias, o que indica alguma robustez do método. Para a maioria das instâncias o processo construtivo começa por obter soluções com uma percentagem de erro que varia entre 30% e 40% e termina com soluções que ficam a cerca de 5 valores percentuais do valor óptimo. As excepções a esta regra

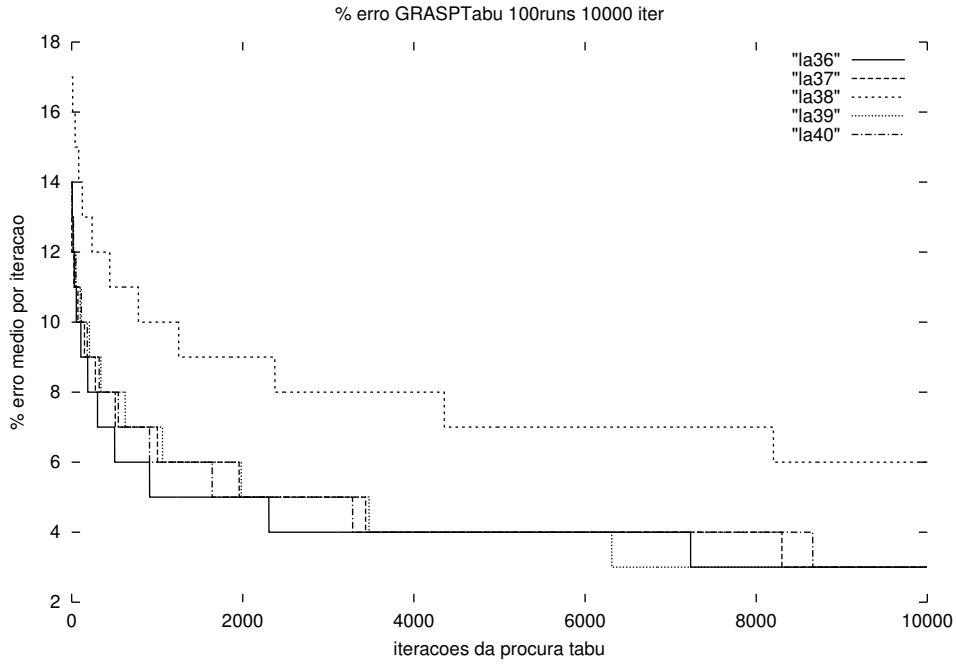
são verificadas na instância *la29*, que começa com soluções com cerca de 45% de erro e termina com 10% de erro, na instância *la19*, onde se atinge a solução óptima e na instância *la*38, em que a melhor solução encontrada fica a um pouco mais de 5% do óptimo. Relembre-se que não se conhece o valor óptimo da instância *la*29, e que por isso o cálculo da percentagem de erro está inflacionado, uma vez que se utiliza o melhor limite inferior conhecido. Note-se que as instâncias para as quais o processo começa com soluções piores não são necessariamente as que terminam com soluções mais afastadas do óptimo.

4.2 Procura tabu

Na implementação do algoritmo de procura tabu é necessário decidir qual o número máximo de iterações permitidas. Nos artigos referidos neste texto, em que se executa procura tabu, este parâmetro varia entre as 1000 e as 12000 iterações.
Para resolver esta questão foram executadas, para cada instância em estudo, 100 corridas independentes do algoritmo GraspTabu com 10000 iterações de procura tabu. Para cada iteração calculou-se o valor médio do *makespan* da melhor solução obtida até então em cada uma das corridas, assim como o tempo médio decorrido até à conclusão de cada iteração. Como seria de esperar, verificou-se que a relação entre o número de iterações realizadas e o tempo decorrido é linear.
Os gráficos construídos relacionam a média da percentagem de erro com o número de iterações decorridas e mais uma vez as instâncias encontram-se divididas em três grupos.



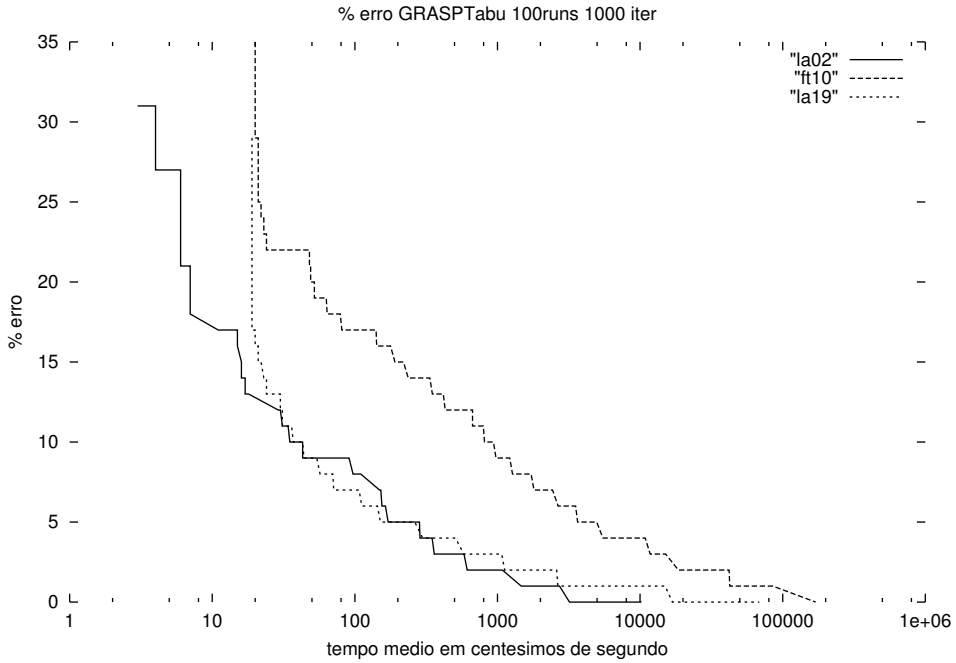


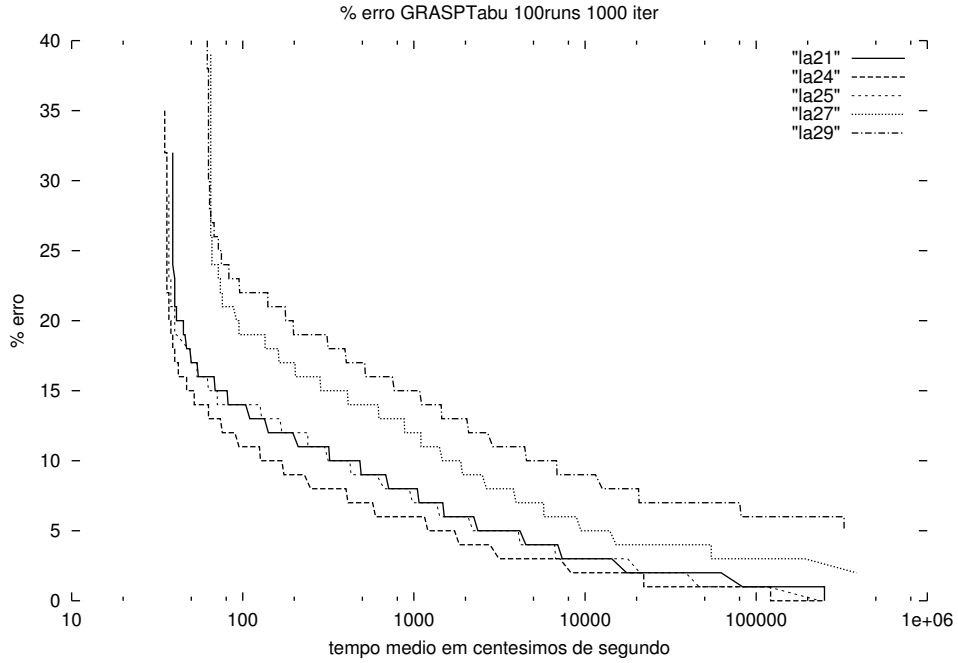


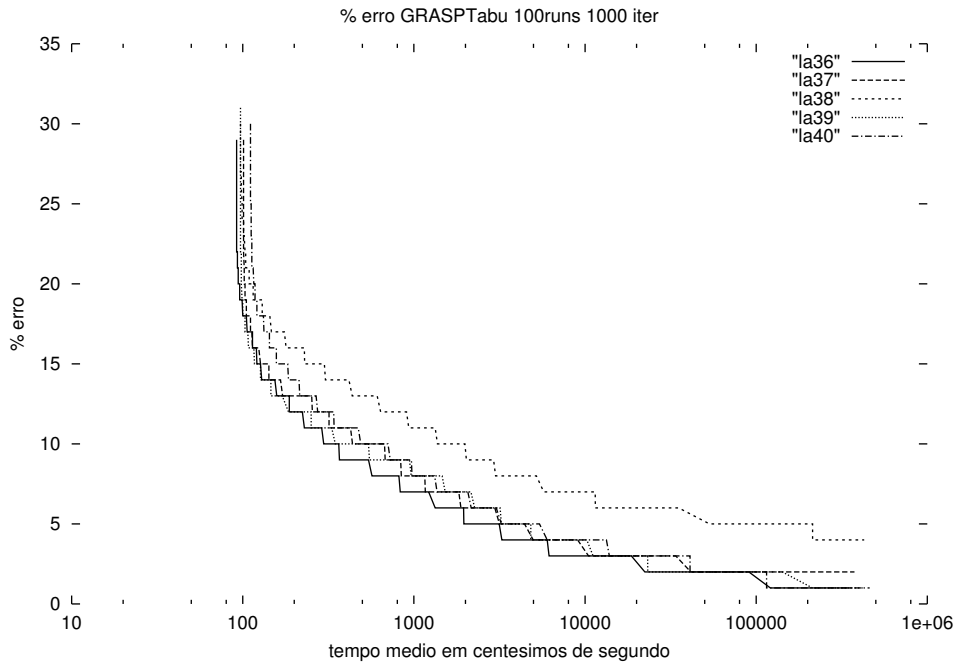
Como se verifica nos gráficos, não existe um número máximo (≤ 10000) de iterações da procura tabu para o qual se possa dizer que, a partir desse valor a média da qualidade da melhor solução obtida não é melhorada. E isto é verdade para todas as instâncias em teste.

Se se pensar na relação existente entre o número de iterações realizadas e o ganho em termos da qualidade da melhor solução obtida até então, talvez 4000 iterações fosse um bom valor de compromisso, dado que, daí até às 10000 iterações o ganho na qualidade é no máximo 1 ou 2% , dependendo da instância.

A decisão de definir o número máximo de iterações da procura tabu igual a 1000, foi tomada considerando dois aspectos. Por um lado a relação entre a qualidade média das soluções e o número de iterações, e o facto do tempo médio decorrido ter uma relação linear com o número de iterações realizadas. Por outro lado, pretende-se incorporar o processo de construção GRASP seguido de procura tabu num algoritmo de religação de soluções elite. Assim o baixo valor máximo fixado para o número de iterações justifica-se com a esperança de que a religação de soluções compense, em termos da qualidade das soluções obtidas, o tempo "poupado"na procura tabu.
Para analisar os resultados obtidos com a execução da procura tabu após a construção de uma solução admissível, foi calculada uma estatística igual à descrita no processo construtivo. São apresentados de seguida, gráficos que relacionam a percentagem de erro das soluções obtidas após 1000 iterações da procura tabu, executada sobre as soluções obtidas pelo algoritmo GRASP, versus o tempo médio de espera (em centésimos de segundo), calculado com base em 100 corridas independentes.





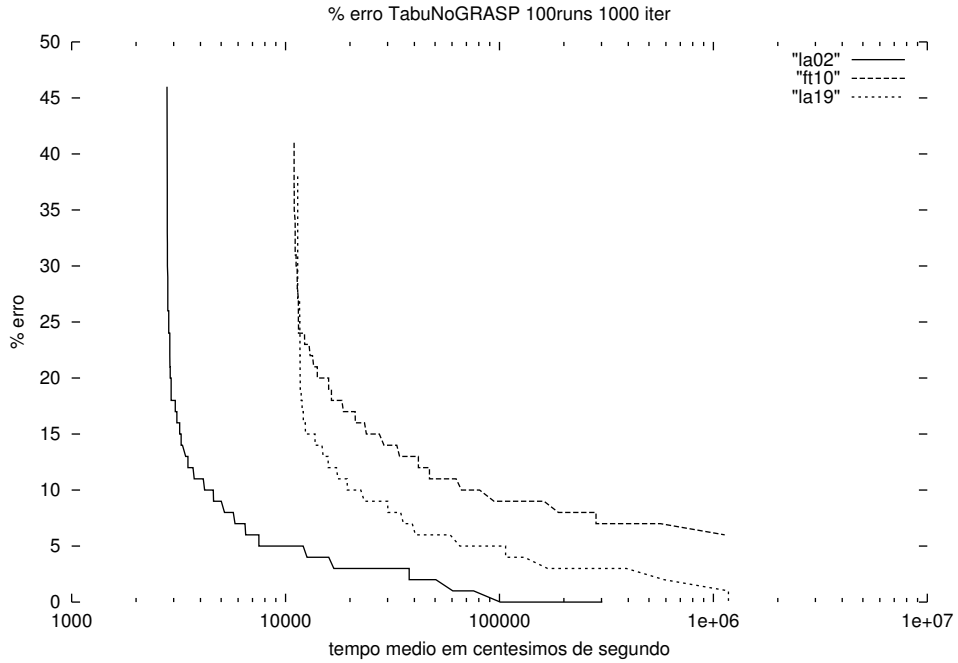


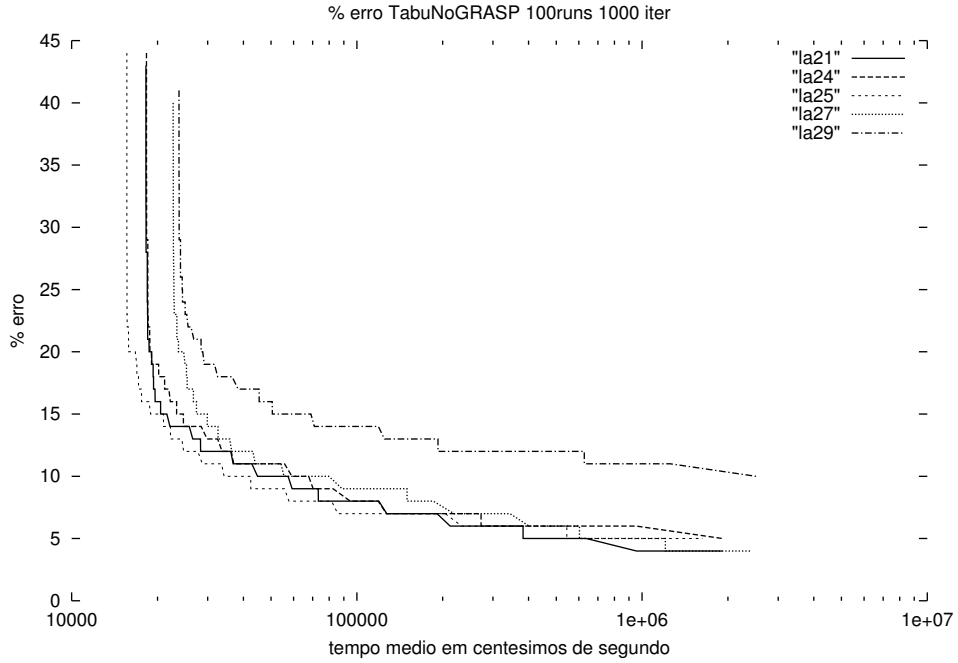
A melhoria da qualidade da melhor solução obtida com o algoritmo GraspTabu relativamente à alcançada com o algoritmo Grasp é evidente. Mesmo assim os resultados poderão estar aquém das potencialidades da procura tabu, pois como referimos o fixar do número máximo de iterações no valor 1000 pode ser um factor limitativo.
O declive da curva descrita pela evolução da percentagem de erro das soluções é mais acentuado no primeiro gráfico, referente às instâncias de menor dimensão. Ainda assim a diferença entre os referidos declives de todas as instâncias não parece ser significativa, pelo que arriscamos concluir que o comportamento do algoritmo é suficientemente robusto relativamente à variação na grandeza das instâncias.

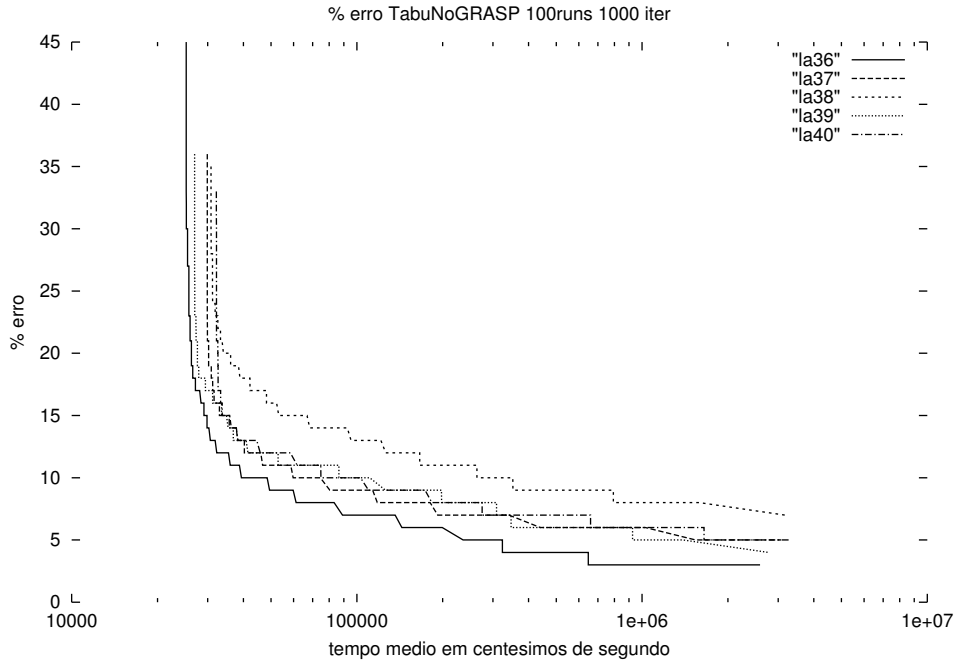
Da observação dos resultados obtidos surge a seguinte questão: Seria preferível incorporar a procura tabu no processo de construção da solução admissível em vez de a executar só sobre soluções completas?
A consciência de que tal transformação iria aumentar significativamente a complexidade do algoritmo global, faz com que se enverede por esta opção só se a vantagem sobre o anterior for esmagadora.

4.2.1 Procura tabu incorporada no Grasp

Para avaliar a performance do algoritmo com a procura tabu incluída no processo construtivo procede-se da mesma forma, utilizando-se a estatística descrita anteriormente.
De seguida são apresentados os gráficos dos tempos médios de espera versus valor da percentagem de erro do *makespan* das soluções.





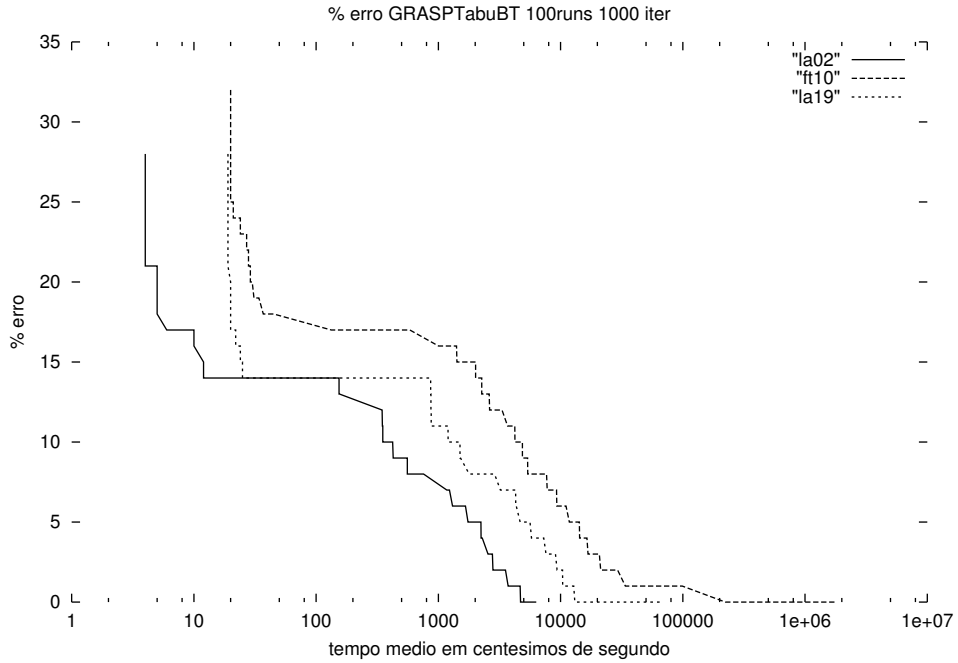


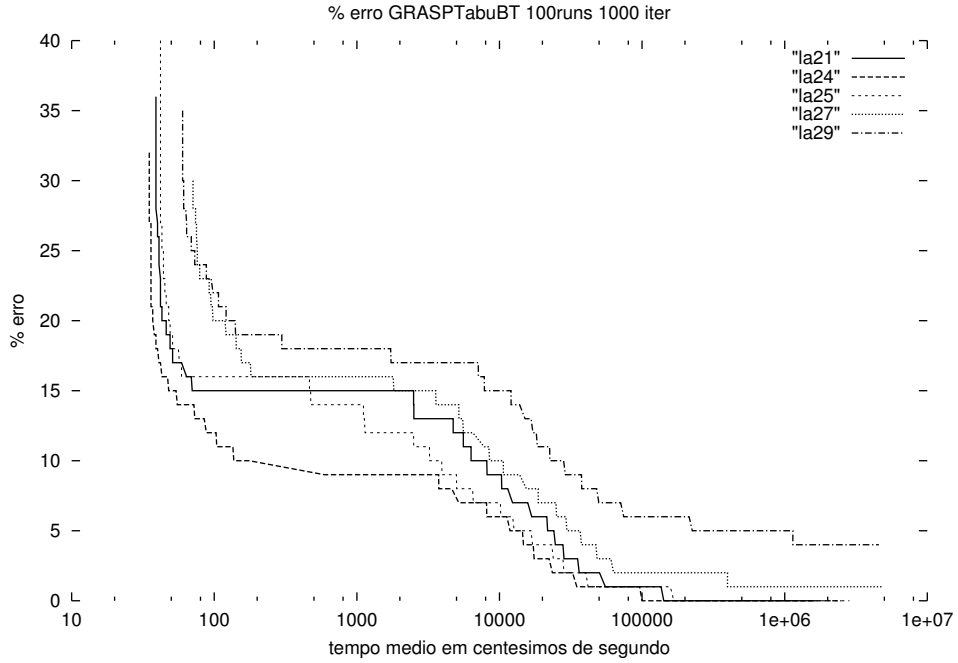
Como facilmente se compreende, este algoritmo consome mais tempo até chegar a uma solução admissível. O facto das curvas da evolução da percentagem de erro apresentarem uma fase praticamente vertical dos 45% aos 20% de erro sugere que, a qualidade das soluções completas obtidas nas várias corridas independentes tem uma variância elevada. Repare-se que aqui a percentagem de erro inicial é, regra geral, superior à verificada com os resultados do GRASP sem procura tabu, logo a inclusão desta no processo construtivo não confirmou a hipótese formulada de que, quanto maior a qualidade das soluções parciais maior a qualidade da solução admissível construída.

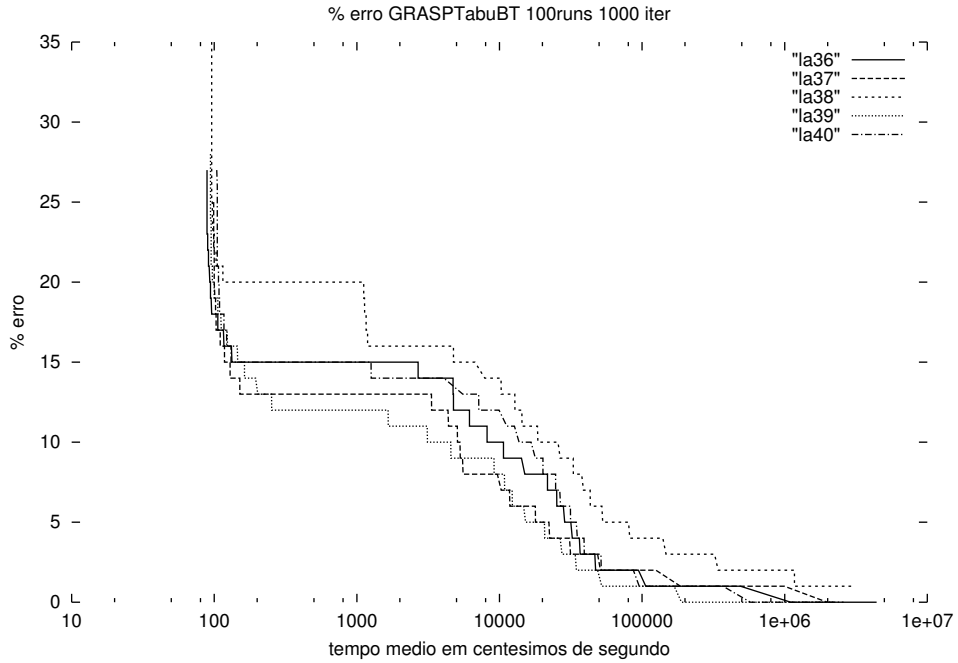
O que acontece é que ao começar o processo construtivo com soluções parciais que correspondem a mínimos locais "mais profundos" (melhores) das soluções na vizinhança definida para a procura tabu, nas fases posteriores é mais difícil ao processo de exploração da vizinhança fugir desses óptimos locais parciais e, como consequência, há maior probabilidade da solução final ter menor qualidade. No algoritmo GRASP as soluções parciais são óptimos locais da vizinhança definida para a procura local, o que não significa que sejam óptimos locais da vizinhança utilizada na procura tabu, daí a maior agilidade do algoritmo de procura tabu no primeiro caso.

4.2.2 Procura tabu com retrocesso

Outra hipótese formulada neste trabalho é a de que diversificar a exploração da região admissível do problema, guardando alguma informação sobre a procura já realizada, ao reiniciar a procura tabu a partir de soluções já visitadas, será mais eficiente do que a diversificação realizada reiniciando a procura tabu a partir de novas soluções construídas.
Os gráficos referentes aos resultados obtidos com o algoritmo GRASPTABUBT são apresentados de seguida.







As partes horizontais das linhas que descrevem a evolução da percentagem de erro da melhor solução encontrada na procura tabu com retrocesso, correspondem a fases de reinicio da procura a partir de soluções já visitadas. Numa primeira análise poder-se-ia concluir que o retrocesso consome tempo inutilmente, uma vez que decorrem algumas iterações sem que a melhor solução encontrada em todo o processo seja actualizada. Nessas fases de reinicio, a procura tabu passa por soluções que embora não sejam melhores são diferentes das soluções já visitadas, permitindo que posteriormente, a partir das novas soluções, se actualize a melhor solução encontrada. Pode então dizer-

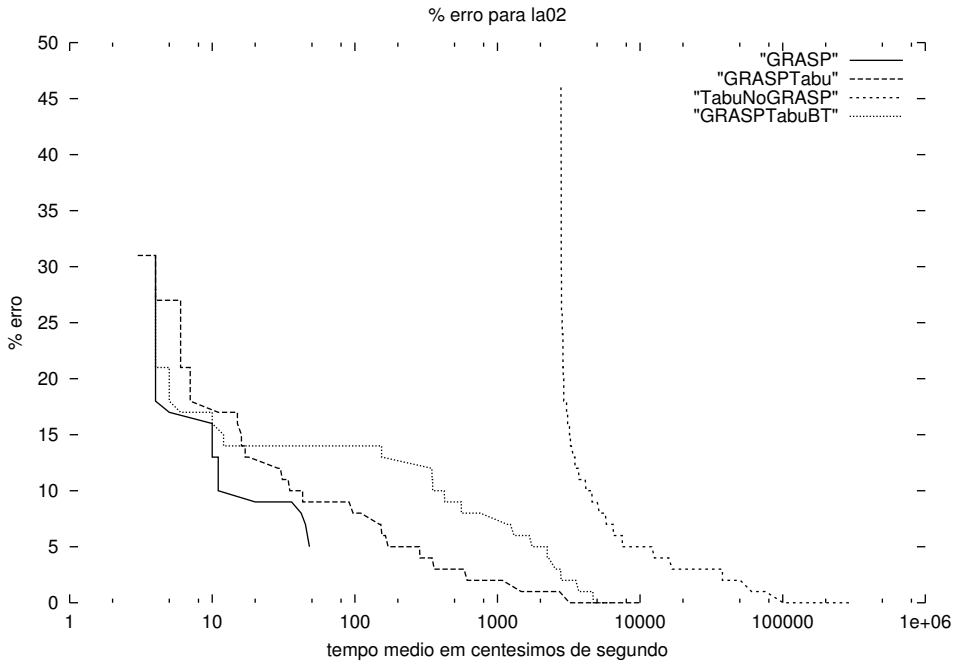
se que o tempo consumido no retrocesso é compensado pela qualidade atingida no final.

4.3 Religação de soluções

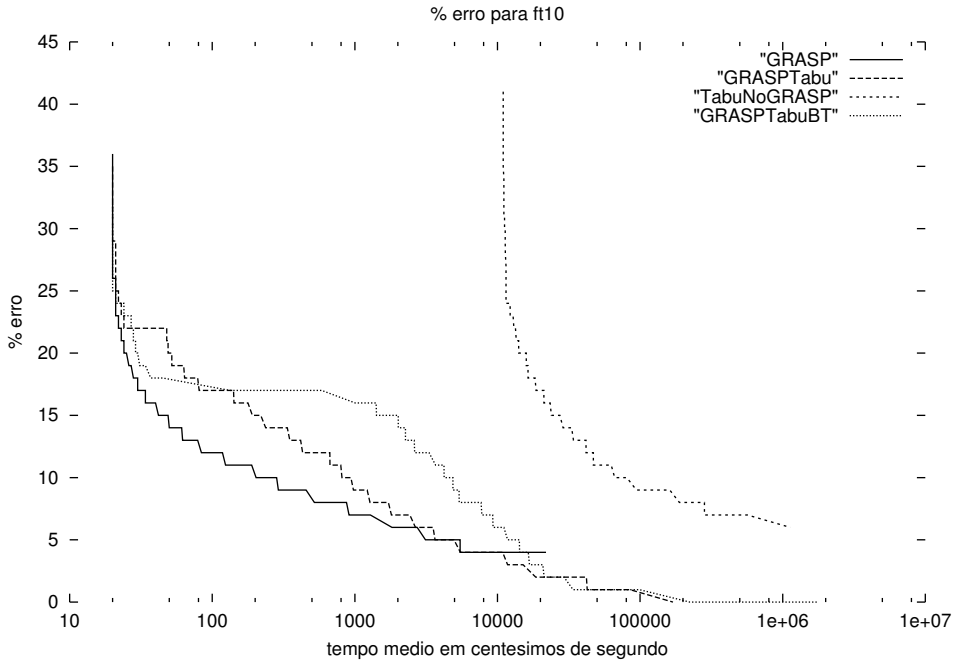
Na experiência computacional realizada, constatou-se que o algoritmo GraspTabuPR nunca conseguiu melhorar as soluções encontradas pela procura tabu. Isto é, nos caminhos construídos entre soluções obtidas pela procura tabu e consideradas elite, nunca em instância alguma se encontrou uma solução que actualizasse a melhor solução visitada.
Porquê este resultado decepcionante mediante as expectativas e o investimento realizado?
Ghamlouche Crainic e Gendreau[GCG02] relatam resultados promissores na aplicação do procedimento de religação de soluções obtidas pela procura tabu, para o problema de desenho de redes. No algoritmo desenvolvido por estes autores, a religação de soluções é implementada não após a conclusão da procura tabu, mas nela incorporada como fase de intensificação, como proposto por Glover[Glo96].
No trabalho de Aiex, Binato e Resende[ABR01] a religação de soluções é executada sobre soluções construídas por um algoritmo GRASP, baseado em regras de sequenciação para o problema de *job shop scheduling*. Comparando o algoritmo GRASP com e sem religação de soluções, os resultados obtidos com a segunda versão são melhores. Também neste caso a religação de soluções é incorporada dentro do GRASP como um processo de intensificação. Aiex, Binato e Resende utilizam uma versão do algoritmo implementada em paralelo.
A opção tomada no algoritmo GraspTabuPR de incluir a religação de soluções como um processo posterior à procura tabu e não como componente integrante do algoritmo de procura tabu, prende-se com preocupações referentes à complexidade do algoritmo resultante.
Talvez o processo de religação de soluções só consiga trazer vantagens, quando as soluções por ele obtidas não sejam consideradas finais, mas sim um ponto de partida para alguma outra forma de exploração do espaço de soluções. No entanto parece mais pacífico acolher a hipótese de que os resultados obtidos se devam a uma gestão pouco eficiente da lista que guarda soluções elite, ou a uma escolha menos acertada da vizinhança subjacente à construção dos caminhos entre duas soluções. A vizinhança que se utilizou na religação de soluções é muito semelhante à descrita em [ABR01], mas aqui as soluções a religar são já de qualidade superior às construídas com o GRASP de Aiex, Binato e Resende. Uma vez que o processo de religar soluções é algo pesado, em termos computacionais, e se optou por religar todas as soluções presentes na lista de elites, com a solução da lista que lhe é mais diferente, a dimensão máxima da lista foi fixada em 5, factor que poderá limitar o processo global.

4.4 Análise comparativa dos algoritmos propostos

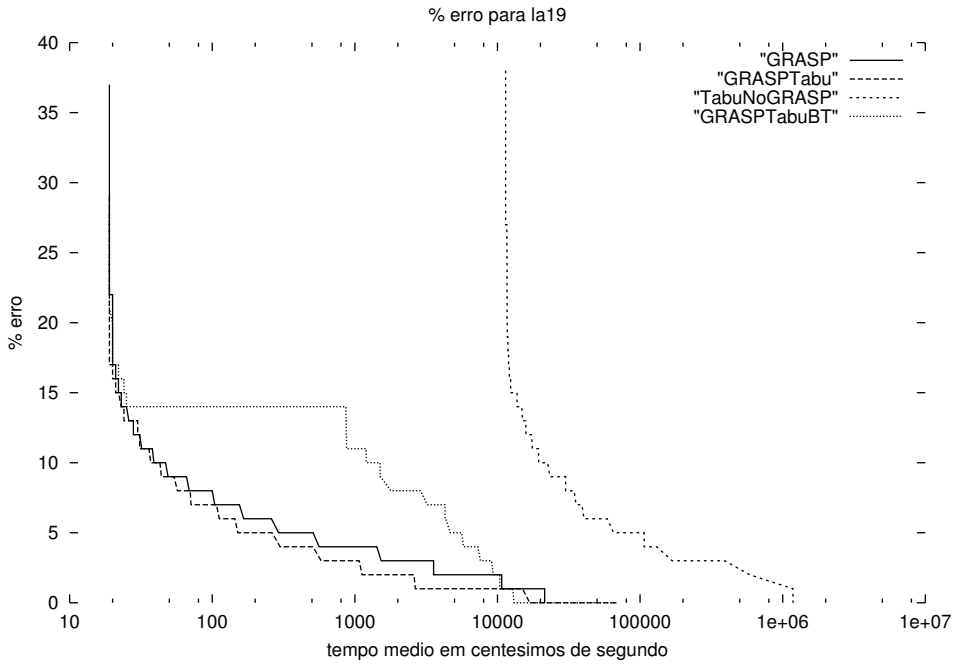
Os gráficos que se seguem apresentam a evolução da percentagem de erro das soluções obtidas com os algoritmos GRASP, GRASPTABU, TABUNOGRASP e GRASPTABUBT, para cada uma das instâncias em estudo.



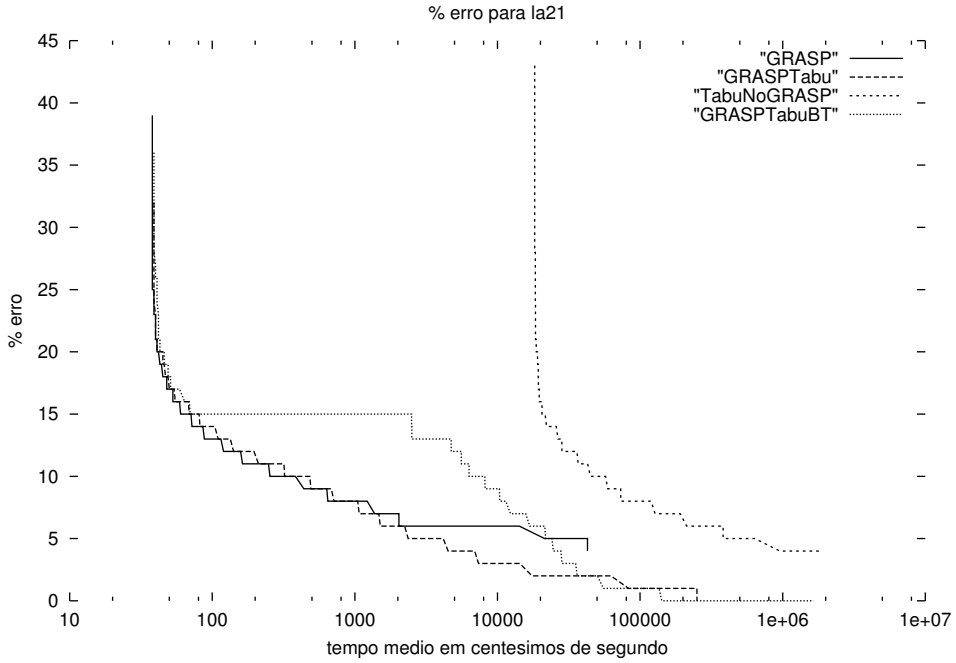
Se tivermos como objectivo obter soluções para esta instância com um erro que poderá ser maior ou igual a 5%, a melhor opção será utilizar o algoritmo GRASP, uma vez que é o que apresenta menor tempo médio de espera por soluções com esta qualidade. Para atingir soluções de qualidade superior será melhor optar pelo algoritmo GRASPTABU.



Também para esta instância a escolha dos algoritmos a utilizar deverá ser; o GRASP se se puder ficar a 5% do óptimo e o GRASPTABU se se pretender soluções de qualidade superior.

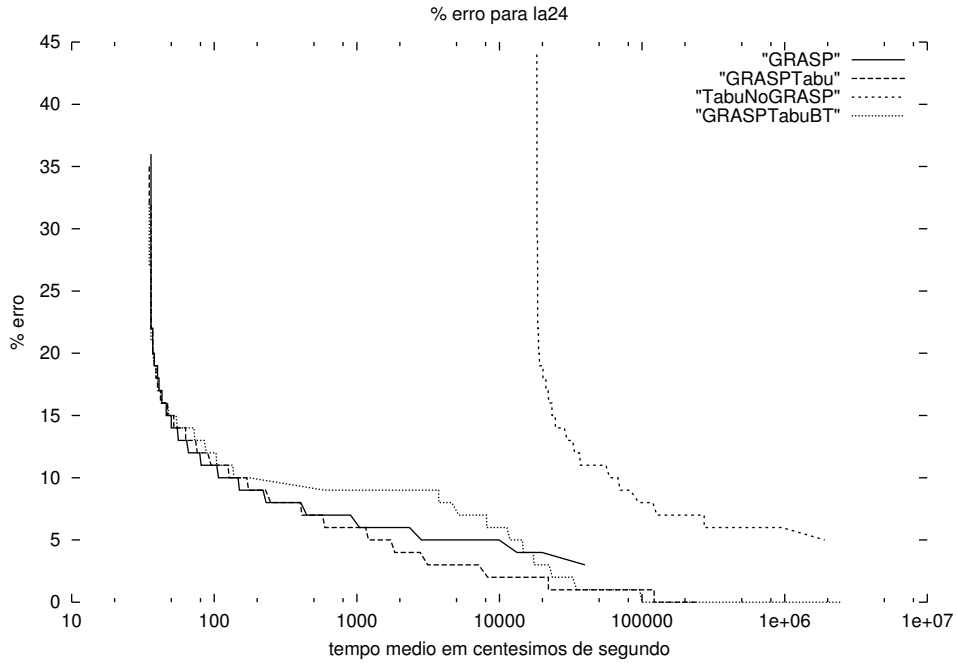


Para a instância *la*19 o algoritmo GRASPTABU domina todos os outros até muito perto do óptimo (cerca de 1% de erro). Para obter a solução óptima, o algoritmo GRASPTABUBT é o que apresenta um tempo médio de espera inferior.

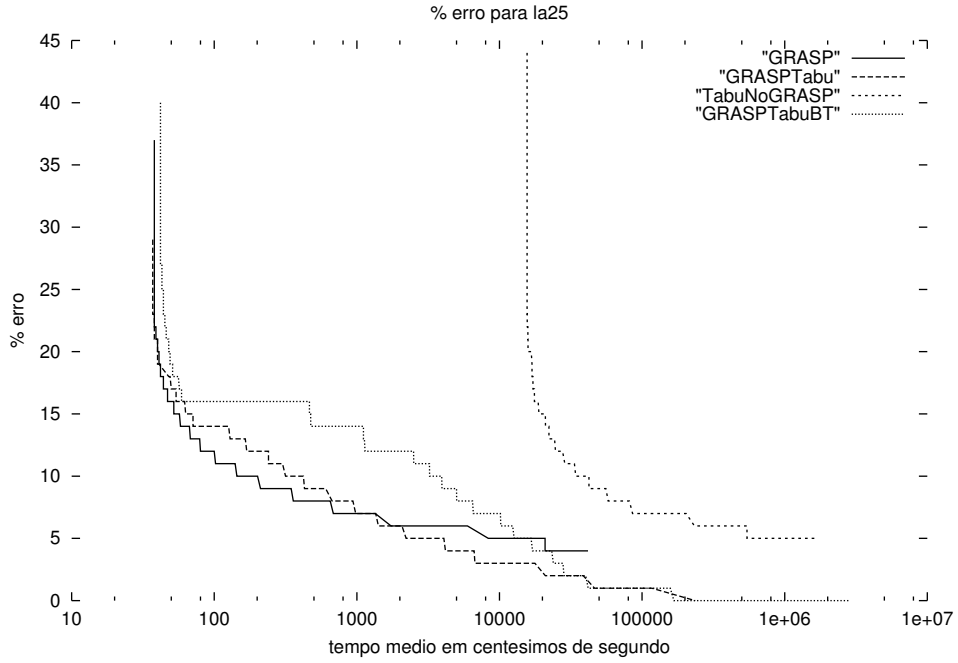


Para obter soluções para a instância *la*21 que poderão ter um erro maior ou igual a 6%, será indiferente utilizar os algoritmos GRASP ou GRASPTABU. Para erros entre

5 e 2%, o algoritmo GraspTabu é o melhor, e daí até ao óptimo o GraspTabuBt domina os restantes.

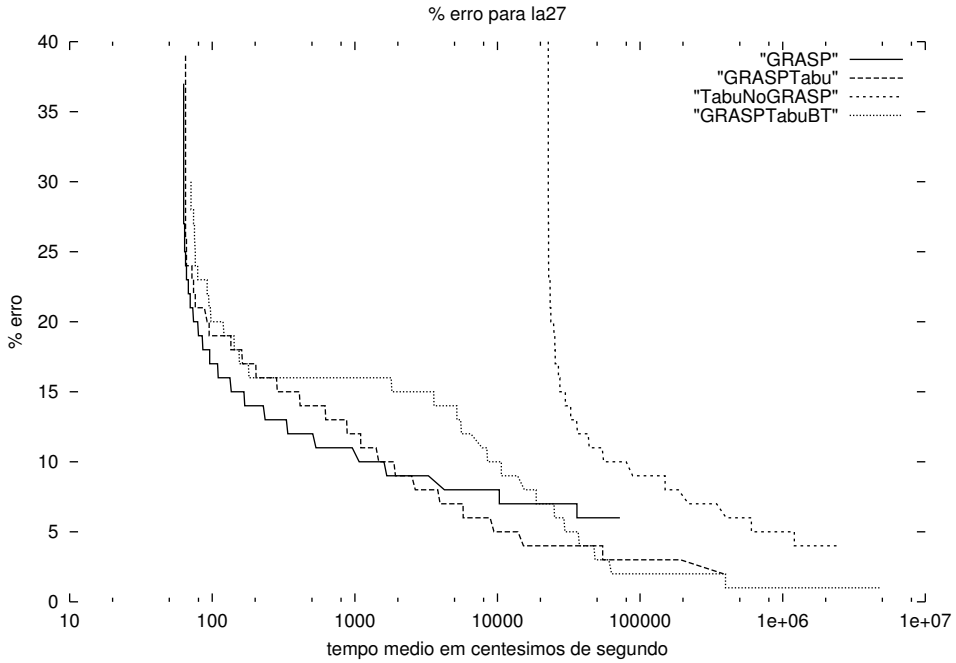


O algoritmo a escolher para resolver a instância *la*24 deverá ser o GraspTabu, a não ser que seja suficiente ficar por soluções com erro não inferior a 7%, neste caso o Grasp é mais rápido.

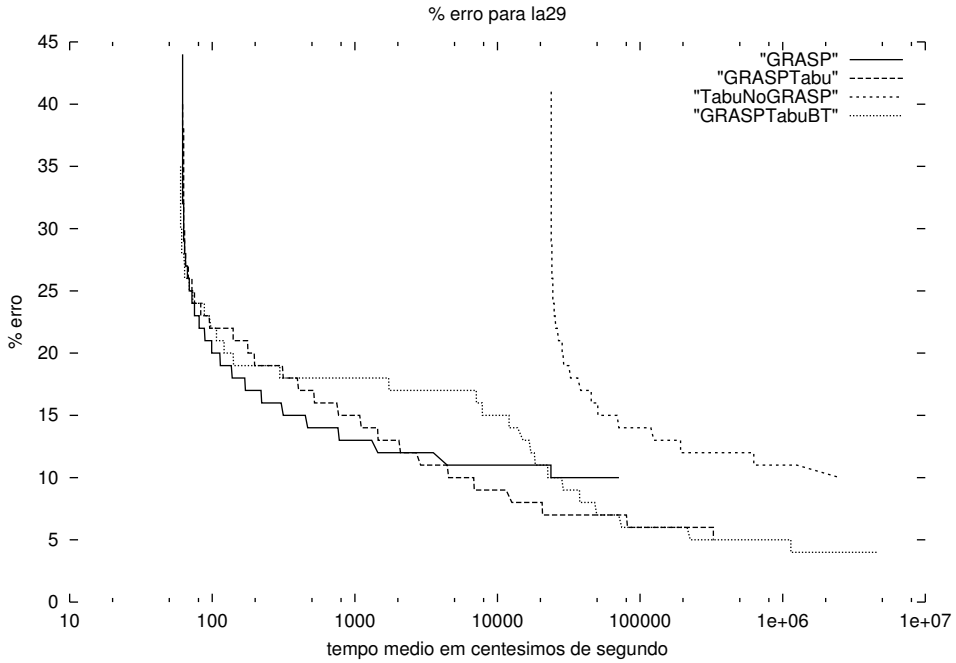


Tal como para a instância anterior, também aqui o algoritmo Grasp domina os outros para erros maiores ou iguais a 7%. Para obter soluções com erros no intervalo [2%,7%]

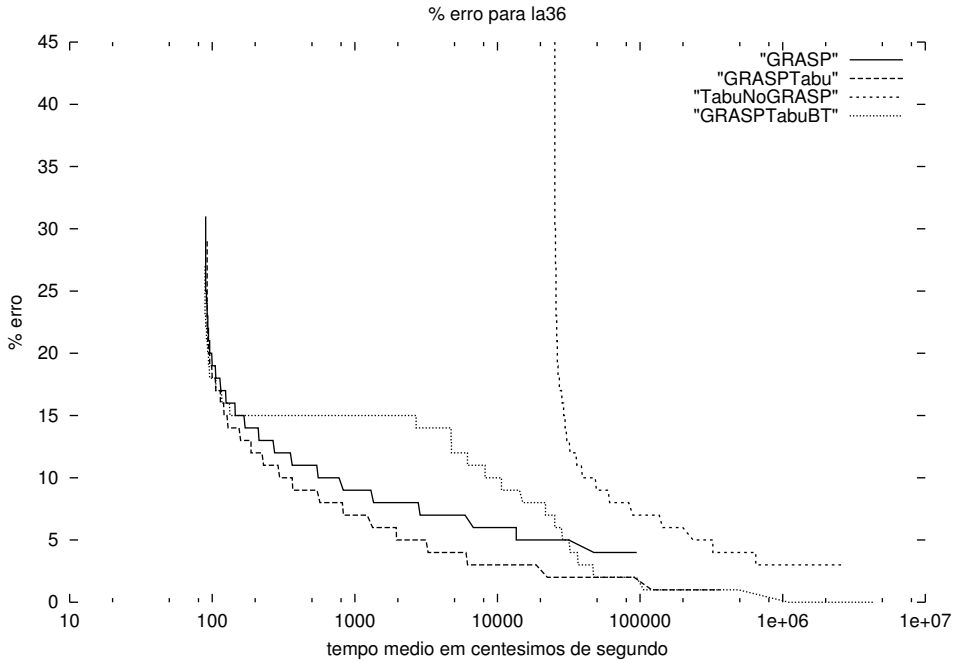
o algoritmo GraspTabu é o escolhido. Para encontrar soluções de qualidade superior deverá utilizar-se o GraspTabuBt.



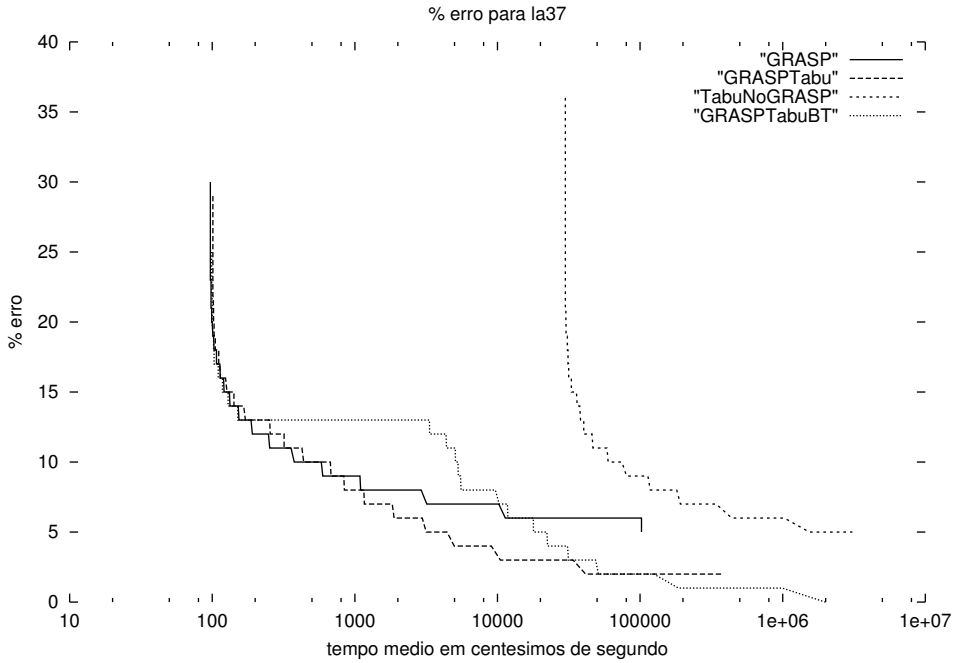
Para a instância *la*27, o algoritmo Grasp apresenta menor tempo médio de espera por soluções com erro maior ou igual a 7%. Para soluções com erros entre 3 e 7%, o GraspTabu é o mais rápido, e para obter soluções com erro não superior a 3% deverá ser utilizado o GraspTabuBt. A melhor solução encontrada para esta instância tem cerca de 1% de erro.



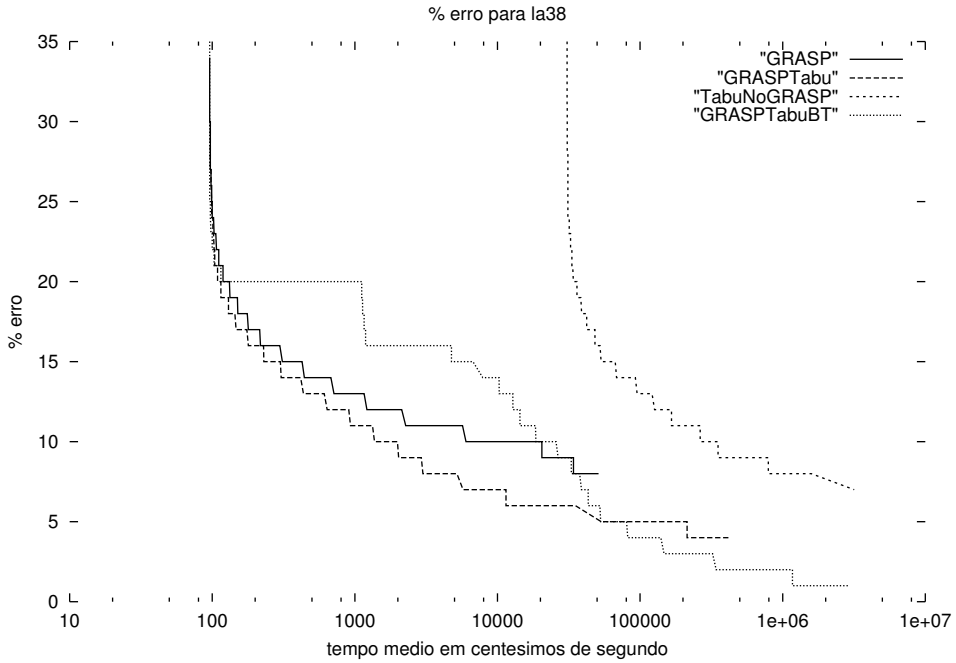
Para a instância *la*29 o algoritmo GRASP é o mais rápido a alcançar soluções que ficam a uma distância de 12% do melhor limite inferior conhecido. Para obter soluções com erros de 12 a 7% o GRASPTABU deverá ser o algoritmo escolhido, e para soluções de qualidade superior terá de utilizar-se o GRASPTABUBT. A melhor solução encontrada para a instância *la*29 apresenta um erro de cerca de 4% para o limite inferior.



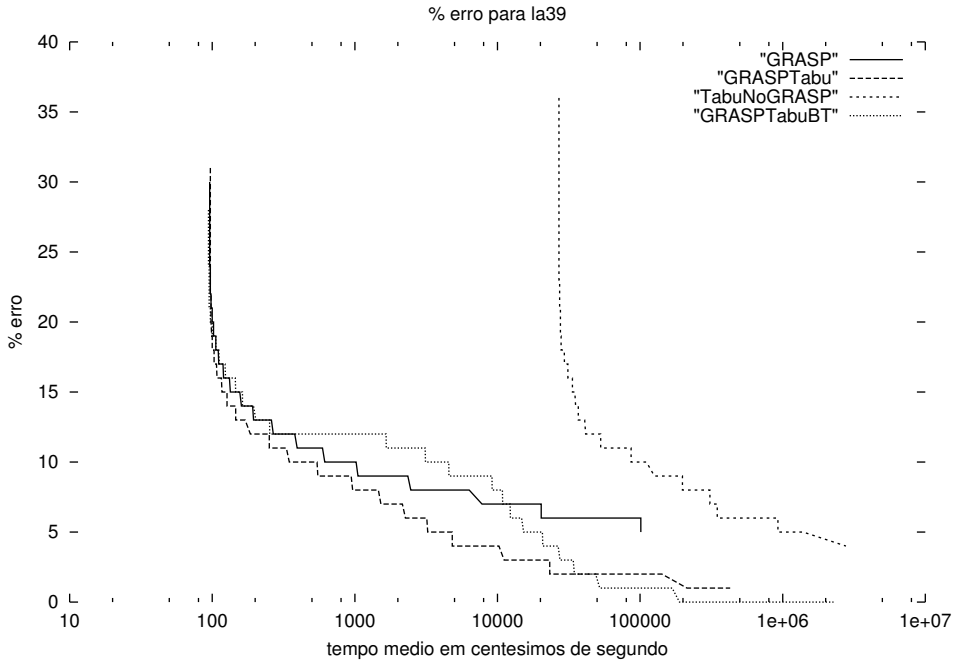
Quando o objectivo é encontrar soluções com uma percentagem de erro que poderá ser superior a 2%, o algoritmo GRASPTABU domina todos os outros. Para soluções de qualidade superior dever-se-à optar pelo GRASPTABUBT.



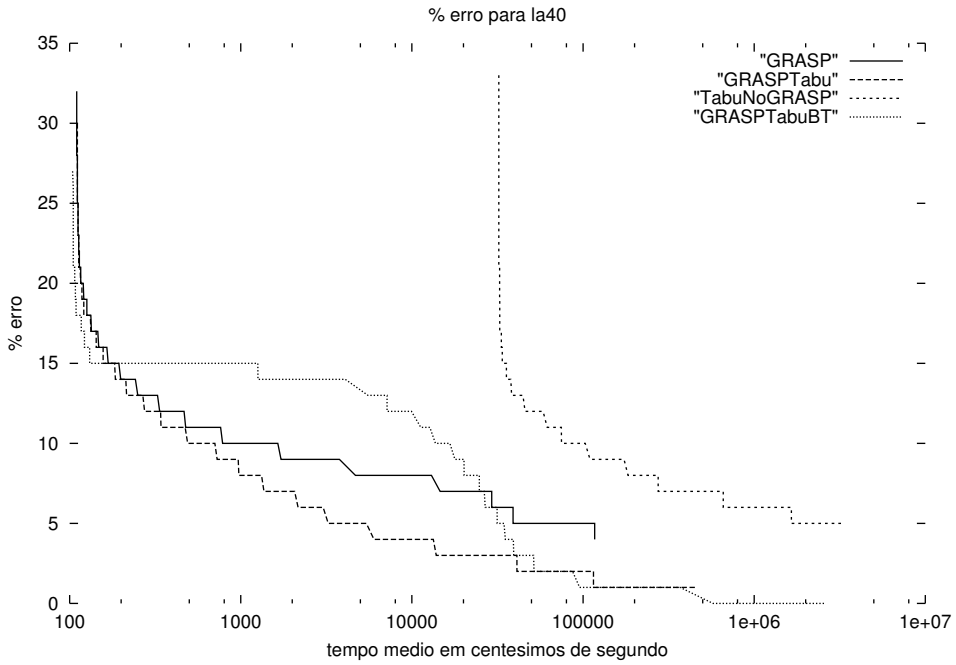
Para esta instância o algoritmo GRASP é o mais rápido a encontrar soluções com erro não inferior a 9%. O GRASPTABU é o que tem menor tempo médio de espera por soluções com erro no intervalo [2%,9%], e o GRASPTABUBT deverá ser o escolhido para atingir erros não superiores a 2%.



O algoritmo GRASPTABU é o mais rápido a alcançar soluções com erro maior ou igual a 5% para a instância *la*38. Para obter soluções com erro inferior aos cinco pontos percentuais deverá utilizar-se o algoritmo GRASPTABUBT, tendo a melhor solução que foi encontrada cerca de 1% de erro.



Para as instâncias *la*39 e *la*40, o algoritmo GraspTabu domina todos os outros na obtenção de soluções com percentagem de erro não inferior a 2. Para soluções de qualidade superior dever-se-à escolher o GraspTabuBt.



Aos observar todos estes gráficos torna-se evidente que o algoritmo TabuNoGrasp é dominado por todos os outros, pois é sempre mais lento e atinge soluções de qualidade inferior. A única excepção é a instância *la*02, onde todos os algoritmos excepto o Grasp atingem a solução óptima.
Extrapolando os resultados obtidos com as instâncias em teste, dir-se-à que o algoritmo GraspTabuBt deverá ser o escolhido quando o objectivo for encontrar uma solução o mais perto possível do óptimo. Se for suficiente obter uma boa solução, deverá utilizar-se o GraspTabu. Para instâncias de menor dimensão poder-se-à optar pelo algoritmo Grasp para encontrar boas soluções. Como se referiu anteriormente, talvez o algoritmo GraspTabu pudesse alcançar melhores resultados se o número máximo de iterações tivesse sido fixado num valor superior.

4.5 Comparação com algoritmos publicados

Nos artigos publicados sobre técnicas aplicadas ao problema *job shop scheduling* que referimos nesta dissertação, os resultados obtidos são apresentados na forma da melhor solução obtida por um conjunto de várias corridas independentes dos algoritmos, e muitas vezes de várias versões implementadas dos algoritmos, com tempos de execução associados.
Como é referido no artigo *Guidelines for Designing and Reporting on Computational Experiments with Heuristic Methods* [BGK$^+$95], a comparação de tempos entre algoritmos testados em máquinas diferentes, implementados por pessoas diferentes em

linguagens de programação distintas e executados em ambientes diferentes é, na melhor das hipóteses, uma comparação pouca fidedigna. Mesmo assim, e para ter algum termo de comparação com as heurísticas que aqui se referiram, são destacadas, para cada instância em teste, a melhor solução obtida nas várias corridas independentes dos algoritmos GRASP, GRASPTABU e GRASPTABUBT. Associados a cada melhor solução obtida estão tempos de processamento médios, em centésimos de segundo. A média dos tempos é calculada dividindo a soma dos tempos de execução das corridas em que o melhor valor é atingido pelo número de parcelas. A Tabela 4.4 apresenta para todas as instâncias, o valor da melhor solução encontrada, a percentagem de erro desse valor, e o tempo médio necessário para o alcançar (em centésimos de segundo), com os algoritmos GRASP, GRASPTABU e GRASPTABUBT.

	GRASP			GRASPTABU			GRASPTABUBT		
Instâncias	valor	%erro	(tempo)	valor	%erro	(tempo)	valor	%erro	(tempo)
FT10	973	4.62	(26)	930	0	(2585)	930	0	(18243)
LA02	694	5.95	(10)	655	0	(866)	655	0	(1579)
La19	842	0	(31)	842	0	(1264)	842	0	(2914)
La21	1095	4.68	(54)	1056	0.96	(2104)	1052	0.57	(18160)
La24	971	3.85	(35)	941	0.64	(2943)	935	0	(62233)
LA25	1023	4.71	(48)	979	0.20	(3324)	979	0.20	(31893)
LA27	1319	6.80	(77)	1271	2.91	(5397)	1250	1.21	(91364)
LA29	1265	10.77	(66)	1209	5.87	(4231)	1189	4.12	(59941)
LA36	1322	4.26	(12)	1291	1.81	(5787)	1274	0.47	(157821)
LA37	1478	5.80	(133)	1425	2.00	(6007)	1408	0.79	(23408)
LA38	1298	8.53	(82)	1249	4.43	(2419)	1208	1.00	(53856)
LA39	1304	5.76	(146)	1252	1.54	(5391)	1237	0.32	(42528)
LA40	1283	4.99	(104)	1240	1.47	(5713)	1229	0.57	(39246)

Tabela 4.4: Melhores *makespans* obtidos, percentagem de erro e respectivos tempos médios por corrida (em centésimos de segundo).

Observando na Tabela 4.4 a qualidade das melhores soluções alcançadas, constata-se que, exceptuando a instância $la19$ para a qual o valor óptimo é atingido pelos três algoritmos, o algoritmo GRASP é superado pelo GRASPTABU em todas as instâncias, que por sua vez é superado pelo GRASPTABUBT, o qual, em comparação com o anterior, apresenta resultados de igual qualidade nas instâncias $ft10$, $la02$, $la19$ e $la25$, e melhor em todas as outras.

A Tabela 4.5 mostra os valores óptimos para cada instância e os melhores valores obtidos com os algoritmos referidos nesta dissertação, nomeadamente, o *shifting bottleneck* de Adams Balas e Zawack [ABZ88] (ABZ), a procura tabu de Dell'Amico e Trubian [DT93] (DT), a procura tabu de Nowicki e Smutnicki [NS96] (NS), o *guided local search* de Balas e Vazacopoulos [BV98] (BV), a procura tabu de Pezzella e Merelli [PM00] (PM) e o GRASP com religação de soluções de Aiex, Binato e Resende [ABR01]

(ABR).

Instâncias	Opt. (LI,LS)	ABZ	DT	NS	BV	PM	ABR
FT10	930	1015	935	930	930	930	930
LA02	655	720	655	655	655	655	655
La19	842	875	842	842	842	842	842
La21	1046	1172	1048	1047	1046	1046	1057
La24	935	1000	941	939	935	938	954
LA25	977	1048	979	977	977	979	984
LA27	1235	1325	1242	1236	1235	1235	1269
LA29	(1142,1153)	1294	1182	1160	1157	1168	1203
LA36	1268	1351	1278	1268	1268	1268	1287
LA37	1397	1485	1409	1407	1397	1411	1410
LA38	1196	1280	1203	1196	1196	1201	1218
LA39	1233	1321	1242	1233	1233	1240	1248
LA40	1222	1326	1233	1229	1224	1233	1244

Tabela 4.5: Melhores *makespans* obtidos por heurísticas referenciadas.

Observando as Tabelas 4.4 e 4.5 verifica-se que o algoritmo GRASP, base dos algoritmos GRASPTABU e GRASPTABUBT, consegue atingir, para todas as instâncias, soluções de qualidade superior ao algoritmo ABZ, que é o algoritmo base dos algoritmos BV e PM.
Comparando os melhores valores atingidos pelo algoritmo GRASPTABU, com os algoritmos da Tabela 4.5, conclui-se que ele não consegue competir com nenhum dos algoritmos de procura tabu, conseguindo apenas um resultado melhor que o algoritmo DT para a instância $ft10$. Relativamente ao algoritmo ABR, o GRASPTABU consegue atingir soluções de qualidade superior para as instâncias $la21$, $la24$, $la25$ e $la40$, alcançado soluções de igual qualidade nas instâncias $ft10$, $la02$ e $la19$.
O comportamento do algoritmo GRASPTABUBT, em termos da melhor solução encontrada, é sempre melhor ou igual que o do algoritmo ABR, verificando-se a igualdade apenas nas três instâncias de menor dimensão, em que o valor óptimo só não é atingido por ABZ. Considera-se que o GRASPTABUBT consegue competir com o algoritmo DT, uma vez que só não encontra soluções de melhor qualidade que este para quatro das instâncias testadas ($la21$, $la27$, $la29$ e $la38$). O algoritmo GRASPTABUBT consegue apenas melhorar o valor alcançado pelo algoritmo NS na instância $la24$, obtendo resultados de igual qualidade nas instâncias $ft10$, $la02$, $la19$, e $la40$. Comparando a melhor solução encontrada pelos algoritmos PM e GRASPTABUBT, este melhora os resultados obtidos por aquele, nas instâncias $la24$, $la37$, $la39$ e $la40$, alcançando valores iguais nas instâncias $ft10$, $la02$, $la19$ e $la25$.
O algoritmo BV é de todos o que apresenta melhores resultados, para todas as instâncias.

Capítulo 5

Conclusões

Nesta dissertação foram desenvolvidos algoritmos heurísticos para a resolução do problema *job shop scheduling*. Um algoritmo construtivo baseado no conceito de máquina limitante do tipo semi-guloso com procura local, designado GRASP; um algoritmo de procura tabu executada sobre várias soluções obtidas com corridas independentes do GRASP, designado GRASPTABU; um algoritmo que executa procura tabu em cada fase do processo de construção de uma solução admissível, designado TABUNOGRASP; um algoritmo de procura tabu que recomeça a partir de soluções já visitadas, aplicada sobre uma solução construída com o GRASP, designado GRASPTABUBT e por último um algoritmo de religação de soluções, que gere uma lista de soluções consideradas elite, obtidas com o GRASPTABU, designado GRASPTABUPR.

Os algoritmos construídos exploram a região de soluções admissíveis, combinando diferentes estruturas de vizinhança e formas distintas de as percorrer. A construção de soluções admissíveis utiliza uma estrutura de vizinhança em que se entende por solução vizinha de outra, toda a solução que se obtém optimizando de forma condicionada a sequência de operações numa única máquina de cada vez; combinada com uma procura local sobre a estrutura de vizinhança definida por trocas (para a frente e para trás) em pares críticos de operações (definidos por Balas e Vazacopoulos). Sobre soluções assim construídas é executada uma procura tabu, sequencial ou com retrocessos, em que a estrutura de vizinhança é definida pela troca na ordem de processamento de operações críticas consecutivas. Finalmente, a componente de religação de soluções visita soluções não admissíveis para o problema, utilizando uma estrutura de vizinhança definida pela troca de posições de quaisquer duas operações, na sequência de processamento numa máquina.

A experiência computacional mostrou que, sendo a construção de soluções admissíveis baseada no conceito de máquina limitante, o recurso ao algoritmo estilo GRASP tem vantagens sobre uma construção determinística, com reoptimização máquina a máquina.

Verificou-se também que a inclusão da procura tabu, tal como foi definida, no processo de construção de soluções admissíveis, se revela contra producente.

Entre diversificar a procura tabu reiniciando-a a partir de novas soluções construídas, ou a partir de soluções já encontradas pela procura tabu, constatou-se que, apesar do acréscimo de tempo consumido, é preferível a segunda versão.

A incorporação do processo de religação de soluções, como componente final do algoritmo com construção GRASP seguido de procura tabu, mostrou-se ineficaz.
Comparando os resultados obtidos com os algoritmos desenvolvidos nesta dissertação, com os relatados na bibliografia, constata-se que a fase construtiva tem qualidade superior; na procura tabu há espaço para melhorias e a religação de soluções necessita de ser reformulada.

Dando continuidade ao trabalho desenvolvido nesta dissertação, e com o objectivo de melhorar os resultados bastante competitivos obtidos, assume-se o compromisso de rever alguns parâmetros dos algoritmos desenvolvidos, como por exemplo: definir o número máximo de iterações da procura tabu como dependente da frequência de actualizações do melhor vizinho encontrado; reduzir o peso da componente aleatória na definição da dimensão da lista tabu, fazendo variar o seu valor estrategicamente; utilizar também a vizinhança definida pelos pares críticos na procura tabu; ponderar a utilização da religação de soluções como estratégia de intensificação da procura.
Já sem limitações de tempo e, espera-se, com acesso mais fácil a equipamento informático indispensável, serão utilizadas em testes futuros todas as instâncias presentes na Livraria de Investigação Operacional.

Referências

[ABR01] R.M. Aiex, S. Binato, and M.G.C. Resende. Parallel grasp with path-relinking for jop shop scheduling. Technical report, AT&T Labs Research, 2001.

[ABZ88] J. Adams, E. Balas, and D. Zawack. The shifting bottleneck procedure for job shop scheduling. *Management Science*, 34(3):391–401, 1988.

[BGK+95] R. S. Barr, B. L. Golden, J. P. Kelly, M. G. C. Resende, and W. R. Stewart. Designing and reporting on computational experiments with heuristic methods. *Journal of Heuristics*, 1(1):9–32, 1995.

[BV98] E. Balas and A. Vazacopoulos. Guided local search with shifting bottleneck for job shop scheduling. *Management Science*, 44(2):262–274, 1998.

[Car82] J. Carlier. The one machine sequencing problem. *European Journal of Operational Research*, 11:42–47, 1982.

[DT93] M. Dell'Amico and M. Trubian. Applying tabu-search to the job-shop scheduling problem. *Annals of Operations Research*, 41:231–252, 1993.

[FGM00] M. Laguna F. Glover and R. Martí. Fundamentals of scatter search and path relinking. *Control and Cybernetics*, 39:656–684, 2000.

[FR95] T. Feo and M. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995.

[Fre82] S. French. *Sequencing and Scheduling: An Introduction to the Mathematics of the Job Shop*. Wiley, New York, 1982.

[GCG02] I. Ghamlouche, T. Crainic, and M. Gendreau. Path relinking, cycle-based neighbourhoods and capacitated multicommodity network design. Technical Report C7PQMR PO2002-01-X, Centre de recherche sur des transports, Université de Montréal, 2002.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, 1979.

[GL97] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, Boston, 1997.

[Glo86] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research*, 13:533–549, 1986.

[Glo89] Fred Glover. Tabu search—part I. *ORSA Journal on Computing*, 1:190–206, 1989.

[Glo90] Fred Glover. Tabu search—part II. *ORSA Journal on Computing*, 2:4–32, 1990.

[Glo96] F. Glover. Tabu search and adaptive memory programming — advances, applications and challenges. In R. Barr, R. Helgason, and J. Kennington, editors, *Interfaces in Computer Science and Operations Research: Advances in Metaheuristics, Optimization, and Stochastic Modeling Technologies*, pages 1–75. Kluwer, 1996.

[GT60] B. Griffer and G. Thompson. Algorithms for solving production scheduling problems. *Operations Research*, 8:487–503, 1960.

[JM99] A. S. Jain and S. Meeran. Deterministic job-shop scheduling: past, present and future. *European Journal of Operational Research*, 113:390–434, 1999.

[LAL88] P. Van Laarhoven, E. Aarts, and J. Lenstra. Job shop scheduling by simulated annealing. Technical Report OSR8809, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1988.

[Mar96] P. Martin. *A time-oriented approach to computing optimal schedules for the job-shop scheduling problem.* PhD thesis, School of Operations Research and Industrial Engineering, Cornell University, New York, 1996.

[MSS88] H. Matsuo, C. Suh, and R. Sulliva. A controlled search simulated annealing method for the general job shop scheduling problem. Technical Report 03-04-88, Department of Management, The University of Texas, Austin, 1988.

[NS96] E. Nowicki and C. Smutnicki. A fast taboo search algorithm for the job shop scheduling problem. *Management Science*, 42(6):797–813, 1996.

[PM00] F. Pezzella and E. Merelli. A tabu search method guided by shifting bottleneck for the job shop scheduling problem. *European Journal of Operational Research*, 120:297–310, 2000.

[Pot80] C. Potts. Analysis of a heuristic for one machine sequencing problems with release dates and delivery times. *Operations Research*, 28(6):1436–1441, 1980.

[RS64] B. Roy and B. Sussman. Les problèmes d'ordonnancement avec constraintes disjonctives. Note DS 9 bis, SEMA, Paris, 1964.

[Sch70] L. Schrage. Solving resource-constrained network problems by implicit enumeration: Non pre-emptive case. *Operations Research*, 18:263–278, 1970.

[Tai94] E. Taillard. Parallel taboo search techniques for the job shop scheduling problem. *ORSA Journal on Computing*, 6(2):108–117, 1994.

[VAL96] R. Vaessens, E. Aarts, and J. Lenstra. Job shop scheduling by local search. *INFORMS Journal on Computing*, 8:302–317, 1996.

[YN96a] T. Yamada and R. Nakano. A fusion of crossover and local search. In *IEEE International Conference on Industrial Technology (ICIT'96)*, pages 426–430, 1996.

[YN96b] T. Yamada and R. Nakano. Scheduling by genetic local search with multi-step crossover. In *Parallel Problem Solving from Nature IV*, Lecture Notes in Computer Science, pages 430–440, Berlin, 1996. International Conference on Evolutionary Computation / PPSN IV.