Escalonamento com restrição de mão-de-obra: heurísticas combinatórias e limitantes inferiores

Cristina Célia Barros Cavalcante

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Escalonamento com restrição de mão-de-obra: heurísticas combinatórias e limitantes inferiores

Cristina Célia Barros Cavalcante

Setembro de 1998

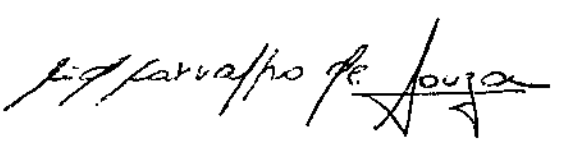
Banca Examinadora:

- Prof. Dr. Cid Carvalho de Souza (Orientador)
- Prof. Dr. Celso Carneiro Ribeiro
  Dep. Informática PUC-RJ
- Prof. Dr. João Carlos Setubal
  Instituto de Computação - UNICAMP
- Prof. Dr. Ricardo Dahab (Suplente)
  Instituto de Computação - UNICAMP

Escalonamento com restrição de mão-de-obra: heurísticas combinatórias e limitantes inferiores

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Cristina Célia Barros Cavalcante e aprovada pela Banca Examinadora.

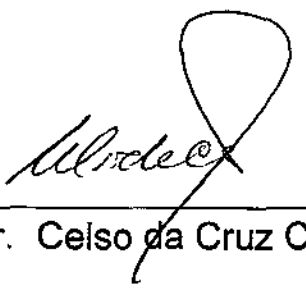
Campinas, 4 de setembro de 1998.



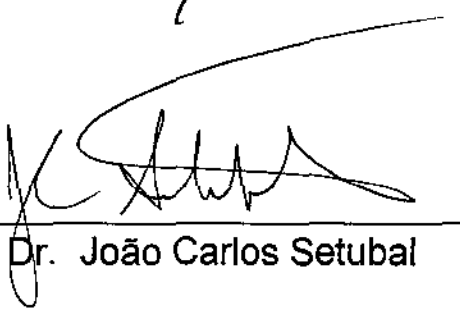
Prof. Dr. Cid Carvalho de Souza (Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

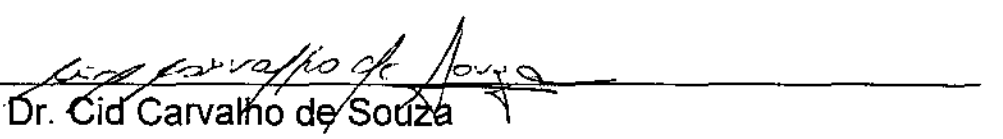
Tese de Mestrado defendida e aprovada em 28 de agosto de 1998 pela Banca Examinadora composta pelos Professores Doutores



Prof. Dr. Celso da Cruz Carneiro Ribeiro



Prof. Dr. João Carlos Setubal



Prof. Dr. Cid Carvalho de Souza

Para Victor, com o meu amor.

Agradecimentos

O meu mestrado certamente não teria sido o mesmo sem a ajuda, os sorrisos, a amizade e a torcida de algumas pessoas:

- meu marido, Victor
- meus pais, Adilson e Terezinha
- minha irmã, Sandrinha, e meu futuro cunhado, Zelmann
- meus segundos pais, Roberto e Darlene
- meus cunhados, Fábio, Roberta e Lucas
- os amigos de todas as horas, André Dantas, André Suzart, Marcos Renato, Toinho, Wesley, Renata, Elaine e Patrícia
- os amigos de longe, Alexandre Moriya, Paulo Magno e Ricardo Valença, e os de muito longe, Hélvio e Daniela, Ronaldo e Sara, Anderson e Eliany
- a amiga Chris Bottura e o Madrigal Cais do Canto
- meu orientador, Cid
- os professores do IC, Tomasz, Lucchesi, Dahab, Célia e Meidanis
- os funcionários da secretaria do IC, Vera e Daniel

Este trabalho tem um pouquinho de cada um de vocês e eu deixo aqui o meu carinhoso muito obrigada!

Resumo

Esta dissertação estuda o problema de escalonamento com restrição de mão-de-obra (SPLC) e apresenta algumas estratégias para obtenção de limitantes inferiores e de limitantes superiores para este problema NP-difícil.

No que diz respeito à obtenção de limitantes inferiores para o SPLC, são apresentadas duas formulações de programação inteira e discutidos os limitantes inferiores obtidos com a relaxação linear de cada uma delas. Um algoritmo de *branch-and-bound* específico para o SPLC é também implementado na tentativa de se obter soluções exatas para este problema. Finalmente, é introduzida uma extensão, baseada em uma formulação de programação inteira, para um método existente na literatura para o cálculo de limitantes inferiores para problemas de escalonamento com restrição de recursos.

Com relação à obtenção de limitantes superiores para o SPLC, são propostas e implementadas quatro estratégias heurísticas seqüenciais (heurística baseada em regras de prioridade, heurística baseada em classes de escalonamento, heurística baseada em programação linear e um algoritmo seqüencial de busca tabu) e duas estratégias paralelas e assíncronas (*A-Team* e um algoritmo paralelo de busca tabu).

Um conjunto de instâncias de teste para o SPLC é gerado e disponibilizado como *benchmark* para ser usado na avaliação da qualidade dos limitantes inferiores e superiores obtidos neste trabalho e em outros disponíveis na literatura.

Embora pouco sucesso tenha sido obtido com as estratégias propostas para obtenção de limitantes inferiores, no caso dos limitantes superiores, as estratégias paralelas e assíncronas propostas e implementadas neste trabalho mostraram-se altamente adequadas à obtenção de soluções de boa qualidade para o SPLC e são, atualmente, responsáveis pelas melhores soluções conhecidas para este problema.

Abstract

This dissertation studies the scheduling problem under labour constraints (SPLC) and presents some strategies to obtain lower and upper bounds for this NP-hard problem.

Concerning the lower bounds, two integer programming formulations are presented and the lower bounds associated with their linear relaxations are discussed. A branch-and-bound algorithm is also implemented as an essay to provide exact solutions for this problem. Finally, integer programming is used as a basis to extend a procedure reported in the literature to compute lower bounds for the resource constrained project scheduling problem.

In order to get upper bounds for SPLC, four heuristic approaches (priority rule based heuristic, schedule set based heuristic, linear programming based heuristic and sequential tabu search algorithm) and two parallel strategies (A-Team and parallel tabu search algorithm) are proposed and implemented.

A benchmark instance data set for SPLC is generated to be used in the evaluation of the lower and upper bounds obtained in this dissertation and in other works available in the SPLC literature.

Although little success has been achieved with respect to the lower bounds, in the case of upper bounds, the parallel strategies proposed in this work have shown the applicability of such techniques in providing high quality solutions for SPLC, and are, presently, responsible for the best known solutions for this problem.

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

1.1 Otimização Combinatória e Problemas de Escalonamento

Acompanhando o crescimento das organizações industriais nos últimos trinta anos, uma constante pode ser verificada como diretriz dos principais processos produtivos: a busca de qualidade e eficiência. Como causa e/ou conseqüência deste fato, observa-se também uma sociedade mais exigente, à procura de soluções rápidas, de baixo custo e de boa qualidade. É dentro deste contexto que aparece a otimização combinatória, um importante aliado no processo competitivo e de modernização industrial.

Numa definição formal [NW88], otimização combinatória é a área que trata de problemas de maximização ou minimização de uma função de uma ou mais variáveis, sujeita a restrições de igualdade e/ou desigualdade e a restrições de integralidade de algumas ou de todas as variáveis.

Um grande conjunto de aplicações de otimização combinatória está relacionado ao gerenciamento e uso eficiente de recursos caros ou escassos com o objetivo de aumentar a produtividade. É nesta classe que se enquadram os problemas de escalonamento.

Em linhas gerais, escalonar diz respeito à alocação de recursos no tempo para executar um conjunto de tarefas. Usualmente, dois tipos de restrições aparecem em problemas de escalonamento: restrições impondo limites na disponibilidade dos recursos e restrições especificando relações tecnológicas entre as tarefas a serem processadas. Assim, encontrar uma solução para um problema de escalonamento envolve decisões de dois tipos:

- Decisões de alocação: que recursos serão destinados para execução de cada tarefa.
- Decisões de seqüenciamento: quando cada tarefa será executada.

No ambiente industrial, problemas de escalonamento aparecem ligados a problemas de planejamento. Numa primeira etapa do processo de produção – a fase de planejamento – o produto a ser fabricado é escolhido, a escala de produção é determinada e a tecnologia a ser usada é especificada. Uma vez estabelecidos o quê e como produzir, a fase de escalonamento é responsável por dizer onde, quando e com quê recurso cada tarefa será executada.

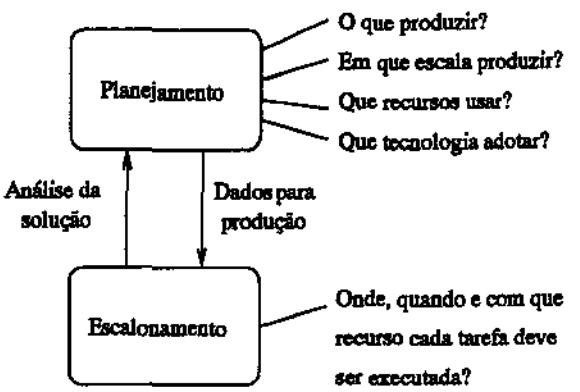


Figura 1.1: Relação entre planejamento e escalonamento.

A Figura 1.1 esquematiza a relação entre planejamento e escalonamento. A solução da fase de escalonamento pode eventualmente desencadear um novo planejamento (revisão da tecnologia usada ou da quantidade de recursos disponíveis). Isto acontece sempre que a qualidade de um escalonamento - medida em termos de utilização eficiente de recursos, resposta rápida à demanda, cumprimento dos prazos de entrega e custos de produção - não é satisfatória.

Devido à natureza intrinsecamente combinatória dos problemas de escalonamento, a escolha de uma estratégia para obtenção de uma solução deve considerar dois aspectos: a complexidade computacional do problema e o tempo de que se dispõe para resolvê-lo.

Com exceção de casos muito especiais (recursos abundantes, por exemplo), problemas de escalonamento em sua grande maioria são NP-difíceis [GJ79]. Pelo desconhecimento de um algoritmo determinístico polinomial para encontrar a solução ótima de todas as instâncias destes problemas, é possível identificar três direções alternativas a serem seguidas [Ree95]:

- Obtenção de limitantes superiores - soluções com custo superior ao da solução ótima - através do uso de métodos aproximados e heurísticos.
- Obtenção da solução ótima através de métodos exatos.
- Obtenção de limitantes inferiores - valores inferiores ao custo da solução ótima - através do uso de relaxações do problema original.

Métodos aproximados são aqueles que produzem soluções viáveis sub-ótimas cujo afastamento em relação à solução ótima é limitado por uma constante. Quando este limite de afastamento

é desconhecido, diz-se que o algoritmo é heurístico. Sem compromisso com a otimalidade, o objetivo de uma heurística é obter soluções viáveis em um tempo computacional pequeno.

Métodos exatos são as estratégias que garantidamente levam a uma solução ótima. Normalmente, para problemas NP-difíceis, estes métodos exigem um elevado tempo computacional o que torna inviável a sua utilização na prática. Avanços recentes nesta área, contudo, têm mostrado bons resultados na aplicação de estratégias exatas à resolução de problemas combinatórios. As principais técnicas exatas usadas são programação dinâmica e algoritmos exatos baseados em programação inteira e na teoria dos poliedros associados a problemas combinatórios [FW96].

A relaxação de um problema diz respeito à relaxação de uma ou mais de suas restrições. A motivação por trás desta técnica está no fato de que muitos problemas combinatórios NP-difíceis são problemas sabidamente polinomiais com o acréscimo de algumas restrições. Relaxando estas restrições "complicadas", o problema modificado é mais simples e sua solução ótima é um limitante inferior para o problema original. As relaxações mais conhecidas são baseadas em formulações do problema com programação inteira:

- Relaxação linear: elimina as restrições de integralidade de algumas, ou de todas, as variáveis.
- Relaxação lagrangeana: traz para a função objetivo cada uma das restrições que se quer relaxar.

A Figura 1.2 mostra a relação entre os métodos para obtenção de limitantes superiores, soluções ótimas e limitantes inferiores para problemas de escalonamento e combinatórios em geral.

Esta dissertação aborda o problema de escalonamento com restrição de mão-de-obra e apresenta estratégias para obtenção de limitantes superiores e inferiores para este problema. As estratégias adotadas e os objetivos deste trabalho são apresentados na próxima seção.

1.2 Objetivos da Tese

O problema de escalonamento com restrição de mão-de-obra ou SPLC, do inglês *Scheduling Problem under Labour Constraints*, diz respeito ao seqüenciamento de um conjunto de jobs (tarefas) sujeitos a restrições de precedência representadas por um grafo direcionado acíclico. Cada job tem uma duração específica durante a qual exige uma quantidade de mão-de-obra para ser executado. A necessidade de mão-de-obra varia a medida que o job é processado. Dado o total de mão-de-obra disponível em cada período, o problema consiste em concluir todos os jobs o mais cedo possível, respeitando as restrições de precedência e disponibilidade de mão-de-obra. Em outras palavras, deseja-se encontrar um escalonamento viável de mínimo *makespan*.

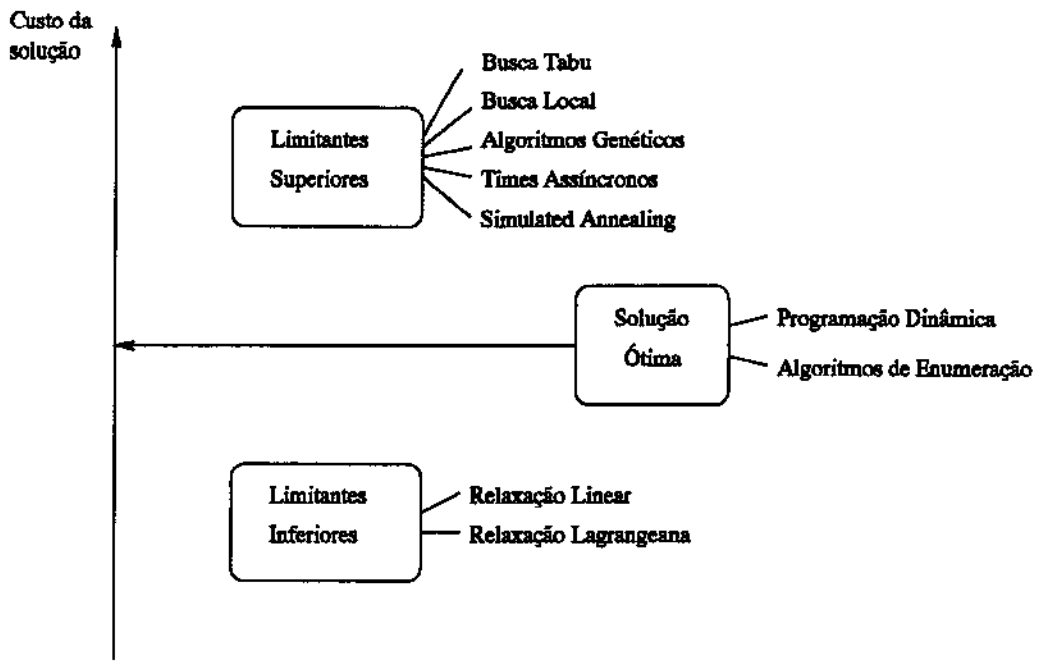


Figura 1.2: Relação entre métodos e tipos de solução para problemas combinatórios.

Este problema apareceu em [Hei95] e é uma simplificação de um problema real da BASF-AG (Alemanha) estudado no âmbito do projeto PAMIPS [PAM].

O SPLC é um exemplo do problema clássico de escalonamento com restrição de recursos e é NP-difícil [BLK83].

Os objetivos principais desta dissertação são os seguintes:

- Estudar o problema de escalonamento com restrição de mão-de-obra, SPLC.
- Desenvolver heurísticas combinatórias para o SPLC e testar a adequabilidade destas estratégias na obtenção de bons limitantes superiores para este problema.
- Estudar formulações de programação inteira mista (MIP) para o SPLC e usá-las na obtenção de limitantes inferiores para este problema.

No que diz respeito ao desenvolvimento de heurísticas combinatórias, o enfoque deste trabalho está centrado em duas abordagens paralelas e assíncronas para o SPLC: times assíncronos [Sou93] e estratégias paralelas e assíncronas de busca tabu [CTG93]. Inicialmente, porém, são propostos também alguns algoritmos seqüenciais para o SPLC: heurísticas baseadas em regras de prioridade, heurísticas baseadas em classes de escalonamento, uma heurística baseada em programação linear e uma busca tabu seqüencial. O objetivo neste caso é duplo: os algoritmos seqüenciais serão usados para compor um conjunto de processos para as estratégias paralelas e

assíncronas; e os resultados das estratégias seqüenciais poderão ser comparados aos das abordagens paralelas e assíncronas. Pretende-se com isso avaliar a potencialidade do paralelismo e assincronismo na obtenção de soluções de qualidade para o SPLC.

Com relação ao estudo de formulações MIP para o SPLC, pretende-se obter limitantes inferiores através de relaxação linear. Além disso, um algoritmo de *branch-and-bound* específico para o SPLC, sugerido por Souza e Wolsey em [SW97], é implementado na tentativa de se obter soluções exatas para o problema estudado. Finalmente, é apresentada uma extensão, baseada em formulações MIP, para o método proposto por Baker [Bak74] para o cálculo de limitantes inferiores para o problema de escalonamento com restrição de recurso.

A opção por trabalhar ao mesmo tempo com limitantes superiores e limitantes inferiores para o SPLC está motivada nos seguintes fatos:

- Bons limitantes superiores para o *makespan* diminuem o tamanho das formulações MIP discretizadas no tempo e podem reduzir a árvore de *branch-and-bound.*
- Limitantes inferiores obtidos com formulações MIP ajudam na análise da qualidade dos limitantes superiores obtidos pelas heurísticas.

1.3 Organização do Texto

Os capítulos que seguem estão organizados da seguinte forma:

O Capítulo 2 define formalmente o problema de escalonamento com restrição de recurso (RCPS - *Resource Constrained Project Scheduling*) e o problema alvo desta dissertação: o problema de escalonamento com restrição de mão-de-obra, SPLC. São apresentados conceitos básicos e complexidade destes problemas. Continuando, é feita uma revisão bibliográfica de algumas estratégias empregadas na solução do RCPS e SPLC. O capítulo termina com a descrição de um conjunto de instâncias de teste (*benchmarks*) geradas neste trabalho para motivar a comparação de resultados obtidos pelos diferentes métodos usados por pesquisadores interessados no SPLC.

O Capítulo 3 apresenta as estratégias usadas na obtenção de limitantes inferiores para o SPLC. Inicialmente, são discutidas duas formulações MIP para o SPLC e apresentados os limitantes inferiores obtidos com a relaxação linear de cada uma delas. Em seguida, é descrito o algoritmo de *branch-and-bound* específico para o SPLC, implementado como sugestão de Souza e Wolsey [SW97]. Finalmente, é apresentada a proposta de extensão para o método de cálculo de limitante inferior para o SPLC adotado em [Bak74] e os resultados obtidos com esta nova estratégia são comparados aos limitantes inferiores obtidos com as formulações MIP.

O Capítulo 4 descreve as estratégias heurísticas seqüenciais implementadas neste trabalho para o SPLC. Inicialmente é apresentada uma heurística baseada em regras de prioridade. Em seguida,

são descritas três heurísticas baseadas em classes de escalonamento. Continuando, é apresentada uma heurística para o SPLC baseada em programação linear. Em seguida, é detalhada uma estratégia seqüencial de busca tabu proposta para o SPLC. O capítulo termina com a comparação dos resultados obtidos por estas estratégias seqüenciais.

O Capítulo 5 apresenta as duas estratégias paralelas e assíncronas propostas para o SPLC. Inicialmente é apresentado o *A-Team* implementado para o SPLC: algoritmos de construção e melhoria que o compõem, memórias de soluções completas e parciais e política de destruição. Os resultados computacionais obtidos com a aplicação desta estratégia às instâncias de teste do SPLC são discutidos. Em seguida, é detalhada a estratégia paralela e assíncrona de busca tabu implementada para o SPLC: processos que a compõem, comunicação entre eles, etapas de intensificação e armazenamento das melhores soluções. Os resultados computacionais do algoritmo paralelo de busca tabu são apresentados no final do capítulo.

O Capítulo 6, finalmente, apresenta as conclusões e contribuições deste trabalho e sugere algumas linhas de pesquisa para trabalhos futuros com o SPLC.

Capítulo 2

O Problema de Escalonamento com Restrição de Mão-de-Obra

Problemas de escalonamento estão entre os mais difíceis problemas de otimização combinatória [BESW93]. Devido à enorme quantidade de situações reais que podem ser modeladas como problemas desta natureza, muito esforço tem sido feito no sentido de aprofundar o conhecimento teórico e estabelecer boas estratégias de solução para estes problemas.

Este capítulo apresenta o problema de escalonamento estudado nesta dissertação: o problema de escalonamento com restrição de mão-de-obra, SPLC. A Seção 2.1 descreve o problema genérico de escalonamento com restrição de recurso, RCPS, do qual o SPLC é um caso especial. A Seção 2.2 traz a definição formal do SPLC. Métodos de solução conhecidos e aplicados ao RCPS e ao SPLC são revisados nas Seções 2.3 e 2.4, respectivamente. A Seção 2.5, por fim, descreve o conjunto de instâncias usadas nos experimentos computacionais com o SPLC.

2.1 O Problema de Escalonamento com Restrição de Recurso (RCPS)

2.1.1 Definição

O problema de escalonamento com restrição de recurso ou RCPS (*Resource Constrained Project Scheduling*) pode ser enunciado da seguinte forma:

Um conjunto de n jobs, $J_1, \ldots, J_n$, compostos por uma ou mais tarefas e relacionados por restrições de precedência deve ser executado em um conjunto de m máquinas, $M_1, \ldots, M_m$. A cada instante, cada máquina pode executar no máximo um job e cada job pode ser processado

por no máximo uma máquina. Para sua completa execução, além da máquina, cada job requer o uso adicional de recursos escassos.

Sejam $R_1, \ldots, R_\ell$ os tipos de recursos existentes. Seja s_{ht}, $h = 1, \ldots, \ell$, o total disponível do recurso R_h no instante t. Seja p_j, $j = 1, \ldots, n$, a duração do job J_j e sejam $T_{j1}, \ldots, T_{jp_j}$ as tarefas que o compõem. Seja r_{hjk} a quantidade de recurso R_h requerida pela tarefa T_{jk} do job J_j.

Um escalonamento, solução do problema de escalonamento, é definido pela atribuição de um instante de início de processamento a cada um dos jobs $J_1, \ldots, J_n$. Um escalonamento para o RCPS é dito viável se ele satisfaz às restrições de precedência entre os jobs e às restrições de recurso:

$$\sum_{T_{jk} \in S(t)} r_{hjk} \leq s_{ht}, \quad \forall t, \ \forall h \in \{1, \ldots, \ell\}$$

onde $S(t)$ é o conjunto de todas as tarefas T_{jk} que estão sendo executadas no tempo t.

O RCPS consiste em encontrar um escalonamento viável que seja ótimo em relação a um critério de desempenho (função objetivo) que se quer otimizar.

A estrutura básica do RCPS envolve, então, quatro elementos: jobs, máquinas, recursos e critério de desempenho. Dependendo de como são especificados estes elementos, surgem diferentes problemas de escalonamento com restrição de recurso. A próxima seção apresenta as caracterizações mais usadas para os elementos básicos de um RCPS de acordo com [Bak74, BESW93].

2.1.2 Caracterização dos Elementos Básicos

Jobs

Cada job J_j pode ser caracterizado pelas seguintes informações [BESW93]:

(i) Tempo de processamento p_j: se $p_j = 1$, diz-se que o job tem duração unitária; caso contrário, fala-se em job com duração múltipla.

(ii) *Ready time* $e_j \geq 0$: indica o primeiro instante de tempo a partir do qual o job J_j está disponível para ser processado.

(iii) *Due date* d_j: indica o instante de tempo limite para conclusão do processamento do job J_j.

(iv) Prioridade w_j: indica a importância do job J_j em relação aos demais jobs.

(v) Necessidade de recursos:

- Discreto: se o job para ser executado precisa de quantidades discretas de recurso a cada instante;
- Contínuo: se o job para ser executado precisa de uma quantidade arbitrária de recursos a cada instante.

Além destas características intrínsecas a cada job, outras informações relativas ao conjunto de jobs como um todo devem ser especificadas:

(vi) Modo de processamento: preemptivo ou não-preemptivo.

Um escalonamento é dito preemptivo se algum job pode ter sua execução interrompida a qualquer momento para a execução de um outro job alocado à mesma máquina. Se isto não puder ocorrer, o escalonamento é não-preemptivo.

(vii) Dependências entre jobs.

Os jobs de um problema de escalonamento são ditos dependentes se a ordem de execução de pelo menos dois deles estiver restrita por relações de precedência. A notação $J_i \prec J_j$ indica que a execução do job J_i deve ser finalizada antes que o job J_j comece a ser processado. Na ausência de restrições de precedência, os jobs são ditos independentes.

Tipicamente, restrições de precedência são representadas através de grafos direcionados acíclicos onde nós representam jobs e arcos indicam restrições de precedência.

Máquinas

Máquinas que podem executar qualquer job são designadas "paralelas". Se, por outro lado, cada máquina é especializada na execução de determinados jobs, fala-se em máquinas "dedicadas".

Máquinas paralelas são divididas em três tipos de acordo com suas velocidades [BESW93]:

- Idênticas: todas as máquinas têm a mesma velocidade de processamento, independente do job que está sendo executado.
- Uniformes: todas as máquinas têm velocidades distintas, que independem do job que está sendo executado.
- Não-relacionadas: cada máquina tem uma velocidade própria, dependente do job que está sendo executado.

Recursos

Recursos são diferenciados de acordo com dois critérios: tipo e categoria [Bak74]. São ditos recursos do mesmo tipo, recursos que têm a mesma função. A caracterização por categorias diz respeito ao limite da quantidade disponível e à divisibilidade de cada recurso.

Com relação à limitação da quantidade disponível, um recurso pode ser:

- Renovável: quando há restrição apenas na disponibilidade temporária do recurso (ele pode ser usado novamente quando liberado por um job que também o requisitou).
- Não-renovável: quando há limite apenas na disponibilidade integral do recurso (uma vez usado por um job, o recurso não pode ser usado em outro job).
- Duplamente limitado: quando o recurso é temporaria e integralmente limitado.

No que diz respeito à divisibilidade, existem duas possibilidades:

- Discreto: quando o recurso pode ser alocado aos jobs em quantidades discretas (uma ou mais unidades).
- Contínuo: quando o recurso pode ser alocado em qualquer quantidade arbitrária, menor ou igual ao total disponível.

A Figura 2.1 exemplifica cada uma das categorias discutidas.

Divisibilidade	**Limitação da quantidade**		
	Renovável	**Não-renovável**	**Duplamente limitado**
Discreto	mão-de-obra	matéria-prima	matéria prima com limite em estoque
Contínuo	energia elétrica	dinheiro	dinheiro limitado por orçamento

Figura 2.1: Exemplo das várias categorias de recurso

Critério de desempenho

Todo escalonamento é caracterizado por algum critério de desempenho ou função objetivo. Esta medida é um indicador da qualidade da solução.

Tipicamente, critérios de desempenho para problemas de escalonamento são uma função do instante de conclusão dos jobs. Assim, se C_j indica o instante no qual o job J_j foi concluído, os critérios mais utilizados como função objetivo (a ser minimizada) para o RCPS são [Bak74, BESW93]:

(i) Tamanho do escalonamento (*makespan*): $C_{max} = \max_j\{C_j\}$

(ii) Tempo médio de fluxo: $\overline{F} = \frac{1}{n}\sum_{j=1}^{n} F_j$, onde $F_j = C_j - e_j$ é o tempo de fluxo do job J_j e indica o tempo total que ele permaneceu no sistema, a contar do instante em que ele ficou disponível para execução.

(iii) Máximo atraso: $A_{max} = \max_j\{A_j\}$, onde $A_j = C_j - d_j$ indica o quão depois do seu limite de conclusão d_j o job J_j foi concluído.

Baker em [Bak74] comenta que o critério (i) é usado quando se deseja uma utilização eficiente de recursos; o critério (ii) quando se deseja analisar a rapidez de resposta a demandas por serviços; e o critério (iii) quando datas e prazos de entrega são um fator importante no problema.

Outros critérios de desempenho para o RCPS envolvem despesas financeiras com o processamento dos jobs. Nesta classe, algumas das funções objetivo mais usadas são:

(iv) Minimizar o custo total de processamento: $Min \sum_j \sum_t G_{jt}$, onde G_{jt} é o custo necessário para começar o job J_j no instante t.

(v) Maximizar o fluxo de caixa: $Max \sum_j \sum_t (G_{jt}^{in} - G_{jt}^{out})$ onde G_{jt}^{in} (G_{jt}^{out}) é o dinheiro que entra em (sai de) caixa quando o job J_j é concluído (iniciado) no instante t.

É comum encontrar no meio industrial RCPSs cuja função objetivo é uma combinação de dois ou mais dos critérios (i)-(v). Neste caso, fala-se em RCPS multi-objetivo.

A próxima seção aborda a complexidade do problema de escalonamento com restrição de recurso.

2.1.3 Complexidade

A determinação da complexidade computacional de um RCPS está diretamente ligada à caracterização dos jobs, máquinas, recursos e critério de desempenho do problema. Uma análise detalhada da influência destes elementos na complexidade do RCPS foge ao escopo desta dissertação e pode ser encontrada em [BCSW86].

É sabido, contudo, que o RCPS é um dos problemas de escalonamento mais difíceis. Mesmo com um número reduzido de máquinas e restrições de recurso muito simples, muitos problemas de escalonamento com restrição de recurso foram provados pertencer à classe dos problemas NP-difíceis [BESW93].

Todavia, existem alguns casos especiais para os quais se conhecem algoritmos determinísticos de complexidade polinomial. Como exemplo, para o RCPS com máquinas paralelas idênticas, jobs independentes, escalonamento não-preemptivo e critério de desempenho mínimo *makespan* - daqui em diante denotado por $RCPS_{ex}$ - os casos polinomiais são [BESW93, Cof76]:

- Duas máquinas; jobs com duração unitária e recursos arbitrários. Pode ser resolvido em tempo $O(\ell n^2 + n^{2.5})$, onde ℓ é o número de recursos e n o número de jobs.
- Número arbitrário de máquinas; jobs com duração unitária; tipos, limites e necessidades de recursos fixados. Pode ser resolvido em tempo linear no número de jobs.

Blazewicz *et. al* em [BESW93] enumeram alguns parâmetros do RCPS que podem torná-lo mais difícil:

- Presença de *ready times* para os jobs.

  O $RCPS_{ex}$ com duas ou tês máquinas; jobs com duração unitária e *ready times*; e apenas um recurso já é NP-difícil [BBKR86, GJ75].

- Presença de relações de precedência entre jobs.

  O $RCPS_{ex}$ com duas máquinas; jobs com duração unitária e necessidade de recurso igual a 1; apenas um recurso com disponibilidade temporária igual a 1; na presença de relações de precedência tipo cadeia [1] é NP-difícil [BLK83].

A próxima seção enuncia o problema alvo desta dissertação. Trata-se de um caso especial do RCPS onde se tem: máquinas paralelas idênticas; jobs dependentes e com duração múltipla; escalonamento não-preemptivo; recurso discreto e renovável; e critério de desempenho baseado no mínimo *makespan*.

2.2 O Problema de Escalonamento com Restrição de Mão-de-Obra (SPLC)

2.2.1 Definição

O problema de escalonamento com restrição de mão-de-obra ou SPLC (*Scheduling Problem under Labour Constraint*) diz respeito ao seqüenciamento de um conjunto de jobs dependentes. Cada job tem uma duração específica, durante a qual exige uma quantidade de mão-de-obra para ser executado. A necessidade de mão-de-obra varia à medida que o job é processado. Dado o total de mão-de-obra disponível em cada período, o problema consiste em concluir todos os jobs o mais cedo possível, respeitando as restrições de precedência e a disponibilidade de mão-de-obra.

Formalizando esta descrição, pode-se enunciar o SPLC da seguinte forma:

Seja I um conjunto de pedidos ($|I| = m$), onde cada pedido está associado a uma máquina e é composto por n_i jobs idênticos. O conjunto de todos os jobs é J ($|J| = \sum_{i=1}^{m} n_i$). Cada

[1] Cada job tem no máximo um predecessor e um sucessor.

job é composto por um conjunto de p_j tarefas de duração 1. As tarefas de um mesmo job devem ser executadas imediatamente uma após a outra. A necessidade de mão-de-obra de um job j é especificada por um vetor $(\ell_{j1}, \ell_{j2}, \ldots, \ell_{jp_j})$, onde ℓ_{js} denota o número de trabalhadores necessários para execução da s-ésima tarefa do job j. Trabalhadores são necessários durante toda a execução de um job e existe um limite L no total de trabalhadores disponíveis em cada instante do horizonte de planejamento. Todos os jobs de um mesmo pedido devem ser executados na mesma máquina e não é permitida a preempção de pedidos (um pedido não pode ter sua execução interrompida para a execução de um outro pedido alocado à mesma máquina). As relações de precedência entre os jobs são representadas através de um grafo direcionado acíclico $G = (V, A)$ onde nós e arcos representam jobs e relações de precedência entre eles, respectivamente. Por definição, o grafo induzido em G por todos os jobs de um mesmo pedido é um caminho orientado simples.

Uma solução para o SPLC é um escalonamento S descrito pelos instantes de início de processamento de todos os jobs $j \in J$. Um escalonamento S é dito viável se: (i) $\forall (i,j) \in A$, o job j não começar antes do término do job i; (ii) o total de trabalhadores solicitados por todos os jobs em processamento em t não exceder L, para todo instante t do horizonte de planejamento.

O objetivo do SPLC é encontrar um escalonamento viável de mínimo *makespan*, isto é, um escalonamento que satisfaz às restrições de mão-de–obra e precedência entre jobs e onde todos os jobs são concluídos o mais cedo possível.

O SPLC, como definido acima, apareceu em [Hei95] e faz parte de um problema de planejamento industrial da BASF-AG (Alemanha) investigado pelo projeto PAMIPS [PAM] da Comunidade Econômica Européia [2]. A principal diferença entre outros tipos de problemas RCPS encontrados na literatura e o SPLC diz respeito à existência neste último de um perfil variável de recurso (mão-de-obra) ao longo da duração de um job.

Um exemplo de instância para o SPLC é mostrado na Figura 2.2. Esta instância será referenciada nos próximos capítulos como *instância exemplo*.

2.2.2 Complexidade

Como um exemplo típico dos mais complexos problemas de escalonamento com restrição de recurso, o SPLC também é NP-difícil. Isto pode ser constatado através dos seguintes passos:

1. O SPLC é um RCPS com máquinas paralelas idênticas, jobs dependentes e com duração múltipla, escalonamento não-preemptivo, um tipo de recurso (com limite e requisição arbitrários) e critério de otimalidade mínimo *makespan*.

[2] PAMIPS (*Parallel Algorithms and Software for Mixed-Integer Programming in Industrial Scheduling*) é um consórcio de quatro indústrias e três universidades (BASF, Buckingham U., CORE, DASH, GESA, IWR e Parsytec)

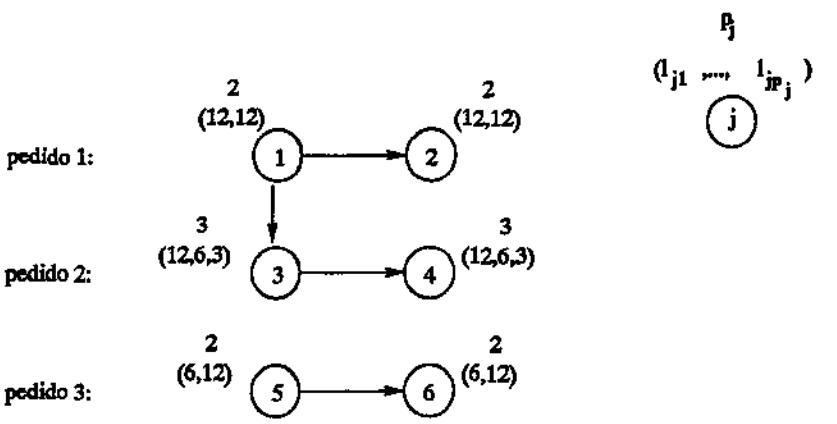


Figura 2.2: *Instância exemplo* do SPLC.

2. O RCPS com máquinas paralelas idênticas, jobs dependentes (restrições de precedência tipo cadeia) e com duração unitária, um tipo de recurso (com limite e requisição arbitrários), escalonamento não-preemptivo e critério de otimalidade mínimo *makespan* é NP-difícil [BLK83].

3. Jobs com duração unitária são um caso particular de jobs com duração múltipla. Assim, o RCPS do passo 2 com jobs de duração múltipla claramente é NP-difícil.

4. Relações de precedência tipo cadeia são um caso particular de relações de precedência quaisquer. Logo, o RCPS do passo 3 com relações de precedência genéricas também é NP-difícil. Como este problema é exatamente o SPLC, conclui-se que o SPLC é NP-difícil.

Assim, da mesma forma que para outros problemas NP-difíceis, não se conhece nenhum algoritmo determinístico polinomial capaz de resolver o SPLC. Esta aparente intratabilidade, contudo, é exatamente o que motiva a pesquisa de estratégias de solução para este problema. As próximas duas seções apresentam uma revisão bibliográfica de alguns métodos empregados na obtenção de soluções para o RCPS e para o SPLC.

2.3 Estratégias de Solução para o RCPS

Comprovadamente reconhecido pela comunidade de pesquisa operacional como intrinsecamente difícil e estrategicamente importante, o problema de escalonamento com restrição de recurso vem sendo amplamente estudado.

Os primeiros trabalhos relacionados ao RCPS apareceram em meados da década de 60. Adotando como critério de desempenho o tamanho do escalonamento, várias heurísticas foram propostas [Pas65, Gon69, Pat73]. Nestes trabalhos, soluções para o RCPS são construídas escolhendo a

cada passo um job para ser escalonado. A escolha é guiada por regras de prioridade que associam a cada job um valor determinante da sua prioridade em relação aos demais: jobs com prioridade mais alta são escalonados primeiro. Entre as regras de prioridade mais empregadas ao RCPS estão:

- MINSLK (*Minimum Job Slack*): jobs com folga menor têm prioridade mais alta. A folga de um job é definida como a diferença entre os instantes mais tarde e mais cedo em que o job pode começar.
- LFT (*Minimum Late Finish Time*): jobs que têm que terminar mais cedo têm prioridade mais alta.
- GRU (*Greatest Resource Utilization*): jobs que resultam numa maior utilização de recursos a cada instante do escalonamento têm prioridade mais alta.
- SIO (*Shortest Imminent Operation*): jobs com menor duração têm prioridade mais alta.

Parece haver um consenso de que nenhuma regra de prioridade domina as demais na qualidade das soluções obtidas [Bak74]. Davis e Patterson em [DP75], contudo, apresentam um estudo de oito destas regras para o RCPS e apontam MINSLK e LFT como as mais efetivas. Neste mesmo trabalho, os autores tentam relacionar a influência da estrutura de um RCPS (número de jobs, restrições de precedência entre eles e limitação de recurso) na qualidade das soluções obtidas por alguma heurística baseada em regra de prioridade. Uma análise detalhada do desempenho de heurísticas deste tipo em função das características do RCPS pode ser encontrada em [Pat76].

Boctor em [Boc90] propõe uma estratégia de múltiplas heurísticas baseadas em regras de prioridade para resolver o RCPS. Uma combinação de três a cinco regras de prioridade é selecionada, o problema é resolvido com cada uma das heurísticas baseadas nas regras escolhidas e a melhor solução obtida é retornada. Várias combinações são testadas e observa-se uma efetiva melhora na qualidade das soluções obtidas pela estratégia múltipla quando comparada às soluções obtidas por cada heurística isoladamente. Uma ordem de escolha das regras de prioridade é sugerida para aumentar a probabilidade de se obter melhores soluções.

Buscando uma alternativa para as estratégias heurísticas baseadas em regras de prioridade, Bell e Han [BH91] apresentam um método heurístico para solução do RCPS. No algoritmo proposto, todos os jobs são escalonados o mais cedo possível considerando apenas restrições de precedência. Até que se obtenha um escalonamento viável, novas restrições de precedência vão sendo acrescentadas entre os jobs para resolver a violação das restrições de recurso. Segundo os autores, os resultados obtidos com esta estratégia são comparáveis favoravelmente ao desempenho das heurísticas baseadas em regras de prioridade [DP75].

Sampson e Weiss em [SW93] apresentam um algoritmo de busca local para o RCPS. Da mesma maneira que a heurística de Bell e Han [BH91], esta estratégia começa escalonando todos os jobs

o mais cedo possível considerando apenas as restrições de precedência. Neste caso, porém, as violações de recurso são penalizadas na função objetivo. A cada iteração, um job tem seu início de processamento alterado de maneira a diminuir os conflitos de utilização de recurso. De acordo com os autores, os resultados obtidos com este algoritmo são melhores que os da heurística de Bell e Han.

Lee e Kim em [LK96] fazem uso para resolver o RCPS de três conhecidas estratégias heurísticas: *simulated annealing*, busca tabu e algoritmo genético [Ree95]. Em todos os três algoritmos propostos, soluções para o RCPS são representadas através de uma seqüência de números cada qual denotando a prioridade de um job. Aqui também, prioridades são usadas para escolher qual job deve ser escalonado primeiro. Soluções vizinhas são obtidas através da permutação das prioridades de dois jobs escolhidos arbitrariamente. Os resultados obtidos com a aplicação de cada uma destas meta-heurísticas ao RCPS são superiores aos da estratégia proposta por Sampson e Weiss [SW93].

Tentativas de se obter soluções exatas para o RCPS usando formulações de programação inteira não obtiveram êxito [DH92]. Mesmo para instâncias do RCPS de tamanho moderado (30 a 40 jobs), o número de variáveis e restrições de tais formulações tornou impraticável, em termos de memória e tempo computacional, o uso desta estratégia. Não obstante, vários algoritmos de enumeração implícita foram propostos para o RCPS e conseguiram resolver algumas de suas variações de forma ótima.

Davis e Heidorn em [DH71], por exemplo, apresentam um algoritmo exato para o RCPS com critério de desempenho mínimo *makespan*, onde a necessidade de recursos pode variar à medida que os jobs são processados. O RCPS abordado é transformado no problema de encontrar o caminho mais curto entre dois nós de um grafo direcionado G', construído de tal forma que: nós representam subconjuntos de tarefas e arcos ligam subconjuntos que podem ser processados em instantes adjacentes. Técnicas de enumeração parcial são utilizadas para encontrar os subconjuntos correspondentes aos nós de G'. O algoritmo foi testado com sucesso em instâncias de até 30 jobs.

Para problemas RCPS onde os recursos disponíveis para execução dos jobs não são muito limitados e onde se dispõe de pouca memória computacional para execução de uma estratégia de solução para o problema, Talbot e Patterson [TP78] propõem um algoritmo de *branch-and-bound*. O procedimento consiste numa enumeração sistemática dos possíveis instantes de conclusão de todos os jobs. Limitantes inferiores baseados nas restrições de precedência e recurso, e um artifício chamado "corte de rede" são usados para acelerar o processo de enumeração. O algoritmo é considerado uma boa estratégia de solução para instâncias com até 50 jobs.

Em trabalhos mais recentes, Bell e Park [BP90] e Demeulemeester e Herroelen [DH92] propõem estratégias exatas alternativas para solução do RCPS. Bell e Park apresentam um algoritmo de busca exaustiva, a busca A^*. Neste procedimento, cada nó da árvore de *branch-and-bound* re-

presenta um escalonamento completo (todos os jobs estão escalonados) satisfazendo às restrições de precedência. Restricões de recurso podem estar violadas e vão ser resolvidas pela adição de novas restrições de precedência. Cada escalonamento é obtido minimizando o *makespan* com as restrições de recurso relaxadas. A principal desvantagem deste algoritmo é o seu alto tempo computacional para instâncias com recursos muito limitados.

O *branch-and-bound* de Demeulemeester e Herroelen [DH92] introduz o conceito de "alternativas minimais de atraso" para resolver conflitos de utilização de recursos pelos jobs. Da mesma forma que o algoritmo de Bell e Park, novas restrições de precedência são adicionadas para tratar violações de recurso. Neste algoritmo, contudo, cada nó da árvore de *branch-and-bound* corresponde a um escalonamento parcial (apenas um subconjunto dos jobs está escalonado) satisfazendo às restrições de precedência. Um escalonamento completo é construído à medida que novos nós vão sendo criados pela adição de um job aos escalonamentos parciais existentes. A enumeração é guiada por regras de dominância e este algoritmo é, atualmente, uma das melhores estratégias exatas conhecidas para o RCPS.

Buscando uma abordagem heurística comparável, em termos da qualidade das soluções obtidas, ao *branch-and-bound* de Demeulemeester e Herroelen [DH92], Naphade et al. em [NWS93] propõem uma nova estratégia de solução aproximada para o RCPS: um algoritmo de busca no espaço do problema. Por este método, a cada iteração é feita uma pertubação nos dados de entrada do problema. Uma solução é construída com base nas informações do problema pertubado e o seu custo é avaliado considerando os dados originais. A melhor solução obtida ao longo do processo é retornada. Pertubações são feitas de tal maneira que soluções viáveis para o RCPS pertubado são também soluções viáveis para o RCPS original. De acordo com os autores, os resultados obtidos com este algoritmo têm qualidade, senão igual, bem próxima a das soluções exatas obtidas com o *branch-and-bound* de Demeulemeester e Herroelen.

Na literatura disponível para o RCPS, são apresentadas também outras técnicas de solução para as seguintes variações deste problema:

- RCPS com escalonamento preemptivo [Slo80].
- RCPS com função multi-objetivo [Slo81, NDWS95].
- RCPS com critério de desempenho "fluxo de caixa" [ZP95, ZP96, IE96]
- RCPS com múltiplos modos de execução para cada job [PTSW90].

A próxima seção apresenta as estratégias de solução já aplicadas à variação do RCPS estudada nesta dissertação: o problema de escalonamento com restrição de mão-de-obra.

2.4 Estratégias de Solução para o SPLC

As estratégias conhecidas na literatura para solução do SPLC podem ser divididas em duas classes: algoritmos usando programação por restrições e algoritmos baseados em programação matemática. As próximas duas seções apresentam os métodos de cada uma destas classes aplicados ao SPLC.

2.4.1 Programação por Restrições

O primeiro estudo sobre o SPLC aparece em [Hei95]. Neste trabalho, Heipcke faz uso de programação por restrições para obter soluções viáveis para uma única instância deste problema.

Programação por restrições ou *constraint programming* (CP) é uma área que se desenvolveu principalmente a partir da década de 80 combinando idéias e técnicas de diferentes origens como pesquisa operacional, programação lógica, inteligência artificial e matemática.

Os princípios básicos de CP são os seguintes:

- Cada variável está associada a um intervalo (ou união de intervalos) que descreve seus possíveis valores: o seu domínio.
- Uma restrição é uma relação entre variáveis e expressa limitações nos valores que estas podem assumir.
- O par (variáveis,restrições) define um sistema de restrições.
- A consistência de um sistema de restrições é verificada da seguinte forma: os limites do domínio de cada variável são testados em cada restrição. Um algoritmo de propagação de restrições é usado neste processo: limitações impostas por uma restrição no domínio de uma variável são propagadas para todas as demais restrições nesta variável. Valores inconsistentes, que resultam em violação de restrições, são removidos dos domínios das variáveis. Isto pode, em alguns casos, quebrar o intervalo do domínio de uma variável em uma união de intervalos. Se durante este processo o domínio de alguma variável ficar vazio, o sistema é dito inconsistente. Caso contrário, o sistema é consistente.
- Uma solução para um sistema de restrições é uma atribuição de valores, no sentido de aritmética de intervalos, a cada uma das variáveis de maneira que todas as restrições sejam satisfeitas. Assim, numa solução, o valor de uma variável não é necessariamente um intervalo unitário mas, normalmente, uma união de intervalos.
- Uma solução efetiva para um sistema de restrições é uma solução onde cada variável está atribuída a exatamente um único valor (um intervalo unitário).

- Dada uma solução para um sistema de restrições, uma solução efetiva é obtida através de um procedimento enumerativo como, por exemplo, um algoritmo de *branch-and-bound.*

Na estratégia proposta por Heipcke em [Hei95], o SPLC é modelado como um sistema de restrições. Soluções efetivas para o problema são encontradas usando o algoritmo de *branch-and-bound* da biblioteca COME (*Smalltalk*). A cada nó da árvore de enumeração, a consistência do sistema de restrições é verificada. O *branching* é feito sempre em cima da variável de menor domínio. A desvantagem desta estratégia é que, devido ao excessivo tempo computacional, o algoritmo de *branch-and-bound* é interrompido antes de chegar à solução efetiva ótima.

Em [HC97], Heipcke e Colombani fazem uso novamente de programação por restrições para resolver o SPLC. Neste caso, é usada a biblioteca *SchedEns* (C), desenvolvida especialmente para resolver problemas de escalonamento através de sistemas de restrições em uniões de intervalos inteiros. O sistema de restrições do SPLC é construído usando as primitivas de *SchedEns* para restrições de precedência e mão-de-obra. Um *branch-and-bound* é usado para encontrar as soluções efetivas. Os resultados obtidos com esta estratégia são superiores, em qualidade e eficiência, aos resultados apresentados em [Hei95]. Em algumas instâncias pequenas (20 a 40 jobs) é possível, inclusive, provar a otimalidade das soluções efetivas encontradas. Em instâncias maiores, o algoritmo de *branch-and-bound* pára após 20000 nós retornando a melhor solução efetiva e o melhor limitante inferior conhecidos.

2.4.2 Programação Matemática

Assim como a programação por restrições, a programação matemática também foi usada na obtenção de soluções heurísticas e limitantes inferiores para o SPLC.

Savelsbergh, Wang e Wolsey em [SWW96] apresentam uma heurística baseada em programação linear para o SPLC. Nesta estratégia, o SPLC é modelado através de uma formulação MIP com variáveis inteiras discretizadas no tempo. A solução da relaxação linear desta formulação é usada para obter uma permutação dos jobs. A idéia é capturar da solução fracionária uma medida de onde (instante no qual) a principal parte de um job já foi escalonada. Este valor é usado para ordernar os jobs e, em seguida, escaloná-los na ordem obtida. Duas possibilidades são testadas:

- *Schedule-by-S_j*: os jobs são ordenados em ordem não-decrescente dos seus instantes de início de processamento e escalonados respeitando as restrições de precedência e de mão-de-obra e a restrição adicional de que nenhum job j deve ser iniciado antes de um job i se j aparece depois de i na ordem considerada.
- *Schedule-by-Fixed-α*: para cada job j é determinado o seu ponto α, $0 \leq \alpha \leq 1$, como o primeiro instante, na solução fracionária da formulação indexada no tempo, onde uma

fração α de j já foi executada. Os jobs são ordenados em ordem não-decrescente dos seus pontos α e escalonados respeitando as restrições de precedência e de mão-de-obra e a restrição adicional de que nenhum job j deve ser iniciado antes de um job i se j aparece depois de i na ordem considerada. Originalmente, nesta heurística, α está fixado em 0.5.

Após a obtenção de um escalonamento viável, a heurística continua com uma fase de melhoria. Duas estratégias de busca local são aplicadas para melhorar a solução. Na primeira, dois jobs consecutivos na permutação obtida a partir da relaxação linear têm suas posições trocadas entre si sempre que esta nova ordem resultar num escalonamento viável de menor *makespan*. Na segunda estratégia de melhoria, são testadas mudanças sucessivas nos instantes de início de processamento de cada job.

Aprimorando a estratégia anterior, Savelsbergh em [SUW97] propõe a seguinte mudança para a heurística de [SWW96]: ao invés de um α fixado em 0.5, 100 valores distintos de α são considerados ($\alpha = 0.01, 0.02, \ldots, 1.00$). Usando *Schedule-by-Fixed-α* com cada um destes valores, 100 escalonamentos viáveis são construídos e o de menor *makespan* é retornado. Esta nova estratégia é denominada *Schedule-by-Best-α* pelos autores [SUW97]. Os mesmos algoritmos de melhoria propostos em [SWW96] são aplicados ao escalonamento obtido com *Schedule-by-Best-α*. Os resultados obtidos desta forma apresentam significativa melhora em relação aos resultados de [SWW96].

Formulações inteiras e desigualdades válidas para o SPLC são apresentadas por Wang e Wolsey em [WW96] e por Souza e Wolsey em [SW97] visando a obtenção de bons limitantes inferiores para o SPLC. Os resultados destes trabalhos, contudo, apontam a dificuldade de se usar formulações MIP para o SPLC devido, principalmente, à baixa qualidade dos limitantes inferiores obtidos com a relaxação das formulações propostas. Não obstante este fato, Souza e Wolsey em [SW97] sugerem algumas direções de pesquisa no sentido de continuar usando formulações MIP na obtenção de limitantes inferiores para o SPLC. Estas sugestões foram adotadas e implementadas nesta dissertação e estão detalhadamente descritas no Capítulo 3.

2.5 Instâncias para o SPLC

Inicialmente, apenas duas instâncias do SPLC, fornecidas pela BASF-AG no contexto do projeto PAMIPS [PAM], estavam disponíveis. Com o objetivo de se ter um conjunto maior de instâncias para testar as diferentes estratégias de solução para o SPLC, foi implementado, neste trabalho, um gerador aleatório com as seguintes características:

- Parâmetros: número total de pedidos na instância (m); número mínimo e máximo de jobs em um pedido da instância (m_j, M_j); duração mínima e máxima de um job em um pedido

da instância (m_d, M_d); probabilidade de jobs de diferentes pedidos estarem relacionados (p).

- Saída: instância com m pedidos, onde cada pedido tem entre m_j e M_j jobs idênticos. Todos os jobs de um pedido têm a mesma duração, aleatoriamente escolhida entre m_d e M_d. A necessidade de mão-de-obra de cada tarefa de um job é escolhida do conjunto $\{2, 3, (4), 6, (12), (18)\}$, onde os números entre parênteses são escolhidos com probabilidade $1/12$ e os demais com probabilidade $3/12$. Relações de precedência entre jobs de um mesmo pedido são implícitas. Jobs de pedidos diferentes estão relacionados com probabilidade p se eles pertencem a pedidos consecutivos. Jobs de pedidos não consecutivos não estão relacionados. Um job de um pedido é sucessor (predecessor) de no máximo um job do pedido anterior (seguinte).

O gerador aleatório foi implementado com estas características para que as instâncias por ele geradas tivessem uma estrutura similar a das duas intâncias fornecidas pela BASF-AG. O objetivo, neste caso, era tentar reproduzir o tipo de instância que ocorre na prática.

Foram geradas 23 instâncias. A Tabela 2.1 apresenta as informações gerais sobre cada uma delas. As instâncias *Ins_4o_24j_A* e *Ins_10o_88j_A* são as fornecidas pela BASF-AG.

Todo conjunto de dados está disponível na página WEB [Cav97]. Conforme apresentado em [CCHS97], estas 25 instâncias servem como *benchmarks* para o SPLC e têm sido usadas por outros pesquisadores interessados neste problema.

Instância	Tarefas	Precedências	Duração job	Caminho crítico
Ins_4o_21j_A	126	26	[6,8]	78
Ins_4o_23j_A	114	26	[4,8]	54
Ins_4o_24j_A	49	23	[5,9]	58
Ins_4o_24j_B	109	25	[4,8]	54
Ins_4o_27j_A	121	32	[5,7]	53
Ins_6o_41j_A	295	44	[6,10]	90
Ins_6o_41j_B	245	44	[7,10]	94
Ins_6o_41j_C	249	44	[6,9]	81
Ins_6o_44j_A	224	45	[6,9]	75
Ins_6o_44j_B	296	43	[7,9]	104
Ins_8o_63j_A	504	65	[10,13]	174
Ins_8o_63j_B	601	66	[10,14]	196
Ins_8o_63j_C	658	71	[10,15]	227
Ins_8o_65j_A	581	74	[10,14]	298
Ins_8o_65j_B	641	70	[10,15]	230
Ins_10o_84j_A	953	94	[12,18]	270
Ins_10o_84j_B	973	90	[12,18]	200
Ins_10o_85j_A	1054	87	[13,18]	513
Ins_10o_87j_A	1001	98	[12,18]	194
Ins_10o_88j_A	325	106	[8,37]	362
Ins_10o_100j_A	1795	120	[18,28]	352
Ins_10o_102j_A	1679	107	[15,29]	550
Ins_10o_106j_A	1653	116	[16,29]	383
Ins_12o_108j_A	1845	121	[15,30]	520
Ins_12o_109j_A	2014	113	[16,29]	819

Tabela 2.1: Instâncias de teste para o SPLC. O nome da instância é composto pelo número de pedidos e pelo número total de jobs. Uma letra é usada para distinguir instâncias com o mesmo número de pedidos e jobs. O número total de tarefas e precedências são dados. "Duração job" indica o intervalo de duração dos jobs em todos os pedidos. "Caminho crítico" é o comprimento, medido em tempo de processamento dos jobs, do maior caminho no grafo de precedência.

Capítulo 3

Limitantes Inferiores

Uma das formas de se avaliar a qualidade de uma solução heurística para um problema de otimização combinatória, onde se deseja minimizar (maximizar) uma dada função, é comparar esta solução com um limitante inferior (superior) obtido para aquela função. Quanto mais próximo da solução ótima, mais eficiente é o limitante.

Um limitante inferior trivial para o *makespan* de problemas RCPS é obtido ignorando as restrições de recurso e escalonando todos os jobs o mais cedo possível considerando apenas as restrições de precedência. Este limitante é conhecido como limitante inferior do caminho crítico, ξ_{CC}, e corresponde ao tamanho do maior caminho no grafo de precedência entre jobs.

Este capítulo apresenta algumas alternativas para o cálculo de limitantes inferiores para o SPLC. A Seção 3.1 traz duas formulações MIP para este problema e apresenta os limitantes inferiores obtidos com as relaxações lineares destas formulações. A Seção 3.2, em seguida, descreve um algoritmo de *branch-and-bound* específico para o SPLC, implementado como sugestão de Souza e Wolsey [SW97]. A Seção 3.3, finalmente, introduz uma extensão do método adotado em [Bak74] para o cálculo de limitantes inferiores para o problema de escalonamento com restrição de recursos.

3.1 Formulações MIP

Devido às restrições de mão-de-obra, as formulações MIP para o SPLC baseiam-se na discretização do horizonte de tempo considerado. Desta forma, os instantes em que cada job pode ser iniciado ficam limitados a um número finito de possibilidades. A seguinte notação será usada nas formulações apresentadas nesta seção:

$N = \{1, \ldots, n\}$: conjunto de jobs;

$G = (N, A)$: grafo direcionado acíclico representando as precedências entre jobs;

L: total de mão-de-obra disponível em cada instante;

T: limite do horizonte de planejamento. Os instantes de tempo considerados são $1, 2, \ldots, T$;

p_j: duração do job j;

$(\ell_{j,1}, \ell_{j,2}, \ldots, \ell_{j,p_j})$: perfil de mão-de-obra do job j;

e_j, f_j: instantes mais cedo e mais tarde, respectivamente, em que o job j pode começar, calculados considerando apenas restrições de precedências e T.

Um job *dummy* com duração 0 é criado como sucessor de todos os demais jobs de tal forma que minimizar o *makespan* corresponde a começar este job o mais cedo possível. O conjunto N é tal que o job n corresponde a este job *dummy*.

A Seção 3.1.1, a seguir, apresenta uma formulação MIP para o SPLC inicialmente proposta no contexto do projeto PAMIPS [PAM] e recentemente revisada por Souza e Wolsey em [SW97]. A Seção 3.1.2, na seqüência, introduz uma formulação MIP alternativa para o SPLC, resultado desta dissertação. Finalmente, a Seção 3.1.3 traz os resultados computacionais obtidos com estas formulações.

3.1.1 Formulação x

Para todo job $j \in N$ e todo instante $t \in [e_j, f_j]$, seja x_{jt} uma variável binária igual a 1 se e somente se o job j começa no instante t. Seja ainda s_j a variável que indica o instante de início do job j, para todo $j \in N$. O SPLC pode ser formulado como:

$$
\begin{aligned}
& Minimize && s_n && && \\
& s.a. && && && \\
& \sum_{t=e_j}^{f_j} x_{j,t} && = && 1 \qquad \forall j \in N, && (3.1)\\
& \sum_{t=e_j}^{f_j} t x_{j,t} && = && s_j \qquad \forall j \in N, && (3.2)\\
& s_j && \geq && s_i + p_i \qquad \forall (i,j) \in A, && (3.3)\\
& \sum_{j=1}^{n} \sum_{u=1}^{p_j} \ell_{j,u} x_{j,t-u+1} && \leq && L \qquad \forall t \in [1, \ldots, T], && (3.4)\\
& x_{j,t} \in \{0,1\} && && \forall j \in N, \forall t \in [e_j, \ldots, f_j] && (3.5)
\end{aligned}
$$

A restrição (3.1) assegura que cada job começa exatamente em um único instante de tempo; a restrição (3.2) liga as variáveis $x_{j,t}$ e s_j; a restrição (3.3) cuida das relações de precedência entre os jobs e a restrição (3.4) garante que, a cada instante, a utilização de mão-de-obra não excede o máximo disponível.

Vale observar que, na ausência da restrição de mão-de-obra (3.4), o SPLC se reduz ao problema de determinar o instante de início de processamento de cada job respeitando as restrições de precedência entre eles. Neste caso, como observado por Souza e Wolsey em [SW97], a desigualdade de precedência entre jobs (3.3) pode ser melhor representada por:

$$\sum_{s=1}^{t} x_{i,s} - \sum_{s=1}^{t+p_i} x_{j,s} \geq 0 \qquad \forall (i,j) \in A, t \in T \tag{3.6}$$

A vantagem desta substituição é que o poliedro associado às restrições (3.1), (3.5) e (3.6) é inteiro, ou seja, todos os seus vértices são pontos inteiros. Na formulação x apresentada nesta seção e usada ao longo deste trabalho, contudo, optou-se por manter a restrição de precedência entre jobs representada pela desigualdade (3.3) e não pela desigualdade (3.6), porque na prática o uso desta última resulta em LP's mais lentos e sem nenhum ganho na qualidade da solução obtida com a relaxação linear [SW97].

Na teoria de programação linear, define-se uma desigualdade válida para um conjunto de pontos S como uma desigualdade que é satisfeita por todos os pontos deste conjunto. A motivação para procurar desigualdades válidas para o conjunto de pontos definidos por alguma formulação MIP é tentar, com o acréscimo destas desigualdades à formulação, melhorar a solução obtida com a relaxação linear.

Seja S^x o conjunto dos pontos (x, s) que satisfazem as equações (3.1) a (3.5). A proposição abaixo enuncia uma desigualdade válida, obtida por Souza e Wolsey [SW97], para o conjunto S^x.

Proposição 1 (Desigualdade Caminho) *Seja $j_1, j_2, \ldots, j_r \in N$ uma seqüência de jobs com $(j_i, j_{i+1}) \in A$ para $i = 1, \ldots, r-1$. Para todo $t_1 \geq t_2 \geq \ldots \geq t_r$, com $e_{j_i} \leq t_i \leq f_{j_i}$, $i = 1, \ldots, r-1$, a desigualdade*

$$\sum_{i=1}^{r} \sum_{u=1}^{p_{j_i}} x_{j_i, t_i - u + 1} \leq 1 \tag{3.7}$$

é válida para S^x.

Uma demonstração formal da Proposição 1 pode ser encontrada em [SW97]. De maneira informal, a validade da equação (3.7) pode ser entendida da seguinte forma:

Para cada job j_i, $i = 1, \ldots, r$, o termo $\sum_{u=1}^{p_{j_i}} x_{j_i,t_i-u+1}$ é igual a 1 se e somente se este job começou no intervalo $[t_i - p_{j_i} + 1, t_i]$. O que esta desigualdade diz é que, dada uma seqüência de jobs satisfazendo as condições da proposição, no máximo um dos jobs deste caminho pode começar no intervalo considerado. Para ver isto, considere que o job j_i começou no intervalo $[t_i - p_{j_i} + 1, t_i]$. Neste caso, tem-se:

(i) O mais cedo que um sucessor de j_i pode começar é $t_i + 1$. Os sucessores de j_i na seqüência são os jobs j_k, com $k = i + 1, \ldots, r$. Como o intervalo considerado para cada sucessor j_k de j_i é $[t_k - p_{j_k} + 1, t_k]$, e $t_i \geq t_k$, para todo $k = i + 1, \ldots, r$, nenhum sucessor j_k pode começar neste intervalo.

(ii) O mais tarde que um predecessor j_k de j_i pode ter começado é $t_i - p_{j_k}$. Os predecessores de j_i na seqüência são os jobs j_k, com $k = 1, \ldots, i - 1$. O intervalo considerado para cada predecessor j_k é $[t_k - p_{j_k} + 1, t_k]$. Mas $t_i - p_{j_k} \leq t_k - p_{j_k}$, pois $t_i \leq t_k$. Logo, $t_i - p_{j_k} < t_k - p_{j_k} + 1$ e, portanto, nenhum predecessor j_k de j_i pode ter começado neste intervalo.

3.1.2 Formulação bloco - jobs especiais

Buscando uma alternativa para a formulação x da Seção 3.1.1, chegou-se numa segunda formulação MIP para o SPLC. Antes de apresentá-la, contudo, é necessário introduzir alguns conceitos e definir novas variáveis.

Define-se um bloco B como a maior seqüência de jobs idênticos $j_1, \ldots, j_{n_B}$ tais que $(j_i, j_{i+1}) \in A$ para $i = 1, \ldots, n_B - 1$. Por convenção, $e_B = e_{j_1}$ e $f_B = f_{j_{n_B}}$ denotam, respectivamente, o mais cedo e o mais tarde que um job no bloco B pode começar. Além disso, $p_B = p_{j_1}$ e $\ell_{B,u} = \ell_{j_1,u}$ representam, respectivamente, a duração e a necessidade de mão-de-obra de um job no bloco B.

Dois jobs $i, j \in N$ são ditos especiais se eles pertencem a blocos distintos e se $(i, j) \in A$.

Para cada job j especial, seja $z_{j,t}$ uma variável de job especial tal que $z_{j,t} = 1$ se o job j começou até o instante t (inclusive) e $z_{j,t} = 0$ caso contrário.

Para cada bloco B, seja $Z_{B,t}$ uma variável de bloco que indica o número de jobs do bloco B que começaram até o instante t (inclusive).

Seja $\mathcal{B} = \{1, 2, \ldots, m\}$ o conjunto de todos os blocos, onde m é um bloco *dummy* com um único job especial *dummy*, de duração 0 e sucessor de todos os demais blocos.

A Figura 3.1 ilustra os conceitos de blocos e jobs especiais para a *instância exemplo* do SPLC

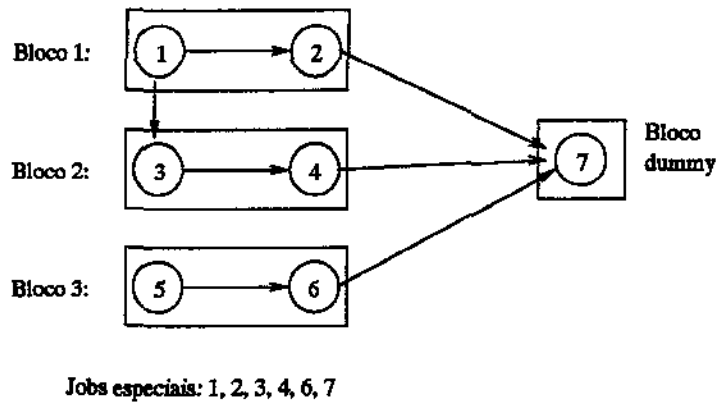


Figura 3.1: Blocos e jobs especiais da *instância exemplo* do SPLC.

(Seção 2.2).

A restrição do SPLC de que cada job deve começar em exatamente um único instante de tempo (restrição 3.1 da formulação x), é representada, em função das variáveis $z_{j,t}$ e $Z_{B,t}$, pelas seguintes desigualdades:

$$-Z_{B,e(B)} \leq 0 \quad \forall B \in \mathcal{B} \tag{3.8}$$

$$Z_{B,f(B)} = n_B \quad \forall B \in \mathcal{B} \tag{3.9}$$

$$Z_{B,t} - Z_{B,t+1} \leq 0 \quad \forall B \in \mathcal{B},\ t = e_B, \ldots, f_B - 1 \tag{3.10}$$

$$Z_{B,t} - Z_{B,t-p_B} \leq 1 \quad \forall B \in \mathcal{B}, t = e_B + p_B, \ldots, f_B \tag{3.11}$$

$$-z_{j,e(j)} \leq 0 \quad \forall j \ especial, \tag{3.12}$$

$$z_{j,f(j)} = 1 \quad \forall j \ especial, \tag{3.13}$$

$$z_{j,t-1} - z_{j,t} \leq 0 \quad \forall j \ especial, t = e_j + 1, \ldots, f_j\ , \tag{3.14}$$

Restrições de precedência entre blocos são tratadas pelos jobs especiais. Assim, para todo $(i,j) \in A$, com i,j jobs especiais, $i \in B$, $j \in B'$, $B \neq B'$, tem-se:

$$z_{j,t+p_i} - z_{i,t} \leq 0, \quad e_i \leq t \leq f_i,\ e_j \leq t + p_i \leq f_j \tag{3.15}$$

Seja $B = \{1, 2, \ldots, n_B\}$ um bloco com jobs especiais $j_1, \ldots, j_r$ nas posições $1 \leq q_{j_1} < \ldots < q_{j_r} \leq n_B$. As restrições de precedência entre jobs especiais dentro do mesmo bloco são tratadas da seguinte forma: para dois quaisquer jobs especiais consecutivos em B, j_l, j_{l+1}, $l = 1, \ldots, r-1$, vale a seguinte desigualdade:

$$z_{j_l,t} \geq z_{j_{l+1},t+(q_{j_{l+1}}-q_{j_l}-1)p_B} \ \forall t, \ e_{j_l} \leq t \leq f_{j_l} \ , \ e_{j_{l+1}} \leq t + p_B \leq f_{j_{l+1}} \tag{3.16}$$

A relação entre as variáveis de bloco $Z_{B,t}$ e as variáveis $z_{j,t}$ dos jobs especiais é dada pela seguinte proposição:

Proposição 2 *Se um bloco* $B = \{1, \ldots, n_B\}$ *contém jobs especiais* $j_1, \ldots, j_r$ *nas posições* $1 \leq q_{j_1} < \ldots < q_{j_r} \leq n_B$, *então*

$$Z_{B,t} \geq \sum_{u=1}^{q_{j_1}} z_{j_1,t+(u-1)p_B} + \sum_{u=1}^{q_{j_2}-q_{j_1}} z_{j_2,t+(u-1)p_B} + \ldots + \sum_{u=1}^{q_{j_r}-q_{j_{r-1}}} z_{j_r,t+(u-1)p_B}, \tag{3.17}$$

$$Z_{B,t} \leq (q_{j_1} - 1) - \sum_{v=1}^{r} \sum_{u=1}^{q_{j_{v+1}}-q_{j_v}} z_{j_v,t-(u-1)p_B} \tag{3.18}$$

com $q_{j_{r+1}} = n_B + 1$

Demonstração:

(i)

$$\begin{aligned} Z_{B,t} &= (z_{1,t} + z_{2,t} + \ldots + z_{j_1,t}) + (z_{j_1+1,t} + \ldots + z_{j_2,t}) + \ldots \\ &\quad +(z_{j_r+1,t} + \ldots + z_{n_B,t}) \\ &\geq z_{j_1,t+(q_{j_1}-1)p_B} + z_{j_1,t+(q_{j_1}-2)p_B} + \ldots + z_{j_1,t} \\ &+ z_{j_2,t+(q_{j_2}-q_{j_1}-1)p_B} + z_{j_2,t+(q_{j_2}-q_{j_1}-2)p_B} + \ldots + z_{j_2,t} \\ &+ \ldots \\ &+ 0 \end{aligned}$$

(ii)

$$\begin{aligned} Z_{B,t} &= (z_{1,t} + z_{2,t} + \ldots + z_{j_1-1,t}) + (z_{j_1,t} + \ldots + z_{j_2-1,t}) + \ldots \\ &\leq (q_{j_1} - 1) + (z_{j_1,t} + z_{j_1,t-p_B} + \ldots + z_{j_1,t-(q_{j_2}-q_{j_1})p_B}) \\ &+ \ldots \end{aligned}$$

□

Finalmente, a restrição de mão-de-obra, em função das variáveis $Z_{B,t}$, pode ser escrita como:

$$\sum_{B \in \mathcal{B}} \left[\sum_{i=1}^{p_B-1} (\ell_{B,i+1} - \ell_{B,i}) Z_{B,t-i} + \ell_{B,1} Z_{B,t} - \ell_{B,p_B} Z_{B,t-p_B} \right] \leq L \tag{3.19}$$

Desta forma, a formulação bloco - jobs especiais completa é:

$$
\begin{aligned}
\textit{Minimize} \quad & \sum_{t=1}^{T} t(Z_{n,t} - Z_{n,t-1}) \\
\textit{s.a.} & \\
(3.8) \ a \ (3.19) & \\
Z_{B,t} \in \{0,1\} \quad & \forall B \in \mathcal{B} \ \ t = e_B, \ldots, f_B
\end{aligned}
$$

Note que a integralidade das variáveis $z_{j,t}$ dos jobs especiais fica automaticamente garantida pelas restrições (3.17), (3.18) e pela integralidade exigida nas variáveis $Z_{B,t}$.

3.1.3 Resultados Computacionais

A motivação inicial do estudo de formulações de programação inteira para o SPLC foi a possibilidade de usar as correspondentes relaxações lineares para obter limitantes inferiores para o *makespan* do problema. Esperava-se, com isto, conseguir alguma melhora em relação ao limitante inferior trivial do caminho crítico, $\underline{\xi}_{CC}$. Contudo, a implementação das formulações x e bloco - jobs especiais, levou aos seguintes resultados:

- O número de variáveis, restrições e coeficientes não-zeros em cada uma das formulações é muito grande, como pode ser constatado pela análise assintótica destes valores apresentada na Tabela 3.1. Isso tem como conseqüência LPs computacionalmente lentos.
- Em todas as instâncias testadas, o limitante inferior dado pelo valor da relaxação linear de cada uma das duas formulações implementadas nunca é superior a $\underline{\xi}_{CC}$. Este valor é bem menor do que o melhor *makespan* conhecido para as instâncias de teste do SPLC [Cav97].
- O uso do CPLEX [CPL95] com suas opções padrão para o *branch-and-bound*, também não possibilita uma melhora nos limitantes inferiores: o valor da função objetivo praticamente permanece inalterado durante milhares de nós. Além disso, soluções viáveis não são encontradas a não ser que o limite T do horizonte de planejamento seja muito maior que o *makespan*.

Outras duas formulações MIP foram propostas para o SPLC por Souza e Wolsey em [SW97]. Contudo, o seu uso, assim como o de algumas desigualdades válidas para os modelos MIP estudados por estes autores e por Wang em [WW96], tiveram pouco ou nenhum impacto na obtenção de limitantes inferiores para o SPLC.

Apesar de todos estes pontos, optou-se pela implementação de um *branch-and-bound* específico para o SPLC, como sugerido por Souza e Wolsey em [SW97]. A idéia era usar informações

	Formulações	
	x	bloco - jobs especiais
# Variáveis	$O(nT)$	$O(nT)$
# Restrições	$O(nT + \|A\|)$	$O((n + \|A\|)T)$
# Não-zeros	$O(nT + \|A\|)$	$O((mn + \|A\|)T)$

Tabela 3.1: Número assintótico de variáveis, restrições e coeficientes não-zero das formulações MIP para o SPLC

próprias da estrutura do problema para acelerar o processo de enumeração e tentar obter melhores limitantes inferiores para o *makespan*. Os detalhes do algoritmo implementado são o assunto da próxima seção.

3.2 Um Algoritmo de *Branch-and-Bound* para o SPLC

Entre as estratégias exatas mais aplicadas a problemas combinatórios está o método de *branch-and-bound* (*B&B*). Como o próprio nome indica, trata-se de um esquema enumerativo fundamentado em duas operações básicas: *branching* e *bounding*.

A operação *branching* diz respeito à decomposição do espaço de soluções S^0 do problema original em subespaços menores que são mutuamente exclusivos e cuja união corresponde a S^0.

A operação *bounding* envolve o cálculo de limitantes inferiores/superiores para a solução ótima em cada um dos subespaços gerados na fase de *branching*. A idéia é eliminar subespaços que não podem levar a soluções melhores que as conhecidas até um certo instante, e assim diminuir o espaço de enumeração.

No *branch-and-bound* dedicado ao SPLC, foi adotada a formulação x (Seção 3.1.1) por ter sido ela a que resultou em LPs menos lentos. A função objetivo foi modificada de min $f(\mathbf{x}, \mathbf{s}) = s_n$ para min $g(\mathbf{x}, \mathbf{s}) = \sum_{j=1}^{n-1} \alpha_j s_j + s_n \sum_{j \in N} T\alpha_j$, onde α_j é o tamanho do caminho crítico do subgrafo induzido em G pelo job j e todos os seus sucessores.

A mudança na função objetivo foi motivada pela observação de que uma das prováveis razões para o mau funcionamento de um algoritmo de *B&B* para o SPLC vem do fato da função objetivo de minimização do *makespan* ser muito pouco informativa para o processo de busca. Em geral, são muitas as soluções $(x, s) \in S^x$ que possuem exatamente o mesmo *makespan*. Este alto grau de degenerescência leva a limitantes inferiores muito pobres comprometendo severamente a busca de soluções viáveis no processo de enumeração. A função g foi introduzida neste trabalho com o objetivo de guiar melhor o processo de busca, reduzindo a degenerescência ao mesmo

tempo em que garante a otimalidade da solução final com respeito ao *makespan*. Este resultado é mostrado a seguir.

Proposição 3 *Se* $(\mathbf{x}^*, \mathbf{s}^*) \in S^x$ *é uma solução que minimiza* g, *então* $(\mathbf{x}^*, \mathbf{s}^*)$ *também minimiza* f *e, portanto, tem mínimo makespan.*

Demonstração:

$$\text{Seja } g^* = g(\mathbf{x}^*, \mathbf{s}^*) = \underbrace{\sum_{j=1}^{n-1} \alpha_j s_j^*}_{A} + s_n^* \sum_{j \in N} T\alpha_j = A + s_n^* \sum_{j \in N} T\alpha_j. \quad (I)$$

Suponha, por contradição, que $(\mathbf{x}^*, \mathbf{s}^*)$ não seja uma solução de mínimo *makespan* e seja $(\mathbf{x}^o, \mathbf{s}^o) \in S^x$ tal que $f^* = f(\mathbf{x}^o, \mathbf{s}^o)$ é mínimo. Assim, tem que existir $k \in Z_+^+$ tal que:

$$f(\mathbf{x}^*, \mathbf{s}^*) = s_n^* = f^* + k > f^* = s_n^o = f(\mathbf{x}^o, \mathbf{s}^o).$$

$$\text{Seja } g^o = g(\mathbf{x}^o, \mathbf{s}^o) = \underbrace{\sum_{j=1}^{n-1} \alpha_j s_j^o}_{B} + s_n^o \sum_{j \in N} T\alpha_j = B + s_n^o \sum_{j \in N} T\alpha_j \quad (II)$$

Como g^* é mínimo, $g^* - g^o \leq 0 \quad (III)$.

De (I) e (II) tem-se que:

$$g^* - g^o = (A - B) + \underbrace{(s_n^* - s_n^o)}_{f^*+k-f^*} \sum_{j \in N} T\alpha_j = (A - B) + k \sum_{j \in N} T\alpha_j.$$

Logo, de (III), tem-se que: $(B - A) \geq k \sum_{j \in N} T\alpha_j$.

Mas, como $B \leq \sum_{j \in N} T\alpha_j$ e $A > 0$, conclui-se que:

$$\sum_{j \in N} T\alpha_j > (B - A) \geq k \sum_{j \in N} T\alpha_j \text{ e, portanto, } \sum_{j \in N} T\alpha_j > k \sum_{j \in N} T\alpha_j,$$

o que implica $k < 1$. Isto é uma contradição, pois, por hipótese, k tinha que ser inteiro e positivo. Portanto, $(\mathbf{x}^*, \mathbf{s}^*)$ tem mínimo *makespan*. □

A implementação do *B&B* específico para o SPLC foi feita usando o pacote MINTO [SN96]. O esqueleto do algoritmo está descrito na Figura 3.2. As especificidades do SPLC foram incorporadas às seguintes etapas do *B&B*:

Pré-processamento (Fixação de variáveis)

Algoritmo *Branch-and-Bound* para o SPLC

Variáveis: LB /* limitante inferior */
UB /* limitante superior */
Ativos /* lista de nós ativos da árvore de *B&B* */

Leia o problema;
$LB = 0$;
$UB = \infty$;
Ativos = *raiz*;
Enquanto $LB < UB$ **faça**
Se lista *Ativos* está vazia
Então fim do algoritmo;
Senão
Selecione um nó i da lista *Ativos*;
($\star$) Faça um pré-processamento em i;
Calcule um limitante inferior LB_i para o nó i (relaxação linear);
Se $LB_i > LB$
Então atualize LB;
Calcule um limitante superior UB_i para o nó i (heurística primal);
Se $UB_i < UB$
Então atualize UB;
Se $LB_i > UB$ /* nó i pode ser eliminado */
Então retire o nó i da lista *Ativos*;
Senão
Se existem algumas desigualdades que podem ser
acrescentadas ao nó i (planos de corte)
Então acrescente estas desigualdade ao nó i e volte para ($\star$);
Senão
Faça *branching* no nó i criando dois novos nós i_1, i_2;
Retire o nó i da lista *Ativos*;
Inclua i_1, i_2 na lista *Ativos*;

Figura 3.2: Algoritmo *Branch-and-Bound* para o SPLC

Cada job j está associado a uma janela de tempo $[e_j, f_j]$ que indica o intervalo no qual ele pode começar. O objetivo do pré-processamento em cada nó da árvore de *B&B* é reduzir o tamanho destas janelas. São duas as situações em que isto é possível: (i) Por ocasião de um *branching*, quando $e_{j'}$ e $f_{j'}$ de algum job j' são alterados; (ii) Quando uma nova solução viável de valor $\overline{\xi}$ é encontrada.

No caso (i), sucessores e predecessores de j' devem ter seus e_j e f_j atualizados de modo que:

$$\begin{cases} e_j = \max\{e_j, e_{j'} + p_{j'}\}, & \forall j \text{ sucessor de } j' \\ f_j = \min\{f_j, f_{j'} - p_j\}, & \forall j \text{ predecessor de } j' \end{cases}$$

Recursivamente, sucessores dos sucessores e predecessores dos predecessores de j' também devem ter seus e_j e f_j atualizados.

No caso (ii), variáveis $x_{j,t}$ com $t > \overline{\xi}$ podem ser eliminadas. Os valores de f_j, $\forall$ job j, são atualizados considerando as relações de precedência e o novo limite $T = \overline{\xi}$ do horizonte de planejamento.

Após o pré-processamento, para cada job j, as variáveis $x_{j,t}$ com t fora da janela $[e_j, f_j]$ são fixadas em 0.

Heurística Primal

A idéia básica de uma heurística primal é usar informação da relaxação linear para construir soluções viáveis para o problema.

A heurística primal implementada no *B&B* específico para o SPLC adota a estratégia *Schedule-by-Best-α* de [SUW97] para obtenção de uma permutação dos jobs, $P_{best-\alpha}$, a partir da solução da relaxação linear (Seção 2.4.2).

A construção de um escalonamento viável para o SPLC a partir de $P_{best-\alpha}$ acontece nos seguintes passos:

1. $P_{best-\alpha}$ é transformada em uma nova permutação, $P'_{best-\alpha}$, consistente com as restrições de precedência. Isto é necessário porque nem sempre $P_{best-\alpha}$ satisfaz a estas restrições (um job pode aparecer antes de seus predecessores nesta permutação). $P'_{best-\alpha}$ é determinada pela seleção sucessiva do primeiro job em $P_{best-\alpha}$ que não tem predecessores ou cujos predecessores já foram todos selecionados.
2. A partir de $P'_{best-\alpha}$, são construídos dois escalonamentos viáveis para o SPLC:
   - *directSchedule*: escalonamento direto onde os jobs de $P'_{best-\alpha}$ são escalonados um a um, na ordem direta (do primeiro ao último) em que aparecem nesta seqüência, no instante mais cedo possível considerando as restrições de mão-de-obra;

Algoritmo *BuildDirectSchedule*
Entrada: *sequence* /* permutação dos n jobs */
Instância na ordem direta original
Saída: *start* /* instantes de início de processamento dos n jobs */

/* Inicializa o mais cedo que cada job pode começar */
Para $j = 1, \ldots, n$ **faça** $lbstart[j] = 1$;
/* Inicializa disponibilidade de mão-de-obra */
Para $t = 1, \ldots, T$ **faça** $labour[t] = L$;
/* Escalona os jobs */
Para $i = 1, \ldots, n$ **faça**
$j = sequence[i]$;
$t = lbstart[j]$;
$t = SearchFirstStart(j, t)$; (Figura 3.5)
$start[j] = t$;
/* Atualiza disponibilidade de mão-de-obra */
Para $k = 1, \ldots, p_j$ **faça** $labour[t + k - 1] -= \ell_{j,k}$;
/* Atualiza o mais cedo que os sucessores de j podem começar */
Para todo sucessor s de j **faça** $lbstart[s] = \max\{lbstart[s], t + p_j\}$;

Figura 3.3: Algoritmo para construção de um escalonamento direto viável a partir de uma ordem dos jobs.

- *inverseSchedule*: escalonamento inverso onde os jobs de $P'_{best-\alpha}$ são escalonados um a um, na ordem inversa (do último ao primeiro) em que aparecem nesta seqüência, no instante mais cedo possível considerando as restrições de mão-de-obra;

3. O escalonamento de menor *makespan* entre *directSchedule* e *inverseSchedule* é adotado.

Na construção do escalonamento inverso *inverseSchedule*, trabalha-se com a instância na ordem inversa da instância original. Se I representa a instância original, a instância inversa de I, I', é tal que:

- O grafo de precedências G' de I' é o grafo transposto do grafo G de I.
- O vetor do perfil de mão-de-obra de cada job de I' é o inverso do vetor correspondente em I.

A intuição para construção do escalonamento inverso está no fato de que ele, lido da direita para esquerda, é exatamente uma solução para a instância original.

As Figuras 3.3 e 3.4 mostram os algoritmos *BuildDirectSchedule* e *BuildInverseSchedule* que constroem, respectivamente, um escalonamento direto e um escalonamento inverso para o SPLC

Algoritmo *BuildInverseSchedule*
Entrada: *sequence* /* permutação dos n jobs */
Instância na ordem inversa
Saída: *start* /* instantes de início de processamento dos n jobs */

/* Inicializa o mais cedo que cada job pode começar */
Para $j = 1, \ldots, n$ **faça** $lbstart[j] = 1$;
/* Inicializa disponibilidade de mão-de-obra */
Para $t = 1, \ldots, T$ **faça** $labour[t] = L$;
/* Escalona os jobs */
Para $i = n, \ldots, 1$ **faça**
$j = sequence[i]$;
$t = lbstart[j]$;
$t = SearchFirstStart(j, t)$; (Figura 3.5)
$start[j] = t$;
/* Atualiza disponibilidade de mão-de-obra */
Para $k = 1, \ldots, p_j$ **faça** $labour[t + k - 1] -= \ell_{j,k}$;
/* Atualiza o mais cedo que os sucessores de j podem começar */
Para todo sucessor s de j **faça** $lbstart[s] = \max\{lbstart[s], t + p_j\}$;

Figura 3.4: Algoritmo para construção de um escalonamento inverso viável a partir de uma ordem dos jobs.

a partir de uma ordem dos jobs que respeita as restrições de precedência. Note a presença nestes algoritmos da função auxiliar *SearchFirstStart* descrita na Figura 3.5.

Esta função, calcula o instante mais cedo em que um job pode ser escalonado respeitando as restrições de precedência e mão-de-obra. Como ela é chamada freqüentemente pelos algoritmos *BuildDirectSchedule* e *BuildInverseSchedule*, a sua complexidade computacional é muito importante. Por conta disso, na implementação de *SearchFirstStart* foram usadas algumas técnicas dos algoritmos de casamento de padrão [Man89]. O objetivo era conseguir uma complexidade linear no tamanho do escalonamento no momento em que esta função é chamada. Entretanto, devido ao perfil variável de mão-de-obra necessária para execução de cada job, no pior caso *SearchFirstStart* ainda tem complexidade quadrática no tamanho do escalonamento corrente. Na prática, contudo, observou-se que, em média, a complexidade desta função é bem próxima de linear.

Note-se, por fim, que as soluções obtidas pela heurística primal do *B&B* para o SPLC são limitantes superiores para o escalonamento ótimo. Estes limitantes estarão tanto mais próximos da solução ótima, quanto melhor for a qualidade da permutação $P_{best-\alpha}$ obtida com a relaxação linear.

Algoritmo *SearchFirstStart*
Entrada: job j e instante t a partir do qual j pode começar
Saída: mínimo t no qual j pode ser escalonado sem violar a restrição de mão-de-obra

```
s = t;
canSchedule = false;
Enquanto canSchedule == false faça
    /* Inicializa o mais cedo que cada tarefa k do job j pode começar */
    Para k = 1,...,p_j faça
        Enquanto ℓ_{j,k} > labour[s] faça s++;
        firstStart[k] = s;
        s++;
    /* Tenta escalonar j no intervalo [firstStart[p_j] - p_j + 1, firstStart[p_j]] */
    s = firstStart[p_j] - 1;
    Para k = p_j - 1,...,1 faça
        Se ℓ_{j,k} > labour[s]
            Então s = firstStart[p_j] - p_j + 2;
                  Interrompa laço "Para";
            Senão Se k == 1
                      Então canSchedule = 1;
                            t = s;
                      Senão s--;
```

Figura 3.5: Algoritmo para calcular o mais cedo que um job pode ser escalonado sem violar a restrição de mão-de-obra.

Branching

O esquema padrão de *branching* para problemas de programação inteira com variáveis 0-1 consiste na escolha de uma variável fracionária i (normalmente com $|i - 0.5|$ mínimo) e na criação de dois novos nós: um com i fixada em 0 e outro com i fixada em 1.

Para o SPLC, contudo, o uso deste esquema pode resultar em árvores *B&B* altamente desbalanceadas, o que, em geral, torna mais lento o processo de enumeração [SWW96]. Por conta disto, e tentando capturar mais da estrutura do problema, o seguinte esquema de *branching* foi adotado para o *B&B* dedicado ao SPLC:

- Um job j^*, com $x_{j^*,t}$ fracionário para algum $t \in [e_j, f_j]$, é escolhido para ser usado no *branching*. Jobs no caminho crítico e que ainda não foram fixados são prioritários nesta escolha.

- Dois novos nós são criados: nó 1, onde $\sum_{t=e_j}^{t^*} x_{j^*,t} = 1$ e nó 2, onde $\sum_{t=t^*+1}^{f_j} x_{j^*,t} = 1$.

  O valor de (t^* é calculado por $argmin_{e_j \leq t \leq f_j}\{\sum_{s=e_j}^{t} x_{j^*,s} \geq 0.5\}$)

  Isto equivale a fazer $f_{j^*} = t^*$ no nó 1, e $e_{j^*} = t^* + 1$ no nó 2. Como conseqüência, todas as variáveis $x_{j^*,t}$, $t > t_{j^*}$, no nó 1 e $x_{j^*,t}$, $t \leq t^*$, no nó 2 podem ser fixadas em 0. Os e_j e f_j dos sucessores e predecessores de j^* são alterados no pré-processamento feito nestes nós quando eles forem selecionados durante o processo de enumeração.

Este esquema de *branching* é conhecido na literatura como *GUB branching* [NW88].

Busca na árvore de *B&B*

A cada passo do algoritmo de *B&B* implementado para o SPLC, o nó escolhido para ser avaliado é aquele com menor limitante inferior.

Planos de corte

Devido ao grande número de restrições da formulação x completa, inicialmente não são carregadas as restrições de precedência entre jobs (desigualdade (3.3)). Isto diminui o tamanho dos LP's que, conseqüentemente, são resolvidos mais rápido. As referidas desigualdades vão sendo acrescentadas sempre que forem violadas por alguma solução fracionária, exatamente como em um algoritmo de planos de corte.

3.2.1 Resultados Computacionais

A implementação do *B&B* para o SPLC levou aos seguintes resultados:

- Para quatro das menores instâncias de teste foi possível chegar na solução ótima para um valor de $L = 18$ trabalhadores disponíveis a cada instante (Tabela 3.2).
- Para todas as demais 21 instâncias, o tempo para resolver um único LP já foi computacionalmente excessivo. O algoritmo de *B&B* foi interrompido após 1 hora de processamento e o melhor limitante inferior conhecido ainda era igual ao do caminho crítico.
- Foi detectada uma provável falha do MINTO: embora ele use o CPLEX para resolver seus LPs, o tempo para resolver apenas um LP através do MINTO é mais de 10 vezes maior do que se o mesmo LP for resolvido diretamente pelo CPLEX. Em decorrência deste problema, os testes com o *B&B* específico para o SPLC ficaram inviáveis e tiveram que ser abandonados.

Instância	ξ^*
Ins_4o_21j_A	82
Ins_4o_23j_A	58
Ins_4o_24j_B	72
Ins_4o_27j_A	67

Tabela 3.2: Soluções ótimas obtidas com o algoritmo de *branch-and-bound* implementado para o SPLC.

A próxima seção descreve a última tentativa feita neste trabalho no sentido de usar programação inteira para a obtenção de limitantes inferiores para o SPLC.

3.3 Outros Limitantes Inferiores para o SPLC

Baker em [Bak74] enuncia três teoremas para obter limitantes inferiores para o problema de escalonamento com restrição de recurso. Antes de apresentá-los é necessário introduzir a seguinte notação:

ESS (*Early Start Schedule*): escalonamento onde cada job começa no seu e_j;

LSS (*Late Start Schedule*): escalonamento onde cada job começa no seu f_j;

$r_{ESS}(t)$: total de mão-de-obra utilizada pelos jobs em processamento no tempo t no ESS;

$r_{LSS}(t)$: total de mão-de-obra utilizada pelos jobs em processamento no tempo t no LSS;

$r_S(t)$: total de mão-de-obra utilizada pelos jobs em processamento no tempo t em um escalonamento arbitrário S;

R: total de mão-de-obra necessária para executar todos os jobs;

α: $argmin_{1\leq t\leq T}\{r_{ESS}(t) > L\}$;

β: $argmax_{1\leq t\leq T}\{r_{LSS}(t) > L\}$.

Claramente, $\sum_{u=1}^{t} r_{ESS}(u) \geq \sum_{u=1}^{t} r_S(u) \geq \sum_{u=1}^{t} r_{LSS}(u), \quad \forall t = 1,\ldots,T \quad (1)$

e $\sum_{u=T-t}^{T} r_{ESS}(u) \leq \sum_{u=T-t}^{T} r_S(u) \leq \sum_{u=T-t}^{T} r_{LSS}(u), \quad \forall t = 1,\ldots,T \quad (2)$

Os teoremas de Baker estabelecem condições necessárias para que não exista um escalonamento viável de *makespan* D. São eles:

Teorema 1 *Se* $\sum_{u=1}^{t} r_{LSS}(u) > tL$ *para algum* t, $1 \leq t \leq D$, *então não existe escalonamento viável de makespan* D.

Demonstração: Sob a hipótese do teorema, segue de (1) que $\sum_{u=1}^{t} r_S(u) > tL$. Logo, não há mão-de-obra disponível o suficiente para executar todos os jobs em um escalonamento arbitrário de *makespan* D. □

Teorema 2 *Se* $\sum_{u=D-t}^{D} r_{ESS}(u) > tL$ *para algum* t, $1 \leq t \leq D$, *então não existe escalonamento viável de makespan* D.

Demonstração: Sob a hipótese do teorema, segue de (2) que $\sum_{u=D-t}^{D} r_S(u) > tL$. Logo, não há mão-de-obra disponível o suficiente para executar todos os jobs em um escalonamento arbitrário de *makespan* D. □

Teorema 3 *Seja* $A = \sum_{u=1}^{\alpha-1}[L - r_{ESS}(u)]$ *e seja* $B = \sum_{u=\beta+1}^{D}[L - r_{LSS}(u)]$. *Se* $R > DL - A - B$, *então não existe escalonamento viável de makespan* D.

Demonstração: Claramente, um escalonamento viável de *makespan* D só existe se $R \leq DL$. A parcela A representa a quantidade de mão-de-obra que nunca é usada no começo do ESS, e que, portanto, não é usada por nenhum escalonamento viável. Analogamente, a expressão B representa a quantidade de mão-de-obra que fica ociosa no final de qualquer escalonamento viável. Assim, um limitante superior para o total efetivo de mão-de-obra de que se dispõe para execução de todos os jobs em um escalonamento qualquer de *makespan* D é dado por $DL - A - B$.

□

Começando com D = caminho crítico, pode-se verificar, usando os Teoremas 1, 2 e 3, se é possível existir um escalonamento viável com esta duração. Caso não seja, o valor de D é incrementado - conseqüentemente, o limitante inferior também - e os cálculos são repetidos iterativamente até que nenhum dos teoremas seja violado.

Observando o Teorema 3, é fácil perceber que ele será tanto mais forte quanto maiores forem as parcelas A e B, referentes à mão-de-obra que nunca é usada em um escalonamento viável. Pensando nisto, surgiu a idéia de se usar uma formulação de programação inteira para calcular o número máximo de trabalhadores que pode ser usado nos δ_i primeiros e nos δ_f finais instantes de tempo de um escalonamento viável qualquer. Sejam τ_i e τ_f estes valores.

τ_i é obtido resolvendo :

$$\tau_i = \max\{\sum_{t \leq \delta_i} \sum_{j \in N: e_j \leq \delta_i} \sum_{u=1}^{p_j} \ell_{j,u} x_{j,t-u+1} : (x,s) \in S^x\}, \tag{3.20}$$

onde S^x é tal qual definido na Seção 3.1.1. Note que apenas os jobs j com $e_j \leq \delta_i$ precisam ser considerados no cálculo de τ_i, o que diminui bastante o tamanho do IP resultante.

De maneira similar, τ_f é obtido resolvendo :

$$\tau_f = \max\{\sum_{t \leq \delta_f} \sum_{j \in N: e_j \leq \delta_f} \sum_{u=1}^{p_j} \ell_{j,u} x_{j,t-u+1} : (x,s) \in (S^x)'\}. \tag{3.21}$$

onde $(S^x)'$ é o conjunto de pontos obtidos de maneira similar a S^x só que considerando a instância na ordem inversa. Novamente, apenas os jobs j com $e_j \leq \delta_f$ precisam ser considerados.

Uma vez calculados τ_i e τ_f, os mínimos de mão-de-obra ociosa no início e no final de qualquer escalonamento viável são dados respectivamente por $(\delta_i L - \tau_i)$ e $(\delta_f L - \tau_f)$. A extensão do Teorema 3 proposta neste trabalho faz uso destes valores e é enunciada como:

Teorema 4 *Se $R > DL - (\delta_i L - \tau_i) - (\delta_f L - \tau_f)$, então não existe um escalonamento viável de makespan D.*

Em outras palavras, o menor D tal que pode existir um escalonamento viável com esta duração é dado por:

$$D = \left\lceil \frac{R + (\delta_i L - \tau_i) + (\delta_f L - \tau_f)}{L} \right\rceil \tag{3.22}$$

3.3.1 Resultados Computacionais

A Tabela 3.3 mostra comparativamente os limitantes inferiores obtidos com os Teoremas 1, 2 e 3 de Baker [Bak74] ($\underline{\xi}_1$) e com o Teorema 4 proposto neste trabalho ($\underline{\xi}_2$). No cálculo de $\underline{\xi}_2$ foi usado $\delta_i = \delta_f = 20$. As colunas $\underline{\xi}_{CC}$ e $\underline{\xi}_{CP}$ trazem, respectivamente, os limitantes inferiores do caminho crítico e os obtidos com programação por restrições [HC97]. Para $\underline{\xi}_2$ e $\underline{\xi}_{CP}$ é dado também o tempo computacional gasto para chegar nos valores correspondentes. Todos os valores foram obtidos considerando $L = 18$ trabalhadores disponíveis a cada instante.

Como pode ser observado pelos resultados, o limitante inferior obtido com o Teorema 4 domina aquele obtido com os Teoremas de Baker e é, em 16 das 25 instâncias de teste, significativamente melhor que o limitante inferior do caminho crítico. Por outro lado, apenas em 4 instâncias, $\underline{\xi}_2$ é melhor do que o limitante inferior obtido com programação por restrições.

Instância	ξ_{CC}	ξ_1	ξ_2	(seg)	ξ_{CP}	(seg)
Ins_4o_21j_A	78	78	78	(212)	**82**	-
Ins_4o_23j_A	54	54	54	(206)	**58**	-
Ins_4o_24j_A	58	58	58	(160)	**67**	-
Ins_4o_24j_B	54	56	59	(785)	**72**	-
Ins_4o_27j_A	53	53	57	(202)	**67**	-
Ins_6o_41j_A	90	104	**112**	(775)	109	(1748)
Ins_6o_41j_B	94	94	94	(446)	**102**	(146)
Ins_6o_41j_C	81	88	98	(1621)	**110**	(2169)
Ins_6o_44j_A	75	89	96	(369)	**98**	(1482)
Ins_6o_44j_B	104	104	109	(217)	**124**	(1758)
Ins_8o_63j_A	174	186	**191**	(658)	187	(2919)
Ins_8o_63j_B	196	209	217	(668)	**239**	(5114)
Ins_8o_63j_C	227	227	227	(597)	**271**	(3216)
Ins_8o_65j_A	298	298	298	(93)	**342**	(5541)
Ins_8o_65j_B	230	250	262	(198)	**315**	(8650)
Ins_10o_84j_A	270	379	390	(383)	**394**	(7695)
Ins_10o_84j_B	200	335	343	(172)	**355**	(2701)
Ins_10o_85j_A	513	513	513	(179)	**671**	(12611)
Ins_10o_87j_A	194	371	**379**	(641)	377	(11298)
Ins_10o_88j_A	362	362	362	(758)	-	-
Ins_10o_100j_A	352	670	680	(98)	**830**	(6816)
Ins_10o_102j_A	550	622	633	(233)	**878**	(9004)
Ins_10o_106j_A	383	600	**609**	(270)	578	(8914)
Ins_12o_108j_A	502	691	700	(235)	**838**	(11284)
Ins_12o_109j_A	819	819	819	(199)	**980**	(19864)

Tabela 3.3: Limitantes inferiores para o SPLC.

Obter bons limitantes inferiores para o SPLC realmente não é uma tarefa trivial. Os esforços investidos neste trabalho com esta finalidade permitem fazer as seguintes observações:

- O uso isolado de formulações MIP (Seção 3.1), pelo menos com o que se conhece até o momento, não traz nenhum ganho em relação ao limitante inferior trivial do caminho crítico, $\underline{\xi}_{CC}$.
- A implementação de um *branch-and-bound* específico para o SPLC não tem tanto impacto, como esperado por Souza e Wolsey [SW97], na qualidade dos limitantes inferiores obtidos. Com exceção de quatro das menores instâncias de teste do SPLC, para as quais foi encontrada uma solução ótima, o algoritmo de *B&B* não consegue melhorar em nada o valor de $\underline{\xi}_{CC}$.
- Alguma melhora em $\underline{\xi}_{CC}$ pode ser obtida com o uso de uma formulação inteira no cálculo de um limitante inferior baseado no perfil de utilização de mão-de-obra em um escalonamento viável (Seção 3.3).
- Para a maioria das instâncias de teste do SPLC, os melhores limitantes inferiores conhecidos são os obtidos com programação por restrições [HC97]. Estes valores, contudo, ainda estão muito longe dos melhores limitantes superiores obtidos com as estratégias heurísticas para o SPLC descritas nos Capítulos 4 e 5 a seguir.

Capítulo 4

Limitantes Superiores I - Heurísticas Seqüenciais

No contexto de otimização combinatória, o termo heurística é usado em contraste a métodos exatos. Quando o alto tempo computacional exigido por estes últimos inviabiliza a busca de uma solução ótima global, heurísticas aparecem como uma alternativa bastante atrativa na obtenção de soluções sub-ótimas para problemas NP-difíceis.

Este capítulo apresenta as estratégias heurísticas seqüenciais implementadas neste trabalho para a obtenção de soluções para o SPLC. A Seção 4.1 descreve uma heurística baseada em regras de prioridade. Em seguida, a Seção 4.2 apresenta o conceito de classes de escalonamento e descreve três heurísticas de construção para o SPLC baseadas em escalonamentos ativos e sem atraso. A Seção 4.3 traz uma heurística baseada em programação linear. A Seção 4.4 detalha a estratégia seqüencial de busca tabu proposta para o SPLC. Finalmente, a Seção 4.5 analisa comparativamente os resultados obtidos com todas estas heurísticas.

4.1 Heurísticas Baseadas em Regras de Prioridade

Entre as heurísticas mais aplicadas a problemas de escalonamento estão aquelas que a cada passo selecionam um job para ser escalonado. Num procedimento conhecido por método de um passo ou guloso, uma vez que um job foi selecionado, nenhuma reconsideração desta escolha é permitida.

Uma heurística baseada em regra de prioridade é uma estratégia gulosa onde a seleção do próximo job a ser escalonado é determinada por uma regra de prioridade.

Um job é dito escalonável se todos os seus predecessores já foram escalonados. Na heurística

baseada em regras de prioridade implementada para o SPLC, foram utilizadas as seguintes regras de prioridade para escolher entre os jobs escalonáveis o próximo a ser escalonado:

- MINSLK (*Minimum Slack*): seleciona job que tem a menor folga, onde a folga de um job é dada pela diferença entre os instantes mais tarde e mais cedo em que ele pode começar considerando as restrições de precedência e o limite do horizonte de planejamento;
- LCP (*Longest Critical Path*): seleciona job cujo grafo induzido por ele e seus sucessores no grafo de precedência original tem o maior caminho crítico;
- LOD (*Longest Order Duration*): seleciona job que pertence ao pedido de maior duração total;
- MROD (*Most Remaining Order Duration*): seleciona job que pertence ao pedido com maior tempo de processamento remanescente;
- SPT (*Shortest Processing Time*): seleciona job com menor tempo de processamento;
- LRD (*Least Resource Demand*): seleciona job que necessita de menos mão-de-obra;
- RANDOM (*Random*): seleciona um job arbitrariamente.

A escolha destas sete regras foi guiada pelas observações, disponíveis em [DP75, Pat76], sobre a eficiência de cada uma delas para problemas gerais de escalonamento com restrição de recursos e minimização do *makespan*.

4.2 Heurísticas Baseadas em Classes de Escalonamento

Em teoria, existem infinitos escalonamentos viáveis para qualquer instância do SPLC: desde que preservadas as restrições de precedência e mão-de-obra, é sempre possível introduzir espaços ociosos entre dois quaisquer jobs adjacentes. Na prática, contudo, soluções com este tipo de ociosidade não são boas e claramente podem ser desconsideradas.

De maneira similar àquela proposta por Baker [Bak74] para o JSP (*Job Shop Scheduling Problem*), é possível restringir a busca de soluções de qualidade para o SPLC às seguintes classes de escalonamento:

- Escalonamentos ativos: escalonamentos nos quais nenhum job pode ser iniciado mais cedo sem atrasar o início de outro job ou sem violar a restrição de mão-de-obra;
- Escalonamentos sem atraso: escalonamentos nos quais nenhum trabalhador fica ocioso se ele pode começar a processar algum job.

A Figura 4.1 traz um escalonamento sem atraso e um ativo para a *instância exemplo* do SPLC (Seção 2.2), quando um total de 18 trabalhadores está disponível a cada instante. O escalonamento (a) tem *makespan* 10 e é ativo e sem atraso porque nenhum trabalhador fica ocioso se havia algum job esperando para ser processado. O escalonamento (b) tem *makespan* 9 e é ativo mas não sem atraso (note que no instante $t = 2$, trabalhadores ficaram ociosos quando poderiam ter começado a processar o primeiro job do pedido 3).

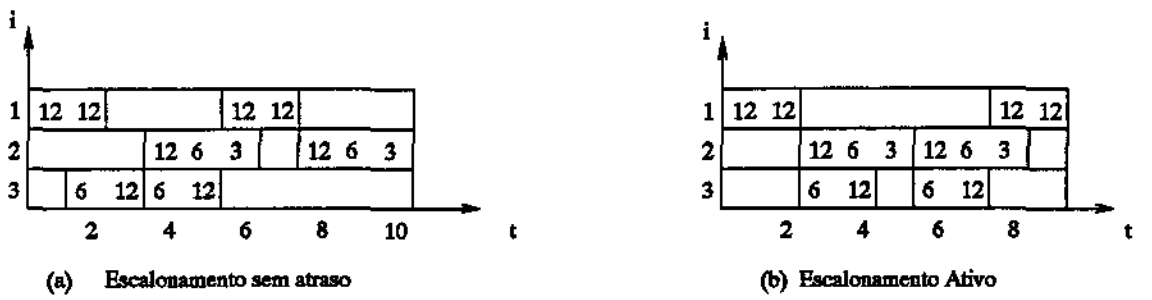


Figura 4.1: Exemplo de escalonamentos sem atraso e ativo para a *instância exemplo* do SPLC.

Escalonamentos ativos são o menor conjunto dominante de soluções para o SPLC, ou seja, pelo menos uma solução ótima pertence a esta classe. Escalonamentos sem atraso são um subconjunto dos escalonamentos ativos que, embora não dominante, pode conter soluções de boa qualidade para o SPLC.

Os diagramas de Venn da Figura 4.2 ilustram a relação entre os escalonamentos ativos e os sem atraso dentro do conjunto (infinito) de todos os escalonamentos viáveis para o SPLC. O símbolo $*$ representa um escalonamento ótimo. Note que em (a) o ótimo é um escalonamento sem atraso e em (b) ele é um escalonamento ativo mas não sem atraso.

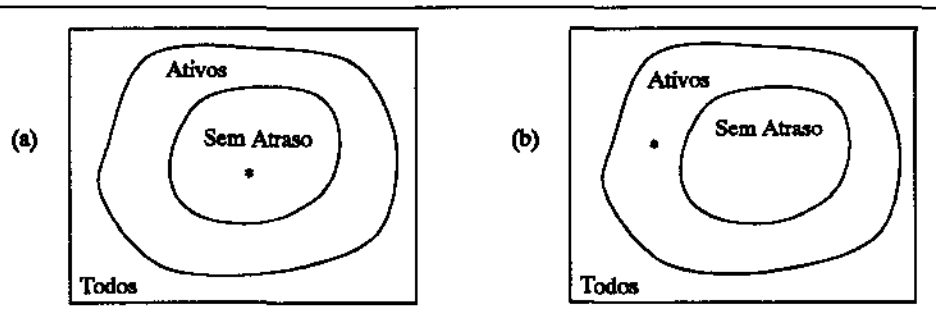


Figura 4.2: Relação entre escalonamentos ativos e sem atraso. O símbolo $*$ indica um escalonamento ótimo.

A seguinte notação será adotada na descrição dos algoritmos desta seção:

σ_j: instante mais cedo em que o job j pode começar considerando restrições de precedência e mão-de-obra;

ϕ_j: instante mais cedo em que o job j pode terminar ($\phi_j = \sigma_j + p_j$);

P_k: escalonamento parcial contento k jobs;

S_k: conjunto dos jobs escalonáveis no estágio k, ou seja, todos os jobs cujos predecessores estão em P_k.

Neste trabalho, foram implementadas três heurísticas seqüenciais baseadas em classes de escalonamento para o SPLC. Todas elas resultaram da adaptação dos dois algoritmos clássicos de Giffler e Thompson (1960) [Bak74], descritos a seguir, que sistematicamente geram todos os escalonamentos ativos e todos escalonamentos sem atraso.

Algoritmo GT-Ativo

1. Seja $k = 0$. Comece com $P_k = \{\}$ e
   $S_k = \{$ todos os jobs que não têm predecessores $\}$.
2. Calcule $\phi^* = \min\{\phi_j : j \in S_k\}$.
3. Para cada job $j' \in S_k$ tal que $\sigma_{j'} < \phi^*$, crie um novo escalonamento parcial P_{k+1} adicionando j' a P_k e começando j' em $\sigma_{j'}$.
4. Para cada escalonamento parcial P_{k+1} criado em (3), faça:
   (a) Remova j' de S_k;
   (b) Atualize σ_j para todo sucessor j de j';
   (c) Incremente k;
   (d) Atualize S_k para $\{$ todos os jobs não escalonados cujos predecessores estão em $P_k\}$;
   (e) Se $k < n$ volte para (2).

Algoritmo GT-Sem-Atraso

1. Seja $k = 0$. Comece com $P_k = \{\}$ e
   $S_k = \{$ todos os jobs que não têm predecessores $\}$.
2. Calcule $\sigma^* = \min\{\sigma_j : j \in S_k\}$.
3. Para cada job $j' \in S_k$ tal que $\sigma_{j'} = \sigma^*$, crie um novo escalonamento parcial P_{k+1} adicionando j' a P_k e começando j' em $\sigma_{j'}$.
4. Para cada escalonamento parcial P_{k+1} criado em (3), faça:
   (a) Remova j' de S_k;
   (b) Atualize σ_j para todo sucessor j de j';
   (c) Incremente k;
   (d) Atualize S_k para $\{$ todos os jobs não escalonados cujos predecessores estão em $P_k\}$;
   (e) Se $k < n$ volte para (2).

A diferença crucial entre estes dois algoritmos está nos passos (2) e (3). Dado que σ^* de GT-Sem-Atraso é menor que ϕ^* de GT-Ativo, o número de jobs escalonáveis j' satisfazendo $\sigma_{j'} = \sigma^*$ é no máximo igual ao número de jobs que satisfazem $\sigma_{j'} < \phi^*$. Desta forma, o conjunto de escalonamentos gerados por GT-Ativo é normalmente maior que o gerado por GT-Sem-Atraso, e não muito raramente, a solução ótima fica na diferença destes dois conjuntos. Por outro lado, o menor número de possíveis escalonamentos sem atraso é exatamente o que torna atrativa a exploração desta classe na busca de boas soluções.

Note que, claramente, os algoritmos GT-Ativo e GT-Sem-Atraso têm complexidade exponencial. Por conta disto, a sua utilização para obter soluções para o SPLC fica inviável. As Seções 4.2.1, 4.2.2 e 4.2.3, a seguir, descrevem como estes algoritmos foram modificados para a obtenção de soluções viáveis para o SPLC.

4.2.1 A Heurística SPLCA

Esta heurística constrói um escalonamento ativo para o SPLC. A idéia central é a mesma do algoritmo **GT-Ativo** de Giffler e Thompson. A diferença é que, em SPLCA, a cada estágio k, se existe mais de um job j' escalonável, satisfazendo $\sigma_{j'} < \phi^*$, uma das regras de prioridade da Seção 4.1 é usada para escolher um único job para ser escalonado. Conseqüentemente, apenas um novo escalonamento parcial P_{k+1} é criado a cada passo e a heurística termina com um único escalonamento ativo construído. O algorimo SPLCA completo é:

Algoritmo SPLCA

1. Faça $k = 0$, $P_k = \{\}$, $S_k = \{$ jobs que não têm predecessores $\}$
   Inicialize a disponibilidade de mão-de-obra: $labour[u] = L, \forall u = 1, \ldots, T$
2. Para todo $j \in S_k$ faça:
   $\sigma_j = \max\{e_j, \min\{t :$ para $u = t, \ldots, t + p_j - 1, \ell_{j,u-t+1} \leq labour[u]\}\}$
   $\phi_j = \sigma_j + p_j$
3. Calcule $\phi^* = \min\{\phi_j : j \in S_k\}$
4. Escolha $j' \in S_k$ tal que $\sigma_{j'} < \phi^*$. Em caso de impasse, resolva usando uma regra de prioridade. Caso persista um impasse, resolva aleatoriamente.
5. Crie um escalonamento parcial P_{k+1} adicionando j' a P_k e escalonando j' em $\sigma_{j'}$
6. (a) Para $u = \sigma_{j'}, \ldots, \sigma_{j'} + p_{j'} - 1$ faça $labour[u] -= \ell_{j',u-\sigma_{j'}}$.
   (b) Para todo sucessor j de j' faça: $e_j = \max\{e_j, \sigma_{j'} + p_{j'}\}$
   (c) Incremente k
   (d) Faça $S_k = \{$ todos os jobs não escalonados cujos predecessores estão em $P_k\}$;
7. Se $k < n$ volte para (2). Caso contrário, pare.

4.2.2 A Heurística SPLCSA

O algoritmo GT-Sem-Atraso de Giffler e Thompson é usado como base desta heurística que constrói um escalonamento sem atraso para o SPLC. De maneira similar a SPLCA (Seção 4.2.1), a cada passo um único escalonamento parcial é construído e regras de prioridade são usadas para resolver conflitos entre os jobs escalonáveis satisfazendo $\sigma_{j'} = \sigma^*$. O algoritmo completo é:

Algoritmo SPLCSA

1. Faça $k = 0$, $P_k = \{\}$, $S_k = \{$ jobs que não têm predecessores $\}$
   Inicialize a disponibilidade de mão-de-obra: $labour[u] = L, \forall u = 1, \ldots, T$
2. Para todo $j \in S_k$ faça:

$\sigma_j = \max\{e_j, \min\{t : \text{para } u = t, \ldots, t + p_j - 1, \ell_{j,u-t+1} \leq labour[u]\}\}$

3. Calcule $\sigma^* = \min\{\sigma_j : j \in S_k\}$
4. Escolha $j' \in S_k$ tal que $\sigma_{j'} = \sigma^*$. Em caso de impasse, resolva usando uma regra de prioridade. Caso persista um impasse, resolva aleatoriamente.
5. Crie um escalonamento parcial P_{k+1} adicionando j' a P_k e escalonando j' em $\sigma_{j'}$
6. (a) Para $u = \sigma_{j'}, \ldots, \sigma_{j'} + p_{j'} - 1$ faça $labour[u] - = \ell_{j',u-\sigma_{j'}}$.
   (b) Para todo sucessor j de j' faça: $e_j = \max\{e_j, \sigma_{j'} + p_{j'}\}$
   (c) Incremente k
   (d) Faça $S_k = \{$ todos os jobs não escalonados cujos predecessores estão em $P_k\}$;
7. Se $k < n$ volte para (2). Caso contrário, pare.

4.2.3 A Heurística SPLCH

Esta heurística combina os algoritmos SPLCA (Seção 4.2.1) e SPLCSA (Seção 4.2.2) fazendo uso de uma proposta híbrida apresentada por Storer *et. al.* em [SWV92]. A idéia desta proposta é a modificação dos algoritmos originais para geração de escalonamentos ativos e sem atraso pela adição de um parâmetro $\delta \in [0, 1]$ tal que: se $\delta = 0$, o algoritmo híbrido gera escalonamentos sem atraso; se $\delta = 1$, o algoritmo gera escalonamentos ativos; e se $0 < \delta < 1$, escalonamentos híbridos de sem atraso e ativos são gerados.

A motivação para uma abordagem híbrida para o SPLC é ter um espectro de soluções maior que o conjunto dos escalonamentos sem atraso sem, contudo, necessariamente considerar todo o espaço de escalonamentos ativos. Desta forma, a heurística SPLCH gera escalonamentos na fronteira dos conjuntos sem atraso e ativos na tentativa de encontrar soluções melhores que as produzidas pelos algoritmos "puros" SPLCA e SPLCSA.

O algoritmo SPLCH é descrito a seguir.

Algoritmo SPLCH

1. Faça $k = 0$, $P_k = \{\}$, $S_k = \{$ jobs que não têm predecessores $\}$
   Inicialize a disponibilidade de mão-de-obra: $labour[u] = L, \forall u = 1, \ldots, T$
2. Para todo $j \in S_k$ faça:
   $\sigma_j = \max\{e_j, \min\{t : \text{para } u = t, \ldots, t + p_j - 1, \ell_{j,u-t+1} \leq labour[u]\}\}$
   $\phi_j = \sigma_j + p_j$
3. Calcule $\sigma^* = \min\{\sigma_j : j \in S_k\}$ e
   $\phi^* = \min\{\phi_j : j \in S_k\}$
4. Escolha $j' \in S_k$ tal que $\sigma_{j'} \leq \sigma^* + \delta(\phi^* - \sigma^*)$, $\delta \in [0, 1]$. Em caso de impasse, resolva usando uma regra de prioridade. Caso persista um impasse, resolva aleatoriamente.
5. Crie um escalonamento parcial P_{k+1} adicionando j' a P_k e escalonando j' em $\sigma_{j'}$
6. (a) Para $u = \sigma_{j'}, \ldots, \sigma_{j'} + p_{j'} - 1$ faça $labour[u] - = \ell_{j',u-\sigma_{j'}}$.
   (b) Para todo sucessor j de j' faça: $e_j = \max\{e_j, \sigma_{j'} + p_{j'}\}$
   (c) Incremente k
   (d) Faça $S_k = \{$ todos os jobs não escalonados cujos predecessores estão em $P_k\}$;
7. Se $k < n$ volte para (2). Caso contrário, pare.

4.3 Heurística Baseada em Programação Linear - SPLCPL

Apesar dos resultados não muito bons do algoritmo de *branch-and-bound* dedicado ao SPLC (Seção 3.2) na obtenção de limitantes inferiores para este problema, pensou-se em utilizar algumas das idéias deste algoritmo para construir limitantes superiores para o SPLC de maneira similar à feita por Savelsbergh em [SUW97].

A heurística SPLCPL essencialmente é a heurística primal do *B&B* (Seção 3.2) aplicada à solução fracionária obtida com a relaxação linear da formulação x (Seção 3.1.1). A implementação foi feita com a biblioteca *Callable Library* do CPLEX [CPL95] e o pseudo-código está descrito a seguir:

Algoritmo SPLCPL

1. Carregue a formulação x.
2. Resolva o LP.
3. Obtenha ordem $P'_{best-\alpha}$ dos jobs a partir da solução fracionária usando a estratégia *Schedule-by-Best-α* (Seção 3.2).
4. Construa *dirSchedule* usando o algoritmo *BuildDirectSchedule* (Figura 3.3) com $P'_{best-\alpha}$ como parâmetro e a instância na ordem direta.
5. Construa *invSchedule* usando o algoritmo *BuildInverseSchedule* (Figura 3.4) com $P'_{best-\alpha}$ como parâmetro e a instância na ordem inversa.
6. Retorne o escalonamento de menor *makespan* entre *dirSchedule* e *invSchedule*.

4.4 Estratégia Seqüencial de Busca Tabu para o SPLC

Esta seção descreve a estratégia seqüencial de busca tabu proposta neste trabalho para o SPLC. A Seção 4.4.1, inicialmente, apresenta os aspectos genéricos da metaheurística busca tabu de acordo com [Glo89, Glo90, GL95]. Em seguida, a Seção 4.4.2 detalha como estes aspectos foram implementados para o SPLC.

4.4.1 A Metaheurística Busca Tabu

Metaheurísticas são heurísticas de caráter genérico, aplicáveis a uma grande variedade de problemas combinatórios e tipicamente centradas em métodos de busca em vizinhança [Ree95]. A metaheurística busca tabu (TS) é um processo empregado para guiar métodos de busca local de forma a evitar que eles fiquem presos em soluções ótimas locais. Para isso, TS faz uso de mecanismos que continuamente permitem a exploração do espaço de soluções, mesmo quando não existe nenhum movimento que melhore a solução corrente. Ao escapar de mínimos locais da função objetivo, a busca tabu pretende aumentar as chances de se encontrar uma solução ótima global.

Define-se a *vizinhança* $N(s)$ de uma solução s como o conjunto de soluções obtidas a partir de s pela aplicação de operações elementares denominadas *movimentos*. Cada movimento é caracterizado por um ou mais *atributos de movimento* que correspondem às mudanças feitas numa solução por ocasião da sua aplicação.

De maneira similar aos métodos de busca local, uma busca tabu parte de uma solução inicial e, até que algum critério de parada seja satisfeito, a cada iteração aplica um movimento à solução corrente s transformando-a em uma solução $s' \in N(s)$. Ao contrário dos métodos descendentes tradicionais, que só aceitam movimentos que resultam em soluções de custo melhor (menor), uma busca tabu permite também movimentos que levam a soluções de custo pior (maior). Além disso, três estruturas flexíveis de memória são usadas para manter informações sobre a trajetória da busca e controlar a exploração do espaço de soluções:

- *Memória de curto prazo*, que guarda uma história seletiva das soluções já examinadas pela busca e oferece mecanismos para estender ou reduzir a vizinhança de uma solução;
- *Memória intermediária*, que guarda as características de soluções historicamente detectadas como boas e estimula o uso destas informações em etapas de intensificação da busca;
- *Memória de longo prazo*, que permite identificar áreas do espaço de soluções ainda não exploradas e, em fases de diversificação, direciona a trajetória da busca para estas regiões.

Tipicamente, as informações mantidas pela memória de curto prazo dizem respeito a atributos de movimento que ocorreram no passado e estão intimamente relacionadas ao conceito de restrições tabu.

Restrições tabu são regras definidas pelos atributos de movimento de maneira a evitar, por um certo tempo, movimentos que revertam ou repitam as mudanças representadas por atributos de movimentos ocorridos anteriormente. Ao impedir a reversão de mudanças já realizadas, o objetivo é assegurar que a busca não entre em ciclo após visitar uma solução ótima local. Por outro lado, ao evitar a repetição de atributos de movimento já efetuados, restrições tabu direcionam a busca para soluções diferentes das já visitadas.

Uma restrição tabu é ativada quando os atributos de movimento a ela associados ocorreram dentro de um certo número de iterações anteriores (*restrição tabu baseada em recenticidade*) ou com uma certa freqüência ao longo de um conjunto de iterações anteriores (*restrição tabu baseada em freqüência*).

O tempo durante o qual uma restrição tabu permanece ativa é chamado *prazo tabu* e normalmente é expresso em número de iterações. Existem dois tipos de prazo tabu: *estático*, quando ele permanece fixo durante todo o processo de busca; e *dinâmico*, quando ele é calculado em função dos atributos de movimento que definem a restrição tabu ou simplesmente varia sistematica ou

aleatoriamente em um intervalo estabelecido. Experiências práticas indicam que prazos tabu dinâmicos são mais efetivos que os estáticos [GL95].

Uma restrição tabu ativa pode ser desconsiderada sempre que a execução dos atributos de movimento por ela proibidos levar a uma solução atrativa. A medida de quão atrativa deve ser uma solução para que, mesmo proibida por uma restrição tabu, ela possa ser visitada é determinada pelos chamados *critérios de aspiração.*

A exploração do espaço de soluções por uma busca tabu acontece da seguinte forma: seja s a solução corrente em um certo estágio da busca e seja H o conjunto de informações mantidas pelas memórias de curto prazo, intermediária e de longo prazo sobre as características das soluções visitadas até este momento. Com base nestes conhecimentos históricos, a vizinhança $N(s)$ é substituída por uma *vizinhança modificada* $N(H,s)$ e a próxima solução a ser visitada pela busca é escolhida deste novo conjunto de soluções.

Em estratégias de busca tabu que usam apenas memória de curto prazo, $N(H,s) \subseteq N(s)$ e restrições tabu servem exatamente para identificar as soluções em $N(s)$ que devem ser excluídas de $N(H,s)$. Em estratégias que fazem uso das memórias intermediárias e de longo prazo, por outro lado, $N(H,s)$ pode conter soluções que não estão em $N(s)$. Por exemplo, numa etapa de diversificação, $N(H,s)$ pode conter soluções de boa qualidade encontradas em vários pontos do processo de busca.

Para problemas de grande porte, onde a vizinhança modificada de uma solução é muito grande ou onde a análise de todos os elementos em $N(H,s)$ é computacionalmente cara, é necessário isolar um subconjunto de $N(H,s)$ chamado de *soluções candidatas* ($CS(s)$). Este novo conjunto representa as soluções vizinhas de s que serão efetivamente consideradas pelo processo de busca. A definição de $CS(s)$ tem um impacto significante na qualidade das soluções visitadas e, portanto, boas estratégias devem ser usadas para escolher as soluções integrantes deste conjunto. Na prática, muitas vezes, ao invés de se construir $CS(s)$, opta-se simplesmente por guardar uma lista com todos os movimentos que, uma vez aplicados a s, levam às soluções deste conjunto. Esta lista é chamada de *lista de movimentos candidatos.*

A *função de custo* a ser otimizada pode incorporar penalidades ou premiações dependendo do estado em que se encontra a busca. No caso de uma etapa de diversificação pode-se, por exemplo, penalizar soluções com características que ocorreram com uma freqüência alta durante a busca e premiar soluções com características que raramente foram encontradas. Numa abordagem conhecida como *oscilação estratégica,* fronteiras de viabilidade podem ser cruzadas e a busca é sistematicamente guiada para regiões de soluções viáveis e inviáveis.

Critérios de parada são utilizados para interromper o processo de busca. Entre os mais comuns estão: limite no número máximo de iterações e limite no número máximo de iterações sem melhoria na função de custo.

O pseudo-código do procedimento de busca tabu genérico está descrito a seguir. A essência do método está na maneira como é definido e usado o conjunto H dos conhecimentos adquiridos ao longo do processo. A determinação do conjunto de soluções candidatas é outro fator importante e merece atenção especial.

Procedimento Busca Tabu

1. Selecione uma solução inicial s^0.
2. Faça $c^* = c(s^0)$; $s^* = s^0$; $s = s^0$.
3. Inicialize o conjunto H de informações mantidas pelas memórias de curto prazo, intermediária e de longo prazo: $H = \emptyset$
4. Enquanto o *critério de parada* não for satisfeito faça
   a. Determine o conjunto de *soluções candidatas* $CS(s)$ como subconjunto da *vizinhança modificada* $N(H, s)$.
   b. Selecione $s' \in CS(s)$ tal que $c(s')$ é mínimo neste conjunto.
   c. Se $c(s') < c^*$
      Então $s^* = s'$; $c^* = c(s')$.
   d. $s = s'$.
   e. Atualize o conjunto H com os conhecimentos adquiridos nesta iteração.
5. Retorne s^*.

A metaheurística busca tabu tem sido amplamente aplicada [Aie96, Glo89, Glo90, Lag95, LK96, PR95, ZP96] e os resultados confirmam o potencial desta estratégia na obtenção de soluções de qualidade para problemas combinatórios NP-difíceis. A próxima seção descreve os detalhes da estratégia de busca tabu proposta e implementada para o SPLC.

4.4.2 O Algoritmo TSSPLC

Na estratégia seqüencial de busca tabu proposta para o SPLC, uma solução é representada por uma seqüência s de jobs.

O algoritmo começa com um escalonamento inicial construído por alguma das heurísticas das Seções 4.1, 4.2 e 4.3. Em seguida, a cada iteração, uma (a melhor) solução da lista de soluções candidatas é escolhida. As soluções desta lista são resultantes de operações *Insert* ou *Swap*, descritas mais adiante, aplicadas a pares de jobs da solução corrente de maneira a não violar as restrições de precedência. Uma memória de curto prazo é usada para guardar os movimentos mais recentes que levaram a soluções de pior custo. Restrições tabu podem ser ignoradas sempre que a execução do movimento a elas associado resultar numa solução de custo melhor do que a melhor conhecida até o momento. O novo escalonamento obtido ao final de cada iteração passa a ser o escalonamento corrente. A busca termina quando um dos critérios de parada é atingido. A melhor solução obtida ao longo do processo é retornada. Estratégias de intensificação e diversificação não são exploradas.

A seguir são discutidos os principais elementos do algoritmo TSSPLC.

Solução Inicial

Três estratégias foram adotadas para escolha do escalonamento inicial para o algoritmo TSSPLC:

- *Best_PRH*: melhor solução obtida com a heurística baseada em regras de prioridade da Seção 4.1;
- *Best_SSH*: melhor solução obtida com alguma das heurísticas SPLCSA, SPLCA e SPLCH da Seção 4.2;
- *Sol_PLH*: solução obtida com a heurística baseada em programação linear da Seção 4.3.

A seqüência s representativa do escalonamento inicial é obtida da seguinte forma: seja n o número total de jobs a serem escalonados e seja $\sigma_{s[k]}$ o instante de início de processamento do job $s[k]$ no escalonamento representado por s. A seqüência s é construída para satisfazer $\sigma_{s[i]} \leq \sigma_{s[j]}, \forall i < j, i, j \in \{1, \ldots, n\}$.

Estratégia de Vizinhança

Estudos anteriores mostram que a definição da vizinhança tem um impacto significativo na qualidade das soluções obtidas e é um dos aspectos cruciais de uma busca tabu [Lag95].

A dificuldade em encontrar uma boa vizinhança reside em determinar uma estratégia simples e rápida de geração de movimentos que dê ao processo de busca a chance de visitar soluções diferentes e de alta qualidade.

No algoritmo TSSPLC, foram adotados os dois tipos de movimento mais usados em problemas de escalonamento. Ambos fazem uso da representação de uma solução como uma seqüência e são descritos como segue:

- *Insert(i, j)*: Insere o job j imediatamente na frente do job i na seqüência (Figura 4.3 (a));
- *Swap(i, j)*: Troca as posições dos jobs i e j na seqüência (Figura 4.3 (b)).

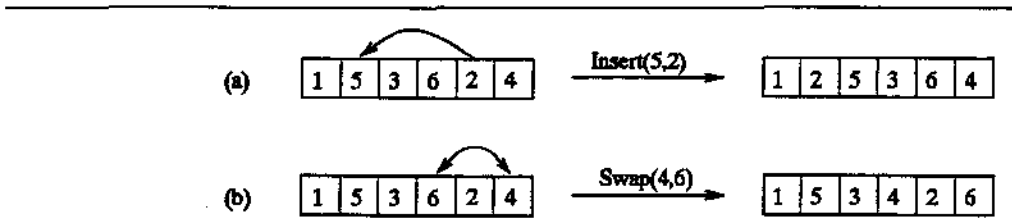


Figura 4.3: Exemplo dos movimentos *Insert* e *Swap*.

Assim, dada uma solução s, uma solução s' vizinha de s é obtida pela aplicação de um movimento *Insert* ou *Swap* a algum par de jobs (i, j) de s. O conjunto de todas as soluções vizinhas de s define a sua vizinhança $N(s)$.

Uma seqüência s', representativa de uma solução, é dita viável se todo job em s' aparece à direita de todos os seus predecessores.

A vizinhança reduzida $RN(s)$ de uma solução s é definida por $RN(s) = \{s' \in N(s) : s' \text{ é viável }\}$. São, portanto, desconsiderados todos os movimentos *Insert* ou *Swap* que resultam em seqüências inviáveis. Uma análise das relações de precedência entre os jobs permite identificar rapidamente estes movimentos.

Seja PPL_j uma lista de todos os jobs que não guardam relações de precedência com o job j. Dada uma seqüência viável s, uma seqüência inviável s' só é obtida nos seguintes casos:

(i) Um movimento *Insert(i, j)* é aplicado a s, sendo que $j \notin PPL_i$ e j não é predecessor direto de i;

(ii) Um movimento *Swap(i, j)* é aplicado a s, sendo que $j \notin PPL_i$.

Por razões de eficiência, portanto, sempre que o algoritmo TSSPLC precisa obter alguma solução s' na vizinhança de uma solução s, apenas os seguintes movimentos são considerados:

(iii) *Insert(i, j)* onde $j \in PPL_i$ ou j é predecessor direto de i;

(iv) *Swap(i, j)* onde $j \in PPL_i$.

Note, contudo, que as condições (iii) e (iv) sozinhas não são suficientes para garantir que o movimento correspondente leva a uma seqüência viável s'. Isto depende da posição dos predecessores e sucessores dos jobs i e j na seqüência s e só pode ser verificado em tempo de execução. Para isso, foram implementadas duas funções auxiliares, descritas a seguir, que verificam, dada uma seqüência viável s, se um movimento *Insert* satisfazendo (iii) e um movimento *Swap* satisfazendo (iv) realmente levam a uma seqüência viável. Claramente, a complexidade destas funções é $O(n)$ onde n é o número total de jobs.

Função *CheckFeasibilityOfInsert*
Entrada: seqüência s e jobs i, j.
Saída: *viável* se *Insert(i, j)* aplicado a s resultar em s' viável; *inviável* c.c.

```
/* Inicializa vetor posInS com a posição de cada job na seqüência s */
1. Para k = 1, ..., n faça
       posInS[s[k]] = k;
2. Se posInS[i] < posInS[j]
       Então Para todo predecessor k de j faça
                 Se posInS[k] > posInS[i]
                    Então retorne inviável;
             Retorne viável;
3. Se posInS[i] > posInS[j]
```

Então Para todo sucessor k de j faça
Se $posInS[k] < posInS[i]$
Então retorne *inviável*;
Retorne *viável*;

Função *CheckFeasibilityOfSwap*
Entrada: seqüência s e jobs i, j.
Saída: *viável* se *Swap(i, j)* aplicado a s resultar em s' viável; *inviável* c.c.

/* Inicializa vetor com a posição de cada job na seqüência s */
1. Para $k = 1, \ldots, n$ faça
$posInS[s[k]] = k$;
2. Para todo sucessor k de i faça
Se $posInS[k] < posInS[j]$
Então retorne *inviável*;
3. Para todo predecessor k de j faça
Se $posInS[k] > posInS[i]$
Então retorne *inviável*;
4. Retorne *viável*;

Mesmo tendo um número menor de soluções do que $N(s)$, a vizinhança reduzida $RN(s)$ ainda tem tamanho $O(n^2)$. Portanto, analisar todas as soluções deste conjunto é computacionalmente caro. Por conta disso, quatro alternativas foram propostas para serem usadas como soluções candidatas a partir de uma solução s:

- $CS_1(s)$: subconjunto de $RN(s)$ formado por 10 soluções cada qual obtida a partir de s pela aplicação do movimento *Insert* a um par de jobs (i, j) satisfazendo: i é escolhido aleatoriamente; j é escolhido aleatoriamente entre os jobs de PPL_i e os predecessores direto de i; *CheckFeasibilityOfInsert(s, i, j)* retorna *viável*.
- $CS_2(s)$: subconjunto de $RN(s)$ formado por 10 soluções cada qual obtida a partir de s pela aplicação do movimento *Swap* a um par de jobs (i, j) satisfazendo: i é escolhido aleatoriamente; j é escolhido aleatoriamente entre os jobs de PPL_i; *CheckFeasibilityOfSwap(s, i, j)* retorna *viável*.
- $CS_3(s)$: subconjunto de $RN(s)$ formado pela aplicação do movimento *Insert* a todos os pares (i, j) para um único j escolhido aleatoriamente e todos os i's satisfazendo: $i \in PPL_j$ ou i é sucessor direto de j; *CheckFeasibilityOfInsert(s, i, j)* retorna *viável*.
- $CS_4(s)$: subconjunto de $RN(s)$ formado pela aplicação do movimento *Insert* a todos os pares (i, j) para um único i escolhido aleatoriamente e todos os j's satisfazendo: $j \in PPL_i$ ou j é predecessor direto de i; *CheckFeasibilityOfInsert(s, i, j)* retorna *viável*.

O algoritmo TSSPLC foi testado com cada um dos quatro conjuntos de soluções candidatas definidos acima. A cada iteração a solução de menor custo entre as candidatas resultantes de

movimentos não tabu é escolhida. O único caso onde uma solução candidata resultante de um movimento tabu pode ser escolhida é quando o critério de aspiração, definido mais adiante, é atingido.

Memória de Curto Prazo e Restrições Tabu

A estratégia de busca tabu proposta para o SPLC faz uso de uma memória de curto prazo e adota restrições tabu baseadas em recenticidade. Assim, são designados tabu-ativo todos os atributos de movimento que foram recentemente executados e resultaram em um aumento na função de custo. Movimentos que contêm atributos tabu-ativos são considerados tabu e não podem ser executados a não ser que o critério de aspiração seja satisfeito.

Os seguintes atributos de movimento foram definidos para os movimentos *Insert* e *Swap* usados no algoritmo TSSPLC:

- A_1: o job j foi movido;
- A_2: o job j foi inserido imediatamente na frente do job i na seqüência;
- A_3: os jobs i, j trocaram de posição.

Associadas com estes atributos, são definidas três restrições tabu de tal forma que no algoritmo TSSPLC um movimento é tabu se:

- R_1: o job j for movido sendo A_1 tabu-ativo;
- R_2: o job j for inserido imediatamente na frente do job i na seqüência sendo A_2 tabu-ativo;
- R_3: os jobs i e j trocarem de posição sendo A_3 tabu-ativo.

Note que a restrição tabu R_1 é mais forte (restritiva) do que as outras duas. Isto acontece porque R_1 classifica como tabu todos os movimentos do job j. R_2 e R_3, por outro lado, proibem apenas os movimentos de j onde i também está envolvido.

Prazo Tabu

O prazo tabu especifica o número de iterações durante as quais um atributo de movimento fica tabu-ativo e, conseqüentemente, a restrição tabu a ele associada fica ativa. No algoritmo TSSPLC foram definidos, empiricamente, três prazos tabu dinâmicos, um para cada uma das restrições tabu:

- T_1: inteiro aleatoriamente escolhido entre $\lfloor 0.5\sqrt{n} \rfloor$ e $\lceil 0.8\sqrt{n} \rceil$;
- T_2: inteiro aleatoriamente escolhido entre $\lfloor 1.2\sqrt{n} \rfloor$ e $\lceil 1.5\sqrt{n} \rceil$;

- T_3: inteiro aleatoriamente escolhido entre $\lfloor 0.9\sqrt{n} \rfloor$ e $\lceil 1.1\sqrt{n} \rceil$;

T_1 é menor que T_2 e T_3 coerentemente com o fato de R_1 ser mais restritiva que R_2 e R_3. Restrições tabu mais fortes devem permanecer ativas por um número menor de iterações [Glo89].

Critério de Aspiração

O critério de aspiração adotado no algoritmo TSSPLC permite que uma restrição tabu seja relaxada sempre que o movimento correspondente levar a uma solução melhor do que a melhor solução obtida pela busca até o momento.

Função de Custo

O objetivo do SPLC é encontrar uma solução de mínimo *makespan*. Neste sentido, duas alternativas de função de custo a ser minimizada foram propostas para o algoritmo TSSPLC:

- $F_1(s) = makespan$
- $F_2(s)$ = função objetivo modificada da Seção 3.2.

Para cada job j, seja σ_j o instante de início do seu processamento; p_j a sua duração e α_j o tamanho do caminho crítico do subgrafo induzido em G por j e todos os seus sucessores. F_1 e F_2 são dadas por:

- $F_1(s) = \max\{\sigma_j + p_j : j = 1, \ldots, n\}$
- $F_2(s) = \sum_{j=1}^{n} \alpha_j \sigma_j + (F_1(s) + 1) \sum_{j \in N} T\alpha_j$, onde T é o limite do horizonte de planejamento.

Em qualquer um dos casos, dada uma seqüência viável s, um escalonamento é obtido pegando-se o escalonamento de menor *makespan* entre os dois contruídos com as funções *BuildDirectSchedule* (Figura 3.3) e *BuildInverseSchedule* (Figura 3.4) com s como parâmetro.

Critério de Parada

O algoritmo TSSPLC pára quando uma das seguintes situações acontece:

(i) o número máximo de MAX_TOTAL_ITER iterações é atingido;

(ii) o número máximo de MAX_BAD_MOVES iterações sem melhoria na função de custo é atingido.

4.4.3 Configurações do TSSPLC

Dadas as diferentes alternativas de soluções candidatas, restrições tabu e prazos tabu, oito configurações foram definidas para o algoritmo TSSPLC, como mostra a Tabela 4.1. Cada uma destas configurações foi testada com as duas funções de custo (F_1 e F_2) e com as três possíveis soluções iniciais (*Best_PRH*, *Best_SSH* e *Sol_PLH*).

Configuração	Soluções Candidatas	Restrição Tabu	Prazo Tabu
$TSSPLC_1$	CS_1	R_1	T_1
$TSSPLC_2$	CS_1	R_2	T_2
$TSSPLC_3$	CS_2	R_1 aplicada aos jobs i e j	T_1
$TSSPLC_4$	CS_2	R_3	T_3
$TSSPLC_5$	CS_3	R_1	$T1$
$TSSPLC_6$	CS_3	R_2	T_2
$TSSPLC_7$	CS_4	R_1	T_1
$TSSPLC_8$	CS_4	R_2	T_2

Tabela 4.1: Configurações do algoritmo TSSPLC.

O pseudo-código do algoritmo TSSPLC está descrito na Figura 4.4.

A função *FinalTuning* (Figura 4.5) é uma função de ajuste fino chamada ao final do algoritmo TSSPLC para tentar reduzir ainda mais o *makespan* da melhor solução s^* obtida bela busca tabu. Basicamente, o que ela faz é tentar, por um certo número de iterações (MAX_ITER), encontrar para cada um dos jobs da instância do SPLC a sua melhor posição na seqüência s^*. Os jobs são examinados na ordem $1, \ldots, n$. Sempre que a mudança de posição de um job resultar numa seqüência s' de melhor *makespan*, esta nova seqüência é adotada como seqüência corrente.

4.5 Resultados Computacionais

As heurísticas seqüenciais para o SPLC descritas neste capítulo foram implementadas na linguagem C++. Os códigos foram compilados usando o compilador g++ da GNU. Todos os resultados apresentados a seguir foram obtidos em uma máquina SUNW,SPARC 1000, com 300MB de memória e oito processadores. Os resultados apresentados foram obtidos considerando $L = 18$ trabalhadores disponíveis a cada instante.

A Tabela 4.2 traz os resultados obtidos com a heurística baseada em regras de prioridade da Seção 4.1. Conforme descrito naquela seção, foram sete as regras testadas para o SPLC. Cada regra foi aplicada à instância na ordem direta e na ordem inversa e apenas o menor *makespan* entre os dois obtidos é mostrado. O símbolo ' é usado para indicar que o valor correspondente

Algoritmo TSSPLC

1. Gere solução inicial s^0.
2. Inicialize a memória de curto prazo *movimentos_tabu*.
3. Faça $s = s^0$; $s^* = s^0$; $c^* = c(s^0)$.
4. Faça $nIter = 0$; $nBadMoves = 0$.
5. Enquanto (($nIter <$ MAX_TOTAL_ITER) e ($nBadMoves <$ MAX_BAD_MOVES)) faça
   a. Determine soluções candidatas $CS(s)$.
   b. $c' = \infty$
   c. Para toda solução $\overline{s} \in CS(s)$ faça
      Se (($\overline{s}$ não foi obtida por um movimento tabu) ou ($c(\overline{s}) < c^*$))
      Então Se ($c(\overline{s}) < c'$)
      Então $c' = c(\overline{s})$ e $s' = \overline{s}$.
   d. Se ($c' > c(s)$)
      Então atualize *movimentos_tabu*.
   e. Se ($c' < c^*$)
      Então $nBadMoves = 0$;
      $s^* = s'$; $c^* = c'$.
      Senão $nBadMoves++$.
   f. $s = s'$.
   g. $nIter++$.
6. $s^* = FinalTuning(s^*)$
7. Retorne s^*

Figura 4.4: Algoritmo de busca tabu para o SPLC.

Procedimento *FinalTuning*
Entrada: seqüência viável s
Saída: seqüência viável s' tal que $makespan(s') \leq makespan(s)$

1. $s' = s$; $iter = 1$; $improv =$ TRUE;
2. Enquanto ($iter \leq$ MAX_ITER) e ($improv ==$ TRUE) faça
   $improv =$ FALSE;
   Para $j = 1$ até n faça
   Para todo $i \in PPL_j$ ou i sucessor direto de j faça
   Se ($CheckFeasibilityOfInsert(s', i, j) == viável$)
   Então seja $\overline{s}$ a solução obtida pela aplicação do movimento $Insert(i, j)$ a s'.
   Se ($makespan(\overline{s}) < makespan(s')$)
   Então $s' = \overline{s}$;
   $improv =$ TRUE.
3. Retorne s'.

Figura 4.5: Função de ajuste fino do *makespan* da melhor solução obtida pelo algoritmo TSSPLC.

foi obtido com a instância na ordem inversa. No caso da regra de prioridade RANDOM os três valores mostrados correspondem, respectivamente, à melhor, à mediana e à pior solução obtida com cinco execuções na ordem direta e cinco na ordem inversa. O tempo de CPU gasto em cada execução da heurística baseada em regras de prioridade foi menor que 5s para as instâncias com menos de 80 jobs e menor que 10s para as demais instâncias.

Analisando os resultados da Tabela 4.2, percebe-se que a regra LCP foi a que teve melhor desempenho entre as sete regras testadas. Em catorze das vinte e cinco instâncias, foi esta regra que chegou nas soluções de menor *makespan*. Por outro lado, as regras SPT e LRD mostraram-se as de pior qualidade, pois em oito e onze das vinte e cinco instâncias, respectivamente, elas foram as responsáveis pelas soluções de maior *makespan*.

Note, ainda, que, com exceção da regra MINSLK que em nenhuma das instâncias chegou na solução de menor *makespan*, nenhuma das outras seis regras domina as demais na qualidade das soluções obtidas. Esta não dominância das regras de prioridade já era um fato conhecido e esperado, conforme observado por Davis Patterson em [DP75].

As Tabelas 4.3, 4.4 e 4.5 apresentam os resultados obtidos com as heurísticas baseadas em classes de escalonamento da Seção 4.2. Cada uma das heurísticas SPLCSA, SPLCH e SPLCA foi testada com seis regras de prioridade e aplicada à instância na ordem direta e na ordem inversa. Apenas o melhor resultado obtido com cada regra é mostrado. Os resultados da heurística SPLCH na Tabela 4.4 são também os melhores resultados obtidos quando o parâmetro δ deste algoritmo foi atribuído aos valores $0.1, 0.2, \ldots, 0.9$. O tempo de CPU gasto em cada execução das heurísticas baseadas em classes de escalonamento foi menor que 1s para as instâncias com menos de 100

Instância	LOD	MROD	LCP	SPT	LRD	MINSLK	RANDOM		
Ins_4o_21j_A	87'	**82'**	87'	87	87'	87'	**82**	87	92
Ins_4o_23j_A	74	63'	64	68'	72'	64	**60**	65	68
Ins_4o_24j_A	**71'**	75'	72'	**71'**	**71'**	78	75	76	80
Ins_4o_24j_B	82	80'	82	81'	84	81	**77**	79	80
Ins_4o_27j_A	86'	80	**69**	85	81	82	74	76	82
Ins_6o_41j_A	179	**158'**	**158**	189	191	169	160	169	170
Ins_6o_41j_B	143	133'	**122**	151	139	146	126	132	137
Ins_6o_41j_C	156	148'	142	155	159	153	**138'**	146'	148'
Ins_6o_44j_A	133'	131'	127	127	135	135	**124**	129	130
Ins_6o_44j_B	171'	166'	**155'**	181'	181'	168	162	166	171
Ins_8o_63j_A	336'	303	**296**	324	311	315	303	305	312
Ins_8o_63j_B	**375**	384	383	406'	403'	401	377	379	389
Ins_8o_63j_C	373	363'	**334**	408	402	383'	351'	355'	361'
Ins_8o_65j_A	491'	446'	**425**	485	483	471	450'	460'	463'
Ins_8o_65j_B	463'	468	453	474'	513	470	**438'**	451'	472'
Ins_10o_84j_A	871'	743	731	861	884	767	**712**	741	757
Ins_10o_84j_B	736	658	**643**	740	748	677	649	655	668
Ins_10o_85j_A	1165'	1004	**965**	1142'	1204'	1060	989'	995'	1034'
Ins_10o_87j_A	751'	659	**656**	749	753'	680	659'	679'	688'
Ins_10o_88j_A	569	548	532	551	543'	556	**524'**	546'	556'
Ins_10o_100j_A	1924	1651'	1655'	1964'	1884'	1699	**1582'**	1632'	1664'
Ins_10o_102j_A	1461'	1394	**1321'**	1469	1456	1397	1348	1356	1426
Ins_10o_106j_A	1384'	1271'	**1215'**	1399	1371	1352	1249	1271	1291
Ins_12o_108j_A	1607	1466'	**1435**	1648'	1611'	1569	1459'	1474'	1486'
Ins_12o_109j_A	1726	1554	**1522**	1741	1746'	1638	1580	1599	1613

Tabela 4.2: Resultados da heurística baseada em regras de prioridade. Os três valores da coluna RANDOM são, respectivamente, a melhor, a mediana e a pior solução obtidas em cinco execuções. O símbolo ' após um valor indica que a solução foi obtida com a instância na ordem inversa.

Instância	LOD	MROD	LCP	SPT	LRD	MINSLK
Ins_4o_21j_A	**82'**	**82'**	**82'**	87	**82'**	**82'**
Ins_4o_23j_A	**58'**	**58'**	**58'**	**58'**	**58'**	**58'**
Ins_4o_24j_A	76'	74'	**71'**	76	78	75
Ins_4o_24j_B	78	78	**73**	79'	78	**73**
Ins_4o_27j_A	71	71	**69**	70	70	**69**
Ins_6o_41j_A	154'	**152'**	155'	153	168	164'
Ins_6o_41j_B	123	123	**116'**	119	119	120'
Ins_6o_41j_C	147'	144'	140'	**137'**	145'	152
Ins_6o_44j_A	125'	126'	124'	**123**	124	128
Ins_6o_44j_B	153	152	**151**	156	153	**151**
Ins_8o_63j_A	297	289	**283**	307	300	**283**
Ins_8o_63j_B	**350**	360	353	358	358	353
Ins_8o_63j_C	340	340	**325**	329	341	325
Ins_8o_65j_A	430'	435	437	**426**	434	437
Ins_8o_65j_B	459'	450	**419**	468'	490'	**419**
Ins_10o_84j_A	817'	**722**	732	803'	727	732
Ins_10o_84j_B	646'	631'	628'	639	**620'**	639'
Ins_10o_85j_A	968	**901**	912	927	916	912
Ins_10o_87j_A	652'	**618'**	631'	655'	653'	648
Ins_10o_88j_A	509'	513'	507'	525'	519'	**502'**
Ins_10o_100j_A	1608'	1593'	**1592**	1636	1655	**1592**
Ins_10o_102j_A	1287'	1302'	**1239'**	1345	1351	1271
Ins_10o_106j_A	1250	1203'	**1191'**	1220	1234	1232
Ins_12o_108j_A	1442	**1392'**	1396'	1430	1438'	1418
Ins_12o_109j_A	1514'	1462'	**1440'**	1542'	1531	1556'

Tabela 4.3: Resultados da heurística SPLCSA. O símbolo ' após um valor indica que a solução foi obtida com a instância na ordem inversa.

jobs e menor que 5s para as demais instâncias.

Da mesma forma que na heurística baseada em regras de prioridade, as melhores soluções de cada uma das heurísticas SPLCSA, SPLCH e SPLCA foram, em sua maioria, obtidas com a regra LCP. O importante de se observar com relação às melhores soluções obtidas por estas heurísticas baseadas em classes de escalonamento é a clara superioridade da heurística SPLCH. Em dezoito das vinte e cinco instâncias de teste, a melhor solução obtida por esta heurística tem *makespan* estritamente menor do que o das correspondentes melhores soluções obtidas por SPLCSA e SPLCA. Nas sete instâncias restantes, as melhores soluções de SPLCH têm qualidade igual a das melhores obtidas por SPLCSA e SPLCA.

Os resultados obtidos com a heurística baseada em programação linear, SPLCPL, da Seção

Instância	LOD	MROD	LCP	SPT	LRD	MINSLK
Ins_4o_21j_A	**82'**	**82'**	**82'**	**82**	**82'**	**82'**
Ins_4o_23j_A	64	**58'**	**58'**	**58'**	**58**	**58**
Ins_4o_24j_A	71'	71'	70'	**69'**	71'	72
Ins_4o_24j_B	78	74	**73**	79'	76	**73**
Ins_4o_27j_A	76	71	**69**	81'	73	72
Ins_6o_41j_A	162'	151'	**150'**	156	157	158
Ins_6o_41j_B	129	120	**115**	128	122	**115**
Ins_6o_41j_C	144	**136**	139	139'	139'	137
Ins_6o_44j_A	126	125	126'	128	126	**123**
Ins_6o_44j_B	161'	148	**146**	162	158	**146**
Ins_8o_63j_A	292	290'	**283**	312'	301	284
Ins_8o_63j_B	364	361	360	366'	**347'**	372
Ins_8o_63j_C	344	334	326	340	345	**325**
Ins_8o_65j_A	433'	428'	**421**	426	438	**421**
Ins_8o_65j_B	447'	437'	410	474'	460	**408**
Ins_10o_84j_A	775'	714	**708'**	794'	748	712
Ins_10o_84j_B	638'	609'	**606'**	632	622'	614
Ins_10o_85j_A	973'	917	**879**	938	927	903
Ins_10o_87j_A	665	**615'**	624	680	663'	631
Ins_10o_88j_A	528'	513'	**493**	522'	509	507
Ins_10o_100j_A	1553'	1535'	**1519**	1593	1614	1544
Ins_10o_102j_A	1237'	1308'	**1226'**	1389	1368	1292'
Ins_10o_106j_A	1249'	1190	**1163'**	1244	1214	1169
Ins_12o_108j_A	1477'	1398	1365	1394	1433	**1341**
Ins_12o_109j_A	1599	1497'	**1416'**	1544'	1604'	1492

Tabela 4.4: Resultados da heurística SPLCH. O símbolo ' após um valor indica que a solução foi obtida com a instância na ordem inversa.

Instância	LOD	MROD	LCP	SPT	LRD	MINSLK
Ins_4o_21j_A	87'	**82'**	87'	87	87'	87'
Ins_4o_23j_A	74	**63'**	64	69'	72'	68
Ins_4o_24j_A	71'	74	72'	**71'**	**71'**	80
Ins_4o_24j_B	82	**79'**	**79**	81'	82	84
Ins_4o_27j_A	85'	74'	**69**	85	78	77
Ins_6o_41j_A	173	**152'**	158	189	191	165
Ins_6o_41j_B	145	127'	**124'**	140'	139	131
Ins_6o_41j_C	159	146	**144'**	155	155	152
Ins_6o_44j_A	133'	128'	127	**133**	135	**125**
Ins_6o_44j_B	165'	161	154	162	178'	**153**
Ins_8o_63j_A	324'	302	**299**	321	320	300
Ins_8o_63j_B	**377**	368	379	406'	403'	379
Ins_8o_63j_C	369'	353'	**332**	399	395	340
Ins_8o_65j_A	466'	439	**426**	480	483	440
Ins_8o_65j_B	447'	454'	**432**	474'	513	444
Ins_10o_84j_A	832'	**732**	734'	869	868	734
Ins_10o_84j_B	668'	642	**638**	714	684	666
Ins_10o_85j_A	1130'	1000	**967'**	1142'	1204'	1058
Ins_10o_87j_A	756'	**661'**	**661**	743'	753'	685
Ins_10o_88j_A	560	555	**523**	540	539'	537
Ins_10o_100j_A	1874'	**1621**	1638'	1964'	1884'	1709
Ins_10o_102j_A	1373'	1377	**1279'**	1454	1456	1404'
Ins_10o_106j_A	1379'	1253'	**1219'**	1400	1365	1241
Ins_12o_108j_A	1579	1473'	**1455**	1648'	1611'	1524
Ins_12o_109j_A	1721	1559'	**1493**	1737'	1711'	1569

Tabela 4.5: Resultados da heurística SPLCA. O símbolo ' após um valor indica que a solução foi obtida com a instância na ordem inversa.

Instância	*Makespan*	α	Segundos
Ins_4o_21j_A	87	0.01	< 1
Ins_4o_23j_A	63	0.01	< 1
Ins_4o_24j_A	73	0.27	1
Ins_4o_24j_B	77	0.97	< 1
Ins_4o_27j_A	73	0.03	1
Ins_6o_41j_A	152	0.03	9
Ins_6o_41j_B	123	0.02	4
Ins_6o_41j_C	140	0.28	12
Ins_6o_44j_A	124	0.01	7
Ins_6o_44j_B	152	0.32	6
Ins_8o_63j_A	299	0.07	38
Ins_8o_63j_B	369	0.59	137
Ins_8o_63j_C	328	0.01	63
Ins_8o_65j_A	425	0.36	87
Ins_8o_65j_B	466	0.28	227
Ins_10o_84j_A	719	0.41	1623
Ins_10o_84j_B	639	0.34	1255
Ins_10o_85j_A	964	0.37	458
Ins_10o_87j_A	646	0.30	1861
Ins_10o_88j_A	519	0.17	291

Tabela 4.6: Resultados da heurística baseada em programação linear.

4.3 aparecem na Tabela 4.6. Para cada instância, além do valor da melhor solução obtida, é mostrado também o valor de α que chegou nesta solução, e o tempo total de CPU gasto em cada execução da heurística. Esta heurística não pode ser testada com as instâncias de mais de cem jobs devido ao tamanho das formulações MIP a elas associadas.

Em termos de qualidade da solução obtida, os resultados da heurística SPLCPL não são tão bons como os da heurística SPLCH. Por outro lado, como poderá ser observado na Tabela 4.9, em treze instâncias a solução obtida por SPLCPL tem *makespan* estritamente menor do que pelo menos uma entre as melhores soluções obtidas por SPLCSA, SPLCA e pela heurística baseada em regra de prioridades.

A última estratégia heurística seqüencial implementada para o SPLC neste trabalho foi o algoritmo de busca tabu TSSPLC descrito na Seção 4.4.2. Os resultados obtidos com a sua aplicação ao conjunto de intâncias de teste do SPLC são apresentados a seguir.

Inicialmente, o algoritmo TSSPLC foi testado em um subgrupo das instâncias para o ajuste dos valores de MAX_TOTAL_ITER e MAX_BAD_MOVES usados no seu critério de parada. O limite MAX_TOTAL_ITER no número total de iterações executadas por TSSPLC foi testado com os valores 100, 500 e 1000.

O limite MAX_BAD_MOVES no número máximo de iterações sem melhoria na função objetivo foi testado com os valores 50, 100 e 200. Os valores que se mostraram mais adequados, em termos do compromisso "tempo total de execução e qualidade da solução obtida", foram MAX_TOTAL_ITER = 500 e MAX_BAD_MOVES = 100. Assim, estes foram os valores adotados nos demais testes computacionais do algoritmo seqüencial de busca tabu implementado para o SPLC .

O algoritmo TSSPLC foi testado com cada uma das suas oito configurações (Tabela 4.1). Cada configuração, por sua vez, foi executada cinco vezes com cada uma das seis combinações possíveis de solução inicial e função objetivo para o algoritmo TSSPLC: (*Best_PRH*, F_1), (*Best_PRH*, F_2), (*Best_SSH*, F_1), (*Best_SSH*, F_2), (*Sol_PLH*, F_1) e (*Sol_PLH*, F_2). A Tabela 4.7 apresenta as melhores soluções obtidas por cada configuração ao final de todos estes testes. Ao lado do valor do *makespan* de uma solução apresentada nesta tabela, mostra-se, esquematicamente, o par (solução inicial, função objetivo) que levou à solução correspondente. A seguinte convenção foi adotada:

$\star$: (*Best_PRH*, F_1); $\star\star$: (*Best_PRH*, F_2)

o: (*Best_SSH*, F_1); oo: (*Best_SSH*, F_2)

$\bullet$: (*Sol_PLH*, F_1); $\bullet\bullet$: (*Sol_PLH*,F_2)

Analisando os resultados da Tabela 4.7 pode-se fazer os seguintes comentários:

- Nenhuma das oito configurações do algoritmo TSSPLC domina as demais na qualidade das soluções obtidas. Cada uma delas foi a responsável por pelo menos uma das melhores soluções obtidas com a busca tabu seqüencial para as instâncias de teste do SPLC.
- Em termos da influência do tipo de solução inicial no desempenho do algoritmo TSSPLC, o que se pode observar dos resultados da Tabela 4.7 é que as melhores soluções obtidas por cada uma das oito configurações, em 50% dos casos, correspondem a uma solução inicial do tipo *Best_SSH* (melhor solução obtida com alguma das heurísticas baseadas em classes de escalonamento). Este fato não surpreende pois, entre as três alternativas de solução inicial para o algoritmo TSSPLC, *Best_SSH*, é, sem dúvida, a solução de melhor qualidade.
- Com relação ao desempenho do algoritmo TSSPLC de acordo com o tipo de função objetivo adotado, o que se observa é que 50% das melhores soluções obtidas por cada uma das oito configurações, corresponde a função objetivo F_1 (*makespan*) e os 50% restantes à função objetivo F_2, (função objetivo modificada da Seção 3.2). Este fato mostra como foi interessante usar no tabu seqüencial a função objetivo modificada inicialmente proposta para o algoritmo de *branch-and-bound* implementado para o SPLC.

A Tabela 4.8 apresenta as melhores soluções obtidas pelo algoritmo TSSPLC juntamente com o tempo de CPU necessário para chegar até elas.

Instância	TSSPLC$_1$		TSSPLC$_2$		TSSPLC$_3$		TSSPLC$_4$		TSSPLC$_5$		TSSPLC$_6$		TSSPLC$_7$		TSSPLC$_8$	
Ins_4o_21j_A	**82**	⋆	**82**	⋆	**82**	○	**82**	●	**82**	●	**82**	⋆	**82**	○	**82**	○
Ins_4o_23j_A	**58**	⋆	**58**	○	**58**	○	**58**	○	**58**	○	**58**	⋆	**58**	○	**58**	⋆
Ins_4o_24j_A	**68**	●	**68**	○○	69	○	**68**	●●	69	○	69	○	**68**	⋆⋆	**68**	○○
Ins_4o_24j_B	**72**	⋆	**72**	⋆	**72**	⋆⋆	**72**	●	**72**	⋆⋆	**72**	⋆	**72**	⋆⋆	**72**	●●
Ins_4o_27j_A	**67**	●	**67**	⋆	**67**	●	**67**	○	**67**	○○	**67**	○	**67**	⋆	**67**	⋆
Ins_6o_41j_A	143	●	143	●	143	●	144	⋆	144	⋆⋆	145	●●	**141**	●	143	⋆⋆
Ins_6o_41j_B	112	●	112	●	112	●●	**110**	⋆⋆	113	●●	113	●●	112	●	112	●●
Ins_6o_41j_C	**128**	●	**128**	●	129	●●	129	●●	130	⋆⋆	129	●	**128**	○○	**128**	⋆
Ins_6o_44j_A	**117**	⋆⋆	**117**	○	**117**	○	**117**	○	118	●	118	●	118	⋆⋆	**117**	●
Ins_6o_44j_B	140	⋆⋆	140	○	139	○	139	⋆⋆	140	⋆⋆	140	○○	**137**	●●	139	●
Ins_8o_63j_A	262	●	265	○	**261**	○○	**261**	●●	264	⋆⋆	264	⋆	262	○	263	○
Ins_8o_63j_B	321	●	320	●●	320	●●	319	●●	323	●	320	●	322	●	**316**	●●
Ins_8o_63j_C	**296**	○○	297	○○	**296**	○○	297	○	302	●	301	⋆⋆	297	○	298	○○
Ins_8o_65j_A	408	○○	407	⋆⋆	408	○○	**406**	○	409	⋆	411	○	407	○○	410	○○
Ins_8o_65j_B	387	○	385	○	390	○○	393	○○	393	○	**384**	○○	387	○○	387	○○
Ins_10o_84j_A	**636**	○○	637	○○	638	⋆⋆	641	⋆⋆	641	○	642	○○	640	⋆	642	○○
Ins_10o_84j_B	561	●●	**556**	⋆⋆	568	⋆	567	⋆⋆	560	⋆⋆	565	⋆⋆	561	⋆⋆	563	●●
Ins_10o_85j_A	**791**	○○	**791**	○○	814	○○	806	●●	813	○○	810	○○	801	⋆⋆	801	○○
Ins_10o_87j_A	588	⋆	591	○○	588	⋆⋆	590	○	584	⋆	**582**	●●	584	●●	585	⋆⋆
Ins_10o_88j_A	**460**	●●	465	●	464	○○	463	○	467	○○	463	○	467	●●	464	○
Ins_10o_100j_A	**1468**	○	**1468**	○○	**1468**	○○	**1468**	○○	1469	○○	1469	○○	1469	○○	1469	○
Ins_10o_102j_A	1170	○	1171	○	1168	○	1175	○	**1166**	○	**1166**	○○	**1166**	○	1174	○○
Ins_10o_106j_A	1096	○	**1094**	○○	1101	⋆⋆	1099	○○	1107	⋆⋆	1108	○	1103	⋆⋆	1099	○○
Ins_12o_108j_A	**1277**	○○	1280	○	1286	○○	1279	○○	1282	○○	1284	○	1280	○○	1279	○○
Ins_12o_109j_A	1361	⋆	1359	○	1356	○○	1349	○	1352	○	1371	⋆	**1343**	⋆	1358	⋆

Tabela 4.7: Resultados obtidos com cada configuração do algoritmo de busca tabu seqüencial TSSPLC.

Instância	*Makespan*	Segundos
Ins_4o_21j_A	**82**	< 1
Ins_4o_23j_A	**58**	< 1
Ins_4o_24j_A	**68**	8
Ins_4o_24j_B	**72**	14
Ins_4o_27j_A	**67**	9
Ins_6o_41j_A	**141**	80
Ins_6o_41j_B	**110**	10
Ins_6o_41j_C	**128**	10
Ins_6o_44j_A	**117**	12
Ins_6o_44j_B	**137**	29
Ins_8o_63j_A	**261**	133
Ins_8o_63j_B	**316**	152
Ins_8o_63j_C	**296**	28
Ins_8o_65j_A	**406**	165
Ins_8o_65j_B	**384**	182
Ins_10o_84j_A	**636**	337
Ins_10o_84j_B	**556**	478
Ins_10o_85j_A	**791**	637
Ins_10o_87j_A	**582**	123
Ins_10o_88j_A	**460**	510
Ins_10o_100j_A	**1468**	87
Ins_10o_102j_A	**1166**	291
Ins_10o_106j_A	**1094**	1277
Ins_12o_108j_A	**1277**	859
Ins_12o_109j_A	**1343**	381

Tabela 4.8: Melhores soluções obtidas pelo algoritmo seqüencial de busca tabu implementado para o SPLC. Ao lado de cada solução é dado o tempo de CPU gasto para chegar até ela.

A influência do componente aleatório dos conjuntos de soluções candidatas do algoritmo TSSPLC na qualidade das soluções obtidas pode ser visualizada no gráfico da Figura 4.6. Este gráfico mostra, para cada instância, a percentagem, acima da melhor solução, da pior solução e da mediana obtidas em cinco execuções deste algoritmo. Em todas estas execuções, para cada instância, foi usada a combinação de configuração, solução inicial e função objetivo do algoritmo TSSPLC que levou ao melhor resultado na Tabela 4.7. Claramente, a aleatoriedade presente no algoritmo TSSPLC não acarreta uma variação significativa na qualidade das soluções obtidas em várias execuções. Em todas as instâncias testadas, a pior solução obtida em cinco execuções do algoritmo TSSPLC não chegou a ser 3% maior do que a melhor solução obtida.

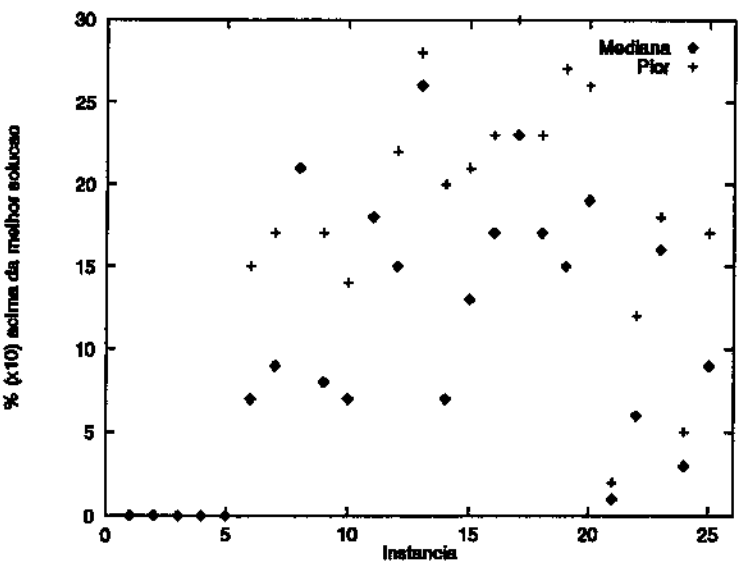


Figura 4.6: Relação entre a melhor, a mediana e a pior solução obtidas em cinco execuções do algoritmo TSSPLC.

O gráfico da Figura 4.7 ilustra a relação entre o tempo de CPU levado para chegar na melhor solução e o tempo de CPU total gasto até o término do algoritmo TSSPLC. Para cada instância, os valores mostrados foram obtidos em uma única execução do algoritmo TSSPLC usando a combinação de configuração, solução inicial e função objetivo que levou à melhor solução para esta instância na Tabela 4.7.

Finalmente, para efeito comparativo, a Tabela 4.9 traz as melhores soluções (limitantes superiores) obtidas por cada uma das estratégias heurísticas seqüenciais implementadas neste trabalho e por outras duas estratégias disponíveis na literatura. As colunas $\overline{\xi}_{PRH}$, $\overline{\xi}_{SSH}$, $\overline{\xi}_{PLH}$ e $\overline{\xi}_{TS}$, desta tabela trazem, respectivamente, as melhores soluções obtidas com a heurística baseada em regras de prioridade (Seção 4.1) , com as heurísticas baseadas em classes de escalonamento (Seção 4.2), com a heurística baseada em programação linear (Seção 4.3) e com o algoritmo seqüencial de busca tabu (Seção 4.4.2) implementados para o SPLC. A coluna $\overline{\xi}_{CP}$ traz a melhor solução obtida com programação por restrições [HC97] (Seção 2.4.1) e, por fim, a coluna $\overline{\xi}_{PM}$ apresenta as

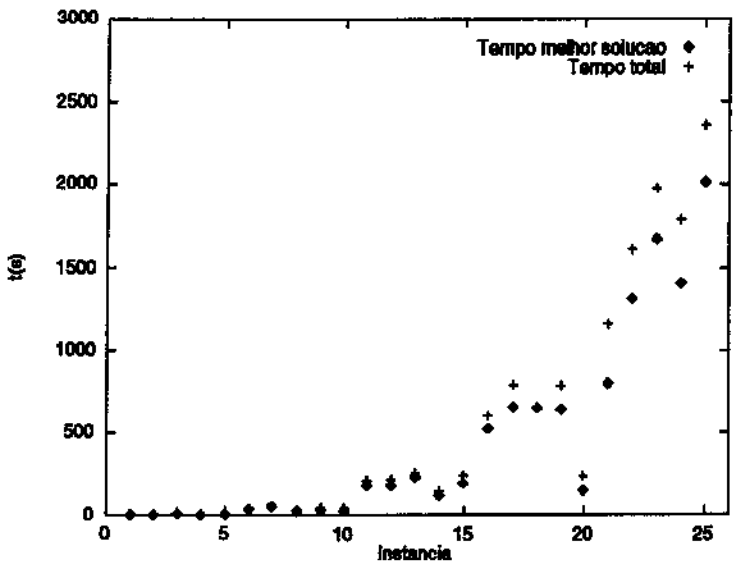


Figura 4.7: Relação entre o tempo de CPU gasto para chegar na melhor solução e o tempo total gasto pelo algoritmo TSSPLC.

melhores soluções obtidas com a heurística baseada em programação matemática de Savelsbergh [SUW97] (Seção 2.4.2).

Analisando os valores da Tabela 4.9 é fácil ver a superioridade do algoritmo seqüencial de busca tabu implementado neste trabalho para o SPLC. As soluções por ele obtidas dominam em qualidade qualquer outra obtida pelas demais estratégias apresentadas. Isto confirma a potencialidade do algoritmo de busca tabu na obtenção de soluções de qualidade para o SPLC.

O Capítulo 5, a seguir, mostra como algumas das heurísticas seqüenciais implementadas neste trabalho para o SPLC foram integradas em estratégias paralelas e como os esforços conjuntos destes algoritmos permitiram chegar em soluções de qualidade ainda melhor para as várias instâncias do SPLC.

Instância	ξ_{PRH}	ξ_{SSH}	ξ_{PLH}	ξ_{TS_*}	ξ_{CP}	ξ_{PM}
Ins_4o_21j_A	**82**	**82**	87	**82**	**82**	**82**
Ins_4o_23j_A	60	**58**	63	**58**	**58**	**58**
Ins_4o_24j_A	71	69	73	**68**	**68**	**68**
Ins_4o_24j_B	77	73	77	**72**	**72**	**72**
Ins_4o_27j_A	69	69	73	**67**	**67**	**67**
Ins_6o_41j_A	158	150	152	**141**	152	142
Ins_6o_41j_B	122	115	123	**110**	**110**	112
Ins_6o_41j_C	138	136	140	**128**	134	130
Ins_6o_44j_A	124	123	124	**117**	122	118
Ins_6o_44j_B	155	146	152	**137**	149	**137**
Ins_8o_63j_A	296	283	299	**261**	281	273
Ins_8o_63j_B	375	347	369	**316**	344	323
Ins_8o_63j_C	334	325	328	**296**	344	308
Ins_8o_65j_A	425	421	425	**406**	445	411
Ins_8o_65j_B	438	408	466	**384**	411	402
Ins_10o_84j_A	712	708	719	**636**	730	–
Ins_10o_84j_B	643	606	639	**556**	616	–
Ins_10o_85j_A	965	879	964	**791**	912	–
Ins_10o_87j_A	656	615	646	**582**	610	–
Ins_10o_88j_A	524	493	519	**460**	473	–
Ins_10o_100j_A	1582	1519	–	**1468**	1587	–
Ins_10o_102j_A	1321	1226	–	**1166**	1239	–
Ins_10o_106j_A	1215	1163	–	**1094**	1166	–
Ins_12o_108j_A	1435	1341	–	**1277**	1412	–
Ins_12o_109j_A	1522	1416	–	**1343**	1476	–

Tabela 4.9: Melhores soluções obtidas para o SPLC com as heurísticas seqüenciais do Capítulo 4 e com duas outras estratégias disponíveis na literatura.

Capítulo 5

Limitantes Superiores II - Estratégias Paralelas e Assíncronas

O surgimento de arquiteturas paralelas e a evolução de sistemas distribuídos marcam o início de um novo paradigma na busca de soluções para problemas combinatórios NP-difíceis: os algoritmos paralelos.

Algoritmos paralelos são procedimentos que permitem que dados de um problema e/ou tarefas de métodos de solução para o mesmo sejam divididos entre vários processos que eventualmente se comunicam entre si. Desta forma, potencialmente, uma exploração mais efetiva do espaço de soluções é alcançada.

Dependendo da estratégia de paralelização adotada, costuma-se classificar métodos paralelos em três grupos [CT97]:

- *Tipo 1*: algoritmos paralelos que diferem das correspondentes versões seqüenciais pela paralelização de tarefas que exigem alto esforço computacional.
- *Tipo 2*: algoritmos paralelos que se caracterizam pela divisão do domínio ou do espaço de soluções do problema entre vários processos.
- *Tipo 3*: algoritmos paralelos formados por vários processos que exploram simultaneamente o espaço de soluções e podem trocar informações entre si de acordo com algum grau de cooperação e sincronização estabelecidos.

Em se tratando de algoritmos paralelos heurísticos, pode-se dizer que, no que se refere à qualidade da solução obtida, algoritmos paralelos do tipo 1 não apresentam ganho em relação às versões seqüenciais correspondentes: eles chegam na mesma solução destes últimos, apenas mais rápido.

Algoritmos do tipo 2, por outro lado, se baseiam no princípio de que se cada um dos seus processos estiver dedicado a resolver um problema menor do que o original, procedimentos mais criteriosos podem ser usados na solução destes subproblemas. A solução do algoritmo paralelo é a melhor solução entre as obtidas por cada um dos seus processos e, possivelmente, tem qualidade superior à do método seqüencial correspondente.

Estratégias paralelas do tipo 3, contudo, são as que têm se mostrado mais robustas em termos da qualidade da solução obtida [TCG96]. Nestes algoritmos, vários processos exploram ao mesmo tempo o espaço de soluções. Os processos podem ser independentes ou cooperar através da troca de informações obtidas por cada um deles. Neste último caso, comunicações síncronas ou assíncronas podem ser adotadas.

No contexto de computação paralela, sincronismo indica um modo de operação no qual todos os processos do algoritmo paralelo devem alcançar um determinado estágio (número de iterações ou tempo de processamento, por exemplo) antes que a próxima etapa de processamento seja iniciada [CTG95]. Assincronismo, por outro lado, é um modo de operação onde não existem pontos em que um processo fica aguardando os demais para continuar a sua execução. Em estratégias assíncronas, os processos de um algoritmo paralelo são autônomos e estabelecem comunicação com outro(s) processo(s) apenas a partir da sua necessidade de fornecer ou receber informações.

Além da redução do custo com comunicação entre processos e do tempo em que estes ficam ociosos, algoritmos paralelos cooperativos e assíncronos do tipo 3 têm uma outra vantagem: eles são menos sensíveis do que estratégias seqüenciais a variações nas características das instâncias do problema que se está resolvendo. A coexistência de processos diferentes neste tipo de algoritmo paralelo, aliada à cooperação entre eles, permite que as eventuais deficiências de um dos processos para um certo tipo de instância possa ser suprida pelos demais processos ou pela troca de informações entre eles.

Este capítulo apresenta as duas estratégias paralelas do tipo 3 propostas neste trabalho para o SPLC. Ambas fazem uso de processos cooperativos que se comunicam assincronamente. São elas: *A-Team* e estratégia paralela e assíncrona de busca tabu. As Seções 5.1 e 5.2, a seguir, descrevem, respectivamente, os detalhes de cada uma destas estratégias e apresentam os resultados computacionais obtidos com a sua aplicação ao conjunto de instâncias de teste do SPLC.

5.1 *A-Team* para o SPLC

A escolha de que estratégia adotar na busca de soluções para um problema combinatório não é uma tarefa simples. Diferentes abordagens levam a diferentes algoritmos com desempenhos (tempo computacional e qualidade da solução obtida) provavelmente diferentes. A opção pela utilização de um único algoritmo impede a exploração das vantagens dos outros métodos.

A estratégia Times Assíncronos ou *A-Teams* (do inglês *Asynchronous Teams*) é uma organização de *software* que possibilita a cooperação entre vários algoritmos diferentes com o intuito de que, juntos, eles possam chegar a soluções melhores do que as obtidas por cada um deles isoladamente [Sou93].

A Seção 5.1.1, a seguir, apresenta os conceito básicos da metodologia *A-Teams*. Em seguida, as Seções 5.1.2 a 5.1.6 descrevem a estrutura e as características do *A-Team* proposto neste trabalho para o SPLC. A Seção 5.1.7, por fim, traz os resultados computacionais obtidos com a aplicação desta estratégia às instâncias de teste do SPLC.

5.1.1 Conceituação Básica de *A-Team*

O método *A-Team* para solução de problemas combinatórios foi introduzido por Souza em [Sou93]. Como o próprio nome sugere, trata-se de uma técnica que utiliza um "time" de algoritmos (agentes) que exploram simultaneamente o espaço de soluções de um problema trocando informações entre si através de memórias compartilhadas.

O princípio básico de *A-Team* é que a combinação dos esforços de vários agentes construindo e modificando soluções potencialmente conduz a soluções de melhor qualidade do que aquelas obtidas por cada agente isoladamente.

Dentro da estrutura de um *A-Team*, o controle é descentralizado. Os agentes são autônomos e agem nas memórias compartilhadas (lendo ou escrevendo soluções) de acordo com seus próprios critérios. Desta forma, as comunicações são assíncronas e toda troca de informações entre os agentes acontece através das memórias compartilhadas.

Tipicamente, são quatro os tipos de agentes em um *A-Team* [Sou93, Cav95]:

- Agentes de Construção: algoritmos que geram soluções completas para o problema que se está resolvendo;
- Agentes de Desconstrução: algoritmos que, a partir de uma ou mais soluções completas, geram uma solução parcial para o problema;
- Agentes de Melhoria: algoritmos que modificam soluções já existentes;
- Agentes de Destruição: algoritmos que de acordo com alguma política de destruição eliminam soluções de uma memória compartilhada.

Num *A-Team* genérico, os quatro tipos de agentes definidos acima estão continuamente ativos: novas soluções são construídas, soluções já existentes são desconstruídas ou modificadas e soluções de qualidade inferior são eliminadas. Memórias compartilhadas armazenam soluções completas ou parciais e podem ser acessadas a qualquer momento por qualquer um dos agentes.

Graficamente, costuma-se representar agentes de um *A-Team* como arcos e memórias compartilhadas como retângulos. A Figura 5.1 traz um exemplo de um *A-Team* que pode ser descrito da seguinte forma: os agentes A e B lêem da memória 1 e escrevem, respectivamente, nas memórias 1 e 3. O agente C lê da memória 2 e escreve na memória 1. O agente D lê da memória 3 e pode escrever tanto na memória 1 quanto na memória 2 (note o retângulo externo a estas memórias). O agente E lê da memória 2 e escreve na memória 3. O agente F inicia as memórias 1 e 2 e o agente G remove soluções destas memórias.

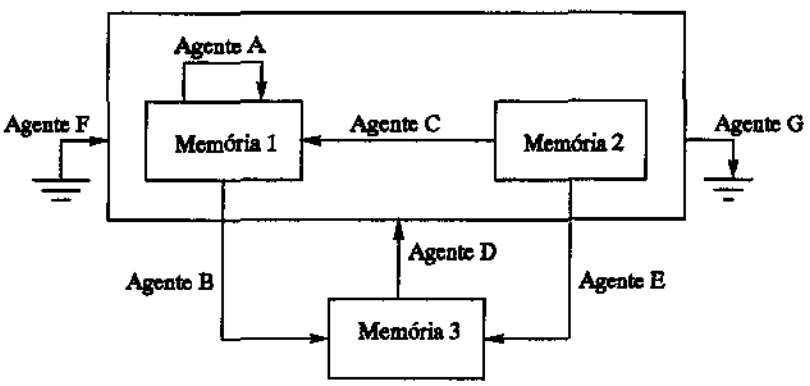


Figura 5.1: Exemplo de um *A-Team*.

Uma característica fundamental de *A-Teams* é o fluxo cíclico de dados dentro da sua estrutura. Agentes recuperam, modificam e armazenam informações nas memórias compartilhadas. Soluções geradas por um algoritmo são eventualmente usadas como entrada de outros algoritmos. Este fluxo contínuo de informações permite a interação entre os agentes e, potencialmente, o refinamento das soluções por eles obtidas.

Note-se, por fim, a intrínseca adequação de *A-Teams* ao processamento paralelo. A sua estrutura com agentes autônomos, comunicações assíncronas e controle descentralizado permite a execução simultânea dos diferentes algoritmos nos diversos processadores de uma rede de computadores.

Aplicações de *A-Teams* têm obtido êxito na busca de boas soluções para problemas combinatórios como: caixeiro viajante [Sou93], *flow shop scheduling* [Pei95] e *job shop scheduling* [Cav95, Had96]. Este fato serviu como motivação para investigar a adequabilidade de *A-Teams* como estratégia de solução também para o SPLC.

O *A-Team* proposto neste trabalho para o SPLC está esquematicamente apresentado na Figura 5.2. Existem duas memórias compartilhadas: CSM, a memória de soluções completas e PSM, a memória de soluções parciais. Um agente inicializador gera o conjunto inicial de soluções completas da memória CSM. Agentes de desconstrução usam soluções de CSM para gerar soluções parciais que são então armazenadas na memória PSM. Agentes de construção lêem soluções de PSM e as transformam em soluções completas que são armazenadas em CSM. Agentes de melhoria lêem soluções da memória CSM e, caso consigam melhorá-las, escrevem as novas soluções

nesta mesma memória. Finalmente, um agente destrutor é responsável por eliminar soluções da memória CSM para dar lugar às soluções completas obtidas pelos agentes de construção e melhoria.

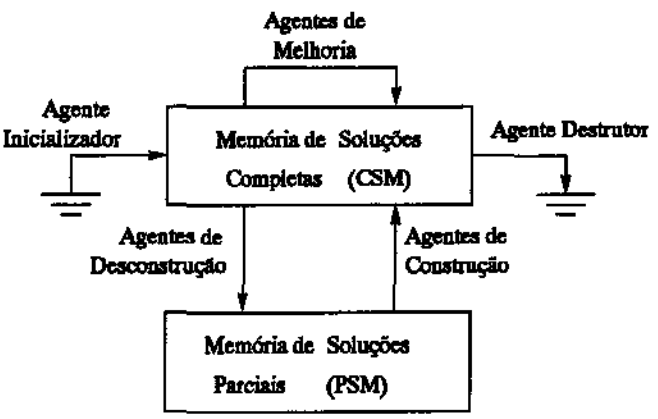


Figura 5.2: Estrutura do *A-Team* proposto para o SPLC.

Em termos de implementação, a memória de soluções parciais e os agentes de construção, desconstrução e melhoria correspondem, cada um, a um processo diferente. A memória de soluções completas e os agentes inicializador e destrutor, por outro lado, estão juntos em um único processo.

As Seções 5.1.2 a 5.1.6, a seguir, detalham cada um dos processos que implementam os agentes e as memórias de soluções que compõem o *A-Team* proposto para o SPLC.

5.1.2 Processo ATC - Memória de Soluções Completas, Agente Inicializador e Agente Destrutor

Este processo tem cinco funções básicas:

(i) Inicializar a memória de soluções completas;

(ii) Dar início a todos os demais processos que compõem o *A-Team* para o SPLC;

(iii) Enviar soluções completas para os agentes de desconstrução e melhoria;

(iv) Receber soluções completas obtidas pelos agentes de construção e melhoria;

(v) Determinar o fim do *A-Team*.

Como cada uma destas funções é desempenhada pelo processo ATC está descrito a seguir.

Inicialização da Memória de Soluções Completas

A memória de soluções completas do *A-Team* proposto para o SPLC é mantida pelo processo ATC como uma lista, *csm*, de tamanho CSM_SIZE. O primeiro passo do processo ATC é exatamente a inicialização desta lista de maneira a disponibilizar para os demais agentes do *A-Team* um conjunto inicial de soluções completas.

As primeiras soluções completas inseridas na lista *csm* pelo processo ATC são:

- A melhor solução obtida com a heurística baseada em regras de prioridade da Seçao 4.1;
- A melhor solução obtida com alguma das heurísticas baseadas em classes de escalonamento da Seção 4.2;
- A solução obtida com a heurística baseada em programação linear da Seção 4.3.

As outras soluções necessárias para integralizar o tamanho CSM_SIZE da memória de soluções completas são geradas usando a heurística baseada na regra de prioridade RANDOM da Seção 4.1.

A utilização desta estratégia de inicialização foi feita com o objetivo de se ter ao mesmo tempo diversidade e boa qualidade de soluções na memória inicial de soluções completas.

Cada elemento da lista *csm* contém o custo e a seqüência representativa de uma solução completa. As soluções desta lista são mantidas em ordem crescente dos seus custos e não são permitidas soluções repetidas. Uma solução completa só é inserida na lista *csm* após a constatação de que ela é diferente de todas as demais soluções desta lista.

A igualdade de duas soluções é verificada comparando-se inicialmente os seus custos. Se estes forem iguais, as seqüências representativas de cada uma das soluções são comparadas. Duas soluções serão idênticas se e somente se elas tiverem o mesmo custo e forem representadas pela mesma seqüência.

Início dos Outros Processos do *A-Team*

Após a inicialização da memória de soluções completas, o próximo passo do processo ATC é dar início a todos os demais processos que compõem o *A-Team* proposto para o SPLC. São eles: processo ATP (memória de soluções parciais); processos IBC e DBC (agentes de desconstrução); processos SPLCSA, SPLCH e SPLCA (agentes de construção); e processos IMPJ, IMPCP, IMPALL e IMPSWP (agentes de melhoria).

Uma vez iniciados, cada um destes processos segue sua execução autonomamente. O processo ATC fica, então, a espera de mensagens indicando se ele deve receber uma nova solução obtida por um agente de melhoria ou de construção, ou se ele deve enviar soluções completas já existentes para um agente de melhoria ou de desconstrução.

Recebimento de Solução

No *A-Team* proposto para o SPLC, quando um agente de construção ou um agente de melhoria encontram uma nova solução completa, eles enviam para o processo ATC uma mensagem (*CANDIDATE*) contendo o custo desta solução. Ao receber a mensagem, o processo ATC decide se vai aceitar a nova solução com base no seguinte **critério de aceitação de soluções:**

> "Somente serão aceitas soluções completas cujo custo seja no máximo γ% maior do que o custo da melhor solução da memória de soluções completas."

O valor γ é chamado de limite percentual do critério de aceitação e o seu objetivo é evitar que sejam consideradas soluções com qualidade muito inferior à da melhor solução da memória de soluções completas. Com isto, pretende-se diminuir o fluxo de soluções para o processo ATC de forma que ele possa ficar mais dedicado a receber e tratar outras mensagens.

Uma vez avaliado o custo da solução de acordo com o critério de aceitação de soluções, o processo ATC responde ao agente que enviou a mensagem *CANDIDATE* dizendo se aceita ou não receber a nova solução completa. Em caso positivo, o processo ATC permanece esperando para efetivamente recebê-la.

Ao receber uma nova solução completa, o processo ATC verifica se ela já não existe na lista *csm*. Caso exista, a solução é desconsiderada. Caso contrário, o processo ATC elimina uma das soluções desta lista e insere a nova solução recebida, preservando a ordenação da lista pelo custo das soluções.

A escolha de que solução eliminar da lista *csm* para dar lugar à solução recebida é feita de acordo com uma distribuição de probabilidade linear: a melhor solução tem probabilidade 0 de ser removida e as demais têm probabilidade crescente até a pior solução. Esta política de destruição consegue manter a diversidade na memória de soluções completas e ao mesmo tempo mostra um bom nível de compromisso entre a proporção de boas e más soluções eliminadas. Como observado por Cavalcante em [Cav95] esta característica tem se mostrado eficaz para a convergência do *A-Team* para boas soluções.

Esquematizando o que foi dito, as Figuras 5.3 e 5.4 trazem, respectivamente, o pseudo-código dos procedimentos usados pelo processo ATC para tratar mensagens do tipo *CANDIDATE* e para receber soluções completas enviadas por um agente de construção ou de melhoria.

Envio de Solução

Ao longo da execução do *A-Team* proposto para o SPLC, tanto os agentes de desconstrução como os de melhoria continuamente solicitam soluções completas ao processo ATC. A diferença entre estas solicitações é que, enquanto os agentes de melhoria pedem uma única solução completa a cada vez, os agentes de desconstrução pedem duas destas soluções.

Procedimento *TreatCandidateMessage*$(c(s'), pId)$

1. Se $(c(s') \leq (1+\gamma)c_p^*)$
   Então *accept* = TRUE.
   Senão *accept* = FALSE.
2. Envie mensagem *ACCEPT_CANDIDATE*(*accept*) para o agente (de melhoria ou construção) *pId*.
3. Se (*accept* == TRUE)
   Então *ReceiveSolution*(*pId*).

Figura 5.3: Procedimento usado pelo processo ATC para tratar mensagens do tipo *CANDIDATE*.

Procedimento *ReceiveSolution*(*pId*)

1. Aguarde mensagem *SOLUTION*(s) do agente (de melhoria ou construção) *pId*.
2. Se s for diferente de todas as soluções da lista *csm*
   Então elimine, usando uma distribuição de probabilidade linear, uma solução da lista *csm*.
   Insira s na lista *csm*.
   $nInsertedSol++$.

Figura 5.4: Procedimento usado pelo processo ATC para receber soluções completas dos agentes de melhoria e construção.

Assim, são dois os tipos de mensagens recebidas pelo processo ATC indicando solicitação de solução: *REQUEST_ONE_SOLUTION*, enviada por um agente de melhoria, e *REQUEST_TWO_SOLUTIONS*, enviada por um agente de desconstrução.

As Figuras 5.5 e 5.6 trazem o pseudo-código das funções usadas pelo processo ATC para tratar cada uma destas mensagens. Em qualquer um dos casos, as soluções enviadas como resposta aos agentes solicitantes são escolhidas aleatoriamente entre todas as soluções da memória de soluções completas. O objetivo com isto é dar aos agentes de melhoria e de desconstrução a chance de trabalhar com todas e com todas as combinações de duas soluções, respectivamente, da memória de soluções completas.

Procedimento *TreatRequestOneSolutionMessage*(*pId*)

1. Escolha aleatoriamente uma solução s da lista *csm*.
2. Envie mensagem *ONE_SOLUTION*(s) para o agente de melhoria *pId*.

Figura 5.5: Procedimento usado pelo processo ATC para tratar solicitação de uma solução.

Procedimento *TreatRequestTwoSolutionsMessage(pId)*

1. Escolha aleatoriamente duas soluções distintas s_1, s_2 da lista *csm*.
2. Envie mensagem *TWO_SOLUTIONS*(s_1, s_2) para o agente de desconstrução *pId*.

Figura 5.6: Procedimento usado pelo processo ATC para tratar solicitação de duas soluções.

Critério de Finalização do *A-Team*

Apesar da autonomia de execução intrínseca aos processos que compõem um *A-Team*, no *A-Team* proposto para o SPLC optou-se por deixar a cargo do processo ATC determinar o instante em que esta estratégia paralela deve terminar.

Assim, quando MAX_SOLUTIONS soluções completas tiverem efetivamente sido inseridas na memória de soluções completas, o processo ATC encerra a execução de todos os demais processos do *A-Team*. A sua própria execução termina em seguida e a melhor solução da memória de soluções completas é retornada.

A Figura 5.7 apresenta o pseudo-código completo do processo ATC.

5.1.3 Processos IBC e DBC - Agentes de Desconstrução

Um agente de desconstrução em um *A-Team* é um agente que parte de uma ou mais soluções completas e gera uma solução parcial para o problema que se quer resolver. O *A-Team* proposto para o SPLC tem dois agentes de desconstrução, cada um deles baseado em um algoritmo de consenso.

Define-se um algoritmo de consenso como aquele que usa informações de duas ou mais soluções para gerar uma terceira [Sou93]. Os exemplos mais comuns deste tipo de algoritmo são [Cav95]: consenso baseado em intersecção e consenso baseado em diferença. Cada um dos dois agentes de desconstrução implementados no *A-Team* para o SPLC usa exatamente um destes dois tipos de consenso. São eles:

- Processo IBC: algoritmo de consenso baseado em intersecção;
- Processo DBC: algoritmo de consenso baseado em diferença.

Estes processos continuamente solicitam duas soluções completas ao processo ATC. As informações (semelhanças ou diferenças) destas soluções são usadas para gerar uma solução parcial que, por sua vez, é enviada para o processo ATP (Seção 5.1.4) onde ficará armazenada até ser solicitada por um dos agentes de construção (Seção 5.1.5).

Processo ATC

1. Inicialize a memória de soluções completas (lista *csm*).
2. Dispare os processos ATP, IBC, DBC, SPLCSA, SPLCH,SPLCA, IMPJ, IMPCP, IMPALL e IMPSWP.
3. Faça $nInsertedSol = 0$; /* número de soluções inseridas na lista csm */
4. Enquanto ($nInsertedSol <$ MAX_SOLUTIONS) faça
   a. Aguarde até receber uma mensagem M de um dos agentes de melhoria, desconstrução ou construção.
   b. Faça $pId =$ identificador do processo que mandou M.
   c. Verifique o tipo da mensagem M
      Caso *CANDIDATE*($c(s')$):
      TreatCandidateMessage($c(s'), pId$).
      Caso *REQUEST_ONE_SOLUTION*:
      TreatRequestOneSolutionMessage(pId).
      Caso *REQUEST_TWO_SOLUTIONS*:
      TreatRequestTwoSolutionsMessage(pId).
5. Encerre os processos ATP, IBC, DBC, SPLCSA, SPLCH, SPLCA, IMPJ, IMPCP, IMPALL e IMPSWP.
6. Retorne a melhor solução da memória de soluções completas.

Figura 5.7: Algoritmo executado pelo processo ATC do *A-Team* para o SPLC.

Uma solução parcial pode ser visualizada como "pedacinhos" de uma solução completa. Assim, se uma solução completa é representada por uma seqüência de n jobs, uma solução parcial pode ser representada por uma lista de blocos, onde cada bloco é uma seqüência de $k < n$ jobs. A forma como os agentes de desconstrução obtêm os blocos de uma solução parcial a partir das soluções completas é explicado a seguir.

Sejam s_A e s_B as duas seqüências representativas das soluções completas usadas como base para o consenso em um certo instante de um dos agentes de desconstrução.

No caso do processo IBC, os blocos da solução parcial gerada a partir de s_A e s_B são os "pedaços" resultantes da intersecção, posição a posição, destas duas seqüências. A motivação para um consenso baseado em intersecção é tentar isolar os aspectos (no caso, grupos de jobs consecutivos numa seqüência) que se repetem nas soluções completas. A esperança é que, ao longo da execução do *A-Team*, estas repetições indiquem características desejáveis em soluções de boa qualidade.

No caso do processo DBC, a idéia é explorar as diferenças entre as soluções s_A e s_B de acordo com a seguinte idéia: se s_A é diferente de s_B e, por exemplo, s_A tem custo melhor do que s_B, então é porque há algo de bom em s_A que não existe em s_B. Assim, os blocos da solução parcial gerada a partir de s_A e s_B no algoritmo de consenso baseado em diferença são os "pedaços" da

solução s_A que não estão na intersecção posição a posição de s_A e s_B.

A Figura 5.8 ilustra a geração de uma solução parcial a partir de duas soluções completas usando consenso baseado em interseção e consenso baseado em diferença.

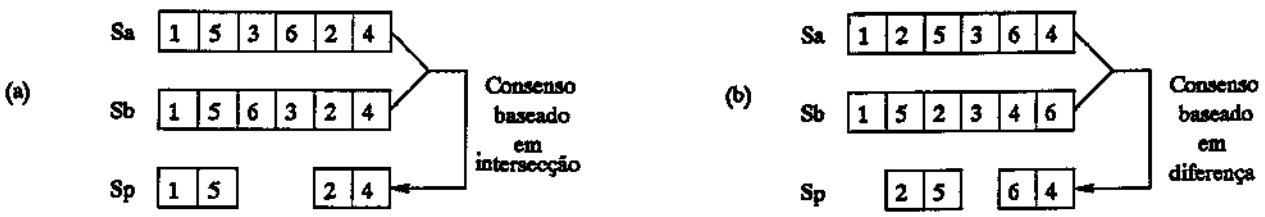


Figura 5.8: Exemplo da geração de uma solução parcial s_p a partir de duas soluções completas, s_a e s_b, usando consenso baseado em intersecção e consenso baseado em diferença.

A Figura 5.9 traz o pseudo-código do algoritmo executado pelos agentes de desconstrução do *A-Team* implementado para o SPLC.

5.1.4 Processo ATP - Memória de Soluções Parciais

Este processo é responsável pelo gerenciamento da memória de soluções parciais do *A-Team* proposto para o SPLC. Basicamente, ele é uma espécie de *buffer* que recebe soluções parciais dos agentes de desconstrução (Seção 5.1.3) e as envia, quando solicitadas, para os agentes de construção (Seção 5.1.5). A seguir são apresentados os principais aspectos do processo ATP.

Armazenamento das Soluções Parciais

Para guardar as soluções parciais, o processo ATP mantém uma lista chamada *psm*. Inicialmente vazia, esta lista cresce com a inserção das soluções parciais recebidas dos agentes de desconstrução, e decresce quando estas soluções são enviadas aos agentes de construção. Cada elemento

---

Agente de Desconstrução IBC ou DBC

Enquanto (TRUE) faça
a. Envie mensagem *REQUEST_TWO_SOLUTIONS* para o processo ATC solicitando duas soluções completas.
b. Aguarde mensagem *TWO_SOLUTIONS*(s_1, s_2) do processo ATC.
c. Gere uma solução parcial s usando consenso baseado na intersecção (ou na diferença) de s_1, s_2.
d. Envie mensagem *PARTIAL_SOLUTION*(s) para o processo ATP.

Figura 5.9: Algoritmo executado pelos agentes de desconstrução IBC, DBC do *A-Team* implementado para o SPLC.

Processo ATP

1. Faça *psm* = NULL.
2. Enquanto (TRUE) faça
   a. Aguarde até receber uma mensagem *M* de um dos agentes construção ou desconstrução.
   b. Faça *pId* = identificador do processo que mandou *M*.
   c. Verifique o tipo da mensagem *M*
      Caso *PARTIAL_SOLUTION*(*s*):
         Insira *s* no final da lista *psm*.
      Caso *REQUEST_PARTIAL_SOLUTION*:
         Se (*psm* $\neq$ NULL)
            Então *accept* = TRUE.
            Senão *accept* = FALSE.
         Envie mensagem *REQUEST_ANSWER*(*accept*) para o agente de construção *pId*.
         Se (*accept* == TRUE)
            Então remova a primeira solução, *s*, da lista *psm*.
                  Envie mensagem *PARTIAL_SOLUTION*(*s*) para o agente de construção *pId*.

Figura 5.10: Algoritmo executado pelo processo ATP do *A-Team* para o SPLC.

da lista *psm* corresponde a uma solução parcial e é uma lista dos blocos de jobs que formam esta solução.

Recebimento de Soluções Parciais

Sempre que um agente de desconstrução obtém uma solução parcial, ele envia uma mensagem (*PARTIAL_SOLUTION*) para o processo ATP contendo a lista de blocos que formam esta solução. Ao receber a mensagem, o processo ATP simplesmente insere a nova solução parcial no final da lista *psm*.

Envio de Soluções Parciais

Os agentes de construção continuamente mandam uma mensagem (*REQUEST_PARTIAL_SOLUTION*) para o processo *ATP* solicitando uma solução parcial para transformá-la em solução completa.

Ao receber uma mensagem deste tipo, o processo ATP responde ao agente solicitante se é possível ou não atender o seu pedido. Uma solicitação só pode ser atendida quando a lista *psm* é não vazia. Neste caso, o processo ATP remove a primeira solução parcial desta lista e a envia para o agente de construção que fez o pedido.

A Figura 5.10 traz o pseudo-código completo do processo ATP.

5.1.5 Processos SPLCSA, SPLCH e SPLCA - Agentes de Construção

Os agentes de construção do *A-Team* proposto para o SPLC têm como função transformar soluções parciais em soluções completas. Esta transformação é feita de acordo com o seguinte **mecanismo de reconstrução:**

> "Os jobs que ainda não estão na solução parcial vão sendo um a um a ela acrescentados. Cada um destes jobs é inserido numa posição da solução parcial tal que: (i) as posições relativas e consecutivas dos jobs nos blocos já existentes na solução parcial são preservadas; (ii) o job inserido deve ficar sempre a direita de todos os seus predecessores. A ordem em que os jobs "faltantes" são considerados para inserção na solução parcial é determinada pela ordem em que eles aparecem em uma solução completa auxiliar gerada por alguma heurística."

Esta heurística usada no mecanismo de reconstrução é exatamente o que diferencia cada um dos três agentes de construção do *A-Team* para o SPLC. Eles são o processo SPLCSA, o processo SPLCH e o processo SPLCA e usam, respectivamente, as heurísticas seqüenciais SPLCSA, SPLCH e SPLCA da Seção 4.2 no seu mecanismo de reconstrução.

A opção pela utilização nos agentes de construção das heurísticas seqüenciais baseadas em classes de escalonamento foi motivada pela simplicidade, rapidez e boa qualidade das soluções destes algoritmos.

A Figura 5.11 ilustra o mecanismo de reconstrução de uma solução parcial a partir de uma solução completa auxiliar para a *instância exemplo* do SPLC (Seção 2.2). Note que os jobs dos blocos da solução parcial permanecem consecutivos na solução completa resultante.

Figura 5.11: Exemplo da reconstrução de uma solução parcial s_p usando uma solução completa auxiliar s_{aux}.

Um agente de construção pode, então, ser descrito da seguinte forma: continuamente ele solicita uma solução parcial ao processo ATP enviando-lhe a mensagem *REQUEST_PARTIAL_SOLUTION*. Caso o processo ATP possa atender o seu pedido, o agente de construção recebe uma solução parcial e a completa de acordo com o seu mecanismo de reconstrução. Em seguida, ele envia uma mensagem (*CANDIDATE*) para o processo ATC contendo o custo da solução completa obtida. Caso o processo ATC aceite receber esta nova solução, o agente de construção efetivamente a envia para que ela fique armazenada na memória de soluções completas do *A-Team*.

Agente de Construção SPLCSA, SPLCH ou SPLCA

Enquanto (TRUE) faça
- a. Envie mensagem *REQUEST_PARTIAL_SOLUTION* para o processo ATP solicitando uma solução parcial.
- b. Aguarde mensagem *REQUEST_ANSWER*(*accept*) do processo ATP.
- c. Se (*accept* == TRUE)
  - Então aguarde mensagem *PARTIAL_SOLUTION*(s) do processo ATP.
    - Gere solução completa auxiliar s' usando uma das heurísticas SPLCSA, SPLCH ou SPLCA (Seção 4.2).
    - Use s' e o mecanismo de reconstrução para completar a solução parcial s.
    - Envie mensagem *CANDIDATE*($c(s)$) para o processo ATC.
    - Aguarde mensagem *ACCEPT_CANDIDATE*(*accept*) do processo ATC.
    - Se (*accept* == TRUE)
      - Então envie mensagem *SOLUTION*(s) para o processo ATC

Figura 5.12: Algoritmo executado pelos agentes de construção SPLCSA, SPLCH e SPLCA do *A-Team* implementado para o SPLC.

A Figura 5.12 traz o pseudo-código dos algoritmos executados pelos agentes de contrução do *A-Team* implementado para o SPLC.

5.1.6 Processos IMPJ, IMPCP, IMPALL e IMPSWP - Agentes de Melhoria

Os agentes de melhoria do *A-Team* proposto para o SPLC podem ser assim descritos: continuamente eles enviam uma mensagem (*REQUEST_ONE_SOLUTION*) ao processo ATC solicitando-lhe uma solução completa. Ao receber esta solução, eles iniciam uma busca local a partir dela. Caso consigam melhorá-la, enviam uma mensagem (*CANDIDATE*) para o processo ATC contendo o custo da solução completa obtida. Se o processo ATC aceitar receber a nova solução, eles a enviam para que fique armazenada na memória de soluções completas.

São quatro os agentes de melhoria implementados no *A-Team* para o SPLC: processo IMPJ, processo IMPCP, processo IMPALL e processo IMPSWP. A única diferença entre eles está na busca local que cada um realiza a partir das soluções recebidas do processo ATC, como será explicado a seguir.

Seja s a solução recebida do processo ATC em um certo instante da execução de um agente de melhoria e sejam $N_{IMPJ}(s)$, $N_{IMPCP}(s)$, $N_{IMPALL}(s)$ e $N_{IMPSWP}(s)$ as vizinhanças da solução s nos processos IMPJ, IMPCP, IMPALL e IMPSWP, respectivamente. Sejam ainda os movimentos *Insert* e *Swap*, as funções *CheckFeasibilityOfInsert* e *CheckFeasibilityofSwap* e o conceito de PPL_j tais como definidos na Seção 4.4.2. Assim:

- $N_{IMPJ}(s)$ é o conjunto de todas as soluções obtidas a partir de s pela aplicação do movimento *Insert* a todos os pares (i, j) para um único j escolhido aleatoriamente e todos os i's satisfazendo: $i \in PPL_j$ ou i é sucessor direto de j; *CheckFeasibilityOfInsert(s, i, j)* retorna *viável*.

- $N_{IMPCP}(s)$ é o conjunto de todas as soluções obtidas a partir de s pela aplicação do movimento *Insert* a todos os pares (i, j) tais que: j pertence ao caminho crítico do grafo de precedências entre jobs e i satisfaz: $i \in PPL_j$ ou i é sucessor direto de j; *CheckFeasibilityOfInsert(s, i, j)* retorna *viável*.

- $N_{IMPALL}(s)$ é o conjunto de todas as soluções obtidas a partir de s pela aplicação do movimento *Insert* a todos os pares (i, j) para todo $j = 1, \ldots, n$ e todo i satisfazendo: $i \in PPL_j$ ou i é sucessor direto de j; *CheckFeasibilityOfInsert(s, i, j)* retorna *viável*.

- $N_{IMPSWP}(s)$ é o conjunto de todas as soluções obtidas a partir de s pela aplicação do movimento *Swap* a todos os pares (i, j) para um único j escolhido aleatoriamente e todos os i's satisfazendo: $i \in PPL_j$; *CheckFeasibilityOfSwap(s, i, j)* retorna *viável*.

Desta forma, se $N(s)$ for respectivamente igual a $N_{IMPJ}(s)$, $N_{IMPCP}(s)$, $N_{IMPALL}(s)$ e $N_{IMPSWP}(s)$ nos processo IMPJ, IMPCP, IMPALL e IMPSWP, a busca local executada por cada um destes agentes de melhoria pode ser dada por:

"Enquanto existir $s' \in N(s)$ tal que $c(s') < c(s)$ faça $s = s'$"

A Figura 5.13 traz o pseudo-código do algoritmo executado pelos agentes de melhoria do *A-Team* implementado para o SPLC. No passo 'd' deste algoritmo, $N(s)$ é respectivamente igual a $N_{IMPJ}(s)$, $N_{IMPCP}(s)$, $N_{IMPALL}(s)$ e $N_{IMPSWP}(s)$ nos processos IMPJ, IMPCP, IMPALL e IMPSWP.

5.1.7 Resultados Computacionais

O *A-Team* proposto para o SPLC foi implementado na liguagem C++ e o código de todos os agentes foi compilado usando o compilador g++ da GNU. Para realizar as comunicações entre os vários processos desta estratégia paralela, foi usada a biblioteca PVM (*Parallel Virtual Machine*) versão 3.4.beta4. Os testes apresentados nesta seção foram feitos numa máquina SUNW,SPARC 1000, com 300MB de memória e oito processadores. Os resultados apresentados foram obtidos considerando $L = 18$ trabalhadores disponíveis a cada instante.

A configuração do *A-Team* para o SPLC aplicada a todas as instâncias de teste está esquematicamente representada na Figura 5.14.

Agente de Melhoria IMPJ, IMPCP, IMPALL ou IMPSWP

Enquanto (TRUE) faça
- a. Envie mensagem *REQUEST_ONE_SOLUTION* para o processo ATC solicitando uma solução completa.
- b. Aguarde mensagem *ONE_SOLUTION*(s) do processo ATC.
- c. Faça *improv* = FALSE.
- d. Enquanto exisir $s' \in N(s)$ tal que $c(s') < c(s)$ faça
  - $s = s'$.
  - *improv* = TRUE.
- e. Se (*improv* == TRUE)
  - Envie mensagem *CANDIDATE*($c(s)$) para o processo ATC.
  - Aguarde mensagem *ACCEPT_CANDIDATE*(*accept*) do processo ATC.
  - Se (*accept* == TRUE)
    - Então envie mensagem *SOLUTION*(s) para o processo ATC

Figura 5.13: Algoritmo executado pelos agentes de melhoria IMPJ, IMPCP, IMPALL e IMPSWP do *A-Team* implementado para o SPLC.

Os primeiros experimentos realizados com o *A-Team* implementado neste trabalho foram para o ajuste do tamanho da memória de soluções completas (CSM_SIZE), do limite γ do critério de aceitação de soluções nesta memória e do limite do número máximo (MAX_SOLUTIONS) de soluções que podem ser inseridas na memória de soluções completas. Este último valor determina o instante em que o *A-Team* deve terminar.

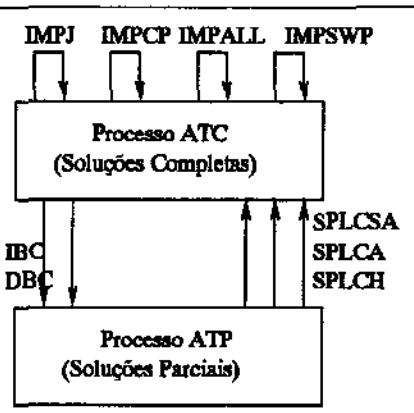


Figura 5.14: Configuração final do *A-Team* aplicado ao SPLC.

Após os testes preliminares, foram adotados os seguintes valores: CSM_SIZE = 100; γ = 10% e MAX_SOLUTIONS = 15000 soluções. Estes números foram escolhidos porque conseguiram manter diversidade dentro da memória de soluções completas ao mesmo tempo em que possibilitaram uma

Instância	*Makespan*	Segundos
Ins_4o_21j_A	82	< 1
Ins_4o_23j_A	58	< 1
Ins_4o_24j_A	68	24
Ins_4o_24j_B	72	9
Ins_4o_27j_A	67	16
Ins_6o_41j_A	143	568
Ins_6o_41j_B	111	451
Ins_6o_41j_C	126	164
Ins_6o_44j_A	116	334
Ins_6o_44j_B	137	296
Ins_8o_63j_A	259	791
Ins_8o_63j_B	316	1023
Ins_8o_63j_C	301	288
Ins_8o_65j_A	403	1248
Ins_8o_65j_B	382	1379
Ins_10o_84j_A	641	1456
Ins_10o_84j_B	567	2652
Ins_10o_85j_A	793	3163
Ins_10o_87j_A	585	2254
Ins_10o_88j_A	456	998
Ins_10o_100j_A	1467	2845
Ins_10o_102j_A	1158	4225
Ins_10o_106j_A	1098	4880
Ins_12o_108j_A	1277	3847
Ins_12o_109j_A	1336	3282

Tabela 5.1: Melhores resultados obtidos com o *A-Team* para o SPLC.

convergência deste conjunto de soluções para soluções de qualidade gradativamente melhores. Além disso, com estes valores observou-se um bom compromisso entre o tempo computacional levado pelo algoritmo paralelo e a qualidade das soluções finais obtidas.

A Tabela 5.1 apresenta os melhores resultados obtidos com a aplicação do *A-Team* a cada uma das instâncias de teste do SPLC. Devido às componentes aleatórias intrínsecas a esta estratégia paralela, cada um dos valores mostrados corresponde à melhor solução obtida em três execuções. Ao lado do valor de cada solução aparece o tempo de CPU gasto para chegar até ela. Este tempo foi medido como o tempo que o processo ATC levou do início da sua execução até o instante em que recebeu a referida solução.

A Figura 5.15 traz um gráfico que mostra, em percentagem, o quanto a pior solução e a solução mediana obtidas para cada instância (com três execuções do *A-Team* para o SPLC) estão acima

da correspondente melhor solução na Tabela 5.1. Como pode ser verificado, em nenhum caso a pior solução foi mais que 1.5% maior do que a melhor solução obtida pelo *A-Team*.

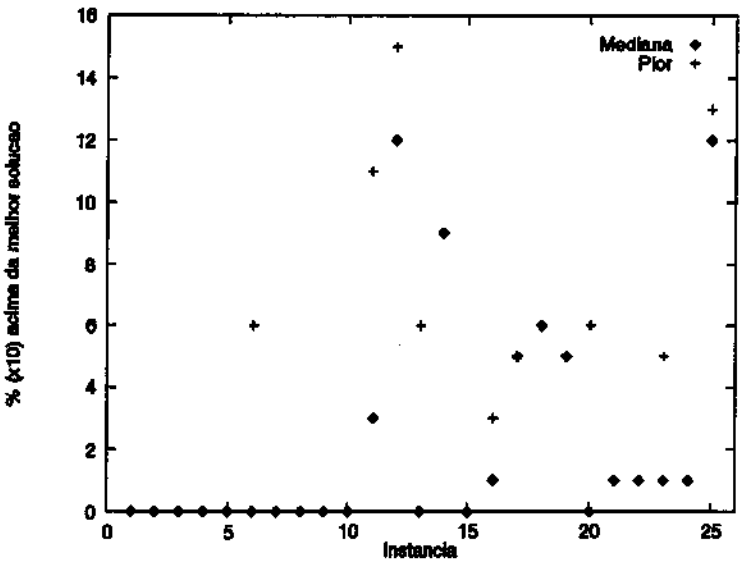


Figura 5.15: Relação entre a melhor, a mediana e a pior solução obtidas em três execuções do *A-Team* para o SPLC.

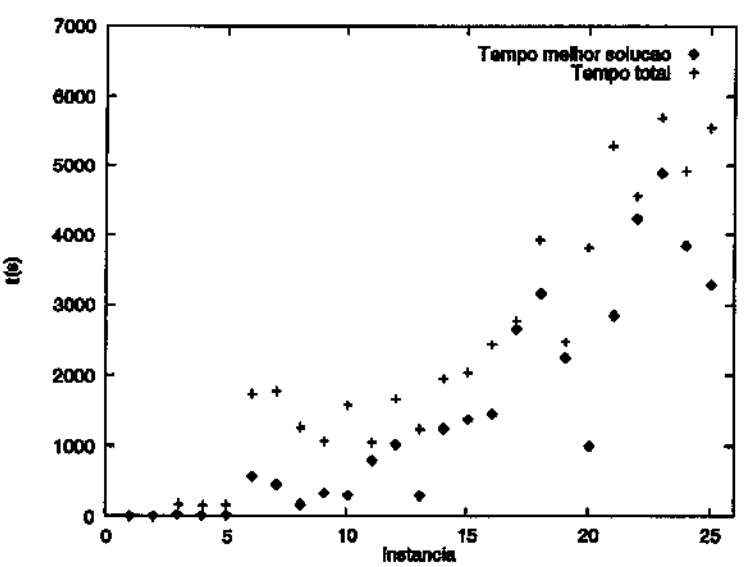


Figura 5.16: Relação entre o tempo de CPU gasto para chegar na melhor solução e o tempo total gasto pelo *A-Team* para o SPLC.

A Figura 5.16 traz um gráfico que ilustra a relação entre o tempo de CPU levado para chegar na melhor solução e o tempo de CPU total gasto até o término do *A-Team*. Este último foi medido como o tempo total de CPU gasto pelo processo ATC. Para cada instância, os valores mostrados correspondem aos tempos obtidos na execução que levou aos resultados da Tabela 5.1.

A análise dos resultados obtidos com o *A-Team* para o SPLC permite fazer os seguintes comentários:

- As melhores soluções obtidas com esta estratégia paralela de solução para o SPLC são, em dezessete das vinte e cinco instâncias, iguais ou melhores do que as melhores soluções obtidas por alguma das heurísticas seqüenciais do Capítulo 4.
- A única heurística seqüencial descrita naquele capítulo que não é completamente dominada pelo *A-Team* implementado para o SPLC é o algoritmo de busca tabu TSSPLC (Seção 4.4.2). Mesmo assim, apenas em oito das vinte e cinco instâncias o *A-Team* chegou numa solução de *makespan* maior do que a melhor obtida por aquele algoritmo seqüencial.

Embora não seja possível avaliar a qualidade das soluções obtidas isoladamente pelos agentes de melhoria e de consenso do *A-Team* implementado para o SPLC, se for feita uma comparação entre as soluções obtidas pelas heurísticas seqüenciais SPLCSA, SPLCH e SPLCA (Seção 4.2) usadas nos agentes de construção, com as soluções obtidas pelo *A-Team*, pode-se facilmente perceber que estas últimas são significativamente melhores. Isto mostra que o trabalho conjunto e cooperativo de vários algoritmos diferentes potencialmente permite encontrar soluções de qualidade melhor do que as obtidas pelos algoritmos isoladamente.

A próxima seção descreve a segunda estratégia paralela implementada para o SPLC neste trabalho. Trata-se de uma estratégia paralela e assíncrona de busca tabu que, em última instância, pode ser vista como um *A-Team* onde todos os agentes são uma busca tabu e onde toda comunicação entre estes agentes acontece através de uma única memória compartilhada.

5.2 Estratégia Paralela e Assíncrona de Busca Tabu para o SPLC

Várias são as possibilidades de paralelização de uma busca tabu. Diferentes estratégias paralelas desta metaheurística surgem dependendo de como são resolvidas as seguintes questões: "Como o espaço de soluções será dividido entre os vários processos? Que etapas da busca estarão alocadas a cada processo? Como será armazenado o conhecimento adquirido durante a busca? Que informações serão trocadas entre os processos e quando acontecerá esta comunicação?"

Tentando capturar todos estes fatores, Crainic, Toulouse e Gendreau em [CTG93] introduzem uma taxonomia para classificação dos procedimentos paralelos de busca tabu. Para situar conceitualmente a estratégia proposta para o SPLC, a Seção 5.2.1 a seguir apresenta resumidamente esta taxonomia. Em seguida, as Seções 5.2.2 e 5.2.3 descrevem os processos que compõem o algoritmo paralelo de busca tabu para o SPLC. A Seção 5.2.4, por fim, traz os resultados computacionais obtidos com esta estratégia.

5.2.1 Uma Taxonomia para Estratégias Paralelas de Busca Tabu

São três as dimensões consideradas na taxonomia de Crainic, Toulouse e Gendreau [CTG93]: (i) número de processos que controlam a busca (1-controle ou p-controle); (ii) tipos de controle e comunicação adotados no algoritmo paralelo (sincronização rígida, sincronização com conhecimento, colegial e colegial com conhecimento); e (iii) diferenciação das estratégias de busca (SPSS, SPDS, MPSS, MPDS). Cada uma destas dimensões é descrita a seguir.

Cardinalidade do Controle

O controle de uma busca tabu paralela pode ficar com um único processo ou ser distribuído entre vários processos. Assim, duas categorias são definidas:

- **1-controle**: Estratégias paralelas de busca tabu onde um processo principal (*mestre*) basicamente executa uma busca tabu seqüencial, transferindo para outros processos (*escravos*) apenas as tarefas que exigem alto tempo computacional. O processo *mestre* é responsável por distribuir as tarefas entre os processos *escravos*, recolher os resultados e determinar o fim do algoritmo.
- **p-controle**: Pertencem a esta categoria estratégias paralelas de busca tabu caracterizadas pelo uso de $p > 1$ processos que controlam simultaneamente p trajetórias de busca. Cada processo é responsável pela execução da própria busca e por estabelecer a comunicação com os outros processos. A busca global é encerrada quando cada um dos seus p processos termina.

Tipos de Controle e Comunicação

Esta dimensão da taxonomia considera a comunicação e a sincronização entre os processos, assim como a maneira como são processadas e compartilhadas as informações adquiridas pela busca tabu paralela. São quatro as categorias definidas sob este critério: *sincronização rígida*, *sincronização com conhecimento*, *colegial* e *colegial com conhecimento*. As duas primeiras estão relacionadas a processos síncronos e as duas últimas a processos assíncronos.

- **Sincronização rígida**: Esta categoria tipicamente inclui estratégias do tipo *mestre-escravo* (1-controle), onde toda informação sobre a trajetória da busca é mantida e manipulada exclusivamente pelo processo mestre. Não há comunicação entre os processos escravos, apenas entre cada um deles e o processo mestre em momentos específicos determinados por este último.

  No caso de estratégias p-controle, pertencem a esta categoria as buscas tabu paralelas formadas por p processos de busca tabu que exploram simultaneamente o espaço de soluções sem que haja qualquer tipo de comunicação ou interação entre eles. A busca tabu paralela

termina quando cada uma das p buscas independentes tiver atingido o seu próprio critério de parada.

- **Sincronização com conhecimento:** Estão nesta classe buscas tabu paralelas síncronas onde ocorre um pouco mais de comunicação e troca de conhecimentos entre os processos.

  Em estratégias 1-controle com sincronização com conhecimento, o processo mestre continua sincronizando e coordenando o trabalho dos processos escravos, que também não se comunicam entre si. A diferença entre abordagens 1-controle com sincronização rígida e 1-controle com sincronização com conhecimento está na complexidade do trabalho delegado aos processos escravos. No caso de sincronização com conhecimento, estes processos executam tarefas mais difíceis e podem até mesmo usar estruturas locais de memória para executar algumas iterações da busca tabu sobre uma região do espaço de soluções.

  Abordagens p-controle com sincronização com conhecimento correspondem a estratégias onde p processos de busca exploram simultaneamente o espaço de soluções parando em momentos pré-determinados (idênticos para todos os processos) para um etapa de comunicação intensiva e troca de informações entre eles. Logo após estes instantes, os processos de busca reassumem suas trajetórias até que uma nova etapa de comunicação seja atingida ou até que a busca global chegue ao fim.

- **Colegial:** Estratégias p-controle onde cada um dos p processos realiza uma busca tabu (com parâmetros eventualmente diferentes) em uma parte do ou em todo domínio. Quando um processo encontra uma solução que minimiza a função objetivo, ele a envia para todos ou para alguns dos demais processos da busca paralela. Alternativamente, esta nova solução pode ser armazenada em um processo central e os demais processos apenas são informados de que ela se encontra disponível. Em qualquer um dos casos, as comunicações são simples e a informação recebida por um processo é idêntica à informação enviada por algum dos demais processos.

- **Colegial com conhecimento:** Estratégias p-controle onde as informações obtidas individualmente por cada processo de busca são analisadas com o objetivo de se tirar conclusões sobre a trajetória global da busca e sobre as características das soluções de alta qualidade. Estas informações adicionais são repassadas aos processos de busca que podem, então, usá-las numa exploração mais eficiente do espaço de soluções. Neste caso, a informação recebida por um processo é resultado da análise de soluções visitadas pelos demais processos e, portanto, é mais rica em conteúdo do que a informação enviada por cada processo individual.

Estratégia de Diferenciação da Busca

A terceira e última dimensão da taxonomia diz respeito ao número de diferentes soluções iniciais e de estratégias (restrições tabu, prazo tabu e soluções candidatas, por exemplo) utilizadas em cada processo da busca paralela. Quatro categorias são definidas:

- **SPSS** (*Single Point Single Strategy*): Estratégias síncronas 1-controle onde uma única solução inicial e uma única estratégia de busca são adotadas.
- **SPDS** (*Single Point Different Strategies*): Abordagens 1-controle ou p-controle onde os processos de busca partem de uma mesma solução inicial mas são guiados por diferentes estratégias na exploração do espaço de soluções.
- **MPSS** (*Multiple Point Single Strategy*): Abordagens p-controle onde os processos de busca partem de diferentes soluções iniciais mas são guiados pela mesma estratégia na exploração do espaço de soluções.
- **MPDS** (*Multiple Point Different Strategies*): Abordagens p-controle onde os processos de busca partem de diferentes soluções iniciais e são guiados por diferentes estratégias na exploração do espaço de soluções.

Aplicações de estratégias paralelas de busca tabu vêm sendo cada vez mais exploradas na literatura. Em muitos casos, ainda, tratam-se de abordagens síncronas para problemas combinatórios como: localização-alocação de comodidades múltiplas [CTG95], escalonamento de tarefas com restrições de precedência em processadores heterogêneos [PR95, PR96] e escalonamento de tarefas (*job shop scheduling problem*) [Tai94].

De acordo com Crainic, Toulouse e Gendreau [CTG93], contudo, estratégias paralelas e assíncronas de busca tabu são muito promissoras e, em seus experimentos, mostraram-se superiores às abordagens síncronas tanto em qualidade de solução como em tempo de processamento.

Confirmando a potencialidade deste tipo de busca tabu paralela, Aiex em [Aie96] propõe uma estratégia paralela e assíncrona de busca tabu para o problema de particionamento de circuitos. Os resultados obtidos com o seu algoritmo paralelo são melhores do que qualquer outro conhecido na literatura para o mesmo problema.

As observações de Crainic, Toulouse e Gendreau e o sucesso do trabalho de Aiex serviram como estímulo para integrar em uma única estratégia paralela as oito configurações do algoritmo seqüencial de busca tabu implementado para o SPLC (Seção 4.4.2). De maneira similar à proposta por Aiex para o problema de particionamento de circuitos [Aie96], a estratégia paralela e assíncrona de busca tabu proposta para o SPLC neste trabalho enquadra-se na classe p-controle colegial com diferenciação da busca tipo MPDS.

Nesta abordagem, os processos de busca partem de soluções iniciais distintas e são guiados por estratégias diferentes na exploração do espaço de soluções. Quando um processo de busca encontra uma solução que minimiza a função objetivo, ele a envia para um processo central que armazena as soluções de qualidade obtidas pelas várias buscas. A cooperação entre os processos de busca é feita por intermédio do processo central e consiste basicamente na utilização de uma solução obtida por um processo de busca para iniciar uma etapa de intensificação em

outro processo. A necessidade de um processo de busca trocar informações com o processo central é determinada apenas pelo seu estado corrente e, desta forma, toda comunicação acontece assincronamente.

Fazendo uma comparação desta estratégia paralela e assíncrona de busca tabu com um *A-Team*, pode-se dizer que o processo central desempenha o papel de uma memória compartilhada de soluções completas e cada um dos processos de busca corresponde a um agente de melhoria.

As Seções 5.2.2 e 5.2.3, a seguir, detalham os processos de busca e o processo central que compõem a estratégia paralela e assíncrona de busca tabu proposta para o SPLC. A Figura 5.17 ilustra esquematicamente esta estratégia.

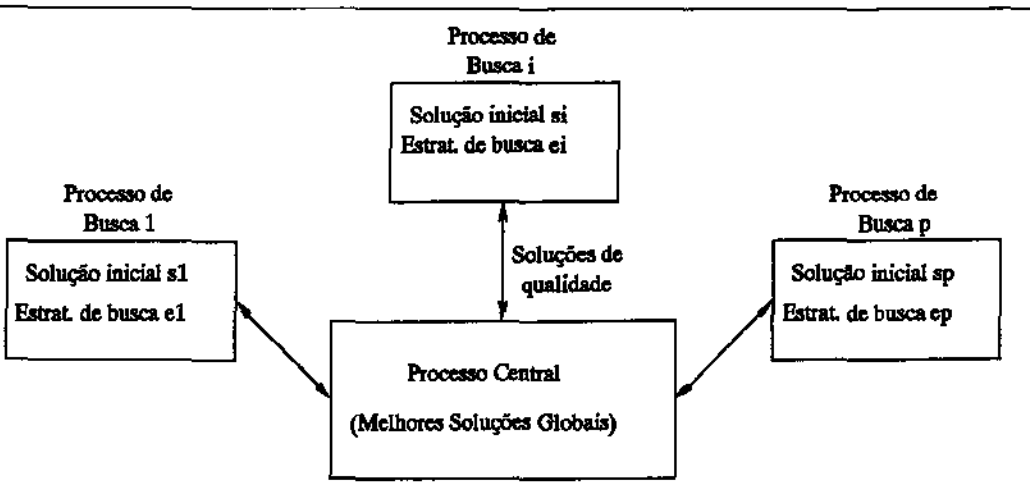


Figura 5.17: Esquema da estratégia paralela e assíncrona de buca tabu para o SPLC.

5.2.2 Processos de Busca

Um processo de busca na estratégia paralela de busca tabu proposta para o SPLC consiste essencialmente no algoritmo TSSPLC (Seção 4.4.2). Algumas modificações, contudo, foram introduzidas para permitir a sua comunicação com o processo central.

São oito os processos de busca utilizados na estratégia paralela. Cada um deles corresponde a uma das oito configurações do algoritmo TSSPLC (Tabela 4.1). Estes processos são diferenciados pelos tipos de soluções candidatas, restrição tabu e prazo tabu adotatos, além da solução inicial (*Best_PRH, Best_SSH ou Sol_PLH*) e da função objetivo (F_1 ou F_2) (Seção 4.4.2) usadas em cada um deles.

Os processos de busca são criados pelo processo central e cada um deles é responsável pela sua própria execução. Embora exista cooperação entre eles, eles não se comunicam diretamente: toda a troca de informações entre os processos de busca acontece por intermédio do processo

central.

São três as situações em que um processo de busca se comunica com o processo central: quando ele quer enviar uma solução de qualidade, quando ele precisa receber uma solução diferente das que ele já visitou para iniciar uma etapa de intensificação e, por fim, para comunicar o término da sua busca. A seguir são detalhados cada um destes aspectos e apresentadas as modificações introduzidas no algoritmo TSSPLC para permitir a interação dos processos de busca com o processo central.

Envio de Solução

Na busca tabu paralela proposta para o SPLC, os processos de busca enviam soluções para o processo central que as armazena junto com outras soluções de qualidade obtidas pelos demais processos.

Uma solução encontrada por um processo de busca é dita de qualidade se ela é um mínimo local de custo inferior ao custo c^* da melhor solução conhecida até então por este processo. Apenas soluções de qualidade são enviadas ao processo central.

A identificação de soluções de qualidade na trajetória de um processo de busca é feita da seguinte forma: define-se uma variável booleana *hillDescending* que assume valor TRUE sempre que uma solução s' de custo inferior a c^* for encontrada. A busca continua normalmente a partir de s' até que seja alcançada uma solução de custo superior ao da solução corrente. Neste instante, a solução corrente corresponde a uma solução de qualidade que deve ser enviada ao processo central e *hillDescending* volta a assumir valor FALSE, indicando que um mínimo local foi atingido.

Quando um processo de busca identifica uma solução de qualidade ele não a envia diretamente para o processo central. Primeiramente, ele lhe manda uma mensagem contendo o custo da solução encontrada e fica aguardando uma resposta. O processo central pode aceitar ou não receber a solução de qualidade. Isto depende do seu critério de aceitação de novas soluções que será discutido na Seção 5.2.3. Apenas no caso de uma resposta positiva do processo central é que o processo de busca lhe envia efetivamente a solução de qualidade encontrada.

A Figura 5.18 descreve o procedimento usado pelos processos de busca para o envio de soluções de qualidade para o processo central.

Solicitação de Solução

Com o objetivo de introduzir uma etapa de intensificação na sua trajetória, um processo de busca periodicamente solicita ao processo central uma nova solução de qualidade para reiniciar a sua busca. Duas são as situações que esta solicitação acontece: (i) após um número máximo de iterações (MAX_BAD_MOVES) sem que o processo de busca consiga melhorar a sua melhor solução; ou (ii) após um número máximo de iterações (MAX_NOT_INTENSIFIED_ITER) desde a última solicitação não atendida.

Procedimento *SendQualitySolution*(s)

1. Envie mensagem *CANDIDATE*($c(s)$) para o processo central.
2. Aguarde mensagem *ACCEPT_CANDIDATE*(*accept*) do processo central.
3. Se (*accept* == TRUE)
   Então Envie mensagem *SOLUTION*(s) para o processo central.

Figura 5.18: Procedimento usado pelos processos de busca para enviar ao processo central uma solução de qualidade.

A solicitação de uma solução é feita nos seguintes passos: o processo de busca envia uma mensagem para o processo central requisitando uma nova solução e fica esperando uma resposta. Ao receber a mensagem, o processo central verifica, entre as soluções de qualidade que ele tem guardadas, se existe alguma que não tenha sido obtida pelo processo de busca solicitante nem tenha sido enviada a ele em uma etapa de intensificação anterior. O processo central responde, então, se é possível ou não atender à solicitação do processo de busca. Em caso positivo, logo em seguida o processo de busca recebe uma nova solução e a transforma na sua solução corrente. A solução recebida é ainda comparada à melhor solução s^* conhecida pelo processo de busca e pode substituí-la caso seu custo seja inferior ao custo de s^*. A memória do processo de busca é reinicializada e a sua busca prossegue normalmente.

Caso o processo central não consiga atender à solicitação feita, o processo de busca reassume a sua trajetória do ponto onde tinha parado e continua as suas iterações até que a próxima solicitação possa ser feita ou até que algum dos seus critérios de parada seja atingido.

A Figura 5.19 descreve o procedimento usado pelos processos de busca para pedir uma nova solução ao processo central.

Critérios de Parada

Dois critérios de parada locais são utilizados para finalizar a execução de um processo de busca:

- O número máximo de MAX_TOTAL_ITER iterações é atingido.
- O processo de busca, por duas vezes consecutivas, não consegue receber do processo central uma nova solução e no intervalo entre estas duas solicitações também não consegue melhorar a sua melhor solução.

Quando um destes critérios é atingido, o processo de busca envia uma mensagem ao processo central para comunicar a sua finalização. A função de pós-otimização do algoritmo TSSPLC (*FinalTuning*) não é chamada pelos processos de busca. Ela foi transferida para o processo central como será explicado na próxima seção.

Procedimento *RequestSolution*

1. Envie mensagem *REQUEST_SOLUTION* para o processo central solicitando uma nova solução.
2. Aguarde mensagem *REQUEST_ANSWER*($exist$) do processo central.
3. Se ($exist$ == TRUE)
   Então Aguarde mensagem *NEW_SOLUTION*(s') do processo central.
   Faça $s = s'$; $c(s) = c(s')$.
   Reinicialize a memória de curto prazo *movimentos_tabu*.
   Se $c(s') < c^*$
   Então $s^* = s'$; $c^* = c(s')$.
   $requestAccepted$ = TRUE.
   $nBadMoves = 0$.
   Senão /* O processo central não conseguiu atender à solicitação do processo de busca */
   Se (($requestAccepted$ == FALSE) e ($improveOnBestSolution$ == FALSE))
   Então /* a solicitação anterior foi recusada e desde então o processo de busca
   não melhorou a sua melhor solução */
   $okIntensification$ = FALSE. /* o processo de busca deve acabar */
   Senão /* o processo de busca continua da mesma solução */
   $requestAccepted$ = FALSE;
4. Faça $improveOnBestSolution$ = FALSE;
   $nIterSinceLastRequest = 0$.

Figura 5.19: Procedimento usado pelos processos de busca para solicitar ao processo central uma nova solução.

A Figura 5.20 traz o pseudo-código do algoritmo TSSPLC modificado e executado por cada processo de busca.

5.2.3 Processo Central

O processo central na estratégia paralela de busca tabu proposta para o SPLC é o correpondente do processo ATC do *A-Team*. Assim, são quatro as funções básicas deste processo: (i) criar cada um dos processos de busca; (ii) gerenciar a memória que contém as soluções de qualidade obtidas por estes processos; (iii) atuar como intermediário na troca de informações entre eles; e (iv) coordenar o término da busca global.

Enquanto os vários processos de busca exploram o espaço de soluções, basicamente o processo central permanece num laço aguardando a chegada de mensagens. São quatro os tipos de mensagens recebidas pelo processo central:

- *CANDIDATE*: avisando que um processo de busca tem uma nova solução de qualidade;
- *SOLUTION*: contendo uma solução de qualidade obtida por um um processo de busca;
- *REQUEST_SOLUTION*: quando um processo de busca solicita uma solução de qualidade para iniciar uma etapa de intensificação;
- *END_SEARCH_PROCESS*: quando um processo de busca é concluído.

Como cada uma destas mensagens é tratada e os principais aspectos do processo central são apresentados a seguir.

Criação dos Processos de Busca

São oito os processos de busca na estratégia paralela de busca tabu proposta para o SPLC. O primeiro passo do processo central é a criação de cada um deles passando como parâmetro a estratégia de busca (restrição tabu, prazo tabu e tipo de soluções candidatas) que deve ser utilizada, a solução inicial para este processo e a função objetivo que deve ser usada ao longo da exploração do espaço de soluções.

Cada um dos processos de busca é criado em correspondência biunívoca a uma das oito configurações do algoritmo TSSPLC (Tabela 4.1).

Armazenamento das Soluções de Qualidade

Sempre que uma solução de qualidade é encontrada por um processo de busca, ela é enviada ao processo central onde fica armazenada para eventualmente ser usada em uma etapa de intensificação de outro processo.

Processo de Busca

1. Gere solução inicial s^0.
2. Inicialize a memória de curto prazo *movimentos_tabu*.
3. Faça $s = s^0$; $s^* = s^0$; $c^* = c(s^0)$.
4. Faça $nIter = 0$; $nBadMoves = 0$; $nIterSinceLastRequest = 0$;
   $improveOnBestSolution$ = FALSE; $okIntensification$ = TRUE;
   $requestAccepted$ = TRUE; $hillDescending$ = FALSE.
5. Enquanto (($nIter <$ MAX_ITER) e ($okIntensification ==$ TRUE)) faça
   a. Determine soluções candidatas $CS(s)$.
   b. $c' = \infty$
   c. Para toda solução $\overline{s} \in CS(s)$ faça
      Se (($\overline{s}$ não foi obtida por um movimento tabu) ou ($c(\overline{s}) < c^*$))
      Então Se ($c(\overline{s}) < c'$)
      Então $c' = c(\overline{s})$ e $s' = \overline{s}$.
   d. Se ($c' > c(s)$)
      Então atualize *movimentos_tabu*.
   e. Se (($hillDescending ==$ TRUE) e ($c' > c(s)$))
      Então *SendQualitySolution*(s);
      $hillDescending$ = FALSE.
   f. Se ($c' < c^*$)
      Então $nBadMoves = 0$;
      $s^* = s'$; $c^* = c'$;
      $hillDescending$ = TRUE;
      $improveOnBestSolution$ = TRUE.
      Senão $nBadMoves++$.
   g. Se (($nBadMoves ==$ MAX_BAD_MOVES) ou
      (($requestAccepted ==$ FALSE) e ($nIterSinceLastRequest ==$ MAX_NOT_INTENSIFIED_ITER)))
      Então *RequestNewSolution*( ).
      Senão $s = s'$.
   h. Faça $nIter++$;
      $nIterSinceLastRequest++$.
6. Enviar mensagem *END_SEARCH_PROCESS* para o processo central.

Figura 5.20: Algoritmo executado pelos processos de busca tabu da estratégia paralela de busca tabu para o SPLC.

Para guardar as soluções de qualidade, o processo central mantém uma lista chamada de *pool*. Inicialmente vazia, esta lista vai crescendo com a inserção das soluções de qualidade enviadas pelos vários processos de busca. As soluções são mantidas no *pool* em ordem crescente dos seus custos e é adotado um limite POOL_SIZE no número máximo de soluções que podem ser armazenadas. Além disso, não são permitidas soluções repetidas. Caso uma mesma solução de qualidade seja enviada ao processo central por mais de um processo de busca, apenas uma cópia dela fica armazenada no *pool*.

Cada elemento da lista *pool* contém o custo e a seqüência representativa de uma solução, além de um vetor de processos proibidos, *forbiddenProcessArray*, que marca os processos que não podem receber a solução correspondente em uma etapa de intensificação.

O vetor de processos proibidos é usado para manter controle sobre os processos que enviaram ou que já receberam cada solução do *pool*. O objetivo é evitar que um processo de busca seja reinicializado a partir de uma solução que ele mesmo gerou ou a partir da qual já iniciou uma etapa de intensificação anterior. Quando uma solução é inserida no *pool*, o seu vetor de processos proibidos tem apenas uma posição marcada: a do processo que a enviou. O processo central é responsável por atualizar este vetor sempre que a solução correspondente for obtida por ou enviada a um processo de busca diferente.

Recebimento de Solução

Quando um processo de busca encontra uma solução de qualidade, ele envia uma mensagem (*CANDIDATE*) para o processo central contendo o custo desta solução. Ao receber a mensagem, o processo central decide se vai aceitar a nova solução com base no seguinte **critério de aceitação de soluções**:

> "Somente serão aceitas soluções de qualidade cujo custo seja no máximo $\gamma\%$ maior do que o custo da melhor solução do *pool*."

O valor γ é chamado de *limite percentual do critério de aceitação*. O objetivo deste critério é evitar que soluções com qualidade muito inferior à da melhor solução do *pool* sejam consideradas.

Uma vez avaliado o custo da solução candidata de acordo com o critério de aceitação de soluções, o processo central responde ao processo de busca dizendo se aceita ou não receber a nova solução. Em caso positivo, o processo central fica aguardando para efetivamente recebê-la.

Ao receber uma solução de qualidade, o processo central verifica se ela já não existe no seu *pool* de soluções. Caso exista, ele apenas atualiza o vetor de processos proibidos a ela associado. Caso contrário, ele a insere no *pool* mantendo a ordenação pelo custo das soluções. Se o limite POOL_SIZE do tamanho do *pool* tiver sido atingido, a solução de pior custo é eliminada para dar lugar à solução a ser inserida. Isto é feito mesmo que a nova solução tenha custo superior ao da solução eliminada. O objetivo é possibilitar uma maior renovação das soluções do *pool* de forma

Procedimento *TreatCandidateMessage*$(c(s'), pId)$

1. Se $(c(s') \leq (1+\gamma)c_p^*)$
   Então *accept* = TRUE.
   Senão *accept* = FALSE.
2. Envie mensagem *ACCEPT_CANDIDATE*(*accept*) para o processo de busca pId.
3. Se (*accept* == TRUE)
   Então *ReceiveSolution*(pId).

Figura 5.21: Procedimento usado pelo processo central para tratar mensagens do tipo *CANDIDATE*.

Procedimento *ReceiveSolution*(pId)

1. Aguarde mensagem *SOLUTION*(s) do processo de busca pId.
2. Se s for idêntica a alguma solução s' do *pool*
   Então atualize o vetor de processos proibidos de s': $s'.forbiddenProcess[pId] = 1$.
   Senão Se ($nSolPool$ == POOL_SIZE) /* *pool* está cheio */
   Então elimine a solução de pior custo do *pool*.
   Insira s no *pool*.
   Faça $s.forbiddenProcess[pId] = 1$.

Figura 5.22: Procedimento usado pelo processo central para receber soluções de qualidade dos processos de busca.

que as solicitações de solução pelos processos de busca possam ser atendidas.

As Figuras 5.21 e 5.22 apresentam, respectivamente, o pseudo-código dos procedimentos usados pelo processo central para tratar mensagens do tipo *CANDIDATE* e para receber soluções de qualidade enviadas por um processo de busca.

Envio de Solução

Quando um processo de busca quer iniciar uma etapa de intensificação, ele envia uma mensagem (*REQUEST_SOLUTION*) ao processo central solicitando uma nova solução. Ao receber a mensagem, o processo central verifica se no seu *pool* de soluções existe uma que possa ser usada para atender à solicitação feita.

Uma solução do *pool* só pode ser enviada a um processo de busca se ela não tiver sido obtida por ele nem tiver sido enviada a ele em uma fase de intensificação anterior. Assim, para atender ao pedido de solução de um processo de busca, o processo central percorre o seu *pool* e marca a solução de menor custo que não é proibida ao processo solicitante. Se não existir tal solução, ou

Procedimento *TreatRequestSolution(pId)*

1. Selecione a solução s' de menor custo no *pool* tal que $s'.forbiddenProcess[pId] = 0$.
2. Se não esistir tal solução s'
   Então *exist* = FALSE.
   Senão *exist* = TRUE.
3. Envie mensagem *REQUEST_ANSWER(exist)* para o processo de busca *pId*.
4. Se (*exist* == TRUE)
   Então envie mensagem *NEW_SOLUTION*(s') para o processo de busca *pId*.
   Atualize o vetor de processos proibidos de s': $s'.forbiddenProcess[pId] = 1$.

Figura 5.23: Procedimento usado pelo processo central para para tratar a solicitação de solução feita por um processo de busca.

seja, se todas as soluções do *pool* não puderem ser enviadas ao processo solicitante, o processo central lhe responde avisando que não é possível atendê-lo. Caso contrário, comunica que a solicitação pode ser atendida e, em seguida, lhe envia uma nova solução.

Após enviar uma solução a um processo de busca, o processo central atualiza o vetor de processos proibidos a ela associado para evitar que ela seja enviada outra vez a este mesmo processo.

A Figura 5.23 apresenta o procedimento usado pelo processo central para tratar a solicitação de solução feita por um processo de busca.

Fase de Pós-Otimização e Critério de Finalização da Busca Global

A estratégia paralela de busca tabu para o SPLC chega ao fim quando cada um dos processos de busca termina. O processo central detecta este instante após receber de cada um destes processos uma mensagem de finalização. Tão logo isto aconteça, o processo central inicia uma fase de pós-otimização aplicando a cada uma das soluções do *pool* o procedimento *FinalTuning* do algoritmo TSSPLC (Figura 4.5).

O processo central termina logo após a etapa de pós-otimização e retorna a melhor solução obtida pela busca tabu paralela.

A Figura 5.24 apresenta o pseudo-código do algoritmo executado pelo processo central.

5.2.4 Resultados Computacionais

A estratégia paralela e assíncrona de busca tabu para o SPLC foi implementada na linguagem C++ e os códigos do processo central e dos processos de busca foram compilados usando o compilador g++ da GNU. Para realizar as comunicações entre os vários processos desta estratégia

Processo Central

1. Crie cada um dos oito processos de busca.
2. Faça $pool = \emptyset$;
   $nActiveProcess = 8$; /* processos de busca que ainda não terminaram */
   $c_p^* = \infty$. /* custo da melhor solução do *pool* */
3. Enquanto ($nActiveProcess > 0$) faça
   a. Aguarde até receber uma mensagem M de um dos processos de busca.
   b. Faça pId = identificador do processo que mandou M.
   c. Verifique o tipo da mensagem M
      Caso *CANDIDATE*($c(s')$):
         TreatCandidateMessage($c(s'), pId$).
      Caso *REQUEST_SOLUTION*:
         TreatRequestSolutionMessage(pId).
      Caso *END_SEARCH_PROCESS*:
         $nActiveProcess - -$.
4. Aplique o procedimento *FinalTuning* (Figura 4.5) a cada solução do *pool*.
5. Retorne a melhor solução conhecida.

Figura 5.24: Algoritmo executado pelo processo central da estratégia paralela de busca tabu para o SPLC.

paralela, foi usada a biblioteca PVM (*Parallel Virtual Machine*) versão 3.4.beta4. Os experimentos computacionais apresentados nesta seção foram realizados numa máquina SUNW,SPARC 1000, com 300MB de meória e oito processadores. Os resultados apresentados foram obtidos considerando $L = 18$ trabalhadores disponíveis a cada instante.

Inicialmente, foram feitos alguns testes com o tabu paralelo para ajustar o tamanho POOL_SIZE do *pool* de soluções mantidas pelo processo central; o limite percentual γ do critério de aceitação de soluções no *pool*; e os valores MAX_NOT_INTENSIFIED_ITER, MAX_BAD_MOVES e MAX_TOTAL_ITER usados pelos processos de busca para, respectivamente, determinar etapas de intensificação e o final da sua execução.

Com relação ao tamanho do *pool* de soluções, foram testados os valores 8, 16 e 32 (múltiplos do número de processos de busca no tabu paralelo implementado). O que se observou foi que, com POOL_SIZE = 8, o processo central freqüentemente não conseguia atender às solicitações de novas soluções por parte dos processos de busca, o que ocasionava um fim precoce do algoritmo paralelo. Com os valores 16 e 32, por outro lado, este problema pareceu contornado. Nos testes realizados com o tabu paralelo foi adotado POOL_SIZE = 16 pelo fato adicional deste valor acarretar um menor custo computacional, em relação POOL_SIZE = 32, para manutenção do *pool* de soluções.

Para o limite percentual γ do critério de aceitação de soluções no *pool*, foram testados os valores

10%, 25% e 50%. Na verdade, o que se observou foi que este parâmetro não tem impacto no desempenho do tabu paralelo implementado. Isto acontece porque como os processos de busca só enviam ao processo central soluções de qualidade, o custo das soluções enviadas por cada um deles é monotonicamente decrescente. Além disso, não há uma diferença significativa entre os custos das soluções de qualidade enviadas pelos vários processos de busca. Assim, mesmo para um valor pequeno de γ o que se observou foi que o processo central nunca recusa uma solução de qualidade obtida por um processo de busca. Os experimentos computacionais do tabu paralelo foram feitos, então, com $\gamma = 10\%$.

Os valores MAX_NOT_INTENSIFIED_ITER, MAX_BAD_MOVES e MAX_TOTAL_ITER usados pelos processos de busca foram fixados em 100, 100 e 1000, respectivamente, porque mostraram um bom nível de compromisso entre o tempo total gasto pela estratégia paralela e assíncrona de busca tabu e a qualidade da solução obtida.

Uma vez feitos os ajustes preliminares acima, foram realizadas duas baterias de teste com o algoritmo paralelo de busca tabu implementado para o SPLC. A diferença entre elas está na maneira como o processo central escolhe a solução inicial e a função objetivo que devem ser usadas em cada processo de busca. São duas as possibilidades:

- *Random Initialization*: Para cada processo de busca, o processo central escolhe aleatoriamente uma solução inicial do conjunto $\{Best\_PRH,\ Best\_SSH,\ Sol\_PLH\}$ e uma função objetivo do conjunto $\{F_1, F_2\}$.
- *Best Initialization*: Para cada processo de busca, o processo central escolhe a solução inicial e a função objetivo que levaram ao melhor resultado na Tabela 4.7 para a configuração do algoritmo TSSPLC correspondente a este processo de busca.

A Tabela 5.2 apresenta os resultados obtidos com o tabu paralelo para o SPLC. Para cada uma das estratégias de inicialização dos processos de busca, *Random* ou *Best*, os valores mostrados nesta tabela correspondem à melhor solução obtida em três execuções deste algoritmo. Ao lado de cada solução é mostrado o tempo de CPU gasto para chegar até ela. Este tempo foi obtido como o tempo de CPU gasto pelo processo de busca que primeiro encontrou esta solução, medido do início da execução deste processo até o instante em que a solução foi encontrada.

A Figura 5.25 traz um gráfico que mostra, em percentagem, o quanto a pior solução e a solução mediana obtidas para cada instância em três execuções do tabu paralelo para o SPLC estão acima da correspondente melhor solução na Tabela 5.2. Em todas estas execuções o algoritmo paralelo de busca tabu usou a estratégia de escolha de solução inicial e de função objetivo para os processos de busca que levou ao melhor resultado naquela tabela. Como pode ser verificado no gráfico apresentado, em nenhum caso a pior solução foi mais do que 1.5% maior do que a melhor solução obtida pelo tabu paralelo.

A Figura 5.26 traz um gráfico que ilustra a relação entre o tempo de CPU para chegar na

Instância	*Random Initialization*		*Best Initialization*	
	Makespan	Segundos	*Makespan*	Segundos
Ins_4o_21j_A	**82**	< 1	**82**	< 1
Ins_4o_23j_A	**58**	< 1	**58**	< 1
Ins_4o_24j_A	**68**	11	**68**	8
Ins_4o_24j_B	**72**	8	**72**	14
Ins_4o_27j_A	**67**	9	**67**	12
Ins_6o_41j_A	141	99	**140**	80
Ins_6o_41j_B	**110**	24	**110**	10
Ins_6o_41j_C	**126**	55	128	10
Ins_6o_44j_A	**117**	22	**117**	12
Ins_6o_44j_B	**137**	30	**137**	29
Ins_8o_63j_A	260	228	**259**	133
Ins_8o_63j_B	316	125	**314**	152
Ins_8o_63j_C	297	208	**294**	28
Ins_8o_65j_A	**406**	244	**406**	165
Ins_8o_65j_B	384	178	**383**	182
Ins_10o_84j_A	635	517	**634**	337
Ins_10o_84j_B	554	560	**550**	478
Ins_10o_85j_A	**783**	637	791	744
Ins_10o_87j_A	582	649	**581**	123
Ins_10o_88j_A	**450**	510	**450**	527
Ins_10o_100j_A	**1468**	593	**1468**	87
Ins_10o_102j_A	1158	1056	**1155**	291
Ins_10o_106j_A	**1087**	1471	**1087**	1277
Ins_12o_108j_A	**1271**	859	1275	1545
Ins_12o_109j_A	**1324**	381	1328	432

Tabela 5.2: Resultados obtidos com a estratégia paralela de busca tabu para o SPLC. A coluna "Segundos" mostra o tempo de CPU gasto para chegar na solução correspondente.

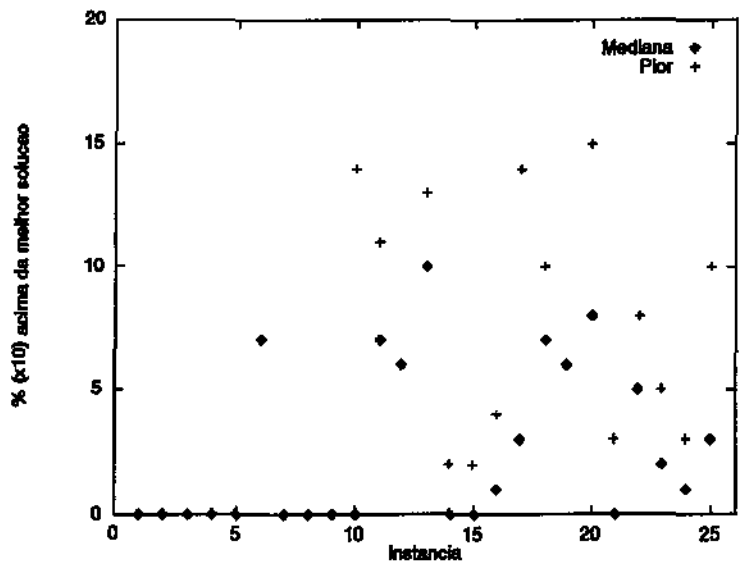


Figura 5.25: Relação entre a melhor, a mediana e a pior solução obtidas em três execuções do tabu paralelo para o SPLC.

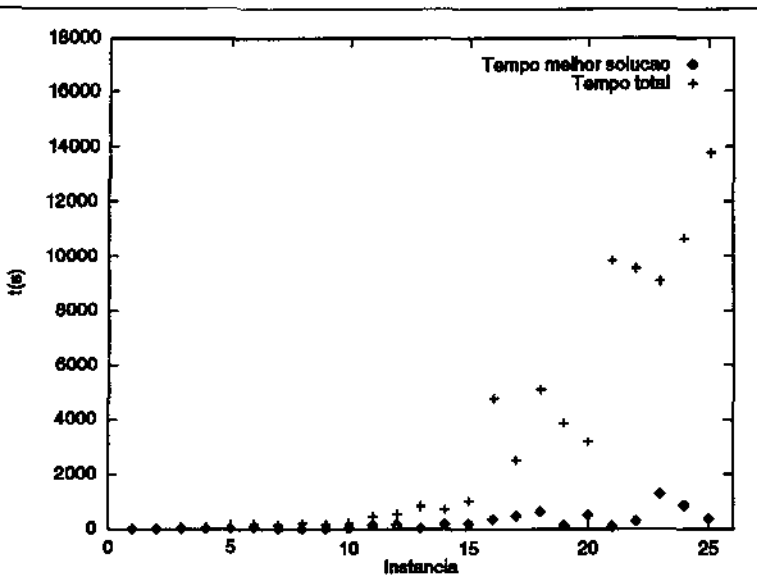


Figura 5.26: Relação entre o tempo de CPU gasto para chegar na melhor solução e o tempo total gasto pelo tabu paralelo implementado para o SPLC.

Instância	Tabu Seqüencial	Tabu Paralelo
Ins_6o_41j_A	**141**	**141**
Ins_8o_65j_B	**383**	**383**
Ins_10o_88j_A	465	**460**
Ins_10o_100j_A	1471	**1468**

Tabela 5.3: Comparação entre as melhores soluções obtidas em 5 minutos de execução do tabu paralelo e em 40 minutos de execução da tabu seqüencial implementados para o SPLC.

melhor solução e o tempo de CPU total gasto até o término da busca tabu paralela (dado pelo tempo máximo de CPU gasto por algum dos processos de busca). Para cada instância, os valores mostrados neste gráfico são os tempos de CPU obtidos na execução que levou ao melhor resultado para esta instância na Tabela 5.2. Note que, de acordo com este gráfico, um melhor critério de parada poderia ter sido desenvolvido para o tabu paralelo implementado para o SPLC. Sobretudo para as instâncias com mais de 80 jobs, o tempo total gasto por este algoritmo foi significativamente maior do que o tempo para chegar na melhor solução.

Uma análise dos resultados obtidos com a estratégia paralela e assíncrona de busca tabu implementada para o SPLC permite fazer os seguintes comentários:

- As melhores soluções obtidas pelo tabu paralelo dominam em qualidade aquelas obtidas pelo algoritmo seqüencial de busca tabu implementado para o SPLC. Na verdade, em quinze das vinte e cinco instâncias, a solução do tabu paralelo foi estritamente melhor do que a correspondente do tabu seqüencial.

- Mesmo limitando o tempo de execução do tabu paralelo a um oitavo do tempo de execução do tabu seqüencial, as soluções obtidas com a estratégia paralela ainda são pelo menos tão boas quanto as soluções do algoritmo seqüencial de busca tabu. Este fato pode ser constatado na Tabela 5.3 que traz para um subconjunto das instâncias de teste os resultados obtidos em cinco minutos de execução do tabu paralelo e em 40 minutos de execução do tabu seqüencial. Note que o produto "número de processos de busca x tempo de CPU", em teoria, é o mesmo nesta comparação entre o tabu seqüencial e o tabu paralelo. Isto dá uma idéia de que, usando aproximadamente a mesma quantidade de recursos computacionais, o esforço da implementação de uma busca tabu paralela vale a pena pela qualidade das soluções que podem ser obtidas com esta estratégia.

- Comparativamente com o *A-Team* implementado para o SPLC, o algoritmo paralelo de busca tabu dominou a qualidadade das soluções obtidas em vinte e uma das vinte e cinco instâncias de teste. Vale observar ainda que, nas únicas quatro instâncias em que o *A-Team* foi melhor do que o tabu paralelo, a solução obtida por este último não chega a ter *makespan* 1% maior do que a obtida pelo *A-Team*.

Instância	ξ_{TS_s}	ξ_{AT}	ξ_{TS_p}
Ins_4o_21j_A	**82**	**82**	**82**
Ins_4o_23j_A	**58**	**58**	**58**
Ins_4o_24j_A	**68**	**68**	**68**
Ins_4o_24j_B	**72**	**72**	**72**
Ins_4o_27j_A	**67**	**67**	**67**
Ins_6o_41j_A	141	143	**140**
Ins_6o_41j_B	**110**	111	**110**
Ins_6o_41j_C	128	**126**	**126**
Ins_6o_44j_A	117	**116**	117
Ins_6o_44j_B	**137**	**137**	**137**
Ins_8o_63j_A	261	**259**	**259**
Ins_8o_63j_B	316	316	**314**
Ins_8o_63j_C	296	301	**294**
Ins_8o_65j_A	406	**403**	406
Ins_8o_65j_B	384	**382**	383
Ins_10o_84j_A	636	641	**634**
Ins_10o_84j_B	556	567	**550**
Ins_10o_85j_A	791	793	**783**
Ins_10o_87j_A	582	585	**581**
Ins_10o_88j_A	460	456	**450**
Ins_10o_100j_A	1468	**1467**	1468
Ins_10o_102j_A	1166	1158	**1155**
Ins_10o_106j_A	1094	1098	**1087**
Ins_12o_108j_A	1277	1277	**1271**
Ins_12o_109j_A	1343	1336	**1324**

Tabela 5.4: Melhores soluções obtidas pelo tabu seqüencial, pelo *A-Team* e pelo tabu paralelo implementados para o SPLC.

A Tabela 5.4 traz lado a lado as melhores soluções obtidas pelo algoritmo seqüencial de busca tabu, pelo *A-Team* e pelo algorimo paralelo de busca tabu implementados para o SPLC. Pode-se ver por estes resultados que, juntas, as estratégias paralelas e assíncronas propostas e implementadas neste trabalho são as responsáveis pelas melhores soluções conhecidas para as instâncias de teste do SPLC. Note que o algoritmo seqüencial de busca tabu já dominava, na qualidade das soluções obtidas, as outras estratégias de solução disponíveis na literatura para o SPLC [HC97, SUW97].

Como desejado no início desta dissertação, fica comprovada então a adequabilidade do uso de estratégias paralelas e assíncronas na obtenção de soluções de boa qualidade para o problema de escalonamento com restrição de mão-de-obra. Vale observar, contudo, que com certeza ainda há muito para se aprender com o SPLC. Como pode ser verificado na Tabela 5.5, a distância entre

Instância	ξ^*	$\overline{\xi}^*$
Ins_4o_21j_A	82	82
Ins_4o_23j_A	58	58
Ins_4o_24j_A	68	68
Ins_4o_24j_B	72	72
Ins_4o_27j_A	67	67
Ins_6o_41j_A	112	140
Ins_6o_41j_B	102	110
Ins_6o_41j_C	110	126
Ins_6o_44j_A	98	116
Ins_6o_44j_B	124	137
Ins_8o_63j_A	191	259
Ins_8o_63j_B	239	314
Ins_8o_63j_C	271	294
Ins_8o_65j_A	342	403
Ins_8o_65j_B	315	382
Ins_10o_84j_A	394	634
Ins_10o_84j_B	355	550
Ins_10o_85j_A	671	783
Ins_10o_87j_A	379	581
Ins_10o_88j_A	362	450
Ins_10o_100j_A	830	1467
Ins_10o_102j_A	878	1155
Ins_10o_106j_A	609	1087
Ins_12o_108j_A	838	1271
Ins_12o_109j_A	980	1324

Tabela 5.5: Melhores limitantes inferiores e superiores conhecidos para o SPLC.

os melhores limitantes inferiores e os melhores limitantes superiores atualmente conhecidos para este problema ainda é significativa. Muito trabalho provavelmente será necessário até que se consiga aproximar significativamente, e quem sabe até igualar, estes valores.

O Capítulo 6, a seguir, formaliza as conclusões e contribuições desta dissertação e apresenta algumas possíveis linhas de pesquisa para trabalhos futuros com o SPLC.

Capítulo 6

Conclusão

Esta dissertação abordou o problema de escalonamento com restrição de mão-de-obra, SPLC. Após um estudo inicial das características deste problema e de técnicas já conhecidas na literatura para sua solução, foram apresentadas novas estratégias de solução para a obtenção de limitantes inferiores e superiores para o SPLC. As conclusões deste trabalho, apresentadas a seguir, aplicam-se ao conjunto de testes realizados com as instâncias *benchmarks* do SPLC.

No que diz respeito aos limitantes inferiores, foram três as abordagens implementadas neste trabalho, todas usando de alguma maneira uma formulação de programação inteira para o SPLC:

1. Obtenção de limitantes inferiores através da relaxação linear de duas formulações MIP estudadas para o SPLC.
2. Obtenção de limitantes inferiores através de um algoritmo de *branch-and-bound* específico para o SPLC.
3. Obtenção de limitantes inferiores com o uso de uma formulação inteira na determinação do perfil de mão-de-obra em um escalonamento viável.

Os resultados obtidos com estas abordagens mostraram que obter bons limitantes inferiores para o SPLC usando formulações de programação inteira não é uma tarefa fácil. Por um lado, o uso isolado destas formulações, pelo menos com o que se conhece até o momento, não traz nenhum ganho em relação ao limitante inferior trivial do caminho crítico. Por outro lado, nem mesmo um algoritmo de *branch-and-bound* implementado especificamente para o SPLC é capaz de chegar a limitantes inferiores de melhor qualidade. Neste trabalho, a única tentativa que conseguiu obter alguma melhora em relação ao limitante inferior do caminho crítico foi a número três acima. Mesmo assim, vale notar que, apesar de todos os esforços investidos neste trabalho, para a

maioria das instâncias *benchmarks* do SPLC, os melhores limitantes inferiores conhecidos ainda são os obtidos com programação por restrições [HC97].

Com relação à obtenção de limitantes superiores para o SPLC, os resultados desta dissertação foram bem mais positivos. Em um primeiro momento, foram implementadas quatro estratégias heurísticas seqüenciais para o SPLC:

1. Heurística baseada em regras de prioridade.
2. Heurística baseada em classes de escalonamento.
3. Heurística baseada em programação linear.
4. Algoritmo seqüencial de busca tabu.

A aplicação destes algoritmos às instâncias *benchmarks* do SPLC levou a soluções de qualidade igual ou melhor a daquelas obtidas pelos outros métodos de solução para o SPLC conhecidos na literatura [HC97, SUW97].

Não obstante os bons resultados dos algoritmos seqüenciais propostos para o SPLC neste trabalho, foram implementadas também duas estratégias paralelas e assíncronas para obtenção de limitantes superiores para este problema:

1. *A-Team*.
2. Estratégia paralela e assíncrona de busca tabu.

O objetivo neste caso era aproveitar alguns dos algoritmos seqüenciais já implementados para compor as estratégias paralelas e assim verificar se o trabalho cooperativo destes algoritmos era capaz de levar a soluções de qualidade ainda melhor.

Os resultados obtidos com o *A-Team* e com o tabu paralelo implementados para o SPLC comprovaram a adequabilidade do uso de estratégias paralelas e assíncronas na obtenção de soluções de boa qualidade para este problema. Qualitativamente, as melhores soluções entre as obtidas por estes dois algoritmos dominam qualquer outra solução obtida por alguma das demais estratégias de solução conhecidas para o SPLC. Desta forma, juntos, os dois algoritmos paralelos propostos neste trabalho são os atuais responsáveis pelas melhores soluções conhecidas para as instâncias *benchmarks* do SPLC.

Esquematicamente, então, as principais contribuições desta dissertação foram as seguintes:

- Geração e disponibilização, em página WEB, de um conjunto de instâncias *benchmarks* para o SPLC.

- Implementação de um algoritmo de *branch-and-bound* específico para o SPLC.
- Introdução de uma extensão baseada em programação inteira para um método disponível na literatura para o cálculo de limitantes inferiores para problemas de escalonamento com restrição de recursos.
- Proposta e implementação de quatro heurísticas seqüenciais para o SPLC: heurística baseada em regras de prioridade; heurística baseada em classes de escalonamento; heurística baseada em programação linear; e algoritmo seqüencial de busca tabu.
- Proposta e implementação de dois algoritmos paralelos para o SPLC: *A-Team* e estratégia paralela e assíncrona de busca tabu.
- Constatação, através de extensos testes computacionais, da adequação do paradigma paralelo e assíncrono na obtenção de soluções de boa qualidade para o SPLC.

Algumas possíveis linhas de pesquisa para trabalhos futuros com o SPLC são:

- Estudar outras classes de função objetivo que tenham propriedades semelhantes a da função $F_2(s)$, introduzida no algoritmo de *branch-and-bound* e usada nos algoritmos de busca tabu implementados.
- Investigar a incorporação de uma ou mais buscas tabu como agentes de melhoria do *A-Team* proposto para o SPLC.
- Explorar estratégias de diversificação e intensificação no algoritmo seqüencial de busca tabu.
- Explorar o uso de soluções inviáveis nos algoritmos de busca tabu.

A grande distância entre os melhores limitantes inferiores e os melhores limitantes superiores atualmente conhecidos para o SPLC apenas mostra o quanto ainda há para se estudar neste problema. Espera-se que os resultados desta dissertação de alguma forma sirvam como estímulo para continuar a pesquisa com este interessante problema combinatório.

Glossário

α_j: tamanho do caminho crítico do subgrafo induzido em G pelo job j e todos os seus sucessores

$\mathcal{B}$: conjunto de todos os blocos

CP: *Constraint Programming*

e_B: instante mais cedo em que um job do bloco B pode começar considerando apenas as restrições de precedência

e_j: instante mais cedo em que o job j pode começar considerando apenas as restrições de precedência

f_B: instante mais tarde em que um job do bloco B pode começar considerando T e as restrições de precedência

f_j: instante mais tarde em que o job j pode começar considerando T e as restrições de precedência

$G = (N, A)$: grafo de precedência entre jobs

$\ell_{j,s}$: mão-de-obra necessária para execução da s_ésima tarefa do job j

$\ell_{B,s}$: mão-de-obra necessária para execução da s_ésima tarefa de um job no bloco B

L: total de mão-de-obra disponíveis a cada instante

m: total de pedidos

n: total de jobs

n_B: número de jobs no bloco B

n_i: total de jobs no pedido i

N: conjunto de todos os jobs

p_B: duração de um job no bloco B

p_j: duração do job j

MIP: *Mixed Integer Programming*

RCPS: *Resource Constrained Project Scheduling*

s_j: instante de início do processamento do job j

S^x: conjunto dos pontos que satisfazem as desigualdades (3.1) a (3.5) da formulação x

SPLC: *Scheduling Problem under Labour Constraints*

T: limite do horizonte de planejamento

$x_{j,t}$: variável binária igual a 1 se e somente se o job j começa no instante t

$X_{B,t}$: variável binária igual a 1 se e somente se algum job do bloco B começa no instante t

$z_{j,t}$: variável binária igual a 1 se e somente se o job j começou até o instante t (inclusive)

$Z_{B,t}$: número de jobs do bloco B que começaram até o instante t (inclusive)

Bibliografia

[Aie96] R. M. Aiex. Estratégias paralelas e assíncronas de busca tabu aplicadas ao problema de particionamento de circuitos. Tese de Mestrado, Dep. Informática, PUC-RJ, Rio de Janeiro, RJ, 1996.

[Bak74] K. R. Baker. *Introduction to Sequencing and Scheduling.* John Wiley and Sons, New York, 1974.

[BBKR86] J. Blazewicz, J. Barcelo, W. Kubiak e K. Rock. Scheduling tasks on two processors with deadlines and additional resources. *European Journal of Operational Research*, 26:364–370, 1986.

[BCSW86] J. Blazewicz, W. Cellary, R. Slowinski e J. Werglarz. *Scheduling under Resource Constraints: Deterministic Models.* J. C. Baltzer, Basel, 1986.

[BESW93] J. Blazewicz, K. Ecker, G. Schimidt e J. Werglarz. *Scheduling in Computer and Manufacturing Systems.* Spring-Verlag, Berlin, 1993.

[BH91] C. E. Bell e J. Han. A new heuristic solution method in resource constrained project scheduling. *Naval Research Logistics*, 38(3):315–331, 1991.

[BLK83] J. Blazewicz, J. K. Lenstra e A. H. G. Rinnooy Kan. Scheduling subject to resource constraints: classification and complexity. *Discrete Applied Mathematics*, 5:11–24, 1983.

[Boc90] F. F. Boctor. Some efficient multi-heuristic procedures for resource-constrained project scheduling. *European Journal of Operational Research*, 49:3–13, 1990.

[BP90] C. E. Bell e K. Park. Solving resource constrained project scheduling problems by A^* search. *Naval Research Logistics Quaterly*, 37(1):61–84, 1990.

[Cav95] V. F. Cavalcante. Times assíncronos para o *job shop scheduling problem*: heurísticas de construção. Tese de Mestrado, Dep. de Ciência da Computação - IMECC - UNICAMP, Campinas, SP, 1995.

[Cav97] C. C. B. Cavalcante, 1997. http://www.dcc.unicamp.br/~cris/SPLC.html.

[CCHS97] C. C. B. Cavalcante, Y. Colombani, S. Heipcke e C. C. Souza. Scheduling under labour constraints, 1997. Submetido para o jornal *Constraints* (Kluwer Academic Publishers).

[Cof76] E. G. Coffman. *Computer and Job-Shop Scheduling Theory.* John Wiley & Sons, New York, 1976.

[CPL95] CPLEX Optimization, Inc. Using the CPLEX callable library and CPLEX mixed integer library. Manual, 1995.

[CT97] T. G. Crainic e M. Toulouse. Parallel metaheuristics. Technical report, Centre de Recherche sur les Transports, Université de Montréal, 1997.

[CTG93] T. G. Crainic, M. Toulouse e M. Gendreau. Towards a taxonomy of parallel tabu search algorithms. Publicação 933, Centre de Recherche sur les Transports, Université de Montréal, 1993.

[CTG95] T. G. Crainic, M. Toulouse e M. Gendreau. Synchronous tabu search strategies for multicommodity location-alocation with balancing requirements. Technical report, Centre de Recherche sur les Transports, Université de Montréal, 1995.

[DH71] E. W. Davis e G. E. Heidorn. An algorithm for optimal project scheduling under multiple resource constraints. *Management Science*, 27(12):803–816, 1971.

[DH92] E. Demeulemeester e W. Herroelen. A branch-and-bound procedure for the multiple resource constrained project scheduling problem. *Management Science*, 38:1803–1818, 1992.

[DP75] E. W. Davis e J. H. Patterson. A comparison of heuristics and optimum solutions in resource constrained project scheduling. *Management Science*, 21(8):944–955, 1975.

[FW96] C. E. Ferreira e Y. Wakabayashi. *Combinatória Poliédrica e Planos de Cortes Faciais.* 10a. Escola de Computação, Campinas, 1996.

[GJ75] M. R. Garey e D. S. Johnson. Complexity results for multiprocessor scheduling under resource constraints. *SIAM Journal on Computing*, 4:397–411, 1975.

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

[GL95] F. Glover e M. Laguna. Tabu search. In C. R. Reeves, editor, *Modern Heuristic Techniques for Combinatorial Problems*, capítulo 3, pp. 70–150. McGraw-Hill, 1995.

[Glo89] F. Glover. Tabu search - Part I. *ORSA Journal on Computing*, 1(3):190–206, 1989.

[Glo90] F. Glover. Tabu search - Part II. *ORSA Journal on Computing*, 2(3):4–32, 1990.

[Gon69] L. Gonguet. Comparison of three heuristic procedures for allocating resources and producing schedule, 1969. Apud [DP75].

[Had96] E. G. Haddad. Times assíncronos para o *job shop scheduling problem*: heurísticas de melhoria. Tese de Mestrado, Instituto de Computação - UNICAMP, Campinas, SP, 1996.

[HC97] S. Heipcke e Y. Colombani. A new constraint programming approach to large scale resource constrained scheduling. Workshop on Models and Algorithms for Planning and Scheduling Problems, Cambridge, UK, 1997.

[Hei95] S. Heipcke. A new constraint programming approach to large scale resource constrained scheduling. Diploma-thesis, Mathematisch Geographische Fakultät, Katholische Universität Eichstätt, 1995.

[IE96] O. Icmeli e S. S. Erenguc. A branch-and-bound procedure for the resource constrained project scheduling problem with discounted cash flows. *Management Science*, 42(10):1395–1408, 1996.

[Lag95] M. Laguna. Tabu search tutorial. II Escuela de Verano Latino-Americana de Investigacion Operativa, Mendes, RJ, 1995.

[LK96] J. K. Lee e Y. D. Kim. Search heuristics for resource constrained project scheduling. *Journal of the Operational Research Society*, 47:678–689, 1996.

[Man89] U. Manber. *Introduction to Algorithms - A Creative Approach.* Addison-Wesley, 1989.

[NDWS95] K. S. Naphade, B. H. Doshi, S. D. Wu e R. H. Storer. A bi-level local search algorithm for heavily constrained bi-criterion scheduling problems. 1995. Comunicação particular.

[NW88] G. L. Nemhauser e L. A. Wolsey. *Integer and Combinatorial Optimization.* John Wiley, 1988.

[NWS93] K. S. Naphade, S. D. Wu e R. H. Storer. Problem search algorithms for resource-constrained project scheduling, 1993. Comunicação particular.

[PAM] PAMIPS. http://www.iwr.uni-heidelberg.de/iwr/agbock/doc/pamips.html.

[Pas65] T. L. Pascoe. Alternate methods of project scheduling with limited resources, 1965. Apud [DP75].

[Pat73] J. H. Patterson. Alternate methods of project scheduling with limited resources. *Naval Research Logistic Quartely*, 20:767–784, 1973.

[Pat76] J. H. Patterson. Project scheduling: The effects of problem structure on heuristic performance. *Naval Research Logistics Quaterly*, 20(4):95–123, 1976.

[Pei95] H. P. Peixoto. Metodologia de especificação de times assíncronos para problemas de otimização combinatória. Tese de Mestrado, Dep. de Ciência da Computação - IMECC - UNICAMP, Campinas, SP, 1995.

[PR95] S. C. S. Porto e C. C. Ribeiro. A tabu search approach to task scheduling on heterogeneous processors under precedence constraints. *Int. Journal of High-Speed Computing*, 7, 1995.

[PR96] S. C. C. Porto e C. C. Ribeiro. A case study on parallel synchronous implementations for tabu search based on neighborhood decomposition. Monografias em ciência da computação mcc-03/96, Dep. Informática, PUC-RJ, Rio de Janeiro, 1996.

[PTSW90] J. H. Patterson, F. B. Talbot, R. Slowinski e J. Weglarz. Computational experience with a backtracking algorithm for solving a general class of precedence and resource-constrained scheduling problems. *European Journal of Operational Research*, 49:68–79, 1990.

[Ree95] C. R. Reeves *et. al.*. *Modern Heuristic Techniques for Combinatorial Problems*. McGraw-Hill, 1995.

[Slo80] R. Slowinski. Two approaches to problems of resource allocation among project activities - a comparative study. *Journal of Operational Research Society*, 31:711–723, 1980.

[Slo81] R. Slowinski. Multiobjective network scheduling with efficient use of renewable and nonreneable resources. *European Journal of Operational Research*, 7:265–273, 1981.

[SN96] M. W. P. Savelsbergh e G. L. Nemhauser. Functional description of MINTO, a Mixed INTeger Optimizer. Versão 2.3, 1996.

[Sou93] P. S. Souza. *Asynchronous Organizations for Multi-Algorithm Problems*. Ph.D. dissertation, Electrical and Computer Engineering Department, Carnegie Mellon University, Pittsburgh, PA, 1993.

[SUW97] M. W. P. Savelsbergh, R. N. Uma e J. Wein. An experimental study of lp-based approximation algorithms for scheduling problems. Extended Abstract, Julho de 1997.

[SW93] S. E. Sampson e E. N. Weiss. Local search techniques for the generalized resource constrained project scheduling. *Naval Research Logistics*, 40:665–675, 1993.

[SW97] C. C. Souza e L. A. Wolsey. Scheduling projects with labour constraints. Relatório técnico ic-97-22, IC - UNICAMP, 1997.

[SWV92] R. H. Storer, S. D. Wu e R. Vaccari. New search spaces for sequence problems with applications to job shop scheduling. *Management Science*, 38, 1992.

[SWW96] M. Savelsbergh, Y. Wang e L. A. Wolsey. Computational experiments with a large-scale resource constrained project scheduling problem. Note, Georgia Institute of Technology, Agosto de 1996.

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

[TCG96] M. Toulouse, T. G. Crainic e M. Gendreau. Communication issues in designing cooperative multi thread parallel searches. In I. H. Osman e J. P. Kelly, editores, *Meta-Heuristics: Theory & Applications*, pp. 501–522. Kluwer, Norwell, MA, 1996.

[TP78] B. Talbot e J. H. Patterson. An efficient integer programming algorithm with network cuts for solving resource constrained scheduling problems. *Management Science*, 24(11):1163–1174, 1978.

[WW96] Y. Wang e L. A. Wolsey. Scheduling with labour constraints. Note, CORE, Université Catholique de Louvain, Maio de 1996.

[ZP95] D. Zhu e R. Padman. A cooperative multi-agent approach to constrained project scheduling. Working paper, The Heinz School, Carnegie Mellon University, Pittsburgh, PA 15213, 1995.

[ZP96] D. Zhu e R. Padman. A tabu search approach for scheduling resource-constrained projects with cash flows. Working paper, The Heinz School, Carnegie Mellon University, Pittsburgh, PA 15213, 1996.