Algoritmos Heurísticos para o *Prize Collecting Traveling Salesman Problem*

Wesley Elias Ribeiro

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Algoritmos Heurísticos para o *Prize Collecting Traveling Salesman Problem*

Wesley Elias Ribeiro

Novembro de 1997

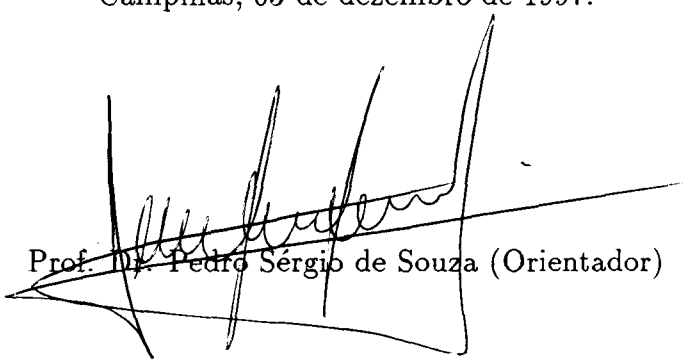
Banca Examinadora:

- Prof. Dr. Pedro Sérgio de Souza (Orientador)
- Prof. Dr. José Nelson Amaral
  Depto. de Eng. Elétrica - Escola Politécnica - PUC - RS
- Prof. Dr. Ricardo Dahab
- Prof. Dr. João Meidanis (Suplente)

Algoritmos Heurísticos para o *Prize Collecting Traveling Salesman Problem*

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Wesley Elias Ribeiro e aprovada pela Banca Examinadora.

Campinas, 03 de dezembro de 1997.



Prof. Dr. Pedro Sérgio de Souza (Orientador)

Prof. Dr. Cid Carvalho de Souza
(Co-orientador)

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

BIBLIOTECA CENTRAL
DESENVOLVIMENTO
COLEÇÃO
UNICAMP

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Ribeiro, Wesley Elias

R354a Algoritmos heurísticos para o prize collecting traveling salesman problem / Wesley Elias Ribeiro -- Campinas, [S.P. :s.n.], 1997.

Orientador : Pedro Sérgio de Souza

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Algoritmos. 2. Otimização combinatória. 3. Heurística. 4. Problema do caixeiro viajante. I. Souza, Pedro Sérgio. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Tese de Mestrado defendida e aprovada em 03 de dezembro de 1997 pela Banca Examinadora composta pelos Professores Doutores

Prof. Dr. José Nelson Amaral

Prof. Dr. Ricardo Dahab

Prof. Dr. Pedro Sérgio de Souza

© Wesley Elias Ribeiro, 1998.
Todos os direitos reservados.

Agradecimentos

À Renata, pelo amor, apoio e compreensão.

À minha família, pelo apoio e incentivo.

Aos amigos da graduação, particularmente: Geraldo, Thomaz, Luís Cláudio, Cláudio, Márcia, Fabíola e Leandro.

À "turma do segundo ano" da UFG, particularmente: William, Ana Paula, Rubens, Anossis e Fabiano.

Ao Cláudio, pela ajuda na revisão da dissertação.

À família Melo Barros (Adílson, Terezinha, Cristina e Sandra), pelo carinho com que sempre me receberam em sua casa.

Ao meu orientador, Prof. Pedro Sérgio, pela seriedade e direções na realização deste trabalho.

A todos os amigos que conheci no mestrado, e que tornaram mais fácil a vida numa nova cidade, particularmente: Victor, Cris, Anderson, George, Clevan, Rosiane, Ana Waleska, Dinalva, Sávio, Walter, Eduardo, Edson, Marcos, Elaine.

A Geraldo, Sílvio e Dalmi, pelas partidas de final de semana.

A todos os demais que contribuíram de alguma forma para a realização deste trabalho.

Resumo

Esta dissertação trata do Problema do Caixeiro Viajante Coletor de Prêmios (*Prize Collecting Traveling Salesman Problem*-PCTSP). Este problema é uma generalização do bastante conhecido Problema do Caixeiro Viajante (*Traveling Salesman Problem*-TSP), em que o caixeiro viajante não precisa visitar, necessariamente, todas as cidades, mas um número suficiente delas para a obtenção de um prêmio mínimo. Além disso, sua função objetivo é dada pela minimização do comprimento da rota adicionada às penalidades pagas por cidades não visitadas. A formulação do PCTSP foi feita com base em uma aplicação prática do problema, o escalonamento de equipamentos em uma indústria siderúrgica.

O presente trabalho apresenta um estudo de algoritmos heurísticos disponíveis na literatura do problema. São, então, apresentadas novas heurísticas de construção e melhoria de soluções desenvolvidas para o PCTSP, e é efetuada uma comparação com o algoritmo de melhor garantia de desempenho encontrado na literatura.

Este trabalho também compreende o desenvolvimento de um Time Assíncrono para o PCTSP. Times Assíncronos compreendem uma abordagem meta-heurística já aplicada com sucesso a diversos outros problemas de Otimização Combinatória. Seu princípio básico é a combinação sinérgica de diversos algoritmos (agentes), comunicando-se através de memórias compartilhadas.

O Time Assíncrono foi implementado de forma distribuída, utilizando-se o pacote PVM (*Parallel Virtual Machine*), baseado em troca de mensagens. Para os testes foram geradas aleatoriamente diversas instâncias de tamanhos variados, e, para efeito de comparação, foram obtidos limites inferiores para estas instâncias utilizando-se o pacote de programação linear/inteira Cplex, aplicado a relaxações do problema desenvolvidas.

Abstract

This dissertation deals with the Prize Collecting Traveling Salesman Problem (PCTSP). This problem is a generalization of the well-known Traveling Salesman Problem (TSP), where the salesman does not need to visit all the cities, but has to visit enough cities in order to obtain a minimum prize. Besides that, the objective function is given by the minimization of the tour lenght plus the penalties paid for unvisited cities. The formulation of the PCTSP was made based on a pratical application of the problem, the scheduling of production units in a steel plant.

The present work presents a study of the heuristic algorithms available in the literature about the problem. It then shows new construction and improvement heurisitcs developed for the PCTSP, and presents a comparison between those new heuristics and the best performance guarantee algorithm found in the literature.

This work also presents a Asynchronous Team developed for the PCTSP. Asynchronous Teams are a meta-heuristic approach already succesfully applied to many other Combinatorial Optimization problems. Its basic principle is the sinergic combination of many algorithms (agents), communicating through shared memories.

The Asynchronous Team was implemented using distributed processing, by using the message-passing based PVM (Parallel Virtual Machine) package. Many instances of different sizes were randomically generated, and, for comparison, lower bounds for these instances were calculated using the Cplex linear/integer programming package, applied to relaxations of the problem.

Conteúdo

Agradecimentos v

Resumo vi

Abstract vii

1 Introdução 1
- 1.1 Objetivos do Trabalho . . . 2
- 1.2 Organização da Dissertação . . . 2

2 Problemas de Otimização Combinatória 4
- 2.1 Definição e Complexidade . . . 4
- 2.2 Métodos de resolução . . . 7
  - 2.2.1 Métodos Exatos . . . 7
  - 2.2.2 Métodos Aproximados . . . 8

3 O PCTSP 13
- 3.1 Definição do Problema . . . 13
  - 3.1.1 Complexidade . . . 17
- 3.2 Algoritmos Heurísticos da Literatura . . . 18
  - 3.2.1 Algoritmo MLP (*Modified Linear Programming Relaxation*) . . . 18
  - 3.2.2 Algoritmo de Goemans e Williamson . . . 22
  - 3.2.3 Algoritmo de Awerbuch, Azar, Blum e Vempala . . . 31
  - 3.2.4 Algoritmo de Inserção/Retirada de Vértices . . . 33
- 3.3 Algoritmos Propostos para o PCTSP . . . 35
  - 3.3.1 Algoritmos de Construção . . . 35
  - 3.3.2 Algoritmos de Melhoria . . . 40
- 3.4 Resultados Comparativos . . . 42

4 Times Assíncronos **49**
4.1 Definição e Características 49
4.2 Parâmetros de Construção 50
4.2.1 Memórias 51
4.2.2 Algoritmos de Iniciação 51
4.2.3 Política de Destruição 52
4.2.4 Algoritmos de Melhoria 53
4.2.5 Algoritmos de Consenso 53
4.2.6 Algoritmos de Reconstrução 53
4.3 Histórico 53

5 Times Assíncronos para o PCTSP **56**
5.1 Estrutura do *A-Team* proposto 56
5.1.1 Memórias 56
5.1.2 Algoritmos de Iniciação 58
5.1.3 Política de Destruição 59
5.1.4 Agentes de Melhoria 61
5.1.5 Algoritmos de Consenso 61
5.1.6 Algoritmos de Reconstrução 62
5.1.7 Implementação 63
5.2 Instâncias para Teste 64
5.2.1 Valores para comparação 65
5.3 Resultados Computacionais 68
5.3.1 Eficiência em Escala 72

6 Conclusão **75**
6.1 Contribuições 76
6.2 Extensões e Trabalhos Futuros 76

Bibliografia **78**

Lista de Tabelas

3.1 Relação entre os números de rotas para instâncias do TSP e PCTSP. . . . 18
3.2 Comparação entre algoritmos de construção propostos e GW-PCTSP. . . . 43

5.1 Resultados obtidos pelo *A-Team* com iniciação pelo Algoritmo Solução Aleatória 59
5.2 Resultados obtidos pelo *A-Team* com iniciação pelo Algoritmo Inserção Arbitrária-PCTSP 59
5.3 Resultados obtidos pelo *A-Team* com iniciação pelo Algoritmo Insere-Vértice Dinâmico Não-Determinístico 59
5.4 Resultados obtidos pelo *A-Team* com $\alpha = 1,5$ 60
5.5 Resultados obtidos pelo *A-Team* com $\alpha = 2,0$ 60
5.6 Resultados obtidos pelo *A-Team* com $\alpha = 3,0$ 60
5.7 Resultados obtidos pelo *A-Team* com $\alpha = 4,0$ 61
5.8 Resultados obtidos pelo *A-Team* para instâncias da classe 10% 69
5.9 Resultados obtidos pelo *A-Team* para instâncias da classe 50% 70
5.10 Resultados obtidos pelo *A-Team* para instâncias da classe 90% 71
5.11 Porcentagem média de distância do *lower bound* por classe de instância . . 72
5.12 Resultados do teste de eficiência em escala para instância t200.5. 73

Lista de Figuras

2.1 Uma possibilidade de relacionamento entre tipos de problemas. 7

3.1 Exemplo de uma solução para o PCTSP. 15
3.2 Exemplo da inserção de um vértice em uma rota qualquer. 16
3.3 Grafo de exemplo da execução do algoritmo GW-PCSTP. 25
3.4 Grafo de exemplo após primeira iteração. 26
3.5 Grafo de exemplo após segunda iteração. 26
3.6 Grafo de exemplo após terceira iteração. 27
3.7 Grafo de exemplo após quarta iteração. 27
3.8 Grafo de exemplo após quinta iteração. 28
3.9 Exemplo da execução do algoritmo Insere-Vértice Estático. 37
3.10 Exemplo da execução do algoritmo Insere-Vértice Dinâmico. 38
3.11 Exemplo da execução do algoritmo Insere-Aresta. 39
3.12 Processo utilizado nos algoritmos Insere e Retira Vértices. 41
3.13 Resultados para instâncias de 50 vértices. 44
3.14 Resultados para instâncias de 100 vértices. 45
3.15 Resultados para instâncias de 150 vértices. 46
3.16 Resultados para instâncias de 200 vértices. 47
3.17 Resultados para instâncias de 500 vértices. 48

4.1 Exemplo de um *A-Team*. 51

5.1 Configuração proposta para o PCTSP. 57
5.2 Exemplo de desigualdade triangular. 65
5.3 Configurações para teste de eficiência em escala. 73
5.4 Mais configurações para teste de eficiência em escala. 74

Capítulo 1

Introdução

No mundo globalizado de hoje em dia, as empresas estão enfrentando uma competição cada vez mais acirrada por espaço no mercado, e estão se dando conta da necessidade de serem mais produtivas e de oferecerem mais qualidade em seus produtos, ao mínimo custo. E existem diversos pontos onde a Ciência da Computação pode ajudar essa busca por eficiência.

No caso específico de indústrias, relacionados ao ambiente de produção aparecem diversos problemas de Otimização Combinatória. Em situações como armazenagem de itens no estoque, distribuição de mercadorias e escalonamento de máquinas é possível divisar diversos problemas deste tipo. Um deles é o chamado Problema do Caixeiro Viajante (*Traveling Salesman Problem*). Sua importância se deve também ao fato de este ser um problema cujo estudo levou a um grande desenvolvimento da Otimização Combinatória [HW85, DFJ54], se tornando uma espécie de "representante" de sua classe.

No TSP, um caixeiro viajante tem um certo número de cidades a visitar, e deseja visitar todas passando apenas uma vez em cada cidade e voltando à cidade inicial. Além disso, já que existe um custo associado à ida de uma cidade para outra, ele deseja minimizar o custo total de sua viagem. A partir de um problema encontrado em um ambiente industrial (uma usina siderúrgica), Balas [Bal89] definiu um problema que é uma generalização do TSP, chamado de Problema do Caixeiro Viajante Coletor de Prêmios (*Prize Collecting Traveling Salesman Problem*). Neste, o caixeiro viajante não necessita mais passar por todas as cidades, mas cada cidade que deixa de visitar implica no pagamento de uma penalidade. Além disso, para cada cidade que visita ele ganha um determinado prêmio. Seu objetivo consiste em achar uma rota que minimize o custo total da viagem mais as penalidades pagas e que contenha cidades suficientes para a obtenção de um prêmio mínimo.

Dada a dificuldade dos problemas, a Otimização Combinatória se tornou um fértil ramo da Ciência da Computação. Foram criadas diversas abordagens para solução destes problemas. Algumas abordagens têm como meta alcançar o valor ótimo comprovado de um problema (são os chamados métodos exatos). Outras abordagens tentam alcançar o melhor valor possível dentro de um tempo computacional aceitável (os chamados métodos aproximados ou heurísticos). Algumas abordagens são específicas para um problema e outras se propõem a serem aplicáveis, em princípio, a todos os problemas desta classe. Uma abordagem enquadrada neste último tipo, denominada de Times Assíncronos, tem como característica marcante a possibilidade de união de diversas abordagens diferentes, todas trabalhando sobre o mesmo problema. A idéia é que os esforços de todos juntos produzam melhores resultados do que qualquer um isoladamente.

1.1 Objetivos do Trabalho

Este trabalho tem basicamente dois objetivos principais, que são o desenvolvimento de heurísticas de construção e melhoria e a aplicação da metodologia de Times Assíncronos ao PCTSP, com conseqüente verificação do desempenho dos algoritmos e do Time Assíncrono.

1.2 Organização da Dissertação

Além desta introdução, esta dissertação consta de mais cinco capítulos:

- O Capítulo 2 discorre sobre os problemas de Otimização Combinatória em geral, dando ênfase em sua complexidade de resolução e em alguns métodos normalmente utilizados para tentar resolvê-los.

- O Capítulo 3 apresenta o *Prize Collecting* TSP, juntamente com alguns algoritmos aplicáveis a ele e disponíveis na literatura. Além disso, são também introduzidos alguns novos algoritmos heurísticos desenvolvidos para o problema, e é mostrada uma comparação de resultados obtidos por estes e o algoritmo da literatura com melhor garantia de desempenho para o problema.

- O Capítulo 4 introduz a meta-heurísitica denominada de Times Assíncronos (*A-Teams*). São definidos os principais componentes de um Time Assíncrono, bem como os itens a serem levados em conta na construção de um *A-team*. Encerramos o capítulo com um breve histórico da aplicação de *A-Teams* a outros problemas.

- O Capítulo 5 apresenta o Time Assíncrono desenvolvido para o Problema do Caixeiro Viajante Coletor de Prêmios, bem como os resultados obtidos com sua aplicação a um grupo de instâncias de teste.
- O Capítulo 6 apresenta as conclusões obtidas neste trabalho, bem como suas principais contribuições e possibilidades de extensão.

Capítulo 2

Problemas de Otimização Combinatória

Neste capítulo apresentamos uma definição do que vem a ser um problema de Otimização Combinatória e enumeramos alguns exemplos desta classe de problemas. Em seguida, discorremos sobre sua complexidade de resolução. Finalmente, apresentamos algumas técnicas que normalmente são utilizadas na tentativa de resolução destes problemas.

2.1 Definição e Complexidade

Problemas de Otimização Combinatória são aqueles cuja resolução compreende a otimização (maximização ou minimização) de uma ou várias funções objetivo, ao mesmo tempo em que devem ser satisfeitas diversas restrições definidas sobre suas variáveis. Além disso, o número de soluções possíveis para um problema desta classe é uma função combinatória do número de suas variáveis.

Numerosos problemas que surgem na Ciência da Computação se enquadram nesta classe, tais como:

- Coloração de grafos;
- Determinação da árvore geradora de peso mínimo de um grafo;
- Fluxo em redes;
- Problema da mochila (*Knapsack Problem*);

- Problema do caixeiro viajante (*Traveling Salesman Problem* - TSP);
- Determinação do caminho mais curto entre dois vértices de um grafo;
- Determinação da clique de tamanho máximo de um grafo.

Todos os problemas de Otimização Combinatória possuem sua "versão de decisão", isto é, uma versão do problema cuja resposta é **sim** ou **não**. Por exemplo, a versão de decisão do problema de determinação da clique de tamanho máximo de um grafo G pode ser enunciada como: dados um grafo $G = (V, E)$ e um inteiro positivo $k \leq |V|$; G possui uma clique de tamanho maior ou igual a k ?

A primeira grande classificação de problemas na Ciência da Computação divide-os em dois grandes grupos, de acordo com a sua possibilidade de resolução (ou não) por um algoritmo:

- Problemas indecidíveis: são problemas para os quais não existe nenhum algoritmo capaz de resolvê-los. Como exemplo, temos o Problema da Parada [Tur36].
- Problemas decidíveis: são problemas que possuem um algoritmo cujo tempo de processamento cresce polinomial ou exponencialmente em função do tamanho da instância do problema.

A teoria da NP-Completude [GJ79] divide os problemas decidíveis de Otimização Combinatória em três grandes grupos, de acordo com a variação do esforço computacional (tempo) necessário para se resolver o problema em relação à mudança do tamanho da instância do problema. São eles:

- Problemas em P (*Polinomial time*): os problemas desta classe dispõe de algoritmos para resolvê-los cujo tempo de processamento cresce polinomialmente em função do tamanho da instância do problema. Este tipo de algoritmo é conhecido como eficiente. Como exemplos de problemas em P, temos: fluxo em redes, determinar a árvore geradora de peso mínimo de um grafo, determinar se um grafo de intervalo próprio é hamiltoniano [Ber83] etc.
- Problemas em NP (*Nondeterministic Polinomial time*): um problema pertence à classe NP se existe um algoritmo determinístico para resolvê-lo cujo tempo de processamento cresce exponencialmente em função do tamanho da instância do problema. Os problemas nesta classe também são caracterizados pela existência de um algoritmo não-determinístico que os resolve em tempo polinomial [GJ79]. Uma propriedade adicional desta classe é que, dada uma solução para determinado problema em NP, é possível checar a validade da solução em tempo polinomial. Como

exemplo de problema em NP, podemos citar o problema de se determinar se um grafo possui um ciclo hamiltoniano.

- Problemas intratáveis: estes problemas são aqueles cujos algoritmos que os resolvem necessitam de um tempo exponencial em relação ao tamanho da instância. Como exemplo de problema intratável, podemos citar o problema de encontrar todas as cliques maximais de um grafo.

Além dos três grandes grupos citados acima, duas outras classes são definidas e complementam a teoria da NP-Completude. Estas classes se baseiam na **transformação (ou redução) polinomial de problemas**. Diz-se que um problema de decisão π_1 é redutível polinomialmente a outro problema de decisão π_2 se e somente se existe uma função $f : \pi_1 \rightarrow \pi_2$ tal que:

1. f pode ser computada em tempo polinomial.
2. para toda instância I de π_1 tem-se que $\pi_1(I)$ possui resposta **sim** se e somente se $\pi_2(f(I))$ possui resposta **sim**.

Com base na definição de transformação polinomial de problemas, e na pertinência ou não de um problema a NP, duas classes adicionais são definidas:

- Problemas NP-difíceis: um problema pertence a esta classe se todos os problemas em NP forem polinomialmente redutíveis a ele. Como exemplos de problemas desta classe temos: o problema do caixeiro viajante, o problema de se determinar uma clique de tamanho máximo de um grafo, o problema da mochila etc.
- Problemas NP-completos: compreendem os problemas que estão na intersecção das classes NP e NP-difícil. Ou seja, são aqueles problemas pertencentes a NP aos quais todos os demais problemas em NP são polinomialmente redutíveis. Nesta classe estão as versões de decisão dos problemas NP-difíceis, como o problema de se determinar se um grafo possui um ciclo hamiltoniano com comprimento menor ou igual a um determinado valor.

Após a formulação destas classes de problemas, restou uma questão ainda não respondida: existe algum problema que pertença a NP e que não pertença a P ? Ou seja, P $\neq$ NP ? Embora ainda não exista nenhuma prova, a maioria dos pesquisadores acredita que a resposta a esta questão é **sim**, o que deixaria a relação entre os tipos de problemas apresentados da forma como é apresentada na Figura 2.1.

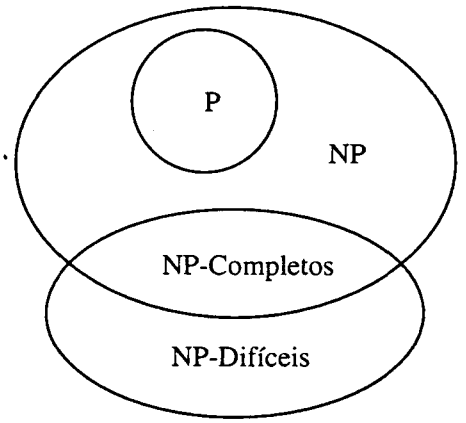


Figura 2.1: Uma possibilidade de relacionamento entre tipos de problemas.

2.2 Métodos de resolução

Como é natural, o interesse dos pesquisadores recai sobre os problemas de Otimização Combinatória NP-difíceis, para os quais não se conhecem algoritmos eficientes para resolvê-los. A seguir apresentamos uma relação de técnicas normalmente usadas na tentativa de resolução deste tipo de problema, divididas em dois grandes grupos: os métodos exatos e os métodos aproximados.

2.2.1 Métodos Exatos

Nesta classe se enquadram os métodos cujo propósito é encontrar uma solução ótima, provada como tal, para as instâncias do problema. Como não se pode garantir que esta solução ótima será atingida em tempo polinomial para todas as instâncias do problema, a pesquisa em torno de métodos exatos se concentra em técnicas para se diminuir o tempo necessário para se encontrar a solução ótima. Estes métodos normalmente são mais difíceis de se implementar do que os métodos aproximados.

Branch-and-Bound

O método *Branch-and-Bound* procura resolver um determinado problema através da enumeração das soluções possíveis, e tenta-se fazer com que o número de soluções enumeradas seja tão pequeno quanto possível [NW88]. Isto é feito através de algoritmos que calculam limites inferiores e superiores a uma solução ótima da instância sendo resolvida: a cada passo do *Branch-and-Bound* o problema é dividido em sub-problemas menores, e verifica-se, através dos limites superior e inferior, a viabilidade e a necessidade de se analisar um ramo da árvore de enumeração.

Cutting Planes

Os *Cutting Planes* (planos de corte) são métodos que começam com uma relaxação do problema que se quer resolver e posterior identificação de restrições que são violadas por esta relaxação. Caso sejam identificadas uma ou mais restrições violadas, então estas são inseridas no conjunto de restrições, e calcula-se novamente uma relaxação do conjunto atual de restrições. Este processo continua até que uma solução viável para o problema seja encontrada ou até que não se consiga encontrar uma restrição violada. Vários algoritmos deste tipo foram propostos por Gomory [Gom60], os quais envolviam a geração automática de cortes no plano com a garantia de que nenhuma solução inteira seria eliminada. Um ponto importante a ser observado em algoritmos de *Cutting Planes* é que os cortes devem ser desenvolvidos de tal modo que, após um número finito de passos, se garanta que a solução inteira obtida seja a solução ótima.

Branch-and-Cut

Este método baseia-se em um algoritmo *Branch-and-Bound* onde em cada nó da árvore de enumeração deste realiza-se uma fase de *Cutting Planes*. Se nesta fase não for identificado nenhum plano de corte, então efetua-se o *branching* (divisão do espaço de soluções correntes em dois ou mais subespaços de soluções disjuntas). Este processo continua até que seja encontrada uma solução ótima para o problema.

Programação Dinâmica

O método de programação dinâmica baseia-se na divisão de um problema em sub-problemas e em sua resolução recursiva. Estes sub-problemas não são disjuntos, isto é, sub-problemas compartilham sub-subproblemas. Portanto, o método de programação dinâmica parte das soluções dos sub-problemas e as combina para formar a solução para o problema original (uma abordagem *bottom-up*).

2.2.2 Métodos Aproximados

Nesta classe se enquadram os métodos cujo objetivo é tentar resolver um problema em um tempo computacional aceitável. Como os métodos aproximados não garantem que a solução ótima será encontrada, os estudos nesta área se concentram em melhorar cada vez mais a qualidade das soluções obtidas, sem contudo comprometer o tempo de execução. Os métodos aproximados em geral são mais fáceis de se implementar do que os exatos. Os métodos aproximados podem ser divididos em dois grupos: heurísticas e meta-heurísticas.

Heurísticas

Heurísticas são métodos aproximados que tentam tirar proveito do conhecimento existente sobre o problema e sua estrutura para tentar resolvê-lo. São, portanto, algoritmos específicos para um determinado problema. Um grande obstáculo para as heurísticas é a existência de ótimos locais na(s) função(ções) sendo otimizadas: vales (ou picos, no caso de problemas de maximização) no espaço de solução dos quais uma determinada heurística não consegue escapar, embora sejam passíveis de melhoria.

Meta-heurísticas

Meta-heurísticas são métodos aproximados que tentam propor uma forma geral, portanto aplicável a qualquer problema, de se evitar a estagnação em ótimos locais. A seguir resumimos algumas meta-heurísticas de uso difundido.

- *Tabu Search*: Desenvolvida por Fred Glover [Glo89] [Glo90a][Glo90b], esta meta-heurística baseia-se na manutenção de uma lista tabu, uma lista de movimentos (modificações na solução) proibidos a cada instante. Assim, a cada passo é efetuada uma mudança na solução (desde que esta mudança não esteja na lista tabu). Esta mudança é normalmente a melhor mudança de uma vizinhança de busca definida.

  Um outro conceito normalmente presente em *Tabu Search* é o de "critério de aspiração". Por este conceito entende-se um critério que, se for atingido por determinado movimento (alteração na solução), este movimento deve ser aceito mesmo que isto viole a lista tabu. O critério de aspiração geralmente empregado é o seguinte: o movimento é aceito caso leve à geração da melhor solução até o momento. Um algoritmo básico de *Tabu Search* é descrito a seguir:

  Algoritmo *Tabu Search*
  1- Gere uma solução inicial.
  2- Crie uma lista tabu.
  3- Enquanto o critério de parada não for satisfeito, faça:
  3.1- Execute o melhor movimento não proibido pela lista tabu ou que satisfaça o critério de aspiração.
  3.2- Atualize a lista tabu.
  3.3- Atualize o critério de aspiração.
  4- Retorne a melhor solução encontrada.

- **Algoritmos Genéticos**: São algoritmos de busca baseados no mecanismo natural de reprodução e seleção, em que uma população de soluções é melhorada após

sucessivas gerações [Gol89].

As soluções em Algoritmos Genéticos são codificadas em estruturas denominadas cromossomos, que são constituídos de genes. Os valores que os genes podem assumir são denominados alelos, e são normalmente binários (0 ou 1). A cada cromossomo é associado um valor indicando o quão bom ele é, ou seja, um valor da função objetivo do problema. Na terminologia de Algoritmos Genéticos esta função é denominada de *fitness function.*

A cada geração (iteração do algoritmo), é criada uma nova população de cromossomos, que pode substituir total ou parcialmente a população anterior. Esta nova população é gerada através de dois processos:

- *Crossover*: é a combinação de dois ou mais cromossomos para a formação de um novo.
- *Mutação*: alterações aleatórias nos genes de um cromossomo que levam à criação de outro.

A seleção dos cromossomos que se reproduzirão é de grande importância em Algoritmos Genéticos, e são comumente usadas seleções aleatórias uniformes ou distribuídas através da *fitness function* (os melhores cromossomos têm mais chances de se reproduzirem).

Uma característica por vezes presente em Algoritmos Genéticos é o *elitismo*, ou seja, os melhores cromossomos da população são preservados na próxima população. Isto garante uma melhoria monotônica da *fitness function* e acelera a convergência do método. Um algoritmo genético seria algo como o descrito a seguir:

Algoritmo Genético
1- Gere uma população inicial.
2- Enquanto o critério de parada não for atingido, faça:
2.1- Selecione os cromossomos para *crossover* e mutação e efetue estas operações.
2.2- Elimine com maior probabilidade os cromossomos piores em relação à *fitness function* para abrir espaço para os novos (elitismo).
3- Retorne o melhor cromossomo encontrado.

- *Simulated Annealing*: Método de otimização baseado nos processos físicos de resfriamento de fluidos. Quando um fluido está se resfriando ele pode se solidificar basicamente com uma de duas estruturas: amorfa (átomos não arranjados de nenhuma

maneira especial) ou cristalina (átomos ordenadamente dispostos em retículos cristalinos). A estrutura cristalina normalmente só é conseguida se a temperatura inicial for suficientemente alta e o resfriamento bastante lento.

Metropolis *et al* [MRR+53] definiram um algoritmo simples para simular uma coleção de partículas durante o processo de resfriamento. Kirkpatrick *et al* [KJV83] adaptaram este algoritmo para a geração de uma seqüência de soluções de um problema de Otimização Combinatória, onde:

- As soluções do problema de Otimização Combinatória correspondem aos estados físicos do sistema.
- As variáveis do problema correspondem às partículas do sistema.
- O valor da função objetivo equivale à energia do estado em que se encontra o sistema.

Em *Simulated Annealing* as novas soluções são geradas a partir de perturbações na solução corrente. Se a solução gerada for melhor do que a corrente, ela é aceita. Se for pior, ela é aceita com uma probabilidade dada pela *distribuição de Boltzman*, uma fórmula da Mecânica Estatística que dá a probabilidade de um sistema em equilíbrio térmico se encontrar em um determinado estado. Além da temperatura, a distribuição de Boltzman é uma função da energia do sistema (que é uma função da posição de suas partículas).

Um algoritmo básico de *Simulated Annealing* é dado a seguir:

Algoritmo Simulated Annealing

1- Gere uma solução inicial.
2- Faça solução corrente igual à solução inicial.
3- Estabeleça uma temperatura T inicial.
4- Enquanto o critério de parada não for atingido faça:

4.1- Gere uma nova solução a partir de uma perturbação na solução corrente e calcule a diferença entre a função objetivo da nova solução gerada e da solução corrente (Δc).

4.2- Se $\Delta c \geq 0$ (nova solução tem valor pior ou igual à corrente)

4.2.1- Gere um número aleatório (α) no intervalo $[0, 1]$.

4.2.2- Calcule a probabilidade de aceitação $P = e^{-\Delta c/T}$.

4.2.3- Se $\alpha \leq P$, aceite a solução gerada.

4.3- Se $\Delta c < 0$ (nova solução melhor do que a corrente), então a nova solução é a solução corrente.

4.4- Se o equilíbrio na temperatura T foi atingido, diminua a temperatura.
5- Retorne a melhor solução obtida.

A decisão de se o equilíbrio foi atingido em uma dada temperatura pode ser tomada com base no número de iterações ocorridas naquela temperatura, número de soluções aceitas naquela temperatura, número de iterações sem ocorrência de aceitação etc. A diminuição da temperatura usualmente é feita multiplicando-se uma constante $0 < \epsilon < 1$ pela temperatura corrente.

Capítulo 3

O PCTSP

Neste capítulo definimos o *Prize Collecting Traveling Salesman Problem* (Problema do Caixeiro Viajante Coletor de Prêmios, ou PCTSP) e fazemos algumas considerações sobre sua complexidade. Em seguida, mostramos alguns algoritmos heurísticos disponíveis na literatura que se aplicam ao PCTSP. Terminamos com a apresentação de algoritmos heurísticos de construção e de melhoria de soluções propostos para este problema e sua comparação com o algoritmo de melhor garantia de desempenho da literatura.

3.1 Definição do Problema

O *Prize Collecting Traveling Salesman Problem* pode ser formulado como segue [Bal89]: um caixeiro viajante que ganha um prêmio w_k em cada cidade k que visita e paga uma penalidade p_l para cada cidade l que deixa de visitar, e viaja entre as cidades i e j a um custo c_{ij} quer minimizar a soma de seus custos de viagem e penalidades pagas, sendo que deve incluir em sua trajetória cidades suficientes para ganhar uma quantidade mínima w_0 de prêmio.

Se fizermos y_i ser 1 se a cidade i está na trajetória e 0 caso contrário, x o vetor de incidência da trajetória (x_{ij} igual a 1 se, e somente se, em sua trajetória o caixeiro viajante vai da cidade i para a cidade j) e denotarmos por n o número de cidades a serem consideradas, então o problema pode ser formulado em um grafo completo dirigido $G' = (N, A)$ como

$$\min \sum_{i \in N} \sum_{j \in N - \{i\}} c_{ij} x_{ij} + \sum_{i \in N} p_i (1 - y_i)$$

sujeito a

$$\begin{aligned}
\sum_{j \in N-\{i\}} x_{ij} - y_i &= 0, \quad i = 1, \ldots, n \\
\sum_{i \in N-\{j\}} x_{ij} - y_j &= 0, \quad j = 1, \ldots, n \\
\sum_{i \in N} w_i y_i &\geq w_0 \\
y_i &\in \{0,1\}, \quad i \in N \\
x_{ij} &\in \{0,1\}, \quad (i,j) \in A
\end{aligned}$$

$$G'(y,x) \text{ é um ciclo}$$

Aqui, $G'(y,x)$ é o subgrafo de G' cujos nós e arcos são aqueles definidos por y e x, respectivamente, e por ciclo entendemos um caminho dirigido fechado.

A fim de verificar mais facilmente o fato de que o PCTSP é uma generalização do TSP, é conveniente modificar a definição anterior da seguinte forma: introduzem-se n novas variáveis $x_{ii} = 1 - y_i, i \in V$, sendo interpretadas como laços (arcos que vão de um nó para o mesmo) de um grafo $G = (N, A \cup O)$ obtido de G' acrescentando a cada nó um laço (O é o conjunto dos laços). O vetor de incidência $(y,x) \in \{0,1\}^{n^2}$ de nós e arcos de G' é então substituído pelo vetor de incidência $x \in \{0,1\}^{n^2}$ de laços e arcos de G. Se definirmos $c_{ii} := p_i, \forall i \in N$, e $U := \sum_{i \in N} w_i - w_0$, o problema pode ser reescrito como:

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij}$$

sujeito a:

$$\begin{aligned}
\sum_{j=1}^{n} x_{ij} &= 1, \quad i = 1, \ldots, n \\
\sum_{i=1}^{n} x_{ij} &= 1, \quad j = 1, \ldots, n \\
\sum_{i=1}^{n} w_i x_{ii} &\leq U \\
x_{ij} &\in \{0,1\}, \quad i,j = 1, \ldots, n
\end{aligned}$$

$$G(x) \text{ tem exatamente um ciclo de comprimento} \geq 2$$

Aqui, $G(x)$ é o subgrafo de G com conjunto N de nós e um conjunto de arcos e laços definido por x. G deve ter apenas um ciclo de comprimento maior ou igual a 2, que seria exatamente a trajetória do caixeiro viajante, mas pode ter vários ciclos de comprimento 1,

ou seja, laços, os quais denotam que aquela cidade não faz parte da trajetória. A Figura 3.1 ilustra um exemplo de solução de uma instância do PCTSP com cinco cidades, onde apenas três participam da trajetória. Note que para $U < \min w_i$ o problema se torna o TSP tradicional, uma vez que nenhum vértice poderá ficar sozinho em um laço sem violar a restrição $\sum_{i=1}^{n} w_i x_{ii} \leq U$.

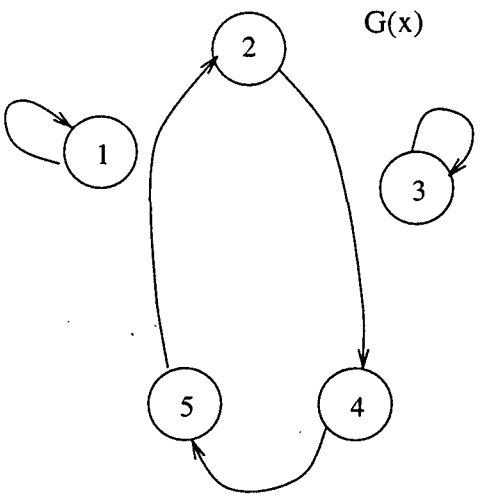


Figura 3.1: Exemplo de uma solução para o PCTSP.

A definição geral do problema assume um grafo dirigido completo. No entanto, certos algoritmos disponíveis na literatura lidam com versões mais reduzidas do problema, como grafo não-dirigido (dando origem à versão simétrica do problema), ou pesos nas arestas obedecendo à chamada desigualdade triangular, ou seja, para quaisquer três arestas (i,j), (j,k) e (i,k) em G, $c_{ij} \leq c_{ik} + c_{kj}$.

Um fato interessante a ser notado no PCTSP é a relação que pode ser feita entre as penalidades dos vértices e o seu custo de inserção em uma rota: seja C^i_{jk} o custo (em termos de acréscimo no comprimento da rota) da inclusão do vértice i entre os vértices j e k em uma rota qualquer, e p_i a penalidade do vértice i (veja a Figura 3.2). Se para todo vértice i do grafo valer a relação $p_i > \max\{C^i_{jk}\} \forall j, k \neq i$, então a solução ótima desta instância do PCTSP deverá ser uma rota que envolva todos os vértices do grafo (o problema se reduz ao TSP, portanto). Isto pode ser visto facilmente: tome-se uma rota qualquer em $k-1$ vértices. A inclusão de um dos vértices restantes nessa rota obrigatoriamente diminuirá o valor da função objetivo, se a relação citada prevalecer. Logo, prosseguindo-se desta maneira, temos que as soluções de menor valor de função objetivo serão aquelas que incluem todos os vértices na rota.

Um outro fato a ser notado é a semelhança entre o ato de escolha das cidades que não participarão da rota (e conseqüentemente daquelas que entrarão na rota) e a resolução de um outro problema clássico de Otimização Combinatória, conhecido como Problema da

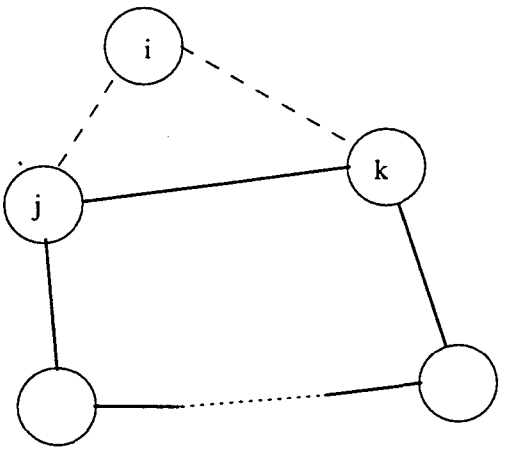


Figura 3.2: Exemplo da inserção de um vértice em uma rota qualquer.

Mochila (*Knapsack Problem*)[Apo91, SCGD92], pertencendo também à classe NP-difícil, com a diferença de que existem algoritmos pseudo-polinomiais para resolvê-lo [MT90].

O *Knapsack Problem* pode ser assim definido: dados n objetos, cada um deles com um peso w_i e um valor v_i associados, e uma mochila de capacidade finita M, devemos escolher quais objetos colocar na mochila de forma a maximizar o valor total obtido ($\sum_{i=1}^{n} v_i$) respeitando a capacidade da mochila ($\sum_{i=1}^{k} w_i \leq M$, onde w_i representa o peso de um objeto colocado na mochila e $k \leq n$ o número de objetos colocados na mochila). No caso do PCTSP podemos considerar os objetos como sendo as cidades e o preenchimento da mochila como sendo a escolha das cidades que não participarão da rota. Assim, a constante U descrita anteriormente fica sendo a capacidade da mochila. A diferença é que, como o PCTSP é um problema de minimização, não podemos considerar como função objetivo apenas a soma das penalidades das cidades dentro da mochila, já que isso levaria à solução trivial em que a mochila fica vazia. Devemos, pois, definir a função objetivo como sendo a soma das penalidades das cidades na mochila com a menor trajetória nas cidades que ficaram de fora. Note que a segunda parte da função objetivo envolve a solução de uma instância do TSP. Isso leva ao problema de que a própria avaliação da função objetivo envolveria a resolução de um problema NP-difícil.

Embora o PCTSP possa, portanto, de certa forma ser "decomposto" no Problema da mochila (escolha das cidades a participarem da rota) e no TSP tradicional (resolução do TSP para estas cidades), esta decomposição não é totalmente possível, uma vez que os dois problemas estão intimamente entrelaçados: a escolha das cidades limita as opções disponíveis para a definição da rota, sendo que os dois problemas devem ser resolvidos simultaneamente.

Como exemplo de algumas aplicações do *Prize Collecting* TSP podemos citar:

- O problema de formação de "caixões" ou "cones" de laminação em indústrias si-

derúrgicas [Bal89]: o aço a ser utilizado pelas indústrias, por exemplo a automobilística, normalmente é produzido em forma de bobinas, que nada mais são do que uma chapa de aço muito comprida enrolada em torno de si mesma diversas vezes. Diversos equipamentos dentro de uma usina siderúrgica efetuam algum processo nas bobinas, tais como revestimento com outros materiais (zinco, estanho, cromo), laminação (diminuição de espessura), limpeza, oleamento, apara de bordas etc. A seqüência de bobinas que passa através de um equipamento destes normalmente deve seguir determinadas regras, tais como: as bobinas devem ser processadas da mais larga para a mais estreita, a diferença de espessura (ou largura) entre duas bobinas consecutivas deve ser menor que determinado limite etc. Além disso, na maioria dos casos o equipamento têm um limite na tonelagem de material que pode processar antes que algum tipo de manutenção (geralmente troca de cilindros) tenha que ser efetuada. Ou seja, dado um conjunto de bobinas para um equipamento, deve ser decidido quais bobinas (atendendo à tonelagem máxima) e em que seqüência elas devem ser processadas, de modo a minimizar os custos associados à transição de uma bobina para outra e que se atenda a data prevista para o término da produção.

- O problema de determinar a seqüência em que peças constituintes de um equipamento devem ser desmontadas para reciclagem, sendo que deve ser determinado também o ponto em que é economicamente mais lucrativo sucatear a parte que restou do equipamento do que investir em mais desmonte [Nav93].

3.1.1 Complexidade

O PCTSP é um problema da classe NP-difícil [BGSW93]. Não existe até o momento, portanto, nenhum algoritmo que o resolva em tempo polinomial no tamanho da instância [GJ79].

Para se ter uma idéia do quão impraticável seria a abordagem mais simples para a resolução exata do problema, ou seja, enumerar todas as soluções possíveis e escolher a de menor valor de função objetivo, apresentamos a seguir o número de soluções possíveis para uma instância do PCTSP com n vértices (S_n), desprezando-se a restrição do prêmio mínimo. Este número é dado pela expressão abaixo, que reflete o número de ciclos em um grafo não dirigido:

$$\begin{aligned} S_n &= \frac{1}{2}\sum_{i=3}^{n}\binom{n}{i}(i-1)! \\ &= \binom{n}{3}\frac{(3-1)!}{2} + \cdots + \binom{n}{n-1}\frac{(n-1-1)!}{2} + \binom{n}{n}\frac{(n-1)!}{2} \end{aligned}$$

A Tabela 3.1 mostra a relação entre o número de soluções possíveis para instâncias de diversos tamanhos do TSP (dado pelo último termo da somatória acima) e do PCTSP. Embora o número de soluções possíveis do PCTSP seja, em média, apenas três vezes maior do que o do TSP, vale ressaltar que ele parece ser mais difícil, devido à restrição de prêmio mínimo e ao fato de a função objetivo envolver não apenas o comprimento da rota, mas também as penalidades dos vértices.

Número de vértices	Número de rotas do TSP	Número de rotas do PCTSP	Relação PCTSP/TSP
3	1	1	1,0
6	60	197	3,28333
10	181440	556014	3,064451
50	3,04e+39	8,44e+39	2.774938
100	4,66e+155	1,28e+156	2,746022
200	1,97e+367	5,38e+367	2,732011
500	1,22e+1115	3,32e+1115	2,723740

Tabela 3.1: Relação entre os números de rotas para instâncias do TSP e PCTSP.

3.2 Algoritmos Heurísticos da Literatura

Nesta seção apresentamos alguns algoritmos heurísticos para o PCTSP disponíveis na literatura, bem como análises sobre suas garantias de desempenho e complexidade de execução.

3.2.1 Algoritmo MLP (*Modified Linear Programming Relaxation*)

Este algoritmo, desenvolvido por Bienstock *et al.* [BGSW93], considera instâncias simétricas do PCTSP, que obedecem à desigualdade triangular e sem a restrição de prêmio mínimo. Para análise deste algoritmo, é conveniente formular o PCTSP da seguinte maneira: sejam Z^* o valor de uma solução ótima de uma instância do PCTSP, e $Z^*(j)$ o valor de uma solução ótima quando o vértice j deve ser incluído na trajetória. Desta forma, claramente temos que:

$$Z^* = \min\{\sum_{i \in V} p_i, \min_{j \in V}\{Z^*(j)\}\}$$

O Problema do Caixeiro Viajante Coletor de Prêmios, quando um determinado vértice $j \in V$ deve estar presente na trajetória, pode ser modelado usando Programação Linear

Inteira da seguinte forma: seja $G = (V, E)$ um grafo completo não dirigido com pesos associados às arestas (c_e é o peso associado à aresta $e \in E$), p_i a penalidade associada ao vértice $i \in V$, $\delta(S)$ o *cut set* do conjunto de vértices S, ou seja, o conjunto de arestas que têm uma extremidade em S e outra em $V \setminus S$, x o vetor de incidência da trajetória e y_i a variável indicando a pertinência ou não da cidade i à trajetória:

Problema $P_1(j)$

$$\min \sum_{e \in E} c_e x_e + \sum_{i \in V} p_i(1 - y_i)$$

sujeito a

$$\begin{aligned}
\sum_{e \in \delta(\{i\})} x_e &= 2y_i, \quad \forall i \in V \\
\sum_{e \in \delta(S)} x_e &\geq 2y_i, \quad \forall i \in V, S \subset V \text{ tal que } |S \cap \{i, j\}| = 1 \\
x_e &\in \{0, 1\}, \quad \forall e \in E \\
y_i &\in \{0, 1\}, \quad i, j \in V, \text{com } i \neq j \\
y_j &= 1
\end{aligned}$$

Assim, se um algoritmo resolve o problema $P_1(j)$ com alguma garantia de desempenho, esta garantia se mantém para o PCTSP, pois bastará aplicar o algoritmo $|V|$ vezes, cada uma obrigando a entrada de determinado vértice do grafo na trajetória, e tomar a solução de menor valor como solução do PCTSP.

O algoritmo MLP se baseia na resolução da relaxação linear do problema $P_1(j)$, onde as restrições $x_e \in \{0, 1\}$ e $y_i \in \{0, 1\}$ são substituídas por $0 \leq x_e \leq 1$ e $0 \leq y_i \leq 1$, respectivamente. Ele usa, ainda, o algoritmo de Christofides para o TSP [Chr76, JP85], que constrói uma trajetória no máximo $\frac{3}{2}$ vezes maior do que o valor de uma solução ótima de uma instância do TSP que obedeça à desigualdade triangular. O algoritmo está descrito a seguir.

Algoritmo MLP

1- Resolvem-se $|V|$ relaxações lineares de $P_1(j)$, onde j é sucessivamente colocado como cada um dos vértices do grafo.

2- Sejam $\bar{x}$ e $\bar{y}$ os vetores de incidência ótimos para a relaxação linear de $P_1(j)$. Constróem-se dois novos vetores da seguinte forma:

$\hat{x}_e = \frac{5}{3}\bar{x}_e, \quad \forall e \in E$

e, para qualquer $i \in V$,

$$\hat{y}_i = \begin{cases} 1, \text{se } \bar{y}_i \geq \frac{3}{5} \\ 0, \text{caso contrário.} \end{cases}$$

3- Seja $T = \{i \in V | \hat{y}_i = 1\}$. Constrói-se uma trajetória sobre os vértices em T usando o algoritmo de Christofides.

4- Seja $L^C(T)$ o comprimento da trajetória retornada pelo algoritmo de Christofides. Definimos $Z^{MLP}(j)$ como sendo o valor da heurística MLP para o problema $P_1(j)$, ou seja,

$$Z^{MLP}(j) = L^C(T) + \sum_{i \in V} p_i(1 - \hat{y}_i).$$

5- Escolhem-se destas $|V|$ soluções heurísticas a que tem o menor valor de função objetivo ou a solução em que nenhum vértice é visitado. Portanto, o valor obtido com a heurística MLP (Z^{MLP}) é dado por:

$$Z^{MLP} = \min\{\sum_{i \in V} p_i, \min_{j \in V}\{Z^{MLP}(j)\}\}.$$

Garantia de desempenho

A heurística MLP apresenta um desempenho de pior caso de 2,5 vezes o valor de uma solução ótima, ou seja, $\frac{Z^{MLP}}{Z^*} \leq \frac{5}{2}$. Para provarmos isto, basta mostrar que este desempenho é obtido em cada problema $P_1(j)$. Para tanto, definimos o seguinte programa linear:

Problema P_2:

$$\min \sum_{e \in E} c_e x_e$$

sujeito a

$$\sum_{e \in \delta(S)} x_e \geq 2, \quad \forall S \subset V \text{ tal que } T \cap S \neq \emptyset, T \cap (V \setminus S) \neq \emptyset \tag{3.1}$$

$$\sum_{e \in \delta(\{i\})} x_e = 2, \quad \forall i \in T \tag{3.2}$$

$$\sum_{e \in \delta(\{i\})} x_e = 0, \quad \forall i \notin T \tag{3.3}$$

$$x_e \geq 0, \quad \forall e \in E \tag{3.4}$$

Note que P_2 com a restrição adicional de que os x_e sejam inteiros é a formulação em programação inteira do TSP sobre os vértices de T [DFJ54, BT85]. Shmoys e Williamson [SW90] provaram que, para a solução ótima de um programa linear da forma de P_2

(Z^{P_2}), vale o seguinte limite, onde $L^C(T)$ é o valor da rota obtida através da aplicação do algoritmo de Christofides aos vértices de T:

$$\frac{Z^{P_2}}{L^C(T)} \geq \frac{2}{3}$$

Goemans e Bertsimas [GB90] provaram que a solução ótima de P_2 não é alterada caso as restrições (3.2) e (3.3) sejam retiradas. Seja P_3 o problema P_2 somente com as restrições (3.1) e (3.4). Assim, denotando por $\acute{x}$ o vetor de incidência de uma solução ótima de P_2 (e P_3), temos que:

$$L^C(T) \leq \frac{3}{2} \sum_{e \in E} c_e \acute{x}_e$$

Devemos mostrar agora que $\hat{x}$ é um vetor de incidência de uma solução viável para P_3. Claramente ele satisfaz (3.4). Para provar que também satisfaz (3.1), considere qualquer $S \subset V$ tal que $i \in T \cap S$ e $j \in T \setminus S$. Pelo fato de que $\bar{x}$ é um vetor de uma solução válida para $P_1(j)$, e pela definição de T, temos:

$$\sum_{e \in \delta(S)} \bar{x}_e \geq 2\bar{y}_i \geq 2(\frac{3}{5}) = \frac{6}{5}, \forall S \subset V, \text{ tal que } T \cap S \neq \emptyset, T \cap (V \setminus S) \neq \emptyset$$

Portanto, para qualquer $S \subset V$ tal que $T \cap S \neq \emptyset$ e $T \cap (V \setminus S) \neq \emptyset$, temos:

$$\sum_{e \in \delta(S)} \hat{x}_e = \frac{5}{3} \sum_{e \in \delta(S)} \bar{x}_e \geq 2$$

Desta forma, $\hat{x}$ satisfaz (3.1). Conseqüentemente, como $\acute{x}_e$ é o vetor de incidência de uma solução ótima para P_3, temos:

$$\sum_{e \in E} c_e \hat{x}_e \geq \sum_{e \in E} c_e \acute{x}_e.$$

Assim, temos:

$$\begin{aligned}
Z^{MLP}(j) &= L^C(T) + \sum_{i \in V} p_i(1 - \hat{y}_i) \\
&\leq \frac{3}{2} \sum_{e \in E} c_e \acute{x}_e + \sum_{i \in V} p_i(1 - \hat{y}_i) \\
&\leq \frac{3}{2} \sum_{e \in E} c_e \hat{x}_e + \sum_{i \in V} p_i(1 - \hat{y}_i) \\
&\leq \frac{3}{2} \sum_{e \in E} c_e \frac{5}{3} \bar{x}_e + \frac{5}{2} \sum_{i \in V} p_i(1 - \bar{y}_i) \\
&= \frac{5}{2} \{ \sum_{e \in E} c_e \bar{x}_e + \sum_{i \in V} p_i(1 - \bar{y}_i) \}
\end{aligned}$$

$$Z^{MLP}(j) \leq \frac{5}{2} Z^*(j).$$

O algoritmo MLP, apesar da boa garantia de desempenho, se atém a uma versão bastante restrita do PCTSP. Uma outra desvantagem deste algoritmo é a necessidade de resolução de $|V|$ problemas de programação linear, o que torna seu desempenho em termos de tempo de execução não muito bom, embora polinomial.

3.2.2 Algoritmo de Goemans e Williamson

Este algoritmo, descrito em [GW92], serve a uma ampla classe de problemas, denominados de *constrained forest problems*. Embora o PCTSP não se enquadre nesta classe, os autores fazem uma pequena adaptação no algoritmo e conseguem aplicá-lo à versão simétrica do *Prize Collecting* TSP, com pesos nas arestas obedecendo à desigualdade triangular e sem considerar a restrição de prêmio mínimo. Além disso, ele considera a versão em que um determinado vértice r, denominado raiz, deve estar presente na trajetória. Como no algoritmo MLP, isto não tira a generalidade do algoritmo, bastando aplicá-lo $|V|$ vezes, considerando cada vértice como raiz, e tomar a melhor solução.

O algoritmo constrói uma solução para o PCTSP com base em uma solução do problema da árvore de Steiner com penalidades associadas aos vértices (ou *Prize Collecting Steiner Tree Problem* - PCSTP) com vértice raiz, que consiste em, dados um grafo $G = (V, E)$ onde $|V| = n$, com pesos associados às arestas e penalidades associadas aos vértices, e um vértice raiz $r \in V$, deve-se encontrar uma árvore sobre os vértices do grafo que conecte o vértice r e minimize uma função objetivo que é dada pela soma do peso da árvore (soma dos pesos das arestas que dela participam) e das penalidades dos vértices que não foram conectados por esta. O PCSTP é uma generalização do problema de Árvore de Steiner tradicional, que pode ser obtido através do PCSTP associando-se uma penalidade igual a 0 para todos os vértices não-terminais e uma penalidade infinita para os vértices terminais. O PCSTP com vértice raiz r pode ser formulado em programação inteira como:

$$\min \sum_{e \in E} c_e x_e + \sum_{T \subset V, r \notin T} z_T \left(\sum_{i \in T} p_i \right)$$

sujeito a

$$\begin{aligned}
\sum_{e \in \delta(S)} x_e + \sum_{T \supseteq S} z_T &\geq 1, \quad S \subset V, r \notin S \\
\sum_{T \subset V, r \notin T} z_T &\leq 1 \\
x_e &\in \{0,1\}, \quad e \in E \\
z_T &\in \{0,1\}, \quad T \subset V, r \notin T
\end{aligned}$$

Na formulação acima, as variáveis z_T são definidas para cada subconjunto T de vértices, e $z_T = 1$ se os vértices de T não são conectados pela árvore, e $z_T = 0$ caso contrário. Uma relaxação linear deste programa inteiro é definida trocando as restrições de integralidade por $x_e \geq 0$ e $z_T \geq 0$ e retirando a restrição $\sum_T z_T \leq 1$. A retirada desta restrição não altera o valor de uma solução ótima [GW92]. Goemans e Williamson mostram que esta relaxação linear é equivalente à seguinte:

$$\min \sum_{e \in E} c_e x_e + \sum_{i \neq r} (1 - s_i) p_i$$

sujeito a

$$\begin{aligned} \sum_{e \in \delta(S)} x_e &\geq s_i, \quad S \subset V, i \in S, r \notin S \\ x_e &\geq 0, \quad e \in E \\ s_i &\geq 0, \quad i \in V, i \neq r \end{aligned}$$

O algoritmo de Goemans e Williamson constrói implicitamente uma solução gulosa (*greedy*) para o dual desta relaxação linear, que é:

$$\max \sum_{S; r \notin S} y_S$$

sujeito a

$$\begin{aligned} \sum_{S: e \in \delta(S)} y_S &\leq c_e, \quad e \in E \\ \sum_{S \subseteq T} y_S &\leq \sum_{i \in T} p_i, \quad T \subset V, r \notin T \\ y_S &\geq 0, \quad S \subset V, r \notin S \end{aligned}$$

O algoritmo de Goemans e Williamson para o problema da árvore de Steiner com penalidades e vértice raiz r é descrito a seguir.

Algoritmo GW-PCSTP

1-$F \leftarrow \emptyset$ (*implicitamente faz* $y_S \leftarrow 0$ *para todo* $S \subset V$)
2-$\mathcal{C} \leftarrow \{\{v\} : v \in V\}$
3- Para cada $v \in V$ faça
  3.1- Desmarque v
  3.2- $d(v) \leftarrow 0$
  3.3- $w(\{v\}) \leftarrow 0$
  3.4- Se $v = r$ então $\lambda(\{v\}) \leftarrow 0$ senão $\lambda(\{v\}) \leftarrow 1$

4- Enquanto $\exists C \in \mathcal{C} : \lambda(C) = 1$ faça

4.1- Encontre a aresta $e = (i, j)$ com $i \in C_p \in \mathcal{C}, j \in C_q \in \mathcal{C}, C_p \neq C_q$ que minimize $\epsilon_1 = \frac{c_e - d(i) - d(j)}{\lambda(C_p) + \lambda(C_q)}$

4.2- Encontre $\tilde{C} \in \mathcal{C}$ com $\lambda(\tilde{C}) = 1$ que minimize $\epsilon_2 = \sum_{i \in \tilde{C}} p_i - w(\tilde{C})$

4.3- $\epsilon = \min(\epsilon_1, \epsilon_2)$

4.4- $w_C \leftarrow w_C + \epsilon.\lambda(C)$ para todo $C \in \mathcal{C}$

(implicitamente faz $y_C \leftarrow y_C + \epsilon.\lambda(C)$*para todo* $C \in \mathcal{C}$*)*

4.5- Para todo $v \in C_r \in \mathcal{C}$

4.5.1- $d(v) \leftarrow d(v) + \epsilon.\lambda(C_r)$

4.6- Se $\epsilon = \epsilon_2$

$\lambda(\tilde{C}) \leftarrow 0$

Marque todos os vértices sem rótulo de $\tilde{C}$ com rótulo $\tilde{C}$

senão

$F \leftarrow F \cup \{e\}$

$\mathcal{C} \leftarrow \mathcal{C} \cup \{C_p \cup C_q\} - \{C_p\} - \{C_q\}$

$w(C_p \cup C_q) \leftarrow w(C_p) + w(C_q)$

Se $r \in C_p \cup C_q$ então $\lambda(C_p \cup C_q) \leftarrow 0$ senão $\lambda(C_p \cup C_q) \leftarrow 1$

5- F' é derivado de F removendo o maior número de arestas possível, mantendo as seguintes propriedades:

(1) todo vértice não rotulado deve estar conectado a r;

(2) se o vértice v com rótulo C está conectado a r, então todo vértice com rótulo $C' \supseteq C$ também deverá estar.

6- X é o conjunto de todos os vértice não conectados por F'.

O algoritmo mantém uma floresta F de arestas, inicialmente vazia. Mantém, também, os vértices do grafo divididos em componentes, cada um deles podendo estar ativo ($\lambda = 1$) ou inativo ($\lambda = 0$). O conjunto de todos os componentes é chamado de $\mathcal{C}$. Inicialmente cada componente tem apenas um vértice, e todos os componentes, exceto o que contém a raiz, estão ativos. O laço principal termina quando não há mais componentes ativos.

A fim de clarificar o mecanismo de funcionamento do algoritmo, damos a seguir um exemplo da aplicação do mesmo sobre o grafo da Figura 3.3. Inicialmente, F é um conjunto vazio e o conjunto $\mathcal{C}$ de componentes contém 5 componentes, cada um contendo um vértice. Além disso todos os vértices ficam sem rótulos, as variáveis d e w são iniciadas com 0 para todos os vértices/componentes e a variável λ é colocada como 1 para todos os componentes exceto o do vértice raiz (A).

As Figuras 3.4, 3.5, 3.6, 3.7 e 3.8 mostram o grafo após as iterações do laço do passo 4 do algoritmo. Os componentes são ilustrados pela envoltória pontilhada para componentes inativos e tracejada para componentes ativos, enquanto que as arestas pertencentes a F

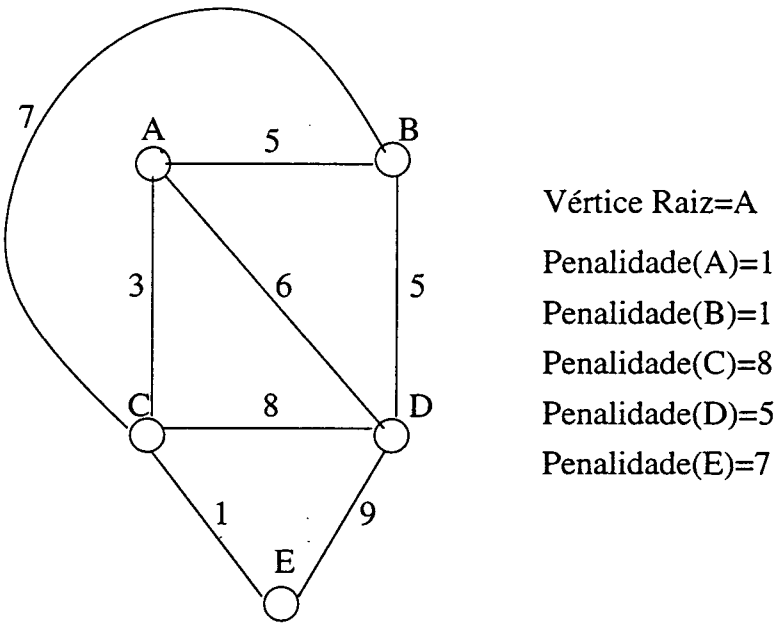


Figura 3.3: Grafo de exemplo da execução do algoritmo GW-PCSTP.

são destacadas.

Após a quinta iteração, não há mais componentes ativos, então o algoritmo passa para o passo 5. Neste passo, a aresta BD não é colocada em F', uma vez que sua remoção não viola nenhuma das propriedades descritas. A árvore resultante do algoritmo é, portanto, composta das arestas AC e CE. O conjunto de vértices fora da árvore é $X = \{B, D\}$.

As escolhas do algoritmo são motivadas por uma construção gulosa de uma solução para o problema dual. Inicialmente, todas as variáveis duais são colocadas em 0. A cada iteração do laço principal o algoritmo aumenta o valor de y_C, para todos os componentes C ativos, por um valor ϵ que é o maior possível sem violar as restrições do problema dual: $\sum_{S:e\in\delta(S)} y_S \leq c_e$, para todo $e \in E$, e $\sum_{S\subseteq T} y_S \leq \sum_{i\in T} p_i$, para todo $T \subset V$. Aumentar de ϵ o valor de y_C para um C ativo vai fazer com que uma das restrições do dual não admita mais folga. Se isso acontece com uma restrição do primeiro tipo, significa que isso foi causado por alguma aresta e entre dois componentes; então adicionamos esta aresta a F. Se foi uma restrição do segundo tipo que deixou de ter folga, significa que o causador foi algum componente C; o algoritmo, então, desativa este componente. O passo final do algoritmo remove o máximo possível de arestas de F, mantendo duas propriedades. Primeiro, todos os vértices sem rótulos devem estar conectados à raiz, uma vez que estes vértices nunca estiveram em nenhum componente desativado, ou seja, o algoritmo nunca os deixou fora da árvore a ser construída. Em segundo, se um vértice com rótulo C está conectado à raiz, então todos os vértices com rótulos $C' \supseteq C$ também devem estar.

O algoritmo para o PCTSP com vértice raiz (que usa o algoritmo anterior) está descrito

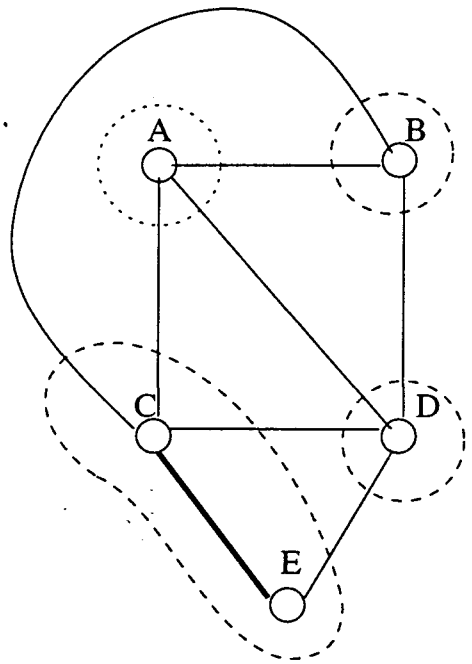


Figura 3.4: Grafo de exemplo após primeira iteração.

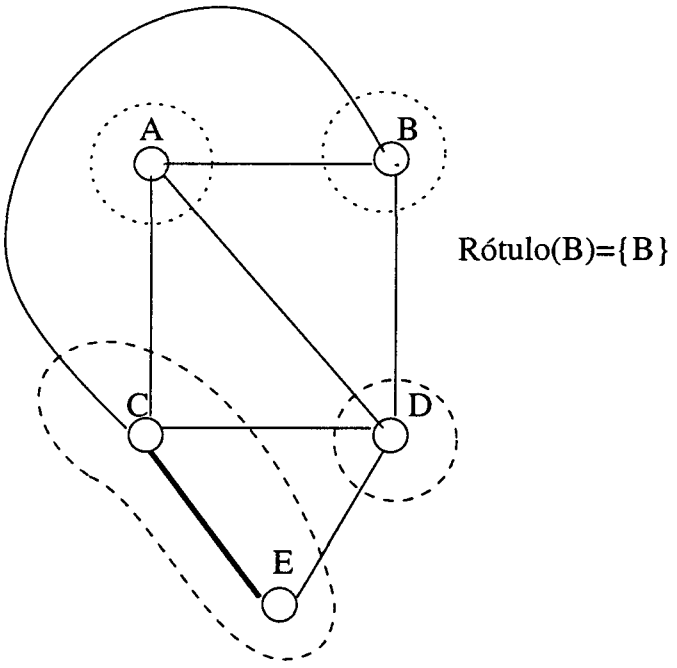


Figura 3.5: Grafo de exemplo após segunda iteração.

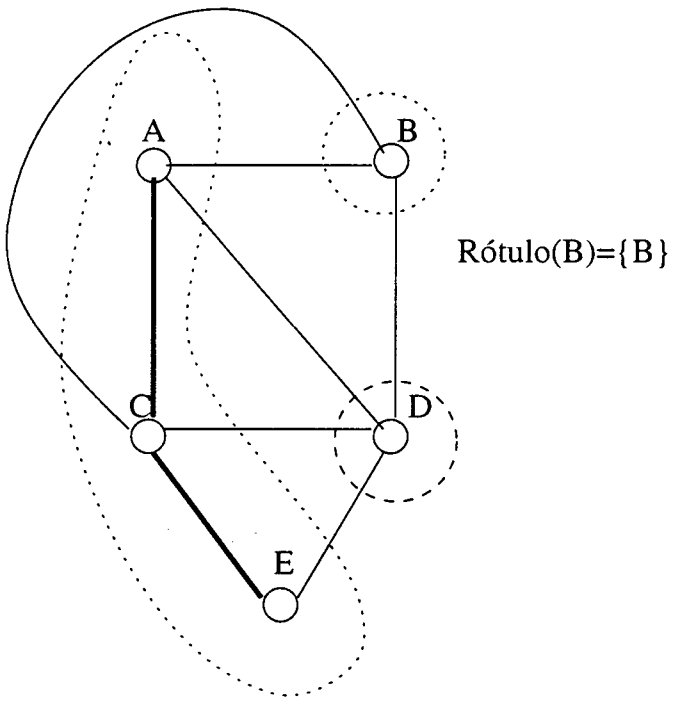


Figura 3.6: Grafo de exemplo após terceira iteração.

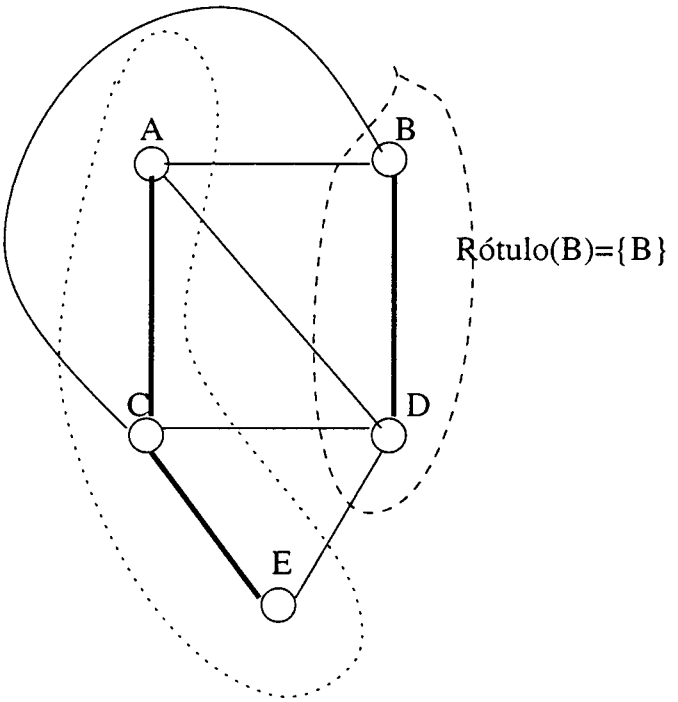


Figura 3.7: Grafo de exemplo após quarta iteração.

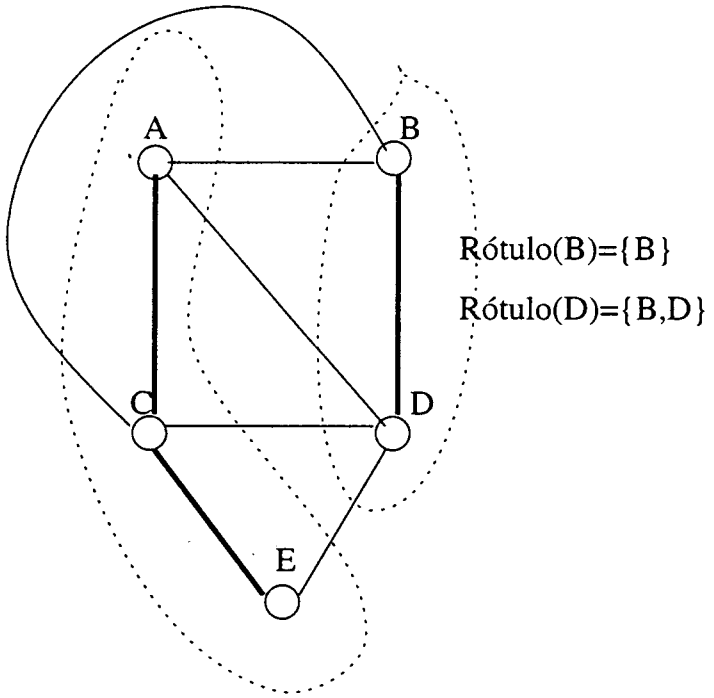


Figura 3.8: Grafo de exemplo após quinta iteração.

a seguir:

Algoritmo GW-PCTSP

1- Aplique o algoritmo GW-PCSTP à instância do problema com grafo G, custos nas arestas c, raiz r e penalidades $p'_i = p_i/2$.
2- Duplique as arestas em F' da árvore retornada para formar um grafo Euleriano T.
3- Faça atalhos em T para formar uma trajetória T'. Seja X o conjunto dos vértices fora da trajetória.

Os atalhos no passo 3 são feitos da mesma forma que no algoritmo de Christofides para o TSP [Chr76, JP85]. Para resolver a versão do problema sem raiz basta, portanto, aplicar o algoritmo acima n vezes, cada uma delas considerando como raiz um vértice diferente e retornando o melhor resultado obtido.

Garantia de desempenho

Denominando de Z^*_{STP} o valor de uma solução ótima para o problema de Steiner com penalidades nos vértices, e sendo y o vetor de variáveis duais da relaxação linear do problema, a árvore F' e o conjunto de vértices X retornados pelo algoritmo GW-PCSTP obedecem à seguinte relação:

$$\sum_{e \in F'} c_e + \sum_{i \in X} p_i \leq (2 - \frac{1}{n-1}) \sum_{S \subset V} y_S \leq (2 - \frac{1}{n-1}) Z^*_{STP}$$

Ou seja, o algoritmo GW-PCSTP produz uma solução válida para o problema de Steiner com penalidades nos vértices e esta solução é no máximo $(2 - \frac{1}{n-1})$ vezes o valor de uma solução ótima, onde n é o número de vértices do grafo.

Não é difícil ver que o algoritmo produz uma solução válida para o PCSTP, uma vez que F' não tem nenhum componente não-trivial que não contenha r, e o componente que contém r é uma árvore.

Pela construção de F', cada vértice não conectado por F', ou seja, os vértices de X, ficou em algum componente desativado em algum ponto da execução do algoritmo. Além disso, se o vértice esteve em algum componente desativado C, então nenhum dos vértices de C estão conectados por F'. Assim, podemos particionar os vértices de X em componentes desativados disjuntos $C_1, \ldots, C_k$. Estes conjuntos são os rótulos maximais dos vértices em X. Já que cada C_j é um componente desativado, temos que $\sum_{S \subseteq C_j} y_S = \sum_{i \in C_j} p_i$, e portanto a desigualdade a ser provada é implicada por $\sum_{e \in F'} c_e + \sum_j \sum_{S \subseteq C_j} y_S \leq (2 - \frac{1}{n-1}) \sum_{S \subset V} y_S$. Além do mais, uma vez que $c_e = \sum_{S: e \in \delta(S)} y_S$ para cada $e \in F'$ pela construção do algoritmo, tudo que devemos provar é:

$$\sum_{e \in F'} \sum_{S: e \in \delta(S)} y_S + \sum_j \sum_{S \subseteq C_j} y_S \leq (2 - \frac{1}{n-1}) \sum_{S \subset V} y_S$$

Ou, reescrevendo a desigualdade acima:

$$\sum_S y_S |F' \cap \delta(S)| + \sum_j \sum_{S \subseteq C_j} y_S \leq (2 - \frac{1}{n-1}) \sum_{S \subset V} y_S$$

Isto pode ser provado por indução no laço principal do algoritmo. Em uma iteração qualquer, seja $\mathcal{C}$ o conjunto de componentes ativos da iteração. Seja H o grafo formado considerando-se componentes ativos e inativos como vértices e as arestas $e \in \delta(C) \cap F'$ para componentes C ativos como as arestas de H. Retire todos os vértices inativos isolados. Sejam: N_a o conjunto de vértices ativos em H, N_i o conjunto de vértices inativos, N_d o conjunto de vértices ativos correspondendo a conjuntos ativos contidos em algum C_j e d_v o grau de um vértice v em H. Note que $N_d = \{v \in N_a : d_v = 0\}$. Nessa iteração, o aumento no lado esquerdo da desigualdade é $\epsilon(\sum_{v \in N_a} d_v + |N_d|)$, enquanto que o aumento no lado direito é $\epsilon(2 - \frac{1}{n-1})|N_a|$. Portanto, o que devemos provar é que $(\sum_{v \in N_a} d_v + |N_d|) \leq (2 - \frac{1}{n-1})|N_a|$. Note que o grau de qualquer vértice correspondendo a um conjunto ativo em algum C_j é zero. Portanto, basta mostrarmos que $\sum_{v \in N_a - N_d} d_v \leq (2 - \frac{1}{n-1})|N_a - N_d|$.

Para tanto, vamos mostrar que todas menos uma das folhas de H devem ser vértices ativos. Suponha que v é uma folha inativa de H, ligada pela aresta e, e seja C_v o componente inativo correspondente a v. Suponha também que C_v não contenha a raiz r.

Como C_v é inativo e não contém r, ele deve ter sido desativado. Uma vez que C_v está desativado, todos os vértices em C_v estão com rótulos. Além disso, como v é uma folha, nenhum vértice em C_v pode estar no caminho entre a raiz e um vértice que deve estar conectado a ela. Então, pela construção de F', $e \notin F'$, o que é uma contradição. Portanto, deve haver ao menos uma folha inativa, a qual deve corresponder ao componente contendo r. Logo, temos:

$$\begin{aligned}
\sum_{v \in N_a - N_d} d_v &\leq \sum_{v \in (N_a - N_d) \cup N_i} d_v - \sum_{v \in N_i} d_v \\
&\leq 2(|(N_a - N_d) \cup N_i| - 1) - (2|N_i| - 1) \\
&= 2|N_a - N_d| - 1 \\
&\leq (2 - \frac{1}{n-1})|N_a - N_d|
\end{aligned}$$

A desigualdade vale já que todos menos um dos vértices inativos tem grau pelo menos dois e o número de componentes ativos é sempre no máximo $n - 1$.

Não é difícil mostar que GW-PCTSP também tem o mesmo limite para o PCTSP. Considere a seguinte relaxação linear do PCTSP com vértice raiz:

$$\min \sum_{e \in E} c_e x_e + \sum_{T \subset V; r \notin T} z_T (\sum_{i \in T} p_i)$$

sujeito a

$$\begin{aligned}
\sum_{e \in \delta(S)} x_e + 2 \sum_{T \supseteq S} z_T &\geq 2 \quad , \quad r \notin S \\
x_e &\geq 0, \quad e \in E \\
z_T &\geq 0, \quad T \subset V, r \notin T
\end{aligned}$$

Este programa linear é uma relaxação de um programa inteiro similar ao do problema de Steiner com penalidades nos vértices, onde $z_T = 1$ para o conjunto de vértices em T não visitados pela trajetória, e $z_T = 0$ caso contrário. A restrição de que cada vértice na trajetória deva ser visitado exatamente duas vezes foi relaxada para pelo menos duas vezes. O dual desta relaxação linear é:

$$\max 2 \sum_{S: r \notin S} y_S$$

sujeito a

$$\sum_{S: e \in \delta(S)} y_S \leq c_e, \quad e \in E$$

$$\begin{aligned} 2\sum_{S\subsetneq T} y_S &\leq \sum_{i\in T} p_i, \quad T \subset V, r \notin T \\ y_S &\leq 0, \quad S \subset V, r \notin S \end{aligned}$$

Note que este dual é bastante similar ao dual do problema de Steiner. A solução dual gerada pelo algoritmo GW-PCSTP para penalidades p' será válida para o programa dual acima com penalidades p. Pela dualidade, temos que $2\sum_{S\subset V} y_S \leq Z^*_{PCTSP}$, onde Z^*_{PCTSP} é o valor da solução ótima do *Prize Collecting* TSP. Dada uma solução F' e X para o problema de Steiner com penalidades, o valor da solução para o PCTSP gerado pelo algoritmo GW-PCTSP é, no máximo, $2\sum_{e\in F'} c_e + \sum_{i\in X} p_i = 2(\sum_{e\in F'} c_e + \sum_{i\in X} p'_i)$. Devido à garantia de desempenho da solução obtida pelo algoritmo GW-PCSTP , temos que $\sum_{e\in F'} c_e + \sum_{i\in X} p'_i \leq (2 - \frac{1}{n-1}) \sum_{S\subset V} y_S$, portanto:

$$2(\sum_{e\in F'} c_e + \sum_{i\in X} p'_i) \leq 2(2 - \frac{1}{n-1}) \sum_{S\subset V} y_S \leq (2 - \frac{1}{n-1}) Z^*_{PCTSP}.$$

Ou seja, o algoritmo GW-PCTSP produz soluções para o PCTSP com um pior caso de $(2 - \frac{1}{n-1})$ vezes o ótimo. Portanto tem uma garantia de desempenho melhor do que a do algoritmo MLP (2,5 contra 2), além de ser puramente combinatorial, não requerendo a resolução de nenhum programa linear diretamente. Sua complexidade de tempo é, inicialmente, $O(n^4)$, entretanto algumas alterações no algoritmo para o problema de Steiner com penalidades podem diminuir a complexidade para $O(n^3 \log n)$ [GW92].

3.2.3 Algoritmo de Awerbuch, Azar, Blum e Vempala

Este algoritmo, descrito em [AABV94], trata da versão simétrica do PCTSP, com instâncias obedecendo à desigualdade triangular. Antes de apresentar algoritmo para o PCTSP, é conveniente mostrarmos alguns outros algoritmos (para problemas correlatos) dos quais ele faz uso. O primeiro destes é um algoritmo para o problema da k-MST. Este problema consiste em, dados: um grafo com n vértices e pesos associados às arestas; e um inteiro $k \leq n$, o objetivo é encontrar uma árvore que conecte exatamente k vértices e que tenha o menor peso total (dado pela soma dos pesos de suas arestas). O algoritmo para a k-MST por sua vez, faz uso do seguinte algoritmo.

Algoritmo *Merge-Cluster*

1- Comece com n componentes, um para cada vértice do grafo.
2- Junte dois componentes, usando o menor caminho entre eles, tais que a razão da distância entre os componentes e o número de vértices no menor seja a menor possível, ou seja, junte os componentes C_i e C_j que tenham o menor valor de $d(C_i, C_j)/\min(|C_i|, |C_j|)$.
3- Repita o passo 2 até que algum componente tenha no mínimo $k/4$ vértices.

O algoritmo descrito a seguir para o problema da k-MST supõe uma versão em que um dado vértice r deve estar na árvore e que o comprimento l da k-MST ótima é conhecido. Isto não tira a generalidade do algoritmo: no caso da raiz, basta executar o algoritmo n vezes, cada uma delas considerando um vértice diferente como raiz; pega-se então a árvore de menor peso dentre as n produzidas. No caso de não se conhecer o peso total da árvore ótima, note que l obedece a $\lambda \leq l \leq k\lambda$, onde λ é a distância do k-ésimo vértice mais distante da raiz. Portanto, podemos executar o algoritmo uma vez com $l = \lambda$ e então dobrar este valor se o número ou cardinalidade dos componentes encontrados não satisfizerem os limites garantidos. Assim, executaremos o algoritmo no máximo $O(\log k)$ vezes para cada raiz.

Além disso, o algoritmo usa o seguinte resultado [BCC+94]: dados um grafo ponderado de n vértices e um $\epsilon > 0$. Se L_ϵ é o comprimento da trajetória mais curta que visita no mínimo $(1-\epsilon)n$ vértices, então pode-se encontrar polinomialmente uma trajetória de comprimento no máximo $6L_\epsilon$ que visita no mínimo $(1-3\epsilon)n$ vértices.

Algoritmo *Improved-Connect-Clusters*

1- Marque como a serem ignorados os vértices a uma distância maior que l da raiz.
2- Execute o algoritmo *Merge-Cluster* várias vezes, sobre os vértices não ignorados. A cada vez, marque para serem ignorados os vértices do componente resultante de *Merge-Cluster*, e diminua o parâmetro k para o número de vértices ainda necessário (por exemplo, para $k - k_1$ da segunda vez, se a primeira execução retornou um componente de k_1 vértices). Faça isso até que no mínimo $\frac{15}{16}k$ vértices tenham sido obtidos.
3- Se a k-MST intersecta menos do que $(1 - \frac{3}{15})$ destes vértices (e portanto contém no mínimo $k/4$ novos vértices), uma aplicação final de *Merge-Cluster*, com argumento $k/4$, encontrará um novo componente com no mínimo $k/16$ vértices, finalizando o processo. Caso contrário, se a k-MST intersecta no mínimo $(1 - \frac{3}{15})$ destes vértices, então aplicando o resultado anterior [BCC+94], podemos encontrar um caminho de comprimento $O(l)$ que visita no mínimo $(1 - \frac{9}{15})\frac{15}{16}k = \frac{3}{8}k$ vértices. Portanto, a MST nestes vértices é uma árvore de custo $O(l)$ em mais de $k/4$ vértices.

Consideremos agora o PCTSP com prêmios associados aos vértices e o prêmio mínimo (w_0) a ser obtido. Seja *Merge-Weighted-Cluster* uma versão de *Merge-Cluster* em que a métrica $\frac{d(C_i,C_j)}{\min(|C_i|,|C_j|)}$ é substituída por $\frac{d(C_i,C_j)}{\min(wt(C_i),wt(C_j))}$, onde $wt(C)$ é a soma dos prêmios dos vértices contidos no componente C. O algoritmo para o PCTSP é o seguinte:

Algoritmo AABV

1- Aplique o algoritmo *Improved-Connect-Clusters* ao grafo usando $k = w_0$ e usando *Merge-Weighted-Cluster* ao invés de *Merge-Cluster*.
2- Construa a *Minimum Spanning Tree* sobre os vértices retornados pelo algoritmo *Im-*

proved-Connect-Clusters e a partir desta, construa uma trajetória sobre estes vértices (desigualdade triangular implica em comprimento no máximo duas vezes o da MST).
3- Concatene a trajetória obtida no passo anterior com a trajetória resultante da aplicação do algoritmo de Goemans e Williamson [GW92] ao grafo (sem prêmios nos vértices, mas com penalidades).

Garantia de desempenho

Tanto o algoritmo *Merge-Cluster* quanto o *Improved-Connect-Cluster* tem uma garantia de $O(\log^2 k)$ do valor ótimo para o problema da k-MST. A modificação na métrica utilizada em *Merge-Cluster* torna possível extender este resultado para o *Prize Collecting* TSP. As provas podem ser encontradas em [AABV94].

O desempenho deste algoritmo não é muito bom, devido à necessidade de pelo menos $O(n \log k)$ iterações de *Merge-Weighted-Cluster*. Além disso, a garantia obtida é logarítmica (não constante). No entanto, este algoritmo tem a seu favor o fato de tratar a versão do PCTSP com prêmios.

3.2.4 Algoritmo de Inserção/Retirada de Vértices

Mittenthal e Noon [MN92] desenvolveram uma heurística para uma variação do TSP, denominada TSSP (sigla para *Traveling Salesman Subset-tour Problem*, ou Problema da Subtrajetória do Caixeiro-Viajante), com uma restrição adicional. É uma heurística que combina construção e melhoria, uma vez que não só gera soluções válidas mas também tenta melhorá-las. O TSSP pode ser definido como o problema de, dadas n cidades, deseja-se encontrar uma trajetória cíclica de comprimento mínimo em um subconjunto dessas n cidades. A restrição adicional é definida sobre as arestas e é usada para garantir a presença na rota de um nível mínimo de algum quantificador, de maneira análoga aos prêmios do PCTSP. O TSSP com a restrição adicional é denominado TSSP+1, e é definido da seguinte forma [MN92]: um caixeiro viajante tem como base de atuação um depósito (cidade 0) e pode visitar algumas cidades de um conjunto $V = \{1, 2, \ldots, n\}$. Seja E o conjunto de arcos de viagem possíveis entre as cidades. Assumimos que a viagem entre quaisquer duas cidades é possível e, além disso, há um arco do depósito para ele mesmo, representando a possibildade de não se visitar nenhuma cidade, ou seja, $E = \{(i,j) | i = 0,1,2,\ldots,n; j = 0,1,2,\ldots,n; i \neq j\} \cup \{(0,0)\}$. Seja c_{ij} o custo de viagem da cidade i para j, para todo $(i,j) \in E$. Assumimos que $c_{00} = +\infty$ e $c_{ij} < +\infty$ para todo outro $(i,j) \in E$. Associamos ainda a cada arco $(i,j) \in E$ um valor a_{ij} que representa a contribuição daquele arco para a aquisição de nível mínimo b de determinada característica. Assim, se fizermos uma variável x_{ij} ser 1 se o arco (i,j) é usado na trajetória do caixeiro-viajante e 0 caso contrário, o TSSP+1 pode ser formulado como:

$$\min \sum_{(i,j)\in E} c_{ij}x_{ij}$$

sujeito a

$$\begin{aligned}
x_{00} + \sum_{j=1}^{n} x_{0j} &= 1 \\
x_{00} + \sum_{i=1}^{n} x_{i0} &= 1 \\
\sum_{\substack{i=0 \\ i\neq j}}^{n} x_{ij} &\leq 1, \quad \forall j \in V \\
\sum_{\substack{i=0 \\ i\neq j}}^{n} x_{ij} - \sum_{\substack{k=0 \\ k\neq j}}^{n} x_{jk} &= 0, \quad \forall j \in V \\
\sum_{i\in S} \sum_{\substack{j\in S \\ j\neq i}} x_{ij} &\leq |S| - 1, \quad \forall S \subseteq V, |S| \geq 2 \\
\sum_{i=0}^{n} \sum_{\substack{j=0 \\ j\neq i}}^{n} a_{ij}x_{ij} &\leq b \\
x_{ij} &\in \{0,1\}, \quad \forall (i,j) \in E
\end{aligned}$$

O TSSP+1 também é um problema da classe NP-difícil [MN92], e isso pode ser visto facilmente, uma vez que se trata de uma generalização do TSP. O TSSP+1 difere do PCTSP em dois pontos: não inclui penalidades (a função objetivo considera apenas o comprimento da rota) e, embora a restrição adicional funcione como o prêmio mínimo do PCTSP, ela é definida sobre as arestas, e não sobre os vértices.

Antes da apresentação da heurística propriamente dita, se fazem necessárias algumas definições quanto à notação utilizada. Representamos por T uma trajetória do TSSP+1, e definimos T_i como sendo a alteração, em T, de uma cidade i, da seguinte forma: se a cidade i já estiver em T, T_i é a trajetória resultante da exclusão de i de T; caso contrário, T_i é a trajetória resultante de T com a cidade i incluída na posição que cause o menor acréscimo ao comprimento da rota.

Além disso, definimos:

- $c(T)$: o valor da função objetivo (comprimento) de uma trajetória T;
- $\bar{b}(T) = b - \sum_{(i,j)\in T} a_{ij}$: a medida do grau de violação da restrição adicional pela trajetória T (se $\bar{b}(T) > 0$, T não é uma solução válida do problema).

Algoritmo de Inserção/Retirada de Vértices

1- Seja T o conjunto de arcos representando a subtrajetória corrente. Se $\bar{b}(T) \leq 0$, vá para o passo 2. Caso contrário, vá para o passo 3.

2- (*A subtrajetória corrente é uma solução válida*)Seja T^{Δ} a subtrajetória T_j tal que $\bar{b}(T_j) \leq 0$ e $c(T_j) \leq c_(T_i)$ para todo T_i satisfazendo $\bar{b}(T_i) \leq 0$. Se não houver nenhum T_i que satisfaça $\bar{b}(T_i) \leq 0$ então pare, nenhuma alteração de uma cidade em T leva a uma solução válida. Se $c(T^{\Delta}) \geq c(T)$, então pare, não há nenhuma alteração de uma cidade em T que leve a uma subtrajetória com custo menor. Caso contrário, vá para o passo 4.

3- (*A subtrajetória corrente não é uma solução válida*) Seja T^{Δ} a subtrajetória T_j tal que $\bar{b}(T_j) \leq \bar{b}(T)$ e $c(T_j) \leq c(T_i)$ para todo T_i satisfazendo $\bar{b}(T_i) \leq \bar{b}(T)$. Se não houver T_i satisfazendo $\bar{b}(T_i) \leq \bar{b}(T)$, então pare, não há alterações de uma cidade que levem a uma solução com menor grau de violação da restrição adicional. Caso contrário, vá para o passo 4.

4- Faça de T^{Δ} a subtrajetória corrente T. Se $\bar{b}(T) \leq 0$ vá para o passo 2, senão vá para o passo 3.

O número de operações necessárias para selecionar a subtrajetória T^{Δ} é $O(n^2)$. Além do mais, como cada iteração do algoritmo leva a um decréscimo diferente de 0 na função objetivo ou na medida de violação da restrição adicional, o número máximo de iterações é finito. Caso o peso (c_{ij}) e o "prêmio" (a_{ij}) associados às arestas obedeçam à desigualdade triangular, pode-se demonstrar que o número máximo de iterações é linear em n [MN92].

3.3 Algoritmos Propostos para o PCTSP

Embora os algoritmos citados anteriormente tenham a seu favor uma garantia de desempenho, normalmente seus tempos de execução são altos, e/ou se aplicam apenas a versões bastante restritas do PCTSP (excluindo-se ora prêmios, ora penalidades). Tendo isso em vista, desenvolvemos algoritmos heurísticos bastante simples e rápidos para o PCTSP, tanto de construção [RdS95] quanto de melhoria de soluções, nos moldes dos vários algoritmos assim desenvolvidos para o TSP [JP85]. Nesta seção descreveremos estes algoritmos heurísticos propostos. Em seguida, mostraremos os resultados computacionais obtidos com os algoritmos.

3.3.1 Algoritmos de Construção

Todos os algoritmos a seguir tem como entrada uma instância do PCTSP consistindo de um grafo completo não-orientado $G = (V, E)$, com pesos não-negativos nas arestas (c_{ij}), prêmios (w_i) e penalidades (p_i) não-negativas associadas aos vértices e um prêmio

mínimo w_0 a ser obtido pelos vértices na rota. Os algoritmos tratam, portanto, de uma versão menos restrita do PCTSP do que os algoritmos descritos na seção anterior.

Algoritmo Insere-Vértice Estático

1- Ordenam-se os vértices do grafo em ordem decrescente segundo o valor de $p_i / \sum_{j \in V, j \neq i} c_{ij}$.
2- Inicia-se a rota com os três primeiros vértices de acordo com a ordem definida no passo anterior (em um grafo simétrico existe apenas uma rota conectando três vértices).
3- Colocam-se os vértices na rota, inserindo-os um a um na posição que implica em menor acréscimo ao comprimento da trajetória, seguindo a ordem definida no passo 1 e até que se satisfaça w_0. Um esboço do trabalho deste algoritmo pode ser visto na Figura 3.9.

Complexidade de tempo: $O(n^2)$

Algoritmo Insere-Vértice Dinâmico

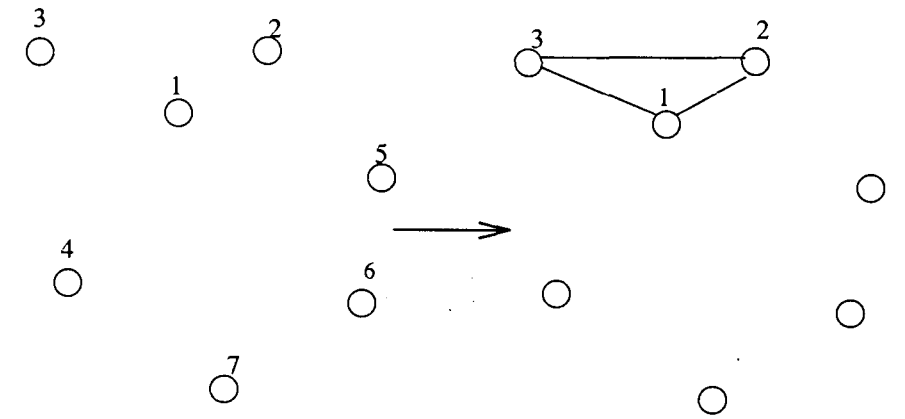
1- Inicie a rota pela aresta com o maior valor de $(p_i + p_j)/c_{ij}$, onde i e j são os vértices da extremidade da aresta.
2- Para cada vértice i fora da rota, calcule $p_i / \min(c_{ij} + c_{ik} - c_{jk})$, onde j e k são vértices já na rota e $\min(c_{ij} + c_{ik} - c_{jk})$ é o menor valor de acréscimo de comprimento da rota causado pela inserção de i.
3- Insira na rota o vértice que apresentar maior valor de $p_i / \min(c_{ij} + c_{ik} - c_{jk})$.
4- Se w_0 não satisfeito, volte ao passo 2. A Figura 3.10 apresenta um exemplo de execução deste algoritmo.

Complexidade de tempo: $O(n^3)$

Algoritmo Insere-Aresta

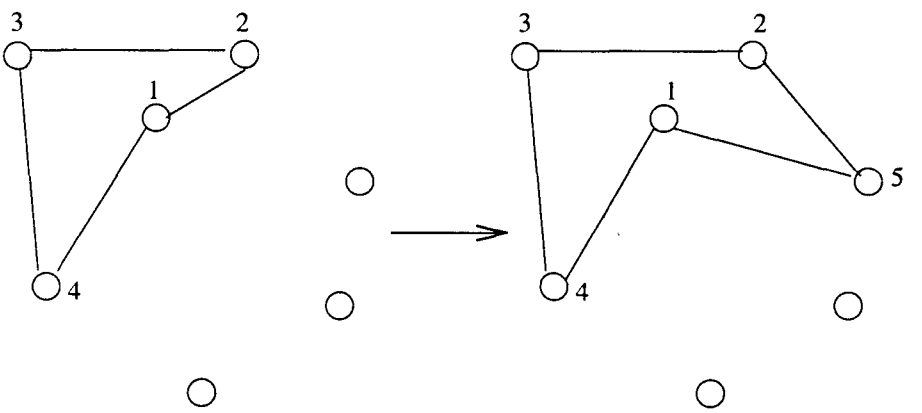
1- Ordene as arestas por ordem decrescente de $(p_i + p_j)/c_{ij}$, onde i e j são os vértices da extremidade da aresta.
2- Insira as arestas na rota nesta ordem, mas sem formar ciclos ou deixar vértices com grau maior que dois na rota, até que os vértices na rota completem w_0.
3- O grafo resultante do passo anterior é uma floresta. Conecte os componentes dessa floresta de forma a resultar em um ciclo. As arestas utilizadas para conectar os componentes são obtidas segundo a ordem estabelecida no passo 1. Um esboço do trabalho deste algoritmo pode ser visto na Figura 3.11.

Complexidade de tempo: $O(n^3)$



Ordenação dos vértices

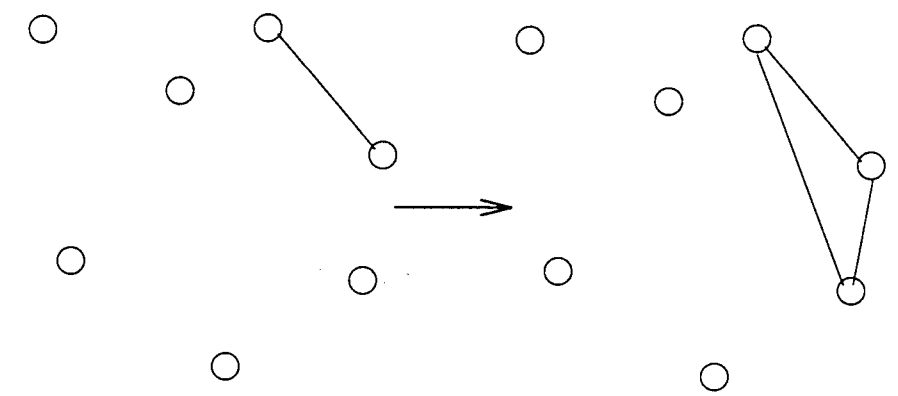
Rota incial



Inserindo quarto melhor vértice
(melhor posição de inserção
entre os vértices 1 e 3)

Inserindo quinto melhor vértice
(melhor posição de inserção entre os vértices 1 e 2)
(prêmio mínimo satisfeito - rota final)

Figura 3.9: Exemplo da execução do algoritmo Insere-Vértice Estático.



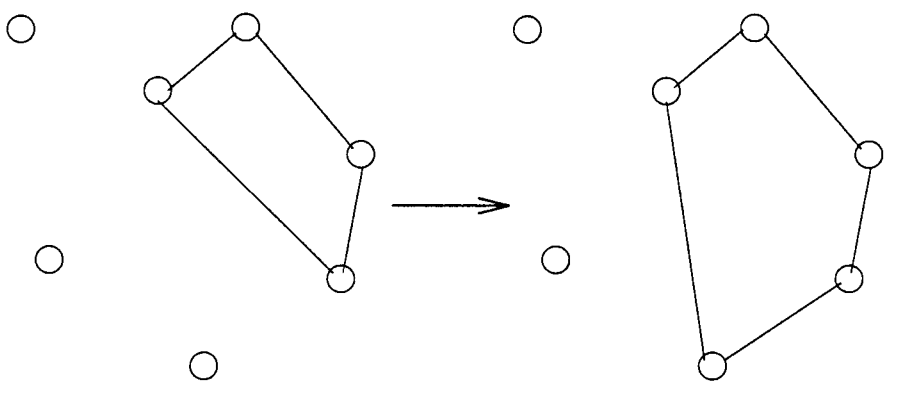


Figura 3.10: Exemplo da execução do algoritmo Insere-Vértice Dinâmico.

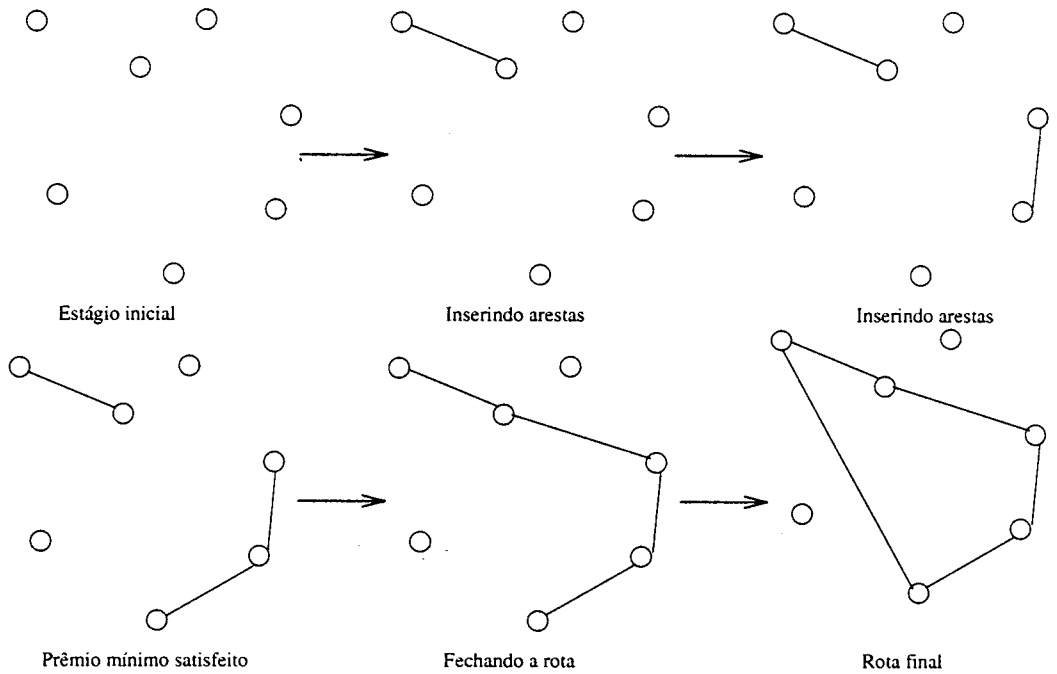


Figura 3.11: Exemplo da execução do algoritmo Insere-Aresta.

Algoritmo Insere-Vértice Ponderado pelos Prêmios

1- Para cada vértice i fora da rota, calcule $(\min(c_{ij} + c_{ik} - c_{jk}) - p_i)/w_i$, onde j e k são vértices já na rota e $\min(c_{ij} + c_{ik} - c_{jk})$ é o menor valor de acréscimo de comprimento da rota causado pela inserção de i. Se ainda não há vértices na rota, calcule para cada vértice o valor p_i/w_i.

2- Insira na rota o vértice que apresentar menor valor calculado acima.

3- Se w_0 não satisfeito, volte ao passo 1. O modo de funcionamento deste algoritmo é basicamente o mesmo explicitado na Figura 3.10. A diferença é a função utilizada para se determinar o melhor vértice a ser inserido em cada iteração.

Complexidade de tempo: $O(n^3)$

Todos os algoritmos apresentados se baseiam em uma ordenação dos vértices ou arestas do grafo, baseada em alguma função quantificadora, e posterior inserção destes vértices ou arestas na rota. No primeiro, os vértices são ordenados *a priori* e esta ordem não muda do decorrer do algoritmo (daí o nome "estático"). A idéia é que os vértices com maior valor de penalidade e menor valor de pesos de arestas ligando-os a outros são melhores para participarem da rota. No segundo algoritmo, a idéia também é esta, com a diferença de que o melhor vértice a ser inserido é calculado a cada iteração do algoritmo, de acordo com a posição onde o vértice considerado seria incluído na rota em construção. Deste fato

deriva o nome "dinâmico".

O terceiro algoritmo também parte do mesmo princípio de que as penalidades dos vértices na rota devem ser maximizadas, e o comprimento das arestas minimizado. Entretanto, seu foco de ação compreende as arestas do grafo. Arestas que ligam vértices de maior penalidade e que têm pesos baixos associados são mais propensas a serem inclusas na rota.

O último algoritmo de construção apresentado é uma modificação do segundo para que os prêmios associados aos vértices sejam levados em consideração. A idéia é dar preferência na rota aos vértices de maior prêmio (levando em conta também as penalidades e o custo de inserção em termos de comprimento de rota).

3.3.2 Algoritmos de Melhoria

Os algoritmos a seguir partem de uma solução válida para o PCTSP e tentam melhorá-la. Portanto, além dos dados descrevendo a instância propriamente dita, eles têm como entrada uma solução válida do PCTSP, ou seja, uma trajetória envolvendo três ou mais vértices de forma que a soma dos prêmios destes vértices seja maior ou igual a w_0 e um conjunto (possivelmente vazio) de vértices fora da rota. É válido notar que todos os algoritmos de melhoria para o TSP, como Lin-Kernighan, Or-opt, k-opt [GS85], podem ser aplicados ao PCTSP para a melhoria do comprimento da rota em um dado conjunto de vértices. Assim sendo, desenvolvemos algoritmos de melhoria voltados para a escolha dos vértices participantes da rota.

Os dois primeiros algoritmos descritos a seguir se baseiam no seguinte princípio: faz-se uma alteração na rota que melhore a função objetivo. Em seguida, são feitas outras alterações na rota até que a melhoria em relação à solução original seja menor ou igual a 0. A solução final do algoritmo é aquela em que a melhoria da função objetivo atingiu seu máximo. Este processo, que também é utilizado no algoritmo de Lin-Kernighan para o TSP [GS85, LK73] é ilustrado pela Figura 3.12. Neste caso, o algoritmo pararia na rota $R7$, que levaria a um acréscimo no valor da função objetivo, e a rota resultante seria a $R5$, que apresentou o maior decréscimo no valor da função objetivo.

Algoritmo Insere Vértices

1- Faça `saldo` igual a 0. Seja R a rota atual. Faça R_1 igual a R e i igual a 1.

2- Para cada vértice v fora da rota R_i faça:

2.1- Verifique qual a posição de inserção de v em R_i que causa o menor acréscimo à função objetivo. Seja `acresc` o valor deste acréscimo.

2.2- Se `saldo+acresc` menor ou igual a 0, insira v na rota na posição de menor acréscimo. Incremente i de 1 e seja R_i a rota com o vértice v acrescentado. Faça

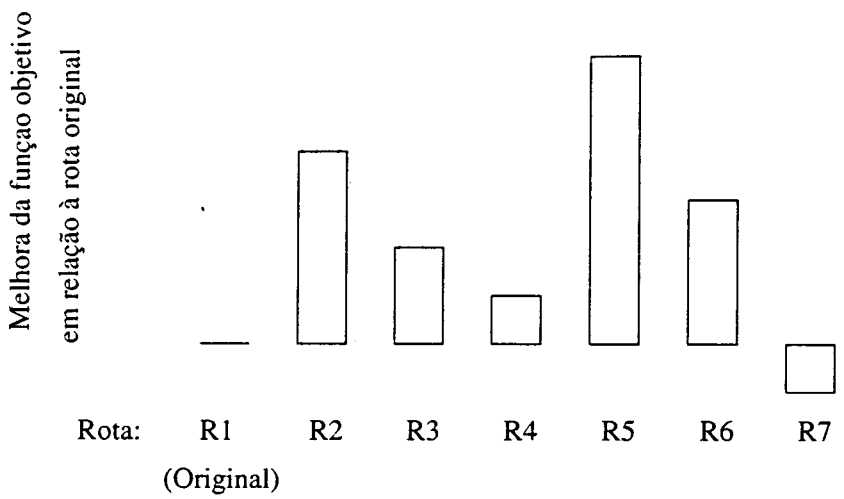


Figura 3.12: Processo utilizado nos algoritmos Insere e Retira Vértices.

`saldo = saldo+acresc`. Volte para o passo 2.

2.3- Se `saldo+acresc` maior que 0, e ainda não houver sido acrescentado nenhum vértice à rota, volte para o passo 2.

2.4- Se `saldo+acresc` maior que 0, e já houver sido acrescentado pelo menos um vértice à rota, vá para o passo 3.

3- Seja a solução resultante a rota R_i de menor valor de função objetivo.

Algoritmo Retira Vértices

1- Faça `saldo` igual a 0. Seja R a rota atual. Faça R_1 igual a R e i igual a 1.

2- Para cada vértice v na rota R_i cuja retirada da rota não viole a restrição de prêmio mínimo, faça:

2.1- Seja `acresc` o valor de acréscimo na função objetivo causado pela retirada de v de R_i.

2.2- Se `saldo+acresc` menor ou igual a 0, retire v da rota. Incremente i de 1 e seja R_i a rota com o vértice v retirado. Faça `saldo = saldo+acresc`. Volte para o passo 2.

2.3- Se `saldo+acresc` maior que 0, e ainda não houver sido retirado nenhum vértice da rota, volte para o passo 2.

2.4- Se `saldo+acresc` maior que 0, e já houver sido retirado pelo menos um vértice da rota, vá para o passo 3.

3- Seja a solução resultante a rota R_i de menor valor de função objetivo.

Algoritmo Substitui Vértices

1- Seja R a rota atual. Para cada vértice v fora de R faça:

1.1- Selecione o vértice w na rota cuja substituição por v gera uma solução válida com o menor valor de função objetivo.

1.2- Se a troca de w por v melhorar a função objetivo, efetue a troca.
2- Se tiver ocorrido alguma troca, volte ao passo 1. Senão, pare.

3.4 Resultados Comparativos

Nesta seção descreveremos os resultados obtidos pela confrontação de nossos algoritmos de construção com o de Goemans e Williamson, que possui a melhor garantia de desempenho para o PCTSP disponível na literatura.

Para comparação com o algoritmo de Goemans e Williamson, geramos 25 instâncias aleatórias do PCTSP com penalidades e pesos nas arestas escolhidos como inteiros no intervalo [0, 100], com tamanhos entre 50 e 500 vértices e com pesos nas arestas obedecendo à desigualdade triangular (necessária para o algoritmo GW-PCTSP). Foi feita, também, uma modificação do critério de parada de nossos algoritmos, já que a restrição de prêmio mínimo foi eliminada para compatibilidade com o algoritmo de Goemans e Williamson. O critério adotado para os dois algoritmos Insere- Vértice (versões Estática e Dinâmica) foi o seguinte: a inserção de vértices na rota continua enquanto melhora a função objetivo. Quando é considerada uma inserção que piore a função objetivo, o algoritmo pára de inserir com probabilidade k/n, onde k é o número de vértices já na rota e n é o número total de vértices do grafo. Uma vez que, no algoritmo Insere-Aresta, não há condições de se avaliar a função objetivo a cada inserção de aresta, pois a rota ainda não está fechada, o critério de parada adotado para este algoritmo foi o seguinte: caso o peso da aresta menos as penalidades dos vértices conectados por ela seja menor que 0, a inserção é feita. Caso contrário, o algoritmo pára com a mesma probabilidade k/n utilizada nos algoritmos anteriores. Não fizemos a comparação com o algoritmo Insere-Vértice Ponderado pelos Prêmios, uma vez que este depende estruturalmente dos prêmios dos vértices.

Os resultados podem ser vistos na Tabela 3.2. Como as versões de nossos algoritmos com critério de parada modificado não são determinísticas, efetuamos cinco execuções de cada um, e mostramos na tabela a média e o melhor valor obtidos. Na Figura 3.13 vemos o gráfico com os resultados obtidos pelo algoritmo GW e as médias obtidas pelos demais algoritmos quando aplicados às instâncias de 50 vértices. O mesmo é visto nas Figuras 3.14, 3.15, 3.16 e 3.17, para as instâncias com 100, 150, 200 e 500 vértices, respectivamente. Nos gráficos destas Figuras, cada um dos cinco grupos de quatro resultados de cada um dos algoritmos (GW, Insere-Vértice Estático/Dinâmico, Insere-Aresta) se refere à aplicação destes a uma instância do tamanho ilustrado pelo gráfico.

Como podemos observar, a versão dinâmica do algoritmo Insere-Vértice sempre obteve resultados melhores do que os obtidos pelo algoritmo de Goemans e Williamson, tanto no melhor resultado quanto na média das execuções. Já a versão estática do algoritmo só perdeu para o GW-PCTSP em três das instâncias testadas, duas de 50 vértices e uma

Número de cidades	GW	1	2	3	4	5	6
50	2206	2139,6	2118	2072,6	2070	2250,6	2231
50	2294	2393,8	2351	2192,6	2192	2433,0	2387
50	1948	1942,0	1845	1864,4	1851	2253,8	2229
50	2352	2347,4	2336	2231,2	2216	2439,4	2417
50	2548	2634,6	2603	2470,4	2465	2520,8	2492
100	4557	4520,4	4263	4133,0	4123	4350,6	4337
100	4544	4346,2	4335	4239,0	4238	4720,6	4709
100	4376	4611,8	4377	4182,4	4163	4428,0	4404
100	4344	4229,4	4135	4307,4	4063	4365,8	4349
100	4737	4590,8	4527	4400,4	4388	4775,4	4773
150	7092	6759,4	6663	6617,0	6617	6914,8	6887
150	7211	6966,4	6693	6537,8	6537	6674,6	6637
150	7293	7276,6	7273	6811,2	6803	7037,8	6999
150	6644	6196,2	6191	6086,0	6085	6500,4	6464
150	6500	6397,0	6396	6271,4	6255	6799,2	6757
200	9473	8813,0	8794	8651,2	8651	8977,0	8967
200	9357	8524,4	8483	8318,2	8318	8802,2	8765
200	9302	8771,6	8709	8575,8	8572	9095,4	9048
200	9475	8869,6	8794	8592,6	8590	9103,0	9103
200	9316	8892,8	8765	8445,2	8445	8878,0	8836
500	24328	22679,8	22067	21647,6	21646	22620,8	22504
500	23284	20752,8	20738	20460,0	20460	21330,6	21329
500	23639	22895,6	21089	20885,0	20885	21695,4	21673
500	23717	21450,8	21072	20792,2	20729	21700,4	21630
500	22982	21135,2	20568	20394,6	20389	21479,0	21479

Legenda: 1- Insere-Vértice Estático (média)
2- Insere-Vértice Estático (melhor)
3- Insere-Vértice Dinâmico (média)
4- Insere-Vértice Dinâmico (melhor)
5- Insere-Aresta (média)
6- Insere-Aresta (melhor)

Tabela 3.2: Comparação entre algoritmos de construção propostos e GW-PCTSP.

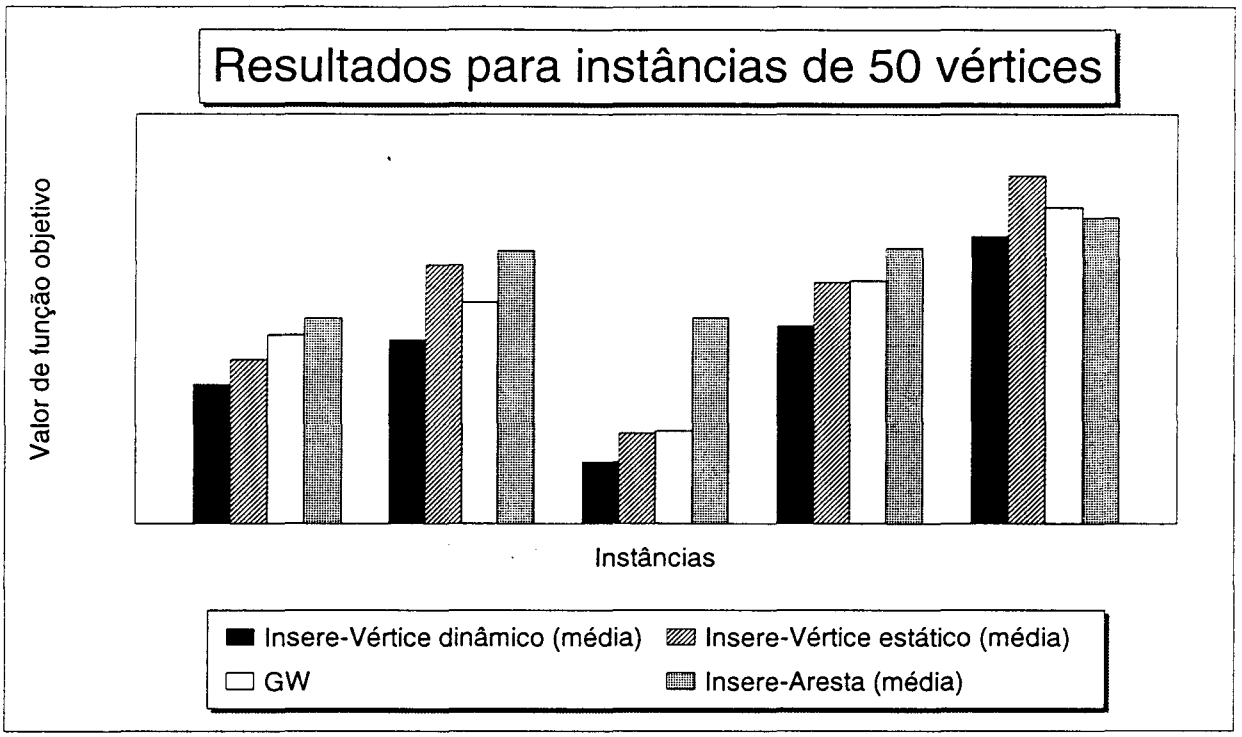


Figura 3.13: Resultados para instâncias de 50 vértices.

de 100 vértices. Nas demais instâncias destes tamanhos e nas maiores ele superou o desempenho do GW-PCTSP. Já o algoritmo Insere-Aresta obteve desempenho superior em todas as instâncias de 200 e 500 vértices, em quatro das cinco de 150 vértices e em uma de 100 e outra de 50 vértices. Dentre os três algoritmos, o que obtém melhor desempenho é a versão dinâmica do Insere-Vértice, que supera os outros dois em todas as instâncias com exceção de uma de 100 vértices. O de pior desempenho é o Insere-Aresta, que somente em algumas instâncias (sete das vinte e cinco) supera a versão estática do algoritmo Insere-Vértice. O desempenho um pouco pior do algoritmo Insere-Aresta pode ser explicado pelo fato de que seu critério de parada é mais dependente do prêmio mínimo do que os dos demais, uma vez que a rota não é válida nos passos intermediários do algoritmo, inviabilizando a quantificação da função objetivo a cada passo.

Devemos lembrar também que nossos algoritmos, além de serem mais simples do que o GW-PCTSP, podem se aplicar a uma versão mais geral do *Prize Collecting* TSP (sem desigualdade triangular e com prêmios), além de apresentarem tempo de execução $O(n^2)$ (Insere-Vértice Estático) e $O(n^3)$ (Insere-Vértice Dinâmico e Insere-Aresta), enquanto que o tempo de execução do algoritmo de Goemans e Williamson é $O(n^3 \log n)$. Este, no entanto, tem a vantagem da garantia de desempenho, e de poder ser aplicado a uma classe vasta de problemas além do PCTSP.

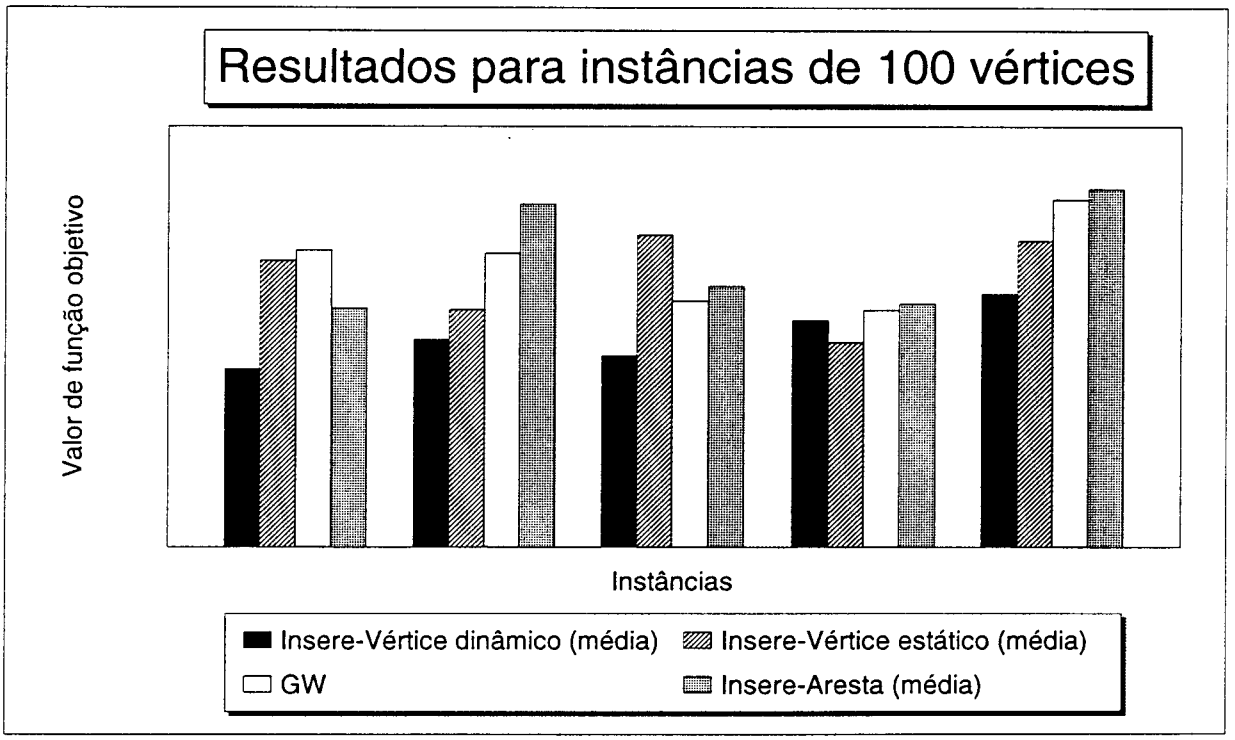


Figura 3.14: Resultados para instâncias de 100 vértices.

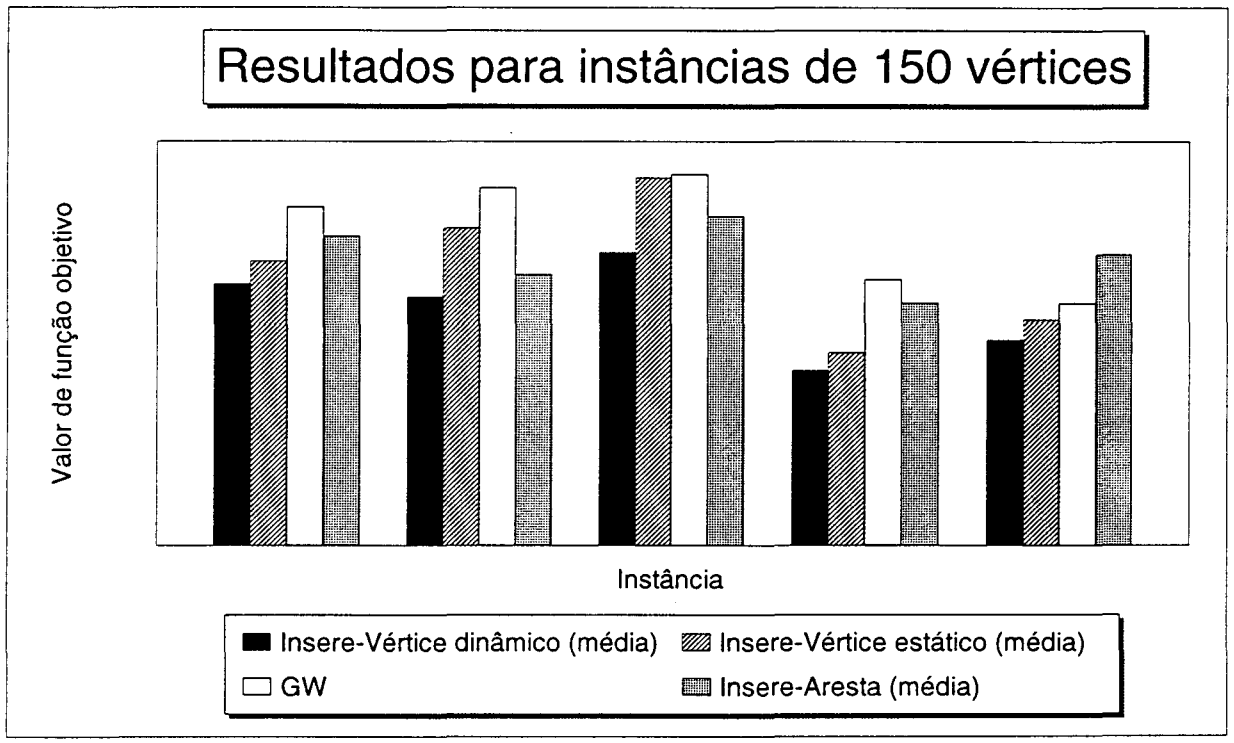


Figura 3.15: Resultados para instâncias de 150 vértices.

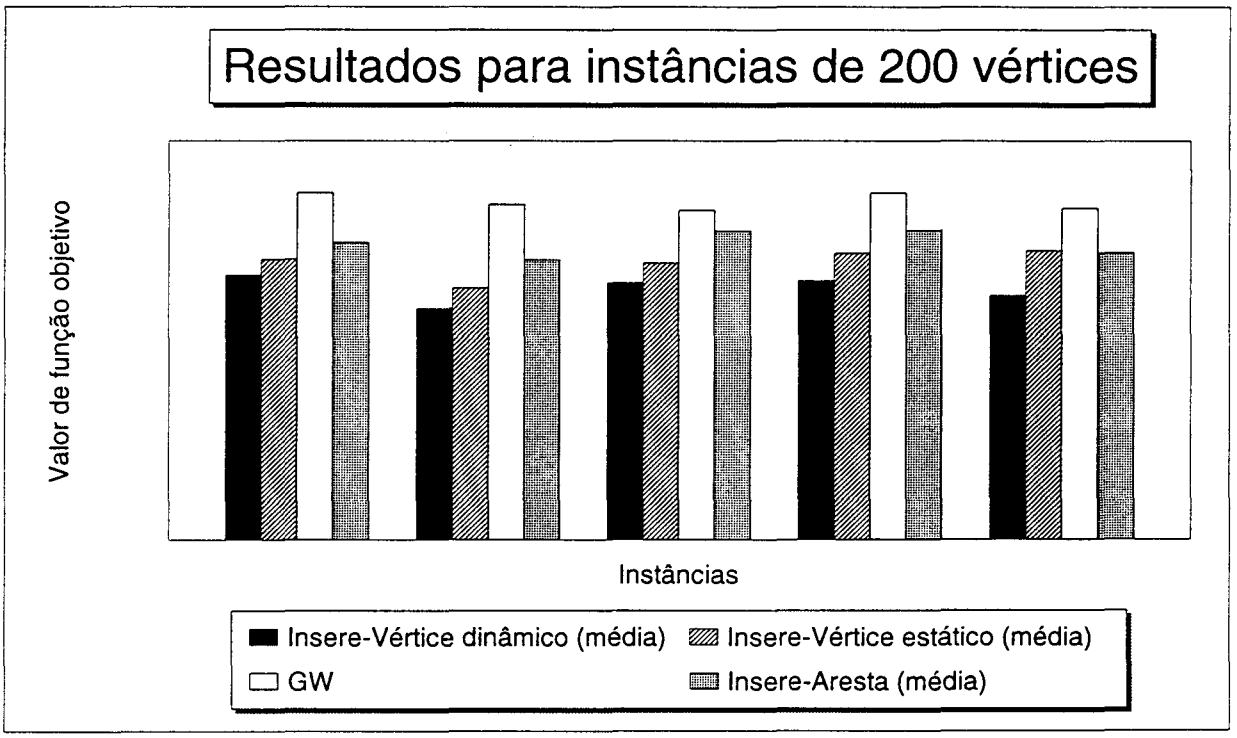


Figura 3.16: Resultados para instâncias de 200 vértices.

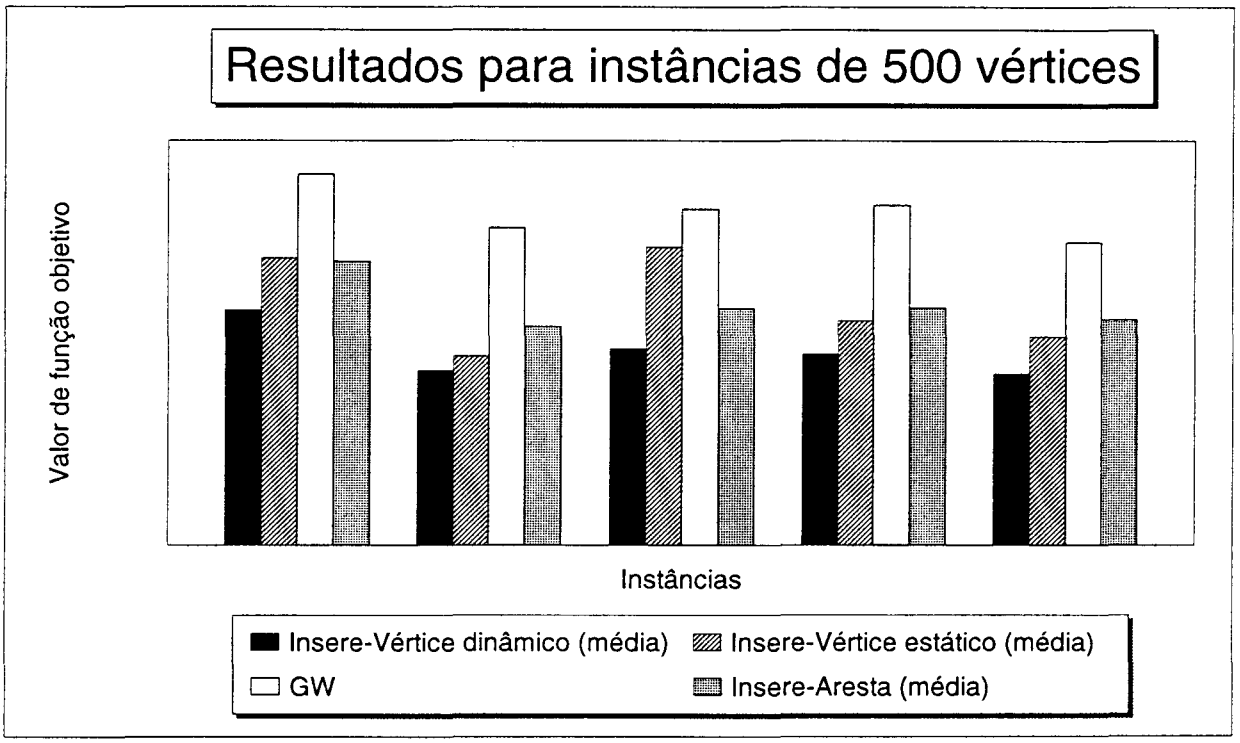


Figura 3.17: Resultados para instâncias de 500 vértices.

Capítulo 4

Times Assíncronos

Neste capítulo descrevemos a técnica multi-algorítmica denominada *Times Assíncronos* (ou *A-Teams*, do inglês *Asynchronous Teams*). Esta técnica é baseada na utilização de diversos algoritmos para a resolução de um problema, e normalmente consegue resultados muito melhores do que os obtidos por algoritmos isolados.

4.1 Definição e Características

Como descrito anteriormente, numerosas abordagens têm sido empregadas na resolução de problemas de Otimização Combinatória. Uma idéia que surge naturalmente é a seguinte: por que não empregar simultaneamente os diversos algoritmos disponíveis para um determinado problema, de modo a aproveitar os pontos fortes de cada um e conseguir melhores resultados ? É uma idéia interessante que, entretanto, leva a algumas perguntas, como: em que ordem aplicar os algoritmos disponíveis ? Quantas vezes aplicar cada um deles ? Para responder estas questões seria necessário um estudo detalhado de cada problema, bem como de cada algoritmo disponível para o mesmo, o que resultaria num esforço provavelmente maior do que o de resolver o problema propriamente dito.

No entanto, estas questões surgem apenas quando se pensa em uma implementação síncrona, onde os algoritmos são aplicados em uma ordem determinada. A proposta de *A-Teams* se baseia em uma interação assíncrona entre os algoritmos; assim, a decisão de qual algoritmo irá agir sobre a instância do problema sendo considerada é aleatória, criando o potencial de, no decorrer do tempo, ocorrerem todas as combinações possíveis de ordens de execução. Times Assíncronos se constituem em uma organização de algoritmos (agentes) autônomos, que se comunicam assincronamente através de memórias compartilhadas e que apresenta um fluxo de dados cíclico [dS93].

As características que definem um *A-Team* são, portanto:

- Autonomia dos agentes: cada agente (algoritmo) de um Time Assíncrono tem autonomia total quanto ao seu funcionamento interno, sua escolha dos dados de entrada e sua política de alocação de recursos. Como os agentes são independentes uns dos outros, podem ser adicionados e/ou retirados agentes do *A-Team* sem que isso interfira no funcionamento dos demais.

- Comunicações assíncronas: a comunicação entre os agentes de um *A-Team* se dá de forma assíncrona, através de escritas/leituras em memórias compartilhadas.

- Fluxo de dados cíclico: isto significa que uma determinada informação armazenada em uma memória é lida, processada por um ou mais agentes (possivelmente sendo armazenada em memórias intermediárias à medida em que é modificada), e depois volta a ser escrita na memória de onde saiu originalmente. Este fluxo cíclico permite que as mudanças realizadas por um agente sobre um dado sejam acessíveis aos demais, permitindo *feedback* e interação entre eles.

Na Figura 4.1 podemos ver a representação gráfica de um Time Assíncrono. Nela, os retângulos representam as memórias compartilhadas (M1 e M2) e as setas os agentes do *A-Team* (A,B,C,D,E,D1,D2). A direção da seta indica a direção do fluxo de dados no agente: por exemplo, o agente D lê soluções da memória M2 e escreve na memória M1. O retângulo maior envolvendo as memórias serve para indicar que o agente A, por exemplo, pode escrever em ambas as memórias. Dado que deve haver uma forma de eliminar soluções das memórias, pois sem isto o espaço ocupado por estas poderia crescer indefinidamente, existem agentes especiais denominados de agentes de destruição (*destroyers*), que se encarregam de manter o tamanho das memórias fixo, eliminando soluções. Estes agentes são indicados por D1 e D2 na figura.

Conforme foi dito, o número de agentes em um *A-Team* pode variar com o tempo, já que estes são autônomos entre si. Isto caracteriza os A-Teams como organizações abertas, como observado por Talukdar e Souza [TdS92]. Uma outra característica normalmente apresentada pelos Times Assíncronos é a chamada "Eficiência em Escala" [TdS92], que se define como uma melhora no desempenho global do *A-Team* à medida em que são acrescentados novos agentes. Ou seja, via de regra observa-se uma interação sinérgica entre os agentes do *A-Team*.

4.2 Parâmetros de Construção

Após a definição da estrutura do *A-Team*, ou seja, da quantidade de memórias e agentes e da forma como eles estarão interligados, torna-se necessária a definição individual dos

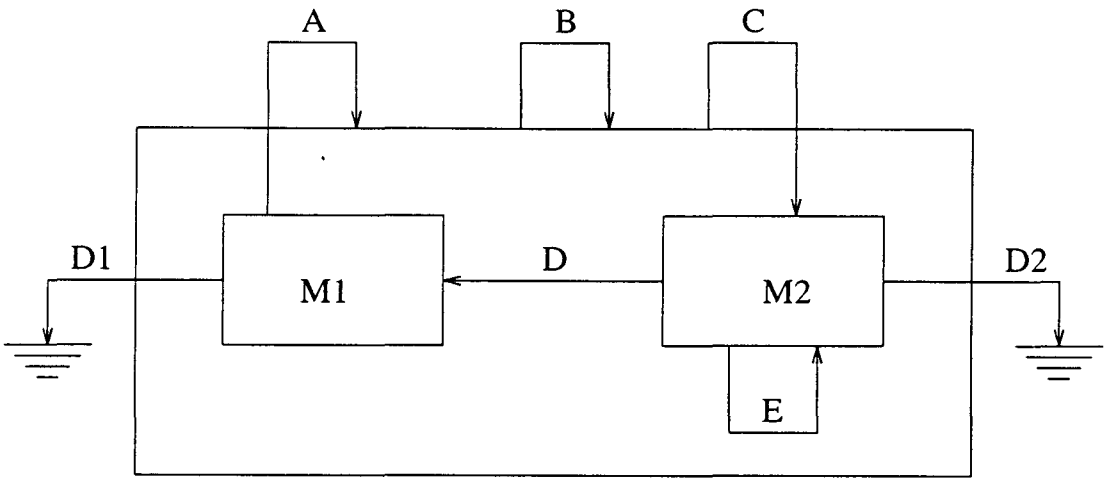


Figura 4.1: Exemplo de um *A-Team*.

diversos componentes do Time. Nesta seção descreveremos os principais fatores a serem levados em conta na definição de cada componente.

4.2.1 Memórias

Não apenas o número de memórias tem que ser decidido, mas também o tipo de solução que a memória irá armazenar (completa ou parcial), bem como o tamanho, em número de soluções, das memórias. Grandes memórias permitem uma maior diversidade de soluções, mas precisam de maior tempo para iniciação e podem fazer com que o *A-Team* demore a convergir para bons resultados. Memórias menores, por outro lado, consomem menos tempo para iniciação e aceleram a convergência do time, mas apresentam uma diversidade menor de soluções, que pode levar o *A-Team* a obter resultados piores. Em um *A-Team* para o Problema do Caixeiro Viajante (*TSP*), Souza [dS93] indica que o ponto de melhor compromisso entre qualidade de solução e tempo de execução ocorre com memórias de tamanho N, onde N representa o número de cidades (vértices do grafo) do problema.

4.2.2 Algoritmos de Iniciação

Assim como os destruidores, existem também os agentes geradores (ou de iniciação) de um *A-Team*, que se encarregam de completar a(s) memória(s) de soluções antes que os demais agentes comecem a executar. A única diferença destes algoritmos para os que participam do *A-Team* é que eles executam apenas no começo, mas nada impede que um algoritmo gerador seja usado também no decorrer da execução normal do *A-Team*. Este preenchimento inicial das memórias poderia ser descartado, mas isto poderia levar a problemas, caso um agente que necessita de uma solução de uma memória fosse executado

com esta estando vazia. Como a execução dos algoritmos é assíncrona, é melhor que se preencha a(s) memória(s) antes, para se evitar atrasos na execução.

4.2.3 Política de Destruição

Os agentes de destruição necessitam de algum critério para eliminar soluções das memórias. Este critério é denominado política de destruição, e se caracteriza pela forma como o agente de eliminação de soluções escolhe uma solução para ser eliminada para dar lugar a outra na memória. Esta escolha é feita normalmente com base em uma distribuição de probabilidade sobre os valores das soluções da memória. Algumas distribuições comuns são:

- Elimina a pior: a pior solução da memória tem probabilidade 1 de ser eliminada, enquanto que todas as demais tem probabilidade 0. Normalmente este tipo de distribuição diminui a diversidade do *A-Team*, já que só elimina as soluções piores, deixando soluções muito parecidas que não sejam muito ruins por um longo tempo na memória.

- Distribuição uniforme: em uma memória com m soluções, cada uma delas tem probabilidade $1/m$ de ser eliminada. Esta distribuição aumenta a diversidade da memória, mas causa a eliminação de soluções boas com muita freqüência, sem que elas tenham tempo de "disseminar" suas qualidades.

- Distribuição linear: a probabilidade de eliminação da melhor solução é 0, enquanto que a probabilidade das demais cresce linearmente conforme se aproximam da pior. Este tipo de distribuição apresenta uma diversidade menor do que a uniforme, mas em compensação tende a preservar por mais tempo as boas soluções da memória, o que tende a melhorar os resultados finais do Time Assíncrono.

A política de destruição que normalmente apresenta melhores resultados é a distribuição linear [dS93, Pei95, Cav95]. Além de se decidir qual solução sairá da memória, é preciso se definir um critério de quando uma solução será aceita. Algumas políticas adotadas são:

- Aceita qualquer uma: qualquer nova solução é aceita, independentemente de seu valor de função objetivo.

- Aceita somente soluções melhores do que a pior: somente aceita uma solução se o seu valor de função objetivo for melhor do que o da pior solução atualmente na memória.

- Aceita dentro de tolerância: uma solução é aceita se seu valor de função objetivo for menor (em se tratando de um problema de minimização) do que α vezes o valor de função objetivo da melhor solução da memória ($\alpha > 1$).

4.2.4 Algoritmos de Melhoria

Este item compreende os agentes que pegam uma solução completa de uma memória, efetuam alguma operação sobre esta solução, na tentativa de melhorá-la, e tentam inserir a solução em alguma memória. Este é justamente o modo de operação de heurísticas de melhoria convencionais.

4.2.5 Algoritmos de Consenso

Por algoritmos de consenso entende-se aqueles que têm como entrada duas ou mais soluções e que produzem uma solução (muitas vezes parcial) através da manipulação das semelhanças e/ou diferenças entre as soluções. A filosofia por trás deste tipo de agente é a tentativa de se extrair as características "boas" de uma solução, ou seja, características que uma solução ótima teria grandes chances de apresentar.

4.2.6 Algoritmos de Reconstrução

Estes agentes partem de uma solução parcial e geram uma solução completa. Normalmente são variações dos algoritmos de iniciação das memórias, e geralmente complementam o trabalho dos algoritmos de consenso, transformando soluções parciais geradas por estes em soluções válidas.

4.3 Histórico

A metodologia de Times Assíncronos tem sido empregada com sucesso em diversos problemas de Otimização Combinatória. Dentre os trabalhos correntes e passados usando *A-Teams* podemos citar:

- Em [TdS90] encontramos a introdução do termo "Times Assíncronos", em um trabalho para a resolução de equações algébricas não lineares. No *A-Team* proposto foram usados o método de Newton-Raphson e um algoritmo genético. Em um trabalho posterior ([dST91]) o método de Newton-Raphson cedeu lugar a duas versões do método de Levenbergh-Marquardt, resultando em um *A-Team* mais eficaz.

- O problema de projeto de um robô manipulador modular foi tratado com *A-Teams* em [Mur92]. Embora se tratasse de um problema com múltiplas funções objetivo, elas foram encapsuladas em apenas uma, e o problema resolvido como mono-objetivo.

- O problema do caixeiro viajante (TSP) é tratado em [dS93], onde se define também o tipo de problemas adequados para resolução através de Times Assíncrono. Neste trabalho diversas instâncias do TSP foram testadas, sendo conseguido o valor ótimo mesmo para a maior delas (532 cidades). O trabalho se baseou em uma implementação paralela de um *A-Team*, conseguido-se um aumento de velocidade praticamente linear em relação ao número de processadores utilizado.

- Em [Pei95] encontramos a aplicação de Times Assíncronos ao problema de seqüenciamento de tarefas conhecido como FSP (*Flow Shop Problem*). Além disto, neste trabalho encontramos uma metodologia para a especificação de Times Assíncronos.

- Os problemas de recobrimento e particionamento de conjuntos (*Set Covering* e *Set Partitioning*) foram tratados usando-se Times Assíncronos em [Lon95].

- O JSP (*Job Shop Scheduling Problem*) foi tema de dois trabalhos envolvendo a metodologia de Times Assíncronos: um deles abordando somente heurísticas de construção (assim como consenso e reconstrução) [Cav95] e outro focando-se em heurísticas de melhoria [Had97]. Os dois *A-Teams* propostos apresentaram bons resultados, tanto isolada quanto conjuntamente.

- O problema de transporte de derivados de petróleo em uma rede de dutos e bases foi tratado utilizando-se *A-Teams* em [Cam95]. Para isto, foi proposta uma modelagem matemática do problema e uma decomposição deste em sub-problemas (dada a complexidade do problema real). Foram conseguidos resultados próximos aos obtidos na prática, embora algumas simplificações tenham sido introduzidas no modelo para torná-lo mais "tratável".

- O escalonamento de trens em uma malha ferroviária foi tratado através de Times Assíncronos em [Tse95]. Neste trabalho foram usadas várias representações diferentes para o problema no mesmo *A-Team*, com cada memória guardando soluções para o problema em representações diferentes.

- Em [Rod96] a metodologia de Times Assíncronos foi aplicada a um problema de Otimização Combinatória com múltiplas funções objetivo (no caso, o problema do caixeiro viajante com múltiplas matrizes de distância). Foi proposta uma adequação das memórias para possibilitar o tratamento dos paretos de soluções.

- Em [NSdS96] encontramos a aplicação de *A-Teams* ao problema de desenho de grafos, onde se tenta encontrar o melhor compromisso entre dois critérios estéticos: maximização da simetria e minimização de cruzamentos entre arestas.

Capítulo 5

Times Assíncronos para o PCTSP

Neste capítulo descrevemos o *A-Team* proposto para o *Prize Collecting* TSP. Discorremos sobre a estrutura do time, seus agentes, memórias e políticas de iniciação e destruição. A seguir, apresentamos alguns detalhes de como o time foi implementado e listamos os resultados obtidos com a sua aplicação a várias instâncias do PCTSP.

5.1 Estrutura do *A-Team* proposto

Após os testes computacionais realizados, a configuração final do *A-Team* ficou como mostrado na Figura 5.1. São três memórias (uma de soluções completas e duas de soluções parciais) e dez agentes (sem contar os de iniciação e destruição), sendo cinco de melhoria (Or-opt, LK, Insere vértices, Remove vértices e Substitui vértices), três de consenso (Dec-1, Dec-2 e Dec-3) e dois de reconstrução (Insere Vértice Dinâmico e Insere Aresta). A seguir descreveremos os diversos elementos do time com mais detalhes.

5.1.1 Memórias

O Time Assíncrono proposto possui três memórias:

- Memória de soluções completas: esta memória guarda soluções para uma instância do PCTSP. Uma solução é representada através de um vetor de inteiros correspondentes aos vértices participantes da rota de uma instância do PCTSP (observe que não necessariamente todos os vértices do grafo estarão na rota). Além disso, cada posição desta memória guarda um vetor booleano onde cada posição indica se determinado vértice está ou não na rota (este vetor torna mais eficiente o trabalho

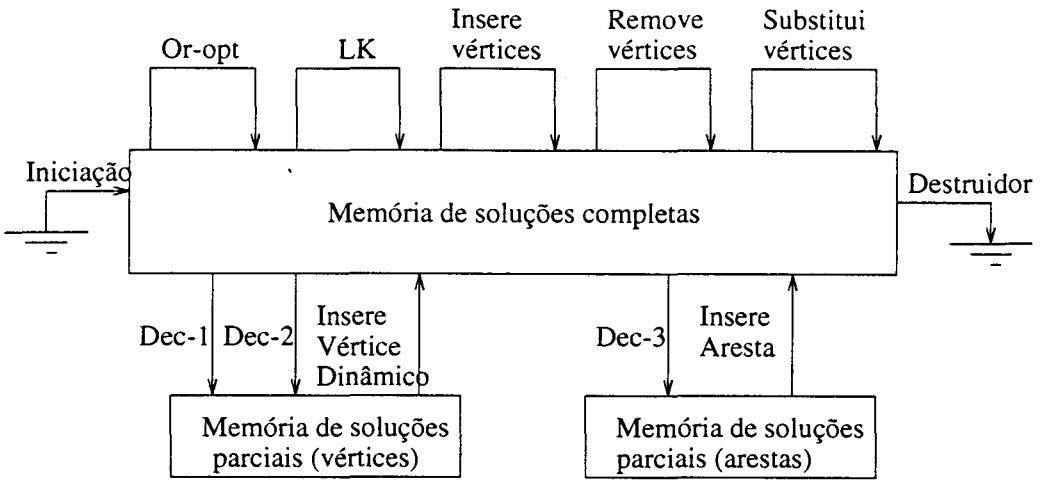


Figura 5.1: Configuração proposta para o PCTSP.

de alguns agentes). Como informações adicionais para cada solução na memória temos: o valor do prêmio atingido por aquela solução, o comprimento total da rota, o total de penalidades (soma das penalidades dos vértices que não estão na rota) e uma indicação de qual agente foi o responsável pela inserção daquela solução na memória. Esta memória de soluções completas é mantida em ordem crescente de valor de função objetivo, ou seja, a melhor solução da memória está na primeira posição. Esta memória tem n posições, onde n é o número de vértices do grafo da instância sendo resolvida.

- Memória de soluções parciais (vértices): esta é uma memória do tipo FIFO (*first-in, first-out*). Cada posição desta representa um conjunto de vértices a serem colocados em uma rota do PCTSP. A solução é parcial porque aqui temos apenas o conjunto de vértices, sem arestas a conectá-los e que inclusive podem não atender ao prêmio mínimo.

- Memória de soluções parcias (arestas): analogamente à anterior, esta é uma memória de soluções parciais em ordem FIFO que guarda arestas do grafo, as quais deverão ser colocadas em uma rota do PCTSP quando a solução parcial for transformada em válida. Novamente a solução é parcial porque as arestas não formam um ciclo no grafo e nem há garantia de que os vértices conectados por estas arestas atendam ao prêmio mínimo. Entretanto, da forma como são produzidas estas soluções parciais, garante-se que elas formam grafos acíclicos onde nenhum vértice tem grau maior do que dois.

5.1.2 Algoritmos de Iniciação

Os três algoritmos considerados para se iniciar a memória de soluções completas (as memórias de soluções parciais não são iniciadas por se tratarem basicamente de filas) são descritos a seguir. O primeiro gera uma solução completamente aleatória para o problema.

Algoritmo Solução Aleatória

1- Dentre os vértices ainda não escolhidos anteriormente, escolha um vértice aleatoriamente.
2- Acrescente este vértice à rota.
3- O prêmio mínimo da instância foi satisfeito ? Se não foi, retorne ao passo 1.
4- Retorne a rota construída como a solução do PCTSP.

O segundo algoritmo de iniciação considerado é uma adaptação do algoritmo *Arbitrary Insertion* [GS85] para o TSP.

Algoritmo Inserção Arbitrária-PCTSP

1- Escolha dois vértices iniciais aleatoriamente.
2- Escolha um vértice k aleatoriamente e não escolhido anteriormente.
3- Encontre o melhor ponto de inserção do vértice k na rota corrente, ou seja, encontre os vértices i e j tais que $c_{ik} + c_{kj} - c_{ij}$ seja mínimo.
4- Insira o vértice k entre os vértices i e j da rota.
5- O prêmio mínimo da instância foi satisfeito ? Se não foi, retorne ao passo 2.
6- Retorne a rota construída como a solução do PCTSP.

O terceiro algoritmo de iniciação é o algoritmo Insere-Vértice Dinâmico apresentado no Capítulo 3 desta dissertação, ligeiramente modificado para se tornar não determinístico.

Algoritmo Insere-Vértice Dinâmico Não-Determinístico

1- Escolha dois vértices aleatoriamente para iniciar a rota
2- Para cada vértice i fora da rota, calcule $p_i / \min(c_{ij} + c_{ik} - c_{jk})$, onde j e k são vértices já na rota e $\min(c_{ij} + c_{ik} - c_{jk})$ é o menor valor de acréscimo de comprimento da rota causado pela inserção de i
3- Insira na rota o vértice que apresentar maior valor de $p_i / \min(c_{ij} + c_{ik} - c_{jk})$
4- O prêmio mínimo da instância foi satisfeito ? Se não foi, retorne ao passo 2
5- Retorne a rota construída como a solução do PCTSP.

Foram feitos testes usando a configuração mostrada na Figura 5.1 e com cada um dos três algoritmos como iniciador da memória. Os resultados destes testes (efetuados com

três instâncias com 200 vértices cada, e com média de três execuções) podem ser vistos nas Tabelas 5.1, 5.2 e 5.3. Nelas são apresentados os tempos necessários para iniciação da memória e os resultados de melhor valor e média obtidos ao final das três execuções do *A-Team* para cada uma das três instâncias testadas. Com base nos resultados obtidos, a iniciação da memória principal ficou a cargo do Algoritmo Inserção Arbitrária-PCTSP

Instância	Média do *A-Team*	Melhor valor do *A-Team*	Tempo médio de CPU para iniciação (em segundos)
t200a.1	1920,0	1920	2,4
t200a.5	1934,7	1934	2,3
t200a.9	1874,0	1874	2,4

Tabela 5.1: Resultados obtidos pelo *A-Team* com iniciação pelo Algoritmo Solução Aleatória

Instância	Média do *A-Team*	Melhor valor do *A-Team*	Tempo médio de CPU para iniciação (em segundos)
t200a.1	1920,0	1920	1,2
t200a.5	1934,0	1934	2,7
t200a.9	1872,0	1872	5,6

Tabela 5.2: Resultados obtidos pelo *A-Team* com iniciação pelo Algoritmo Inserção Arbitrária-PCTSP

Instância	Média do *A-Team*	Melhor valor do *A-Team*	Tempo médio de CPU para iniciação (em segundos)
t200a.1	1920,3	1920	7,3
t200a.5	1934,0	1934	157,4
t200a.9	1873,3	1872	240,8

Tabela 5.3: Resultados obtidos pelo *A-Team* com iniciação pelo Algoritmo Insere-Vértice Dinâmico Não-Determinístico

5.1.3 Política de Destruição

A política de destruição adotada foi a de distribuição linear, ou seja, a probabilidade de eliminação da melhor solução é 0, enquanto que a probabilidade das demais cresce line-

armente conforme se aproximam da pior solução na memória. Como critério de aceitação de uma solução na memória foi usado o de aceitação com tolerância, ou seja, uma solução é aceita se seu valor de função objetivo for menor do que α vezes o valor de função objetivo da melhor solução da memória.

As tabelas 5.4, 5.5, 5.6 e 5.7 mostram os resultados efetuados com vários valores de α. Novamente os testes foram efetuados com a configuração de *A-Team* da Figura 5.1, foram usadas três instâncias com 200 vértices cada, e tomada a média de três execuções. O valor de α definido por estes testes foi 3,0.

Instância	Média do *A-Team*	Melhor valor do *A-Team*
t200.1	2001,3	2000
t200.5	2022,0	2022
t200.9	1985,7	1985

Tabela 5.4: Resultados obtidos pelo *A-Team* com $\alpha = 1,5$

Instância	Média do *A-Team*	Melhor valor do *A-Team*
t200.1	2000,7	2000
t200.5	2021,7	2021
t200.9	1985,7	1985

Tabela 5.5: Resultados obtidos pelo *A-Team* com $\alpha = 2,0$

Instância	Média do *A-Team*	Melhor valor do *A-Team*
t200.1	2000,0	2000
t200.5	2021,7	2021
t200.9	1985,0	1985

Tabela 5.6: Resultados obtidos pelo *A-Team* com $\alpha = 3,0$

Instância	Média do *A-Team*	Melhor valor do *A-Team*
t200.1	2000,7	2000
t200.5	2022	2022
t200.9	1985,7	1985

Tabela 5.7: Resultados obtidos pelo *A-Team* com $\alpha = 4,0$

5.1.4 Agentes de Melhoria

Como agentes de melhoria (leêm uma solução da memória de soluções completas, tentam melhorá-la e a escrevem de volta na mesma memória), utilizamos dois conjuntos de agentes: um deles é composto de agentes que tentam melhorar a solução do PCTSP através da alteração dos vértices participantes da rota (Insere vértices, Remove vértices e Substitui vértices). Estas heurísticas foram desenvolvidas durante este trabalho e são detalhadas no Capítulo 3 desta dissertação. O outro conjunto de agentes é composto de algoritmos para o TSP tradicional cujo objetivo é melhorar a rota, sem alterar os vértices dela participantes (Or-opt e LK):

- Or-opt: trata-se do tradicional algoritmo para o TSP [GS85]. Baseia-se na troca de posição de cadeias de vértices em seqüência na rota. Inicialmente são consideradas cadeias de três vértices. Quando não se consegue mais melhorar a rota trocando estas cadeias de lugar, diminui-se o tamanho da cadeia considerada para dois e finalmente são considerados vértices individuais.

- Lin-Kernighan (LK): este é o algoritmo de LK para o TSP, considerado como a heurística que produz melhores resultados [LK73, GS85]. Baseia-se na troca de arestas na rota por arestas não na rota, decidindo dinamicamente a cada iteração o número de arestas que devem ser trocadas.

5.1.5 Algoritmos de Consenso

O *A-Team* proposto para o PCTSP tem três algoritmos de consenso que alimentam as memórias de soluções parciais: Dec-1 e Dec-2 escrevem na memória de soluções parciais de vértices e Dec-3 escreve na memória de soluções parciais de arestas.

Algoritmo Dec-1

1- Tome duas soluções completas do PCTSP.

2- Selecione todos os vértices que estão presentes nas duas trajetórias tomadas (ou seja,

efetue a intersecção entre os dois conjuntos de vértices). Estes vértices são a solução parcial gerada.

Algoritmo Dec-2
1- Tome duas soluções completas do PCTSP.
2- Selecione todos os vértices que estão presentes em uma das trajetórias (selecionada aleatoriamente) mas não estão na outra (ou seja, efetue a diferença entre os dois conjuntos de vértices). Estes vértices são a solução parcial gerada.

Algoritmo Dec-3
1- Tome duas soluções completas do PCTSP.
2- Selecione todas as arestas que estão presentes nas duas trajetórias tomadas (ou seja, efetua a intersecção entre os dois conjuntos de arestas). Estas arestas são a solução parcial gerada.

Estes algoritmos tentam, através da manipulação de duas soluções, extrair partes da solução que sejam "boas", no sentido de serem partes que uma possível solução ótima ou próxima do ótimo tenha. Dec-1 e Dec-3 fazem isso através da intersecção de soluções, assumindo que o que duas soluções diferentes têm em comum tende a ser algo desejável em uma solução boa. Embora Dec-2 pareça trabalhar contra isso, ao fazer a diferença entre duas soluções, sua idéia é que, dadas duas soluções ruins, ou uma solução ruim e uma boa, a parte comum das duas soluções á algo não desejável em uma solução boa. Além disso, sua estratégia aumenta a diversidade das soluções na memória.

5.1.6 Algoritmos de Reconstrução

Estes agentes geram soluções completas para o PCTSP a partir das soluções parciais. Eles são os algoritmos Insere-Vértice Dinâmico e Insere-Aresta, apresentados no Capítulo 3, porém ligeiramente modificados para partir de uma solução parcial nos moldes daquelas mantidas nas memórias de soluções parciais.

Algoritmo Insere-Vértice Dinâmico
1- Pegue uma solução parcial (conjunto de vértices).
2- Coloque os vértices da solução parcial em uma ordem aleatória.
3- Inicie a rota pela aresta que conecta os dois primeiros vértices da ordenação feita no passo anterior.
4- Para cada vértice i fora da rota, mas no conjunto de vértices parciais, seguindo a ordem definida no passo 2, calcule $p_i / \min(c_{ij} + c_{ik} - c_{jk})$, onde j e k são vértices já na rota e $\min(c_{ij} + c_{ik} - c_{jk})$ é o menor valor de acréscimo de comprimento da rota causado pela

inserção de i.
5- Insira na rota o vértice que apresentar maior valor de $p_i / \min c_{ij} + c_{ik} - c_{jk}$.
6- Se ainda há vértices da solução parcial a serem inseridos na rota, volte para o passo 4.
7- Se o prêmio mínimo foi satisfeito, vá para o passo 11.
8- Para cada vértice i fora da rota, calcule $p_i / \min(c_{ij} + c_{ik} - c_{jk})$, onde j e k são vértices já na rota e $\min(c_{ij} + c_{ik} - c_{jk})$ é o menor valor de acréscimo de comprimento da rota causado pela inserção de i.
9- Insira na rota o vértice que apresentar maior valor de $p_i / \min(c_{ij} + c_{ik} - c_{jk})$.
10- Se o prêmio mínimo não foi satisfeito, retorne ao passo 8.
11- Retorne a rota construída.

Algoritmo Insere-Aresta
1- Pegue uma solução parcial (floresta sem nenhum vértice com grau maior do que dois).
2- Transforme a floresta em um ciclo, conectando os vértices de grau um dos componentes conexos da floresta. Para isso, coloque as $k(k-2)/2$ arestas (k é o número de vértices de grau um) que conectam componentes diferentes (e estão fora da rota) em ordem crescente de $(p_i + p_j)/c_{ij}$, onde i e j são dois vértices de grau um na floresta. A seguir, inclua as arestas na rota seguindo esta ordem de modo a não deixar nenhum vértice com grau maior do que dois.
3- Se o prêmio mínimo foi satisfeito, vá para o passo 7.
4- Para cada vértice fora da rota, calcule o acréscimo mínimo (ou decréscimo máximo) na função objetivo resultante da adição deste vértice à rota.
5- Insira na rota o vértice que cause o maior decréscimo (ou menor acréscimo) à função objetivo.
6- Se o prêmio mínimo não foi satisfeito, volte para o passo 4.
7- Retorne a rota construída.

5.1.7 Implementação

Devido a adequabilidade de *A-Teams* para o processamento distribuído, este foi implementado utilizando-se a biblioteca PVM (de *Parallel Virtual Machine*) [GBD+94]. O PVM provê a utilização de processamento distribuído com troca de mensagens entre os processos. Estes processos podem ser distribuídos através de várias máquinas (que compõem a "máquina virtual" do PVM), bastando, para isso, que as máquinas possuam comunicação via TCP/IP.

Na implementação efetuada, cada agente (exceto os destruidores e iniciadores de memória) foi colocado como um processo. A memória de soluções completas, juntamente com seus agentes destruidores e de iniciação, foi deixada em outro processo, enquanto que as memórias de soluções parciais compartilharam ainda outro processo. Foi definido

um protocolo de comunicação para troca de mensagens entre os processos-agentes e os processos-memórias, para solicitação e recebimento de soluções.

5.2 Instâncias para Teste

Por ser um problema proposto há relativamente pouco tempo, não existem instâncias de teste do *Prize Collecting Traveling Salesman Problem* com valores de melhor solução publicado (instâncias de *benchmarking*). Portanto, para efetuarmos os testes com o *A-Team*, geramos instâncias do PCTSP aleatoriamente. As instâncias geradas obedecem à desigualdade triangular, são simétricas, seus valores de prêmio e penalidades dos vértices e pesos de arestas são inteiros no intervalo $[0, 200]$, e foram obtidas através do seguinte algoritmo:

Algoritmo Gera-Instâncias

1- Para cada vértice da instância, gere um número inteiro aleatório no intervalo $[0, 200]$ como o prêmio do vértice e outro como a penalidade do mesmo.
2- Para cada vértice da instância, gere um par de coordenadas (x,y) aleatoriamente, com valores de coordenadas sendo inteiros no intervalo $[0, 100]$.
3- Inicie o peso das arestas entre os vértices com a distância L_1 entre eles (dadas as coordenadas geradas no passo anterior).
4- Para cada aresta do grafo (pegas em ordem aleatória) faça:
  4.1- TETO $\leftarrow 200$
  4.2- PISO $\leftarrow 0$
  4.3- Para cada vértice k do grafo diferente daqueles conectados pela aresta que liga i e j faça:
    4.3.1- Se $c_{ki} + c_{kj} <$ TETO, então TETO $\leftarrow c_{ki} + c_{kj}$
    4.3.2- Se $|c_{ki} - c_{kj}| >$ PISO, então PISO $\leftarrow |c_{ki} - c_{kj}|$
  4.4- c_{ij} recebe PISO mais um valor inteiro aleatório escolhido no intervalo $[0, (TETO - PISO)]$.

O algoritmo basicamente contrói um grafo onde os vértices são pontos de uma matriz quadrada de tamanho 100 e usa a distância entre os vértices (calculada pela métrica L_1) como base para os pesos das arestas. O algoritmo, então, efetua uma variação aleatória nos pesos das arestas, mas mantendo a desigualdade triangular (garantida pelo uso da métrica L_1). Os passos após 4.2 do algoritmo são responsáveis por essa variação e pela garantia da desiguldade triangular nas arestas da instância. Conforme podemos ver na Figura 5.2, tomando por base a aresta ij e o vértice a, a desigualdade triangular implica que:

$(1) c_{ij} \leq c_{ai} + c_{aj}$
$(2) c_{ai} \leq c_{ij} + c_{aj} \rightarrow c_{ij} \geq c_{ai} - c_{aj}$
$(3) c_{aj} \leq c_{ij} + c_{ai} \rightarrow c_{ij} \geq c_{aj} - c_{ai}$

Das duas últimas desigualdades tiramos que $c_{ij} \geq |c_{ai} - c_{aj}|$ e portanto temos que: $|c_{ai} - c_{aj}| \leq c_{ij} \leq c_{ai} + c_{aj}$.

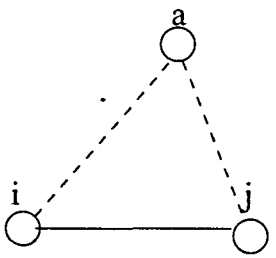


Figura 5.2: Exemplo de desigualdade triangular.

Quanto ao prêmio mínimo, foram geradas três classes de instância: em uma delas, o prêmio mínimo é 10% da soma dos prêmios de todos os vértices, em outra classe o índice definido foi 50% e em outra foi 90%. Estas três classes foram definidas para se obter maior variedade nos testes: à medida em que o índice mencionado aumenta, o PCTSP "converge" para o TSP tradicional (mais vértices precisam participar da rota para a obtenção do prêmio mínimo). Em relação ao tamanho, foram geradas três instâncias de 50, 100, 150 e 200 vértices e uma instância com 500 vértices para cada classe de instância definida. A divisão em classes e a estratégia para se criarem as instâncias são baseadas na abordagem usada por Mittenthal e Noon para os testes de sua heurística para o TSSP+1 [MN92].

5.2.1 Valores para comparação

Como as instâncias testadas foram geradas aleatoriamente, tornou-se necessária a obtenção de algum resultado sobre estas instâncias a fim de possibilitar uma medida da qualidade dos resultados obtidos pelo Time Assíncrono. Com este propósito efetuamos cálculos de limites inferiores (*lower bounds*) para as instâncias geradas.

Estes limites foram gerados com base nas seguintes formulações matemáticas, onde $G = (V, E)$ é um grafo não-dirigido, c_{ij} é o peso associado à aresta $(i, j) \in E$, p_i e w_i são, respectivamente, a penalidade e o prêmio associados ao vértice $i \in V$, w_0 é o prêmio mínimo da instância, x o vetor de incidência da trajetória e y_i a variável indicando a pertinência ou não da cidade i à trajetória:

Problema P_1 :

$$\min \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} + \sum_{i=1}^{n} p_i(1-y_i)$$

sujeito a

$$\begin{aligned}
\sum_{i=1}^{n} y_i w_i &\geq w_0, \\
\sum_{i=1}^{n} x_{ij} - 2y_j &= 0, \forall j \in V \\
x_{ij} &\in \{0,1\}, \forall i,j \in V \\
y_i &\in \{0,1\}, \forall i \in V
\end{aligned}$$

Problema P_2 :

$$\min \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} + \sum_{i=1}^{n} p_i(1-y_i)$$

sujeito a

$$\begin{aligned}
\sum_{i=1}^{n} y_i w_i &\geq w_0, \\
\sum_{i=1}^{n} x_{ij} - 2y_j &= 0, \forall j \in V \\
0 \leq x_{ij} &\leq 1, \forall i,j \in V \\
y_i &\in \{0,1\}, \forall i \in V
\end{aligned}$$

Problema P_3 :

$$\min \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} + \sum_{i=1}^{n} p_i(1-y_i)$$

sujeito a

$$\begin{aligned}
\sum_{i=1}^{n} y_i w_i &\geq w_0, \\
\sum_{i=1}^{n} x_{ij} - 2y_j &= 0, \forall j \in V \\
x_{ij} &\in \{0,1\}, \forall i,j \in V \\
0 \leq y_i &\leq 1, \forall i \in V
\end{aligned}$$

Problema P_4 :

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} + \sum_{i=1}^{n} p_i (1 - y_i)$$

sujeito a

$$\begin{aligned} \sum_{i=1}^{n} y_i w_i &\geq w_0, \\ \sum_{i=1}^{n} x_{ij} - 2y_j &= 0, \forall j \in V \\ 0 \leq x_{ij} &\leq 1, \forall i, j \in V \\ 0 \leq y_i &\leq 1, \forall i \in V \end{aligned}$$

Em todos os problemas acima, V é o conjunto dos n vértices do grafo, c_{ij} é o peso associado à aresta que conecta os vértices i e j, w_i e p_i são, respectivamente, o prêmio e a penalidade associados ao vértice i e w_0 é o prêmio mínimo a ser obtido.

As formulações acima são, portanto, relaxações do PCTSP (cuja formulação pode ser vista no Capítulo 2). Em todas elas foram retiradas as restrições de formação de sub-rotas. Esta relaxação é comumente utilizada para o TSP, e o problema originado é chamado de *Assignment Problem* [Gar85]. O problema P_1 seria, portanto, o *Prize Collecting Assignment Problem*. P_2 é uma relaxação deste, onde as variáveis x não necessitam ser inteiras. Em P_3, as variáveis x devem ser inteiras mas as y não têm a restrição de integralidade. E, finalmente, em P_4 ambos os conjuntos de variáveis são reais. Portanto, denominando de $OPT(PCTSP)$ o valor ótimo do *Prize Collecting* TSP, e de $OPT(P_i)$ o valor ótimo das relaxações descritas acima, temos que:

$$OPT(P_4) \leq OPT(P_3) \leq OPT(P_1) \leq OPT(PCTSP)$$
$$OPT(P_4) \leq OPT(P_2) \leq OPT(P_1) \leq OPT(PCTSP)$$

Embora *a priori* nada possa ser dito sobre a relação entre $OPT(P_2)$ e $OPT(P_3)$, na maioria das instâncias testadas em que pudemos obter os dois valores, encontramos que $OPT(P_3) \geq OPT(P_2)$.

A definição de várias relaxações foi necessária uma vez que, para determinadas instâncias, não foi possível a resolução exata de P_1 com os recursos computacionais disponíveis. Para estas instâncias, tentamos calcular a solução de P_2 e P_3 e usarmos o maior delas como *lower bound*. Caso para alguma instância não fosse possível obter o resultado de nenhuma das três primeira relaxações, usaríamos a solução de P_4 como limite inferior. Isto, entretanto, não foi necessário.

Para a resolução das relaxações foi usado o pacote CPLEX [CPL94], em sua versão 3.0, utilizando um algoritmo do tipo *branch-and-bound* para resolver P_1, P_2 e P_3 (P_4 é um problema de programação linear).

Caso alguma das soluções obtidas com uma relaxação fosse uma solução válida para o PCTSP teríamos encontrado comprovadamente o ótimo da instância. Isto, no entanto, não aconteceu para nenhuma das instâncias testadas.

5.3 Resultados Computacionais

Na Tabela 5.8, apresentamos os resultados obtidos pelo *A-Team* e os *lower bounds* obtidos para as instâncias com prêmio mínimo igual a 10% da soma dos prêmios dos vértices. O mesmo é apresentado nas Tabelas 5.9 e 5.10 para as instâncias com 50% e 90% do prêmio total, respectivamente. Em todas as tabelas, um (2) ao lado do *lower bound* indica valor de limite inferior obtido com a resolução do problema P_2 descrito acima, enquanto que (3) indica *lower bound* obtido pela resolução de P_3. Quando não há indicação, o *lower bound* foi obtido através da resolução de P_1.

Apesar de que, com a implementação em PVM, cada agente do *A-Team* pode executar em uma máquina diferente, os resultados dos testes foram obtidos com todos os agentes e memórias em uma só maquina (uma Sun Sparcserver 1000 com 8 processadores), para evitar diferenças nos resultados causadas por diferenças de carga entre as máquinas.

Por não ser possível a alocação total da máquina para todos os testes, os tempos do *A-Team* apresentados nas tabelas de resultados foram calculados da seguinte forma: efetuaram-se várias medições de tempo para cada tamanho de instância com a máquina totalmente dedicada ao *A-Team*. Assim, obtiveram-se médias de velocidades do *A-Team* em termos de iterações por segundo para cada tamanho de instância (entende-se por iteração o envio ou recebimento de uma solução de/para um agente). Com estas velocidades médias foram calculados os tempos apresentados nas tabelas com base no número de iterações que o Time Assíncrono levou até chegar à melhor solução.

Os resultados obtidos atestam a adequabilidade de Times Assíncronos ao PCTSP. A média de distância entre os resultados obtidos pelo *A-Team* e os *lower bounds* das instâncias foi de 3,21 %. Este parece ser um valor bastante razoável dados os tamanhos das instâncias e a complexidade do problema, visto que, para instâncias com 100 vértices do TSP com desigualdade triangular, heurísticas de construção isoladas ficam, nos melhores casos, a cerca de 6 % do ótimo (com a maioria delas a mais de 10 % do valor ótimo), enquanto que heurísticas de construção com algoritmos de melhoria acoplados chegam em média no máximo a 2 % do valor ótimo, com progressivo aumento desta razão à medida em que aumenta o tamanho da instância [GBDS80, AS83, GS85]. Vale lembrar que os resultados que obtivemos foram alcançados também para instâncias bem maiores, e que

A	B	C	D	E	F	G	H
50	910	957,0	957	3/3	5,16	4,4	247,3
50	966	978,0	978	3/3	1,24	4,1	35,8
50	981	993,0	993	3/3	1,22	0,6	47,4
100	1454	1479,0	1479	3/3	1,72	537,7	65,7
100	1422	1453,3	1452	2/3	2,20	1856,0	157,3
100	1495	1524,0	1524	3/3	1,94	57,0	52,8
150	1617	1675,0	1675	3/3	3,59	37,1	651,2
150	1624	1680,0	1680	3/3	3,45	6669,8	1367,1
150	1718(3)	1753,3	1752	2/3	2,05	1610,9	1281,5
200	1967	2000,7	2000	2/3	1,71	411,6	4745,4
200	1832(2)	1920,0	1920	3/3	4,80	12,0	881,6
200	1898(3)	1968,3	1967	1/3	3,70	2914,3	4233,3
500	2991(2)	3092,7	3091	1/3	3,40	211,1	31584,3

Legenda: A- Tamanho da instância (número de vértices)
B- Valor de *lower bound*
C- Média do *A-Team*
D- Melhor valor do *A-Team*
E- Freqüência do melhor valor
F- % Distância entre média do *A-Team* e *lower bound*
G- Tempo de CPU (segundos) para cálculo de *lower bound*
H- Média de tempo real (segundos) para melhor solução do *A-Team*

Tabela 5.8: Resultados obtidos pelo *A-Team* para instâncias da classe 10%

A	B	C	D	E	F	G	H
50	925	982,0	982	3/3	6,16	7,5	1,5
50	934	979,0	979	3/3	4,82	23,3	19,0
50	930	993,0	993	3/3	6,77	0,5	6,9
100	1356	1392,0	1392	3/3	2,65	112,7	91,6
100	1236(3)	1288,0	1288	3/3	4,21	58,8	26,6
100	1420	1465,0	1465	3/3	3,17	34,2	124,2
150	1640	1681,0	1681	3/3	2,50	131,8	133,0
150	1706(2)	1748,3	1748	2/3	2,48	12,3	1203,4
150	1585	1641,0	1641	3/3	3,53	16,8	92,2
200	1970(3)	2022,0	2022	3/3	2,64	537,9	3367,5
200	1883(2)	1934,0	1934	3/3	2,71	22,7	790,9
200	1818(2)	1888,7	1888	2/3	3,89	12,6	5127,3
500	2865(2)	2974,7	2973	1/3	3,83	152,4	24835,3

Legenda: A- Tamanho da instância (número de vértices)
B- Valor de *lower bound*
C- Média do *A-Team*
D- Melhor valor do *A-Team*
E- Freqüência do melhor valor
F- % Distância entre média do *A-Team* e *lower bound*
G- Tempo de CPU (segundos) para cálculo de *lower bound*
H- Média de tempo real (segundos) para melhor solução do *A-Team*

Tabela 5.9: Resultados obtidos pelo *A-Team* para instâncias da classe 50%

A	B	C	D	E	F	G	H
50	968	1006,0	1006	3/3	3,93	9,6	2,4
50	927	965,0	965	3/3	4,10	0,7	26,2
50	974	1009,0	1009	3/3	3,59	3,4	8,7
100	1347	1384,0	1384	3/3	2,75	17,2	20,6
100	1347	1396,0	1396	3/3	3,64	2734,2	268,3
100	1422	1445,0	1445	3/3	1,62	125,0	30,9
150	1667	1716,3	1716	2/3	2,96	265,6	925,8
150	1686(3)	1723,0	1723	3/3	2,19	7116,2	251,6
150	1709	1736,0	1736	3/3	1,58	977,3	228,6
200	1926(3)	1985,0	1985	3/3	3,06	2874,3	2285,5
200	1814(3)	1873,3	1872	1/3	3,27	7560,7	2685,0
200	1830(3)	1900,0	1900	3/3	3,83	4253,7	4188,5
500	2920(2)	3007,7	3006	1/3	3,00	199,4	26853,3

Legenda: A- Tamanho da instância (número de vértices)
B- Valor de *lower bound*
C- Média do *A-Team*
D- Melhor valor do *A-Team*
E- Freqüência do melhor valor
F- % Distância entre média do *A-Team* e *lower bound*
G- Tempo de CPU (segundos) para cálculo de *lower bound*
H- Média de tempo real (segundos) para melhor solução do *A-Team*

Tabela 5.10: Resultados obtidos pelo *A-Team* para instâncias da classe 90%

representam distância do *lower bound*, e não do valor ótimo.

Quanto às classes de instância testadas, as médias de distância obtidas são mostradas na Tabela 5.11. Não houve diferença significativa entre as médias, mostrando que as três classes de instância apresentam dificuldade de solução igual ou parecida. Em relação ao tamanho das instâncias, os resultados obtidos também não tiveram grandes diferenças. O que realmente variou foi o tempo, ou seja, o *A-Team* conseguiu resultados igualmente bons em todas as instâncias, mas precisou de mais tempo para isto em instâncias maiores.

Classe de instância	Média (%) de distância do *lower bound*
10 % do prêmio mínimo	2,78
50 % do prêmio mínimo	3,80
90 % do prêmio mínimo	3,04

Tabela 5.11: Porcentagem média de distância do *lower bound* por classe de instância

5.3.1 Eficiência em Escala

Conforme dito anteriormente, os Times Assíncronos costumam apresentar eficiência em escala, ou seja, os resultados obtidos pelo time melhoram conforme são acrescentados novos agentes a ele.

Para verificarmos isto, efetuamos diversos testes com o *A-Team* com números diferentes de agentes. As configurações utilizadas neste teste estão ilustradas na Figuras 5.3 e 5.4. São oito configurações diferentes, sendo que a última (H) corresponde à configuração final do *A-Team* (a mesma mostrada na Figura 5.1). A diferença de uma configuração para a outra é a inclusão de um ou mais agentes. Quando foram incluídos mais de um (por exemplo, da configuração G para a H) isto se deveu à necessidade de a inclusão formar mais um ciclo no *A-Team*.

A ordem de inserção dos agentes no *A-Team* foi determinada empiricamente. A abordagem utilizada foi a de se incluir em seqüência agentes que tivessem o mesmo tipo de "estratégia" de montagem de solução, e depois que todos que usassem de uma mesma estratégia fossem incluídos, passou-se para um outro conjunto de agentes de estratégias parecidas, e assim sucessivamente.

Estas configurações foram aplicadas sobre uma instância de 200 vértices e com 50% do prêmio total como prêmio mínimo. Os resultados podem ser vistos na Tabela 5.12, e comprovam a melhoria dos resultados, tanto na média (de três execuções), quanto no melhor valor obtido.

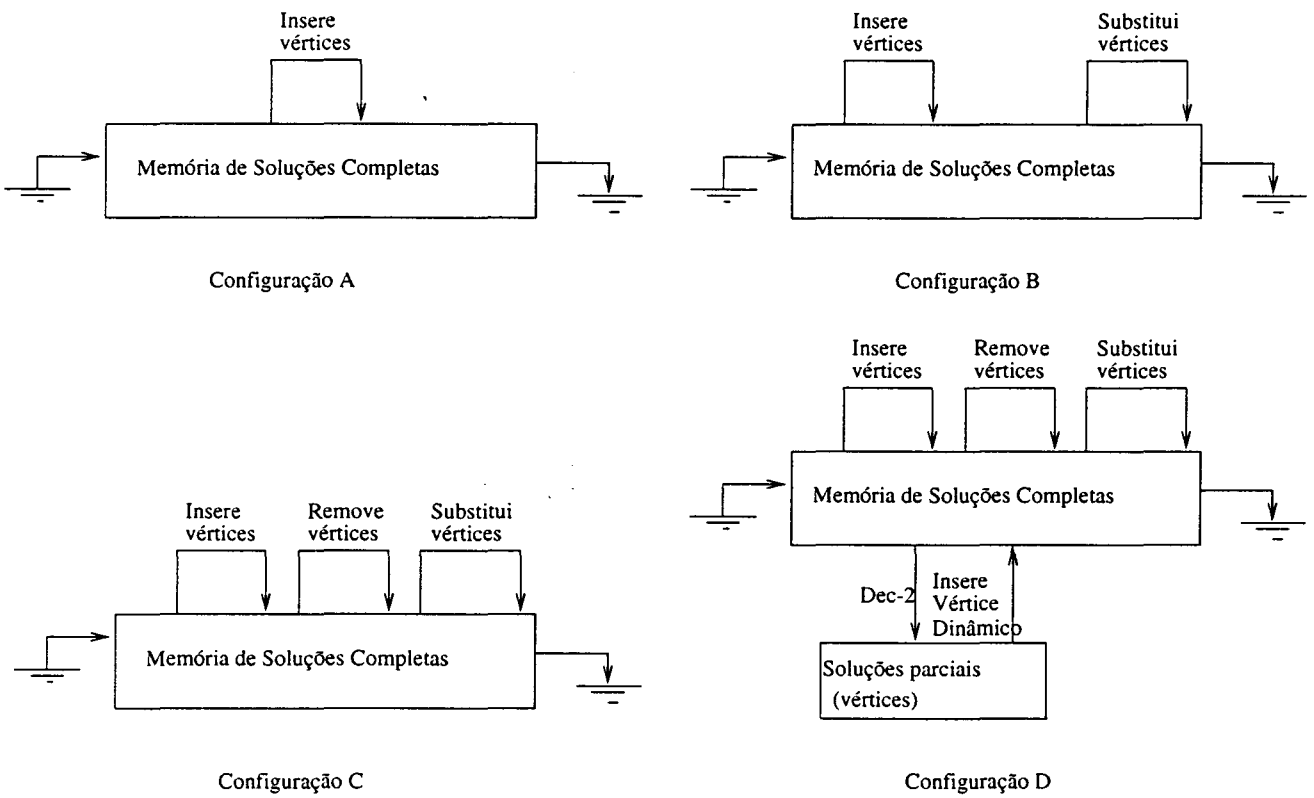
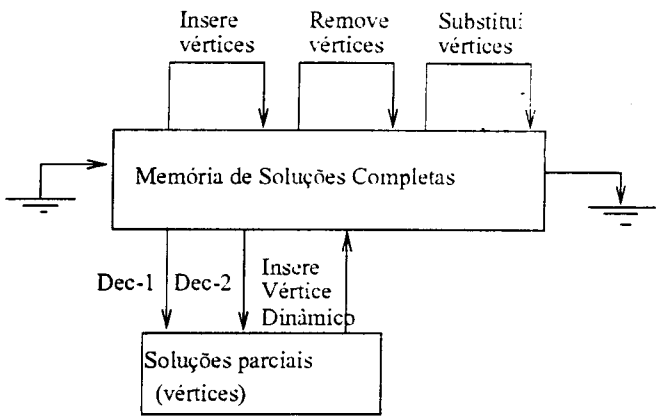


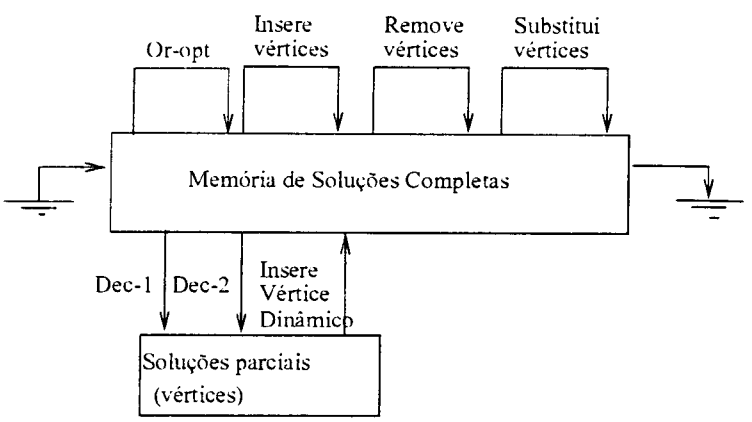
Figura 5.3: Configurações para teste de eficiência em escala.

Configuração	Média do *A-Team*	Melhor valor do *A-Team*
A	2162,0	2159
B	2161,0	2153
C	2150,7	2148
D	2149,7	2137
E	2142,0	2135
F	2078,0	2074
G	2023,3	2023
H	2022,0	2022

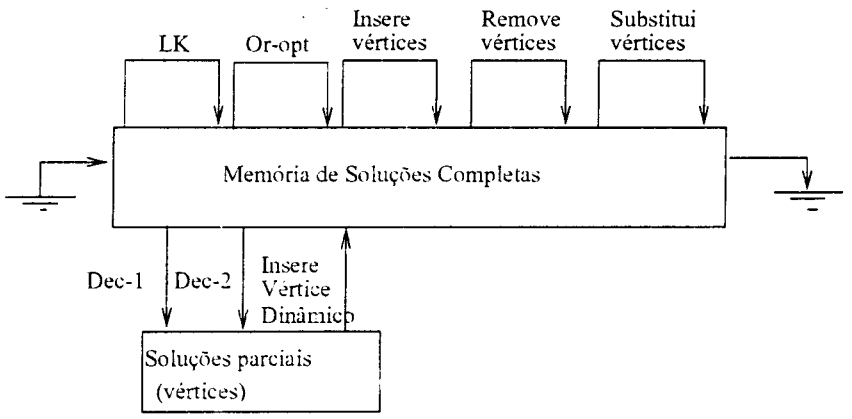
Tabela 5.12: Resultados do teste de eficiência em escala para instância t200.5.



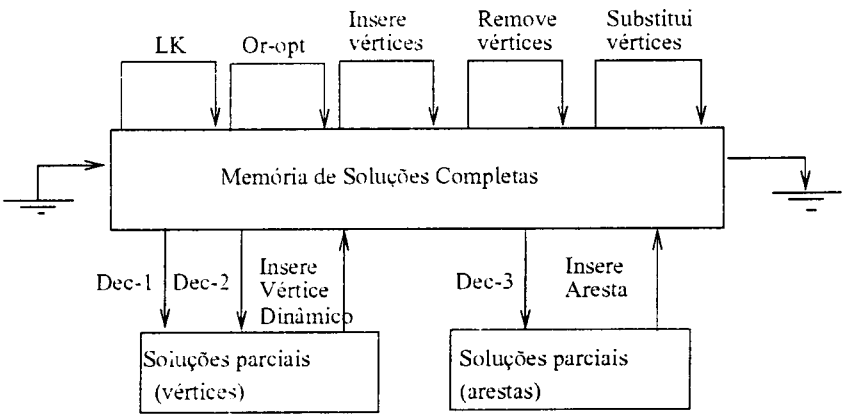
Configuração E



Configuração F



Configuração G



Configuração H

Figura 5.4: Mais configurações para teste de eficiência em escala.

Capítulo 6

Conclusão

Este trabalho propõe algoritmos heurísticos de construção e melhoria de soluções para o Problema do Caixeiro Viajante Coletor de Prêmios (PCTSP), uma generalização do clássico Problema do Caixeiro Viajante (TSP). Após uma revisão de heurísticas disponíveis na literatura do problema, é efetuada uma comparação entre as heurísticas de construção propostas e a que apresenta melhor garantia de desempenho dentre as heurísticas da literatura. Vale ressaltar que as heurísticas propostas se aplicam a versões mais gerais do PCTSP, e que uma delas obteve melhores resultados em quase todas as instâncias do problema testadas.

Além disso, utilizou-se a meta-heurística conhecida como Times Assíncronos, uma organização multi-algorítmica já aplicada com sucesso a diversos problemas de Otimização Combinatória. Foi desenvolvido um Time Assíncrono para o PCTSP, usando como agentes algumas das heurísticas desenvolvidas citadas anteriormente, outras que foram criadas para serem parte do Time e algumas heurísticas da literatura disponível para o Problema do Caixeiro Viajante. Foram realizados diversos testes para se determinar os parâmetros de construção do Time Assíncrono, tais como sua política de aceitação de novas soluções e seus algoritmos de iniciação.

O Time Assíncrono foi implementado de forma distribuída utilizando-se o pacote PVM (*Parallel Virtual Machine*). Foram realizados testes computacionais com instâncias aleatórias criadas. Como parâmetro de comparação, foram implementados modelos de programação matemática de relaxações do PCTSP para estas instâncias. Estes modelos foram resolvidos através do pacote Cplex, e resultaram em limites inferiores para as instâncias testadas.

A média de distância entre os resultados obtidos pelo *A-Team* e os limites inferiores das instâncias testadas foi menor do que 4 % (quatro porcento), atestando a aplicabilidade

de Times Assíncronos ao *Prize Collecting* TSP. Também foi verificada a existência de eficiência em escala no Time Assíncrono implementado ao PCTSP, ou seja, a adição de agentes produziu melhores resultados.

6.1 Contribuições

As principais contribuições deste trabalho são:

- concepção e implementação de novas heurísticas de construção e melhoria de soluções para o PCTSP.
- realização de testes computacionais de comparação entre a heurística de construção com melhor garantia de desempenho disponível na literatura e as heurísticas de construção concebidas, resultando na constatação de que as heurísticas concebidas em geral obtem melhores resultados, além de serem aplicáveis a versões mais gerais do problema.
- comprovação da adequabilidade de *A-Teams* ao Problema do Caixeiro Viajante Coletor de Prêmios.
- constatação da existência de eficiência em escala em *A-Teams* quando aplicados ao PCTSP.
- definição de várias instâncias de *benchmark* para o PCTSP, com cálculo de limitantes inferiores para estas instâncias obtidos através de relaxações do problema.
- constatação da facilidade de utilização de recursos computacionais paralelos através de *A-Teams* para problemas de otimização de grande porte.

6.2 Extensões e Trabalhos Futuros

Dentre as possibilidades de pesquisa futura relacionadas a este trabalho, podemos citar:

- desenvolvimento de novas heurísticas para o PCTSP.
- análise quanto a uma possível garantia de desempenho das heurísticas concebidas neste trabalho.
- desenvolvimento de outros fluxos de dados/estruturas para *A-Teams* aplicados ao PCTSP.

- estudo do *A-Team* desenvolvido quanto à verificação do desempenho do mesmo em relação ao número de processadores/máquinas utilizados.

Bibliografia

[AABV94] B. Awerbuch, Y. Azar, A. Blum, e S. Vempala. Improved approximation guarantees for minimum-weight k-trees and prize collecting salesmen. Technical report, School of Computer Science, Carnegie-Mellon University, 1994.

[Apo91] B. Apolloni. Estable heuristics for the 0-1 knapsack problem. *Kibernetika*, 2, 1991.

[AS83] A. Adrabinski e M. M. Syslo. Computational experiments with some approximation algorithms for the traveling salesman problem. *Zastos. Mat.*, 18:91–95, 1983.

[Bal89] E. Balas. The prize collecting traveling salesman problem. *Networks: An International Journal*, 19:621–636, 1989.

[BCC+94] A. Blum, P. Chalasani, D. Coppersmith, B. Pulleyblank, P. Raghavan, e M. Sudan. The minimum latency problem. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, 1994.

[Ber83] A. A. Bertossi. Finding hamiltonian circuits in proper interval graphs. *Information Processing Letters*, 17:97–101, 1983.

[BGSW93] D. Bienstock, M. X. Goemans, D. Simchi-Levi, e D. Williamson. A note on the prize collecting traveling salesman problem. *Mathematical Programming*, 59:413–420, 1993.

[BT85] E. Balas e P. Toth. Branch and bound methods. In E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, e D. B. Shmoys, editores, *The Traveling Salesman Problem - A Guided Tour of Combinatorial Optimization*, pp. 361–401. John Wiley & Sons, 1985.

[Cam95] E. Camponogara. *A-Teams* para um problema de transporte de derivados de petróleo. Master's thesis, Departamento de Ciência da Computação, Universidade Estadual de Campinas, Campinas, SP, 1995.

[Cav95] V. F. Cavalcante. Times assíncronos na solução do *job shop scheduling problem*: heurísticas de construção. Master's thesis, Departamento de Ciência da Computação, Universidade Estadual de Campinas, Campinas, SP, 1995.

[Chr76] N. Christofides. Worst-case analysis of a new heuristic for the traveling salesman problem. Technical Report 388, Graduate School of Industrial Administration, Carnegie-Mellon University, 1976.

[CPL94] CPLEX. *Using the CPLEX callable library - Using the CPLEX linear optimizer with CPLEX Barrier and Mixed Integer Solvers.* CPLEX Optimization, Inc., 1994. Version 3.0.

[DFJ54] G. B. Dantzig, D. R. Fulkerson, e S. M. Johnson. Solution of a large-scale traveling-salesman problem. *Operations Research*, 2:393–410, 1954.

[dS93] P. S. de Souza. *Asynchronous Organizations for Multi-Algorithm Problems.* PhD thesis, Electrical and Computer Engineering Department, Carnegie Mellon Univerisity, Pittsburgh, PA, 1993.

[dST91] P. S. de Souza e S. N. Talukdar. Genetic algorithms in asynchronous teams. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, 1991.

[Gar85] R. S. Garfinkel. Motivation and modeling. In E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, e D. B. Shmoys, editores, *The Traveling Salesman Problem - A Guided Tour of Combinatorial Optimization*, pp. 17–36. John Wiley & Sons, 1985.

[GB90] M. X. Goemans e D. J. Bertsimas. On the parsimonious property of connectivity problems. In *Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms*, 1990.

[GBD+94] A. Geist, A. Begueli, J. Dongarra, W. Jiang, R. Mancheck, e V. Sunderam. *PVM: Parallel Virtual Machine - A users guide tutorial for networked parallel computing.* The MIT Press, 1994.

[GBDS80] B. L. Golden, L. D. Bodin, T. Doyle, e W. Stewart. Approximate traveling salesman algorithms. *Operations Research*, 28:694–711, 1980.

[GJ79] M. R. Garey e D. S. Johnson. *Computers and Intractability: a guide to the theory of NP-Completeness.* Freeman, San Francisco, 1979.

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

[Glo90a] F. Glover. Tabu search: a tutorial. *Interfaces*, 20:74–94, 1990.

[Glo90b] F. Glover. Tabu search II. *ORSA Journal on Computing*, 2(1):4–32, 1990.

[Gol89] D. E. Goldberg. *Genetic Algorithms in search, optimization and machine learning.* Addison-Wesley, 1989.

[Gom60] R. E. Gomory. Solving linear programming problems in integers. In *Proceedings of the Symposium on Applied Mathematics*, volume 10, pp. 211–215, 1960.

[GS85] B. L. Golden e W. R. Stewart. Empirical analysis of heuristics. In E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, e D. B. Shmoys, editores, *The Traveling Salesman Problem - A Guided Tour of Combinatorial Optimization*, pp. 207–249. John Wiley & Sons, 1985.

[GW92] M. X. Goemans e D. P. Williamson. A general approximation technique for constrained forest problems. In *Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, 1992.

[Had97] E. G. Haddad. Times assíncronos para o *job shop scheduling problem*: heurísticas de melhoria. Master's thesis, Instituto de Computação, Universidade Estadual de Campinas, Campinas, SP, 1997.

[HW85] A. J. Hoffman e P. Wolfe. History. In E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, e D. B. Shmoys, editores, *The Traveling Salesman Problem - A Guided Tour of Combinatorial Optimization*, pp. 1–15. John Wiley & Sons, 1985.

[JP85] D. S. Johnson e C. H. Papadimitriou. Performance guarantees for heuristics. In E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, e D. B. Shmoys, editores, *The Traveling Salesman Problem - A Guided Tour of Combinatorial Optimization*, pp. 145–180. John Wiley & Sons, 1985.

[KJV83] S. Kirkpatrick, C. D. Gelatt Jr., e M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, Maio de 1983.

[LK73] S. Lin e B. W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, 21:498–516, 1973.

[Lon95] H. J. Longo. Aplicação de *A-Teams* ao problema de recobrimento. Master's thesis, Departamento de Ciência da Computação, Universidade Estadual de Campinas, Campinas, SP, 1995.

[MN92] J. Mittenthal e C. E. Noon. An insert/delete heuristic for the traveling salesman subset-tour problem with one additional constraint. *Journal of the Operational Research Society*, 43:277–283, 1992.

[MRR+53] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. Teller, e E. Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, junho de 1953.

[MT90] S. Martello e P. Toth. *Knapsack Problems - Algorithms and Computer Implementations.* John Wiley & Sons, 1990.

[Mur92] S. S. Murthy. *Synergy in Cooperating Agents: Designing Manipulators from Task Specifications.* PhD thesis, Electrical and Computer Engineering Department, Carnegie Mellon Univerisity, Pittsburgh, PA, 1992.

[Nav93] D. Navin-Chandra. ReStar: a design tool for environmental recovery analysis. In *Proceedings of the International Conference on Engineering Design (ICED'93)*, Agosto de 1993.

[NSdS96] H. A. D. Nascimento, C. F. X. Souza, e P. S. de Souza. Sinergia em desenho de grafos usando *springs* e pequenas heurísticas. In *XXIII Seminário Integrado de Software e Hardware*, 1996.

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

[Pei95] H. P. Peixoto. Metodologia de especificação de times assíncronos para problemas de otimização combinatória. Master's thesis, Departamento de Ciência da Computação, Universidade Estadual de Campinas, Campinas, SP, 1995.

[RdS95] W. E. Ribeiro e P. S. de Souza. Heurísticas de construção para o Prize Collecting TSP. In *XXVII Simpósio Brasileiro de Pesquisa Operacional*, Vitória-ES, 1995.

[Rod96] R. F. Rodrigues. Times assíncronos para a resolução de problemas de otimização combinatória com múltiplas funções objetivo. Master's thesis, Departamento de Ciência da Computação, Universidade Estadual de Campinas, Campinas, SP, 1996.

[SCGD92] U. K. Sarkar, P. P. Chakrabarti, S. Ghose, e S. C. DeSarkar. A simple 0.5-bounded greedy algorithm for the 0/1 knapsack problem. *Information Processing Letters*, 42, 1992.

[SW90] D. Shmoys e D. Williamson. Analyzing the Held-Karp TSP bound: A monotonicity property with application. *Information Processing Letters*, 35, 1990.

[TdS90] S. N. Talukdar e P. S. de Souza. Asynchronous teams. In *Second SIAM Conference on Linear Algebra: Signals, Systems and Control*, San Francisco, CA, novembro de 1990.

[TdS92] S. N. Talukdar e P. S. de Souza. Scale efficienty organizations. In *IEEE International Conference on Systems, Man and Cybernetics*, Outubro de 1992.

[Tse95] C. K. Tsen. *Solving Train Scheduling Problems using A-Teams.* PhD thesis, Electrical and Computer Engineering Department, Carnegie Mellon University, Pittsburgh, PA, 1995.

[Tur36] A. Turing. On computable numbers with an application to the entscheidungsproblem. In *Proc. of the London Mathematical Society Ser.*, 1936.