Universidade Federal de Mato Grosso do Sul
Centro de Ciências Exatas e Tecnologia
Departamento de Computação e Estatística
Mestrado em Ciência da Computação
Complexidade de Algoritmos
Professor Doutor Celso Carneiro Ribeiro

Problema do Caixeiro Viajante

Travelling Salesman Problem

Delair Osvaldo Martinelli Júnior
Eraldo Luís Rezende Fernandes
Marcos Leal Medeiros

Campo Grande
Agosto, 2002

Conteúdo

1 Introdução **1**

1.1 O Problema 1

1.2 Variações 2

1.3 Abordagem Utilizando Heurísticas 2

2 Heurísticas de Construção **4**

2.1 Introdução 4

2.2 Heurística do Vizinho Mais Próximo 4

2.3 Heurística Gulosa 5

2.4 Heurística de Clarke-Wright 5

2.5 Heurística de Duplicação da Árvore Geradora Mínima 6

2.6 Heurística de Christofides 7

3 Heurísticas de Busca Local **9**

3.1 Introdução 9

3.2 2-Opt 9

3.3 3-Opt 10

3.4 Avaliação Teórica 11

4 Heurística GRASP **12**

4.1 Introdução 12

4.2 Fase de Construção 13

4.3 Fase de Busca Local 13

4.4 Regulagem de Parâmetros 14

5 Implementação 16

5.1 Introdução 16

5.2 Módulo `instancereader` 16

5.2.1 A Classe `tInstanceReader` 17

5.3 Módulo `graph` 17

5.3.1 A Classe `tVertex` 18

5.3.2 A Classe `tEdge` 19

5.3.3 A Classe `tGraph` 19

5.3.4 A Classe `tGraphTour` 21

5.4 Módulo `nearestneighbour` 21

5.5 Módulo `greedy` 21

5.6 Módulo `hub` 22

5.7 Módulo `doublemst` 24

5.8 Módulo `tourlist` 25

5.8.1 A Classe `tTourNode` 25

5.8.2 A Classe `tTourVertexNode` 25

5.8.3 A Classe `tTourEdgeIterator` 26

5.8.4 A Classe `tTourList` 26

5.9 Módulo `localsearch` 26

5.10 Módulo `grasp` 27

6 Resultados Empíricos 29

6.1 Introdução 29

6.2 Avaliação de Tempo de Processamento 29

6.2.1 Heurísticas de Construção 31

6.2.2 Heurísticas de Busca Local 33

6.2.3 Heurística GRASP 34

6.3 Avaliação da Qualidade da Solução 34

7 Conclusão **36**

Bibliografia **37**

Lista de Figuras

3.1 Movimento 2-Opt: circuito original à esquerda e resultante à direita 10

3.2 Dois possíveis movimentos 3-Opt: circuito original à esquerda e resultantes à direita 10

4.1 Pseudocódigo da metaheurística GRASP 12

4.2 Pseudocódigo da fase de construção 13

4.3 Pseudocódigo da fase de busca local 14

4.4 Pseudocódigo da fase de construção utilizando uma lista restrita de candidatos 15

6.1 Comparação entre as três heurísticas mais rápidas. 31

6.2 Comparação da heurística Clarke-Wright com as outras três. 32

6.3 Comparação entre as heurísticas 2-Opt e 3-Opt. 33

6.4 Comparação entre a heurística GRASP utilizando busca local 2-Opt e 3-Opt. 34

6.5 Comparação entre a heurística GRASP variando o fator de qualidade utilizando a instância kroA200. 35

Capítulo 1

Introdução

1.1 O Problema

É conveniente formalizar o problema na linguagem de teoria dos grafos. Em particular, convém recorrer ao conceito de circuito hamiltoniano. Para um grafo G, temos que um caminho hamiltoniano, é um caminho que percorre todos os vértices de G passando uma e somente uma vez sobre cada vértice de G. Um circuito hamiltoniano, é uma caminho hamiltoniano onde todos os vértices do caminho possuem grau 2, isto é, forma um circuito, partimos de um vértice, percorremos o caminho e retornamos para o mesmo vértice. O Problema do Caixeiro Viajante (*Travelling Salesman Problem*), denotado por TSP, é definido da seguinte maneira:

Problema $TSP(G, c)$: Dado um grafo G e um custo $c_e \in Q_{\geq}$ para cada aresta e, determinar um circuito hamiltoniano C que minimize $c(C)$.

TSP é talvez o mais famoso problema de otimização combinatória, em parte graças às conexões com vários outros problemas de otimização. O problema é NP-difícil, ou seja, não se conhece algoritmos eficientes (polinomiais) que encontrem a solução ótima mesmo com $c_e \in \{1, 2\}$ para toda aresta e [GJ79]. Também não é conhecido nenhum algoritmo de aproximação com razão constante para o problema.

Existem algumas variações para o problema e também podemos conseguir soluções muito boas para casos particulares, isto é, podemos restringir o conjunto de entrada usando propriedades que ajudem a encontrar uma solução de forma eficiente.

1.2 Variações

Este é um problema que possui muitas variações, vamos apresentar algumas bem conhecidas. Algumas possuem algoritmos de aproximação muito bons.

Temos variações do TSP em relação à:

- **Simetria**: É dito **simétrico** se a distância da cidade a à cidade b é igual à distância da cidade b à cidade a. Do contrário é dito **assimétrico**.
- **Completude**: É dito **completo** se existe um caminho direto entre todas as cidades. Do contrário é dito **não completo**.

Suponha que o grafo G, com peso nas arestas, é completo. Dizemos que o peso de suas arestas satisfazem a **desigualdade triangular** se

$$c_{ik} \leq c_{ij} + c_{jk}$$

para quaisquer três vértices i,j,k. O TSP restrito ao conjunto de instâncias em que G é completo e o custo de suas arestas satisfazem a desigualdade triangular é conhecido como **métrico**.

Existe ainda o TSP **geométrico**, no qual os vértices possuem localização espacial e o peso de cada aresta é igual à distância relativa às coordenadas dos dois vértices conectados por ela.

Apesar das várias restrições apresentadas, todas estas variações do TSP ainda são NP-difíceis.

Nas heurísticas descritas nos próximos capítulos assumimos que as instâncias são completas e simétricas. E em algumas argumentações sobre performance assumimos a desigualdade triangular.

Utilizamos como restrição para as instâncias utilizadas nas heurísticas descritas que o grafo seja completo, simétrico e que satisfaz a desigualdade triangular.

1.3 Abordagem Utilizando Heurísticas

Heurísticas são algoritmos polinomiais utilizados para encontrar soluções para problemas NP-difíceis. A solução encontrada por uma heurística não é garantidamente a ótima, porém uma boa heurística encontra soluções satisfatoriamente próximas da ótima.

Dois teoremas limitam o comportamento de qualquer heurística para o TSP. Dados uma uma heurística A e uma instância I do TSP, denotaremos A(I) como o custo do circuito obtido utilizando A e OPT(I) como o custo da solução ótima.

Teorema 1: Assumindo que $P \neq NP$, nenhuma heurística de tempo polinomial pode garantir que $A(I)/OPT(I) \leq 2^{P(N)}$ para algum P fixo e qualquer instância I.

Teorema 2: Assumindo que $P \neq NP$, existe um $\epsilon > 0$ tal que nenhuma heurística de tempo polinomial pode garantir que $A(I)/OPT(I) \leq 1 + \epsilon$ para todas as instâncias I que satisfaçam a desigualdade triangular.

Felizmente, a maioria das aplicações impôem restrições substanciais às instâncias permitidas. Em particular, na maioria das aplicações as distâncias obedecem a desigualdade triangular. Neste caso o Teorema 1 não é válido, e o único limite conhecido é o Teorema 2, que comparado ao Teorema 1 impôe apenas uma pequena limitação à performance do algoritmo.

Capítulo 2

Heurísticas de Construção

2.1 Introdução

Heurísticas de construção para o Problema do Caixeiro Viajante são algoritmos que geram um circuito viável partindo de um conjunto inicial (que pode ser vazio) de vértices e/ou arestas, e modificando esse conjunto a cada iteração utilizando algum critério de escolha. Não há garantia de que este circuito seja ótimo.

2.2 Heurística do Vizinho Mais Próximo

É a mais intuitiva das heurísticas tratadas nesse material. Dado um grafo $G(V, E)$ completo e simétrico o Algoritmo do Vizinho Mais Próximo para construir um circuito hamiltoniano T de G é o seguinte:

1. Escolha aleatoriamente um vértice $v \in V(G)$
2. Inclua v em T
3. Seja u o vizinho mais próximo de v que ainda não foi incluído em T
4. Inclua u e a aresta (v, u) em T e faça $v = u$
5. Se $|V(T)| < |V(G)|$ então vá para o passo 3
6. Inclua a aresta entre o primeiro e o último vértices incluídos

O tempo de computação envolvido ao algoritmo do vizinho mais próximo é de $O(n^2)$, para instâncias que satisfazem a desigualdade triangular, este algoritmo possui um resultado melhor que o Teorema 1 da seção 1.3. Em particular podemos garantir $NN(I)/OPT(I) \leq (1/2)(\lceil \log N \rceil + 1)$. Nenhuma garantia substancialmente melhor é possível, no entanto existem instâncias para as quais a razão cresce como $\Theta(\log N)$.

2.3 Heurística Gulosa

Esta heurística é gulosa assim como a heurística anterior. Porém, NN é guloso em relação ao próximo vértice que será incluído na solução, e este algoritmo é guloso em relação à próxima aresta. Dado um grafo $G(V, E)$ completo e simétrico e os custos c_e de cada $e \in E(G)$ o Algoritmo Guloso para construir um circuito hamiltoniano T de G é o seguinte:

1. Faça $V(T) = V(G)$
2. Seja $e \in E(G) - E(T)$ tal que c_e é mínimo e que sua inclusão não implique na formação de um ciclo em T ou deixe algum vértice de T com grau maior que 2
3. Inclua e em $E(T)$
4. Se $|E(T)| < |V(G)|$ então vá para o passo 2

O tempo de computação envolvido ao algoritmo guloso é de $\Theta(N^2 \log N)$, mais lento que NN, porém a solução encontrada geralmente é melhor. Com relação a qualidade da solução encontrada, podemos garantir que $Greedy(I)/OPT(I) \leq (1/2)(\lceil \log N \rceil + 1)$

2.4 Heurística de Clarke-Wright

Em termos do problema do caixeiro viajante, iniciamos com um pseudo-circuito no qual escolhemos um vértice arbitrário, chamado de ***hub***, ao qual ligamos todos os vértices restantes por duas arestas. Este processo cria um grafo especial chamado de multigafo de G.

Nossa tarefa é verificar para cada par de vértices não *hub*, se o caminho direto entre eles é mais curto do que passar pelo *hub*. Depois de encontrado o par, a troca de arestas é feita, a não ser que forme um ciclo com vértices não *hub* ou deixe um vértice não *hub* ligado a mais de dois vértices não *hub*. O processo termina quando restar apenas 2 vértices ligados ao *hub*. Temos então um circuito hamiltoniano.

O tempo de computação envolvido ao algoritmo Clarke-Wright é de $\Theta(N^2 \log N)$. Com relação a qualidade da solução encontrada, podemos garantir que $CW(I)/OPT(I) \leq \lceil \log N \rceil + 1$ (com instâncias que garantam a desigualdade triangular).

2.5 Heurística de Duplicação da Árvore Geradora Mínima

Uma árvore geradora de um grafo G é um subgrafo conexo de G, que possui todos os vértices de G e não possui circuito. Uma **árvore geradora mínima** (AGM) é aquela onde a soma do custo de suas arestas é mínima.

Uma AGM dá uma boa delimitação inferior para o valor ótimo do **TSP métrico** ($TSPM(G, c)$). Vamos definir como $c(S)$ o custo de S, onde S pode ser um conjunto qualquer de arestas. Se removemos uma aresta de um circuito hamiltoniano temos uma árvore geradora de custo não superior ao do circuito. Portanto,

$$opt(G, c) \geq c(T)$$

A heurística de Duplicação da AGM utiliza uma estratégia que compreende três passos:

- Construir uma AGM T de G;
- Duplicar as arestas de T obtendo um multigrafo P;
- Obter um circuito hamiltoniano a partir de P.

Para resolver o passo 1 temos algoritmos eficientes [BM76, CLR92], inclusive um algoritmo cuja complexidade é $O(N^2)$, onde N é o número de vértices do grafo.

A resolução do passo 2 é direta e simples. Bastando criar, para cada aresta de T, duas arestas em P.

O passo 3 da heurística é bem simples, pois no passo 2 garantimos que P é um circuito euleriano (dobramos o grau de todos os vértices, garantindo grau par para todos). O passo 3 se resume no seguinte: percorrer os vértices do circuito P, $(v_1, v_2, \ldots, v_m)$, na ordem em que aparecem em P e armazenar a sequência de vértices não repetidos. O seguinte procedimento ilustra esta idéia:

```
1   w0 ← v0
2   n ← 0
3   para i de 1 até m faça
4       se vi ∉ {w0, ..., wn} então
5           n ← n + 1
6           wn ← vi
```

Como o grafo é completo, a sequência $(w_1, w_2, \ldots, w_n, w_1)$ é um circuito. E o circuito contém todos os vértices do grafo, pois o circuito dado contém todos os vértices (é euleriano). E ainda, o circuito não possui vértices repetidos pois isto é garantido na condição da linha 4. Portanto, o circuito resultante é um circuito hamiltoniano.

Teorema 3: O Algoritmo Double MST é uma 2-aproximação polinomial para o TSP simétrico.

Demonstração: Como P é um ciclo euleriano em $T + E_T$, temos que $c(P) = 2c(T)$. Então, utilizando o fato de $opt(G, c) \geq c(T)$ e que $c(C) \leq c(P)$ temos:

$$c(C) \leq c(P) = 2c(T) \leq 2OPT(G, c)$$

A linha 1 do algoritmo consome $O(\mid V_G \mid^2)$, enquanto que as demais linhas consomem tempo $O(\mid V_G \mid)$. logo, o algoritmo é polinomial.

2.6 Heurística de Christofides

Um emparelhamento em um grafo é um conjunto de arestas sem extremos em comum. Um emparelhamento M é perfeito se cada vértice do grafo pertence a uma aresta de M. O algoritmo de Edmonds [LP86], que denotaremos por EDMONDS, encontra um emparelhamento perfeito de custo mínimo em tempo $O(n^3)$, onde n é o número de vértices do grafo.

O algoritmo de Christofides acrescenta à arvore geradora de T de G um emparelhamento perfeito no subgrafo de G induzido pelos vértices de grau ímpar em T, depois utiliza a técnica do atalho, descrita na seção 2.5 para baixar o custo do tour.

```
Algoritmo:Christofides(G,c)
Saída:    C - Circuito hamiltoniano em G
1 - T <- MST(G,C)
2 - seja I o conjunto os vértices de grau ímpar de T
3 - M <- EDMOND(G[I],c)
4 - T'<- T + M
5 - P <- EULER(T')
6 - C <- ATALHO(P)
7 - devolva C
```

Como M é um emparelhamento perfeito em G[I], todo vértice de $T + M$ tem grau par

e, portanto, o multigrafo T' na linha 4 tem ciclo eureliano. O ciclo é gerado pois T é geradora. Na linha 6 do algoritmo, C é um circuito hamiltoniano de G.

Teorema 3: O Algoritmo TSP-Christofides é uma $\frac{3}{2}$-aproximação polinomial para o TSP simétrico.

Demonstração: Temos que $c(C) \leq c(P)$. Por outro lado, $c(P) = c(T') = c(T) + c(M)$. Usando $OPT(G,c) \geq c(T)$, temos que $c(C) \leq c(T) + c(M) \leq OPT(G,c) + c(M)$. Portanto, para concluir que $c(C) \leq \frac{3}{2}OPT(G,c)$, basta mostrar que

$$OPT(G,c) \geq 2c(M)$$

Seja C* uma solução ótima para TSP. Sejam $u_1, u_2, ..., u_{2k}$ os vértices de I na ordem em que aparecem em C*. Como G é completo, a seqüência $D := (u_1, u_2, ..., u_{2k}, u_1)$ é um circuito em G[I]. Em outras palavras, D pode ser obtido de C* que liga u_i au_{i+1} pela aresta $u_i u_{i+1}$ de G. A desigualdade triangular garante que $c(D) \leq c(C*)$. Além disso, como D tem comprimento par, E_d é a união de dois emparelhamentos perfeitos em G[I] em G[I] mutuamente disjuntos, digamos M' e M". Logo,

$$2c(M) \leq c(M') + c(M") = c(D) \leq c(C*) = OPT(G,c)$$

sendo que a primeira desigualdade vale porque M é um emparelhamento perfeito de custo mínimo.

A linha 3 do algoritmo consome $O(\mid V_G \mid^3)$, enquanto que as demais linhas consomem tempo $O(\mid V_G \mid^2)$. Portanto, o algoritmo é polinomial.

Capítulo 3

Heurísticas de Busca Local

3.1 Introdução

As heurísticas de construção são importantes não somente pela sua perspectiva mas também porque podem ser usadas para gerar pontos de partida usados por outra classe de heurísticas, as de busca local. Os algoritmos de busca local são baseados em modificaçãoes simples no circuito. Dado um circuito hamiltoniano, esses algoritmos tentam fazer trocas para que seu comprimento seja reduzido, até que não seja impossível reduzi-lo mais (um circuito localmente ótimo). Este capítulo descreve as heurísticas de busca local 2-Opt e 3-Opt, que são as mais simples e famosas, além de argumentar sobre o que se sabe sobre elas de um ponto de vista teórico. Além de oferecer bons resultados, essas heurísticas são usadas por muitos pesquisadores como parte de outras técnicas usadas no Problema do Caixeiro Viajante.

3.2 2-Opt

O algoritmo 2-Opt foi proposto primeiramente por [CRO58], embora o movimento básico já tinha sido sugerido por [FLO56]. Esse movimento apaga duas arestas, quebrando o circuito em dois caminhos, e então reconecta esses caminhos da outra maneira possível. O movimento é realizado somente se a outra maneira possível reduz o comprimento do circuito.

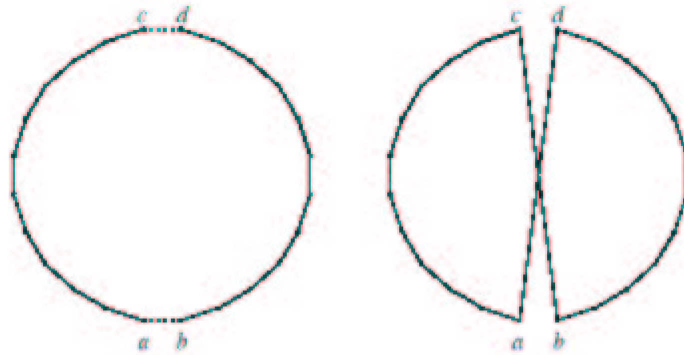

Figura 3.1: Movimento 2-Opt: circuito original à esquerda e resultante à direita

3.3 3-Opt

O algoritmo 3-Opt, proposto por [BOC58], funciona quase da mesma maneira que o algoritmo 2-Opt. A primeira diferença ocorre no movimento básico, onde três arestas são apagadas, quebrando o circuito em três caminhos. Outra diferença está na reconexão destes caminhos, que pode ser feita de duas maneiras diferentes da qual estavam conectados. O movimento é realizado se uma das duas maneiras diferentes reduz o comprimento do circuito. Se as duas maneiras reduzem o circuito, então opta-se por reconectar os caminhos da maneira que implica na maior redução. Além dos algoritmos 2-Opt e 3-Opt, temos o k-Opt que é uma generalização que permite trocas de k arestas.

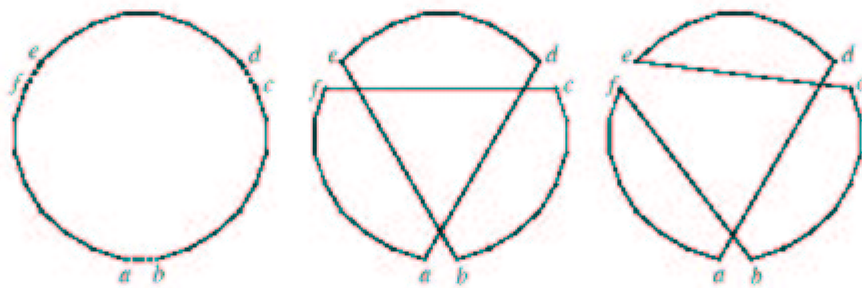

Figura 3.2: Dois possíveis movimentos 3-Opt: circuito original à esquerda e resultantes à direita

3.4 Avaliação Teórica

A primeira questão a se preocupar envolve a qualidade dos circuitos obtidos através de um algoritmo busca local. Para instâncias arbitrárias, esses algoritmos são limitados pelos teoremas citados na seção 1.3, mas algo mais forte pode ser dito: se $P \neq NP$, nenhum algoritmo de busca local que gasta tempo polinomial por movimento pode garantir um limite superior constante para a razão entre o circuito encontrado e o circuito ótimo. Para os caso especificos do 2-opt, 3-Opt e k-Opt($k < 3n/8$), [PS78] mostraram que existem instâncias que possuem um único circuito ótimo e muitos circuitos localmente ótimos, cada um distante do ótimo por um fator exponencial. Para um circuito de partida qualquer, a melhor performance garantida para o 2-Opt é uma razão de no mínimo $(1/4)\sqrt{N}$, e para o 3-Opt é no mínimo $(1/4)\sqrt[6]{N}$. Generalizando, a melhor performance garantida para o k-Opt assumindo a desigualdade triangular é no mínimo $(1/4)\sqrt[2k]{N}$. O lado positivo é que nenhum desses algoritmos produz uma razão pior que $4\sqrt{N}$. A situação fica um pouco melhor se restringirmos atenção às instâncias onde as cidades são pontos em R^d para qualquer d fixo e as distancias são computadas de acordo com alguma regra deste espaço.

Uma outra questão importante sobre os algoritmos de busca local envolve quantos movimentos eles podem fazer antes de atingirem uma solução localmente ótima. Para o 2-opt e o 3-Opt esse número pode ser bem grande, mesmo assumindo que há desigualdade triangular. [LUE76] mostrou que existem instâncias e circuitos de partida nos quais o algoritmo 2-Opt faz $\theta(2^{n/2})$ movimentos. Um resultado similar foi provado para o algoritmo 3-Opt por [CKT94] e extendido para o k-Opt para qualquer k fixo. Esses resultados mostram que não devemos pegar um circuito de partida ruim. Porém, para um k suficientemente grande, o problema de encontrar uma solução ótima local em uma vizinhança k-Opt é PLS-Completo como definido por [JPY88]. Isso significa que o tempo gasto para encontrar um circuito localmente ótimo não pode ser polinomial a menos que todos os problemas PLS sejam resolvidos em tempo polinomial.

Completando a discussão sobre tempo de processamento, temos que considerar também o tempo gasto em um movimento. Isso inclui o tempo gasto para encontrar um movimento que reduza o comprimento do circuito (ou verifique que não há), juntamente com o tempo gasto para realiza-lo. No pior caso, 2-opt e 3-opt requerem tempo $\Omega(N^2)$ e $\Omega(N^3)$ para verificar localidade ótima, assumindo que todos os movimentos possíveis devem ser considerados. Esses custos podem ser ainda maiores, dependendo das estruturas de dados utilizadas.

O comportamento empírico (baseado em resultados práticos) dos algoritmos de busca local contrasta nitidamente com o que é conhecido em teoria sobre eles. Isto ocorre porque muitos resultados teóricos são baseados no pior caso ou em casos muito distantes do caso médio.

Capítulo 4

Heurística GRASP

4.1 Introdução

GRASP (*Greedy Randomized Adaptive Search Procedures*) é uma metaheurística iterativa para problemas combinatórios, onde cada iteração consiste basicamente em duas fases: construção e busca local. A fase de construção gera um circuito hamiltoniano, cuja vizinhança é investigada até que um mínimo local é encontrado durante a fase de busca local. A melhor solução até o momento é mantida como resultado. A condição de parada do algoritmo fica a critério do programador, podendo inclusive ser um número máximo de iterações.

```
procedure GRASP(Max_Iterations,Seed)
1    Read_Input();
2    for k = 1,... ,Max_Iterations do
3         Solution ← Greedy_Randomized_Construction(Seed);
4         Solution ← Local_Search(Solution);
5         Update_Solution(Solution,Best_Solution);
6    end;
7    return Best_Solution;
end GRASP.
```

Figura 4.1: Pseudocódigo da metaheurística GRASP

4.2 Fase de Construção

A cada iteração desta fase, seja o conjunto de elementos candidatos formado por todos os elementos que podem ser incorporados à solução parcial em construção sem destruir a viabilidade. A seleção do próximo elemento é determinada pela avaliação de todos os elementos candidatos de acordo com uma função de avaliação gulosa. Essa avaliação cria uma lista restrita de candidatos (RCL) formada pelos melhores (aspecto guloso da heurística). O elemento a ser incorporado na solução parcial é selecionado randomicamente entre os elementos da lista (aspecto randômico da heurística). Depois que o elemento selecionado é incorporado à solução parcial, a lista de candidatos é atualizada e os custos reavaliados (aspecto adaptativo da heurística). Esta estratégia é similar à heurística semi-gulosa proposta por Hart e Shogan (1987), que também é uma aproximação iterativa baseada em construções gulosas randômicas, mas sem busca local.

```
procedure Greedy_Randomized_Construction(Seed)
1    Solution ← ∅;
2    Evaluate the incremental costs of candidate elements;
3    while Solution is not complete do
4        Build the restricted candidate list (RCL);
5        Select element s from the RCL at random;
6        Solution ← Solution ∪ {s};
7        Reevaluate the incremental costs;
8    end;
9    return Solution;
end Greedy_Randomized_Construction.
```

Figura 4.2: Pseudocódigo da fase de construção

4.3 Fase de Busca Local

As soluções geradas pela construção gulosa randômica não são necessariamente ótimas, até mesmo no que diz respeito a vizinhanças simples. A fase de busca local geralmente melhora a solução construída. Um algoritmo de busca local funciona de modo iterativo trocando sucessivamente a solução corrente por uma solução melhor na sua vizinhança. Ele termina quando nenhuma solução melhor é encontrada. A eficácia de um procedimento de busca local depende de muitos aspectos, tais como a estrutura da vizinhança, a técnica de busca utilizada, a eficiência da função de avaliação dos vizinhos e o percurso de partida. A fase de construção tem um papel importante em relação ao último aspecto, construindo um percurso de partida da boa qualidade. Vizinhanças simples, como a 2-Opt, são geralmente usadas. A busca pode ser implementada usando a estratégia de melhor troca ou de

primeira troca. Na prática, já foi observado que as duas estratégias rumam quase sempre para a mesma solução final, mas num tempo computacional menor quando a estratégia de primeira troca é utilizada.

```
procedure Local_Search(Solution)
1    while Solution not locally optimal do
2        Find s' ∈ N(Solution) with f(s') < f(Solution);
3        Set Solution ← s';
4    end;
5    return Solution;
end Local_Search.
```

Figura 4.3: Pseudocódigo da fase de busca local

4.4 Regulagem de Parâmetros

Uma característica especial da heurística GRASP é a facilidade de implementação. Poucos parâmetros precisam ser inicializados ou ajustados. GRASP tem dois parâmetros principais: um relacionado ao critério de parada e outro em relação à qualidade dos elementos na lista de candidatos.

O critério de parada pode ser um número máximo de iterações. Apesar da probabilidade de encontrar uma solução melhor que a atual reduzir a cada iteração, a qualidade da melhor solução encontrada só pode melhorar. Como a computação não varia muito de iteração para iteração, o total é previsível e aumenta linearmente com o número de iterações. Consequentemente, quanto mais iterações, maior o tempo de computação e melhor a solução encontrada.

Para a construção da lista RCL usada na fase de construção, vamos considerar $c(e)$ o custo de adicionar o elemento $e \in E$ na solução em construção. Em qualquer iteração da heurística GRASP, temos que c_{min} e c_{max} são, respectivamente, o menor e o maior custos. A lista RCL é composta dos elementos $e \in E$ com menores custos. Esta lista pode ser limitada pelo número de elementos ou pela sua qualidade. No primeiro caso, a lista é composta dos p elementos de menor custo, onde p é um parâmetro. Podemos também associar a lista RCL com um parâmetro $\alpha \in [0, 1]$. A lista é composta por todos os elementos $e \in E$ que podem ser inseridos na solução parcial sem destruir a viabilidade e cuja qualidade é superior ao valor limiar, ou seja, $c(e) \in [c_{min}, c_{min} + \alpha(c_{max} - c_{min})]$. Um valor $\alpha = 0$ corresponde a um algoritmo guloso puro, enquanto $\alpha = 1$ é equivalente a uma construção randômica.

```
procedure Greedy_Randomized_Construction(α, Seed)
1     Solution ← ∅;
2     Initialize the candidate set C = E;
3     Evaluate the incremental cost c'(e) for all e ∈ C;
4     while C ≠ ∅ do
5           c^min = min{c'(e) | e ∈ C};
6           c^max = max{c'(e) | e ∈ C};
7           RCL = {e ∈ C | c'(e) ≤ c^min + α(c^max − c^min)};
8           Select element s from the RCL at random;
9           Solution ← Solution ∪ {s};
10          Update the candidate set C;
11          Reevaluate the incremental costs c'(e) for all e ∈ C;
12     end;
13     return Solution;
end Greedy_Randomized_Construction.
```

Figura 4.4: Pseudocódigo da fase de construção utilizando uma lista restrita de candidatos

Capítulo 5

Implementação

5.1 Introdução

Neste capítulo vamos discutir a implementação orientada a objetos das heurísticas apresentadas nos capítulos anteriores. Utilizamos o compilador Borland C++ 5.0 ©.

Nas seções a seguir apresentamos cada módulo da implementação. Cada módulo é constituído por dois arquivos: *.h* (definições) e *.cpp* (implementação).

5.2 Módulo `instancereader`

Neste módulo definimos e implementamos a classe `tInstanceReader` que realiza a leitura de arquivos no formato *EUC-2D*. Este formato foi especificado por [REI91] na *TSPLIB* que é uma biblioteca de instâncias TSP que disponibiliza, além da especificação de cada instância, o melhor resultado encontrado por heurísticas do mundo todo.

Existem vários formatos de arquivos utilizados, e especificados, na TSPLIB mas nossa classe é capaz de reconhecer apenas o formato EUC-2D que é o formato mais comum. Neste formato a instância é especificada por pontos que representam os vértices do grafo. Cada ponto possui um `Id`, que o identifica unicamente dentre todos os pontos, e suas coordenadas cartesianas. Os pesos das arestas do grafo são as distâncias geométricas entre os pontos.

Apesar deste formato restringir somente à instâncias geométricas do TSP, todas as nossas implementações das heurísticas foram construídas para trabalhar com o o TSP completo e simétrico, não sendo necessário satisfazer nem mesmo a desigualdade triangular. Em compensação só realizamos testes com instâncias completas, simétricas e geométricas.

5.2.1 A Classe `tInstanceReader`

Esta classe implementa um *parser* que lê arquivos no formato EUC-2D e armazena em um vetor de `t2DPoint`, a definição desta estrutura é a seguinte:

```
struct t2DPoint
{
  int Id;
  double X;
  double Y;
};
```

Além do conjunto de pontos o *parser* lê e armazena o nome do arquivo, o comentário e a dimensão (número de vértices do grafo). Que podem ser acessados utilizando os métodos abaixo, respectivamente:

```
char* GetFileName();
char* GetComment();
int   GetDimension();
```

O *parser* pode ser disparado chamando o método:

```
bool Run(char* path);
```

onde `path` é o nome do arquivo de entrada.

Para gerar um grafo completo com os pontos lidos e arestas com pesos com as distâncias entre os pontos basta realizar uma chamada ao método:

```
void FillGraph(tGraph* g) const;
```

onde `g` é o grafo que será preenchido. A classe `tGraph` será discutida na seção 5.3.

5.3 Módulo `graph`

Neste módulo definimos e implementamos as classes básicas para representar um grafo. E também uma classe base, `tGraphTour`, que representa um grafo com a capacidade de calcular um **ciclo hamiltoniano** a partir dele próprio.

5.3.1 A Classe `tVertex`

Esta classe representa um vértice de um grafo qualquer. Um vértice tem os seguintes atributos:

- `int Id`: identifica o vértice unicamente entre os vértices do grafo;
- `int Flag`: utilizado por vários algoritmos para vários propósitos. Por exemplo: o algoritmo que cria uma **Árvore Geradora Mínima** do grafo utiliza este atributo para controlar em qual componente o vértice está;
- `tEdgeList AdjacentEdges`: lista duplamente ligada das arestas adjacentes ao vértice.

Os seguintes métodos estão disponíveis nesta classe:

- `tVertex(int id)`: construtor da classe;
- `void InsertAdjacentEdge(tEdge* e)`: insere `e` na lista de arestas adjacentes;
- `void RemoveAdjacentEdge(tEdge* e)`: remove `e` da lista de arestas adjacentes;
- `int GetId() const`: retorna o id;
- `tEdgeList& GetAdjacentEdges()`: retorna uma referência para a lista de arestas adjacentes;
- `void SetFlag(int flag)`: altera a *flag*;
- `int GetFlag() const`: retorna a *flag*;
- `tVertex* GetNeighbour(tEdge* e) const`: retorna o vértice vizinho pela aresta `e`;
- `tEdge* GetEdge(tVertex* v)`: retorna a aresta, se existir, que liga ao vértice `v`;
- `tEdge* GetEdge(int vId)`: retorna a aresta, se existir, que liga ao vértice com id `vId`;
- `int Degree() const`: retorna seu grau.

5.3.2 A Classe `tEdge`

Esta classe representa uma aresta de um grafo qualquer. Uma aresta tem o seguintes atributos:

- `int Id`: identifica a aresta unicamente entre as arestas do grafo;
- `int Weight`: peso da aresta;
- `int Flag`: utilizado por vários algoritmos para vários propósitos;
- `tVertex* V1` e `tVertex* V2`: ponteiros para os vértices que são conectados pela aresta.

A aresta disponibiliza os seguintes métodos:

- `tEdge(int id, int w, tVertex* v1, tVertex* v2)`: construtor de uma aresta com id `id`, peso `w` e que conecta o vértice `v1` a `v2`;
- `int GetId() const`: retorna o seu id;
- `int GetWeight() const`: retorna o seu peso;
- `tVertex* GetV1() const`: retorna `V1`;
- `tVertex* GetV2() const`: retorna `V2`;
- `void SetFlag(int flag)`: altera o valor da *flag*;
- `int GetFlag() const`: retorna o valor da *flag*;
- `tVertex* GetNeighbour(tVertex* v) const`: retorna o vizinho de `v` por esta aresta.

5.3.3 A Classe `tGraph`

Esta classe representa um grafo qualquer. Os atributos desta classe são os seguintes:

- `tEdgeList Edges`: lista duplamente ligada das arestas;
- `tVertexList Vertex`: lista duplamente liga dos vértices.

A classe `tGraph` disponibiliza os seguintes métodos públicos (na especificação dos custos dos métodos abaixo considere n o número de vértices e m o número de arestas do grafo):

- `tGraph()`: construtor padrão;
- `~tGraph()`: destrutor;
- `tVertex* AddVertex(int id)`: adiciona um vértice com id `id`. Custo $O(1)$;
- `tEdge* AddEdge(int id, int w, tVertex* v1, tVertex* v2)`: adiciona uma aresta com id `id`, peso `w` e que conecta o vértice `v1` a `v2`. Este método já adiciona a nova aresta à lista de arestas adjacentes dos vértices. Custo $O(1)$;
- `tEdge* AddEdge(int id, int w, int vId1, int vId2)`: adiciona uma aresta com id `id`, peso `w` e que conecta os vértices com ids `vId1` e `vId2`. Este método já adiciona a nova aresta à lista de arestas adjacentes dos vértices. Custo $O(1)$;
- `void DelEdge(tEdge* e)`: remove a aresta `e` do grafo. Este método já remove a aresta da lista de arestas adjacentes dos vertices `e->GetV1()` e `e->GetV2()`. Custo $O(n + m)$;
- `void ClearEdges()`: remove todas as arestas do grafo. Custo $O(n + m)$;
- `tVertex* GetVertex(int id)`: retorna o vértice com id `id`. Custo $O(n)$;
- `tEdge* GetEdge(int id)`: retorna a aresta com id `id`. Custo $O(m)$;
- `tEdge* GetEdge(int v1Id, int v2Id)`: retorna a aresta que liga os vértices com ids `v1Id` e `v2Id`. Custo $O(m)$;
- `tEdge* GetEdge(tVertex* v1, tVertex* v2)`: retorna a aresta que liga os vértices `v1` e `v2`. Custo $O(n)$ (em um grafo completo);
- `tEdgeList& GetEdges()`: retorna uma referência para a lista de arestas;
- `tVertexList& GetVertex()`: retorna uma referência para a lista de vértices;
- `int GetNVertex() const`: retorna o número de vértices;
- `int GetNEdges() const`: retorna o número de arestas.

5.3.4 A Classe `tGraphTour`

Esta classe é uma especialização de `tGraph` que é capaz de gerar um **circuito hamiltoniano**. É a classe base para a implementação das heurísticas de construção. O único atributo desta classe (além dos atributos de `tGraph` é claro) é um ponteiro para um `tGraph` chamado `Tour` que é usado para armazenar o circuito hamiltoniano.

Observe que como `Tour` apontará para um objeto `tGraph` diferente, os objetos na lista de vértices e arestas também são diferentes do grafo original. Mas deve existir uma relação entre eles, pois para cada vértice do grafo original deve existir um vértice no circuito (`Tour`). Esta relação é garantida pelos ids dos vértices, ou seja, para cada vértice do grafo original existe um vértice com mesmo id em `Tour`. Isto é muito importante pois usamos esta característica na implementação de todas as heurísticas.

5.4 Módulo `nearestneighbour`

Este modulo contém a definição e implementação da classe `tNearestNeighbour` que é derivada de `tGraphTour` e tem a capacidade de construir um circuito hamiltoniano a partir de seus próprios vértices e arestas utilizando a heurística do **Vizinho Mais Próximo** apresentada na seção 2.2.

Não existe nenhum atributo adicional nesta classe. Existem dois métodos que são comentados a seguir.

O método `MakeTour` constrói o circuito hamiltoniano baseado em seus vértices e arestas. Partindo de um vértice aleatório e incluindo as arestas mais baratas adjacentes a este vértice. Ou seja, ele escolhe como próximo vértice corrente o vizinho mais próximo.

Utilizamos o método `NearestVertex` para encontrar o vértice mais próximo. O único parâmetro de entrada deste método é o vértice `v` a partir do qual desejamos encontrar o vizinho mais próximo `vNearest` que está conectado a `v` pela aresta `eNearest`. `vNearest` e `eNearest` são os parâmetros de saída. `vNearest` é encontrado percorrendo a lista de arestas adjacentes de `v`. Utilizamos as *flags* dos vértices para garantir que não fecharemos um ciclo antes de percorrer todos os vértices. No início do algoritmo alteramos as *flags* dos vértices para 0 e quando incluímos um vértice no circuito hamiltoniano então alteramos a *flag* dele para 1.

5.5 Módulo `greedy`

Neste módulo definimos e implementamos a classe `tGraphGreedy` que é derivada de `tGraphTour`. Esta classe tem a capacidade de criar um circuito hamiltoniano a partir

de seus vértices e arestas baseado na heurística **Gulosa por Peso de Aresta** apresentada na seção 2.3.

Como nesta heurística precisamos ir pegando as arestas em ordem não-decrescente de peso fazemos uma ordenação das arestas antes de iniciar a construção do circuito. Essa ordenação é realizada pelo método `SortEdges` que é descrito a seguir.

O método `SorteEdges` cria um vetor de ponteiros para `tEdge` e então usa a função `qsort` da ***Standard Template Library*** do Ansi C++ para ordenar o vetor em ordem não-decrescente de peso. Com isto o vetor é percorrido adicionando-se seus elementos na lista `SortedEdges`, que é uma lista duplamente ligada de ponteiros para `tEdge`.

Como já temos as arestas na ordem desejada só nos resta agora um mecanismo para verificar se uma aresta é válida ou não. Como visto na descrição da heurística *Greedy* são duas características que devemos verificar antes de incluir uma aresta no circuito. A aresta não pode fechar um ciclo (a menos que seja a última aresta) e não pode fazer com que algum dos vértices extremos fique com grau maior que 2. A segunda condição pode ser verificada facilmente observando o grau dos vértices extremos (deve ser menor ou igual a 1).

Para verificar se a aresta forma ciclo usamos a *flag* dos vértices para indicar em qual componente de `Tour` cada um dos vértices se encontra. Antes de iniciar a construção do circuito atribuímos à *flag* de cada vértice o seu id, ou seja, cada vértice está em uma componente distinta. Antes de incluir uma aresta no circuito verificamos se os vértices extremos estão em componentes diferentes (se suas *flags* são diferentes). Assim garantimos que a aresta não formará um ciclo.

Com isso podemos facilmente construir nosso circuito. Para isto vamos removendo as arestas de `SortedEdges` até encontrar uma aresta válida. Então incluímos esta aresta em `Tour` e realizamos a união das componentes dos vértices extremos utilizando o método `ComponentsUnion`.

O método `ComponentsUnion` recebe como parâmetros dois inteiros, `flag1` e `flag2`, que são as *flags* dos vértices que queremos unir na mesma componente. O que este método faz é percorrer todos os vértices do grafo alterando as *flags* dos vértices com *flag* igual à `flag2` para `flag1`.

5.6 Módulo `hub`

Este módulo possui a definição e implementação de uma estrutura, uma classe e algumas funções usadas na implementação da heurística de construção de **Clarke-Wright** apresentada na seção 2.4.

Esta heurística é gulosa pelas maiores economias proporcionadas pela troca das arestas

$u \leftrightarrow hub$ e $u \leftrightarrow hub$ pela aresta $u \leftrightarrow v$, onde *hub* é o vértice *hub* como definido na seção 2.4 e u e v são vértices não-*hub*. Vamos usar o termo *realizar o saving* para nos referir a esta troca.

Por isso criamos a estrutura `tSaving` que representa um *saving*. A definição desta estrutura é a seguinte:

```
struct tSaving
{
  int Saving;

  int E12Id;
  int E12Weight;

  tVertex* V1;
  tVertex* V2;
};
```

`V1` e `V2` são os envolvidos no *saving* e `Saving` é o custo ganho pela realização do *saving*. Os atributos `E12Id` e `E12Weight` são, respectivamente, o id e o peso da aresta $V1 \leftrightarrow V2$ no grafo original e são armazenados aqui para não ser necessário pesquisá-los quando realizarmos o *saving*.

Na implementação desta heurística precisamos realizar a seguinte sequência de operações: criar o multi-grafo (grafo com um vértice *hub* que está ligado por duas arestas a cada um dos outros vértices), computar e ordenar todos os *savings* e então realizar os *savings* até o multi-grafo se tornar um circuito hamiltoniano.

O método `MakeMultiGraph` constrói o multi-grafo inicial utilizado na heurística *Clarke-Wright*. Ele escolhe aleatoriamente um vértice do grafo para ser o *hub* e preenche o atributo `Hub` com um ponteiro para este vértice. Então cria duas arestas de cada vértice (não-*hub*) do grafo para o vértice *hub*.

O método `MakeSavings` aloca o vetor `Savings` e o preenche com as informações referentes ao *saving* de cada par de vértices não-*hub* do multi-grafo e então o ordena.

A partir de agora basta percorrer o vetor `Savings` realizando as devidas trocas até o circuito se tornar hamiltoniano. Mas como discutido anteriormente é necessário realizar uma verificação para garantir que o *saving* é válido.

Para verificar que a troca das arestas não implicará na formação de um ciclo, que não passa pelo vértice *hub*, utilizamos uma idéia semelhante à apresentada na seção 5.5anterior para controle de componentes. Dizemos que dois vértices quaisquer u e v estão na mesma componente se existe um caminho de u para v que não passa pelo vértice *hub*. Então antes de iniciar a construção do circuito atribuímos a *Flag* de cada vértice o seu Id. Assim

dizemos que cada vértice está em uma componente distinta. Se dois vértices estiverem em componentes distintas então a troca de suas arestas não implicará na formação de um ciclo que não passa pelo vértice *hub*. E quando realizamos um *saving* utilizamos o método `ComponentsUnion` para atribuir um mesmo valor para as *Flags* dos vértices das duas componentes.

Mas ainda é necessário verificar se os vértices estão ligados, pelo menos por uma aresta, ao vértice *hub*. Esta operação é direta utilizando a lista de arestas adjacentes dos vértices.

5.7 Módulo `doublemst`

Neste módulo definimos e implementamos a classe `tGraphDoubleMST` que é uma especialização de `tGraphTour` que constrói um circuito hamiltoniano utilizando a heurística *Duplicação da Árvore Geradora Mínima* discutida na seção 2.5. Esta classe possui dois atributos adicionais: `MST` que é um grafo armazenará a AGM do grafo completo e `SortedEdges` que é uma lista de ponteiros para `tEdges` usada para armazenar as arestas do grafo completo em ordem não-decrescente de custo.

O método `MakeMST` de `tGraphDoubleMST` constrói a sua AGM, armazenando-a em `MST`, utilizando o algoritmo de Kruskal. Implementamos o algoritmo de Kruskal da seguinte maneira: antes de iniciar a construção da AGM chamamos o método `SortEdges` que aloca a lista `SortedEdges` e a preenche de acordo com a definição acima. A partir disto as arestas deste vetor são incluídas em `MST`, desde que não impliquem na formação de um ciclo. A idéia para verificar se uma aresta pode ou não ser incluída em `MST` é idêntica a apresentada na seção 5.5 a menos da verificação se algum dos vértices extremos possui grau maior que 2, que neste caso não é necessária.

Agora que já temos a AGM precisamos duplicar suas arestas, transformado-a em um circuito euleriano. O propósito da duplicação é possibilitar a realização de um passeio pelo circuito visitando todos os vértices do grafo e adicionar os vértices não repetidos (e as respectivas arestas) em `Tour`. E para isto não é necessário duplicar realmente as arestas de `MST`. Apenas usamos a *flag* de cada aresta de `MST` para indicar quantas vezes já passamos pela aresta durante o passeio no circuito. Como não utilizamos as *flags* das arestas de `MST` em nenhuma parte do algoritmo então não precisamos iniciá-las com 0 (por padrão uma aresta tem *flag* igual a 0).

Precisamos também controlar quais vértices já foram incluídos em `Tour`. Como incluímos todos os vértices do grafo em `Tour` no início do algoritmo então utilizamos as *flags* dos vértices para indicar se já incluímos (*flag* igual a 1) ou ainda não incluímos (*flag* igual a 0) ele na resposta.

Implementamos um método chamado `NextVertex` que recebe como parâmetro um vértice v de `MST` e retorna o próximo vértice vn (e a aresta (v, vn)) no passeio pelo

circuito euleriano. A implementação deste método é bem simples, basta procurarmos por uma aresta adjacente a v com *flag* igual a 0, se não existir então procuramos por uma com *flag* igual a 1. Esta aresta sempre existirá até que todos os vértices sejam visitados.

Com tudo isto fica simples implementar o método `MakeTour`. O que fazemos é o seguinte: escolhemos um vértice aleatório v de `MST` e alteramos sua *flag* para 1. Então executamos um laço que a cada iteração encontra o próximo vértice u, seguindo o circuito euleriano, que ainda não foi incluído na resposta (para isto utilizamos o método `NextVertex`). Então incluímos a aresta (u, v) em `Tour`, incrementamos sua *flag* e passamos para a próxima iteração do laço. Até que todos os vértices tenham sido incluídos na resposta. No final ainda nos resta adicionar a aresta que liga o primeiro e o último vértices incluídos na resposta.

5.8 Módulo `tourlist`

Neste módulo implementamos uma lista que representa um circuito em um grafo. Usamos esta lista nos algoritmos de **busca local** para percorrer o circuito trocando as arestas que proporcionam alguma economia no custo total. A classe `tTourList` é uma lista circular de nós mistos: nós vértice, nós aresta e nó cabeça. O nó cabeça é único e é a cabeça da lista. Os nós vértice e aresta obviamente representam, respectivamente, os vértices e as arestas do circuito.

5.8.1 A Classe `tTourNode`

Esta classe representa um nó da lista. Nós aresta e o nó cabeça são objetos desta classe. Esta classe possui um atributo que indica o seu tipo (aresta, vértice ou cabeça) e outros dois atributos que são ponteiros para o próximo (`Next`) nó e para o nó anterior (`Prev`).

O método `Reverve` troca os valores de `Next` e `Prev`, ou seja, inverte o sentido do nó na lista.

5.8.2 A Classe `tTourVertexNode`

Esta classe é uma especialização da classe `tTourNode` que representa um nó do tipo vértice. Esta classe possui um atributo adicional que é um ponteiro para o objeto `tVertex` que representa o vértice deste nó.

5.8.3 A Classe `tTourEdgeIterator`

Esta classe implementa um iterador para percorrer as **arestas da lista**. Esta classe possui métodos que retornam os vértices extremos da aresta e um método (sobrecarga do operador de incremento, `++`) que passa para a próxima aresta da lista.

5.8.4 A Classe `tTourList`

Esta classe representa a lista do circuito propriamente dita. Existem métodos para inserir nós aresta e nós vértice. E ainda métodos para trocar a ordem dos vértices da lista de acordo com as heurísticas 2-Opt (`Exchange2`) e 3-Opt (`Exchange3A` e `Exchange3B`).

O único atributo desta classe é um ponteiro para nó cabeça da lista.

5.9 Módulo `localsearch`

Neste módulo definimos e implementamos as classes para as heurística de **busca local**. Definimos uma classe `tLocalSearch` que é a base para os dois algoritmos de busca local. Esta classe recebe como parâmetro no seu construtor um ponteiro para `tGraphTour` que deve ser um grafo completo que já possui um circuito hamiltoniano construído e atribui este ponteiro ao atributo `Complete`. O circuito deste grafo é atribuído ao atributo `Tour`.

Como as duas heurísticas realizam várias comparações entre pesos de arestas para decidir quais arestas serão trocadas então antes de qualquer processamento construímos uma matriz (`WeightMatrix`) com os pesos de todas as arestas do grafo completo. Isto é realzado pelo método `MakeWeightMatrix`.

O método `TourToList` constrói uma `tTourList` que representa o circuito `Tour`. O método `ListToTour` realiza o inverso, ou seja, a partir da lista altera o circuito `Tour`.

O método `CanImprove` é implementado na classe `tLocalSearch2Opt` e `tLocalSearch3Opt` e, em ambas, é responsável por testar se a troca das arestas que conectam determinados vértices (quatro ou seis) implicará na dimunuição do custo do circuito.

O método `ImproveTour` é sobrescrito pelas duas classes e realiza a otimização do circuito `Tour`. A implementação deste método é simples e direta nas duas classes. Em `tLocalSearch2Opt` utilizamos dois iteradores de arestas para percorrer a lista do circuito e trocar, as que são aprovadas pelo método `CanImprove`, utilizando o método `Exchange2` de `tTourList`. Em `tLocalSearch3Opt` a implementação é semelhante mas com três iteradores de arestas e utilizando os métodos `Exchange3A` e `Exchange3B` de `tTourList`.

5.10 Módulo grasp

Neste módulo definimos e implementamos a classe `tGraphGRASP` que é uma especialização da classe `tGraphGreedy`. Utilizamos esta modelagem pois a parte de construção da nossa heurística GRASP é exatamente a heurística Gulosa.

Esta classe possui os seguintes atributos adicionais:

- `unsigned Seed`: semente para gerador de números randômicos;
- `double Alfa`: fator de qualidade da heurística;
- `int MaxIterations`: especifica o número de iterações do algoritmo;
- `int RCLCutWeight`: armazena o peso limite para as arestas que participarão da RCL;
- `int BestTourSize`: armazena o custo do melhor circuito hamiltoniano encontrado durante as iterações do algoritmo;
- `tLocalSearch* Search`: ponteiro para o objeto que realizará a busca local após cada fase de construção do algoritmo.

Os seguintes métodos públicos são disponibilizados nesta classe:

- `void SetAlfa(double alfa)`: altera o fator de qualidade;
- `void SetSeed(unsigned seed)`: altera a semente para o gerador de número randômicos;
- `void SetMaxIterations(int maxIterations)`: altera o número de iterações da heurística;
- `void SetLocalSearch(tLocalSearch* localSearch)`: altera o ponteiro para o objeto que realizará a busca local após cada fase de construção da heurística.

O parâmetro passado para o método `SetLocalSearch` deve ser um objeto que foi criado utilizando o próprio grafo com suas arestas já criadas. Por exemplo, se você criar este objeto passando como parâmetro no construtor um `tGraphGRASP` g então isto deve ser feito após a chamada do método `ir.FillGraph(g)` onde `ir` é um `tInstanceReader`.

Existem, na classe `tGraphGRASP`, os seguintes métodos protegidos, que são utilizados direta ou indiretamente pelo método `MakeTour`:

- `int CheapestWeight()`: retorna o peso do menor elemento da lista `SortedEdges`;

- `int MostExpensiveWeight()`: retorna o peso do maior elemento da lista `SortedEdges`.
- `int CalcRCLLength()`: calcula o tamanho da lista `SortedEdges`;
- `tEdge* GetRCLEdge(int rank)`: retorna o *rank*-ésimo elemento da lista SortedEdges;
- `void Construct()`: realiza a fase de construção da heurística;
- `void LocalSearch()`: realiza a fase de busca local da heurística. A implementação deste método contém apenas um teste verificando se `LocalSearch` é nulo e então, caso contrário, chama `LocalSearch->OptimizeTour()`;

O método `CheapestWeight` varre a lista `SortedEdges`, do início para o final, removendo as arestas até encontrar uma aresta que possa ser incluída na resposta, então retorna o peso desta aresta. `MostExpensiveWeight` realiza o mesmo mas em ordem inversa, ou seja, do final para o início da lista.

`CalcRCLLength` calcula o valor de `RCLCutWeight` combinando os valores retornados pelos métodos `CheapestWeight` e `MostExpensiveWeight` com o fator de qualidade, como discutido na seção 4.4. E então percorre `SortedEdges` a partir do início até encontrar uma aresta com custo maior do que o valor retornado por `MostExpensiveWeight`, contando as arestas válidas e removendo as inválidas.

O método `Construct` é idêntico ao método `MakeTour` de `tGraphGreedy` a menos de onde encontramos a aresta a ser incluída na respota. Em `tGraphGreedy` utilizamos o método `CheapestEdge` que retorna a aresta de menor custo e o método `ValidEdge` para determinar se a aresta é válida ou não. Aqui encontramos a próxima aresta a ser incluída na resposta fazendo uma chamada ao método `GetRCLEdge`, passando como parâmetro um valor no intervalo $[0, CalcRCLLength()]$.

Com isto a implementação do método `MakeTour` é direta. Basta realizarmos um laço com `MaxIterations` iterações e em cada iteração chamar `Construct` e depois `LocalSearch` e armazenar a melhor solução encontrada, que será a solução final.

Capítulo 6

Resultados Empíricos

6.1 Introdução

Neste capítulo vamos apresentar os resultados obtidos a partir dos testes realizados utilizando o nosso programa. Para isto utilizamos um PC com processador AMD K6-II 333MHz com 128MB de memória RAM rodando o sistema operacional Windows 98. As instâncias utilizadas foram obtidas da TSPLIB.

6.2 Avaliação de Tempo de Processamento

Nesta seção vamos apresentar os resultados comparativos em relação ao tempo de processamento consumido pelas heurísticas. Para cada instância testada executamos nossos algoritmos três vezes e calculamos a média.

As instâncias utilizadas nos testes das heurísticas de construção e de busca local foram as seguintes:

Instância	**Tamanho**
kroA200	200
rd400	400
u574	574
rat783	783
pr1002	1002
pcb1173	1173
fl1400	1400

Executamos a heurística GRASP com fator de qualidade α igual a 0.1 e com apenas

1 iteração, utilizando as seguintes instâncias:

Instância	**Tamanho**
eil101	101
kroA200	200
pr299	299
rd400	400
d493	493
rat575	575

6.2.1 Heurísticas de Construção

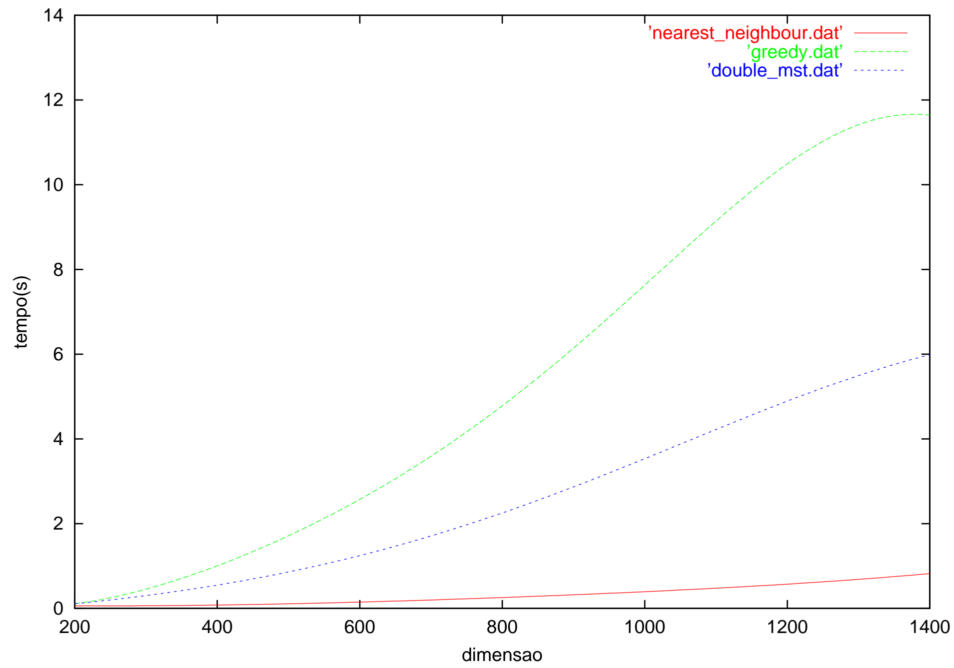

Figura 6.1: Comparação entre as três heurísticas mais rápidas.

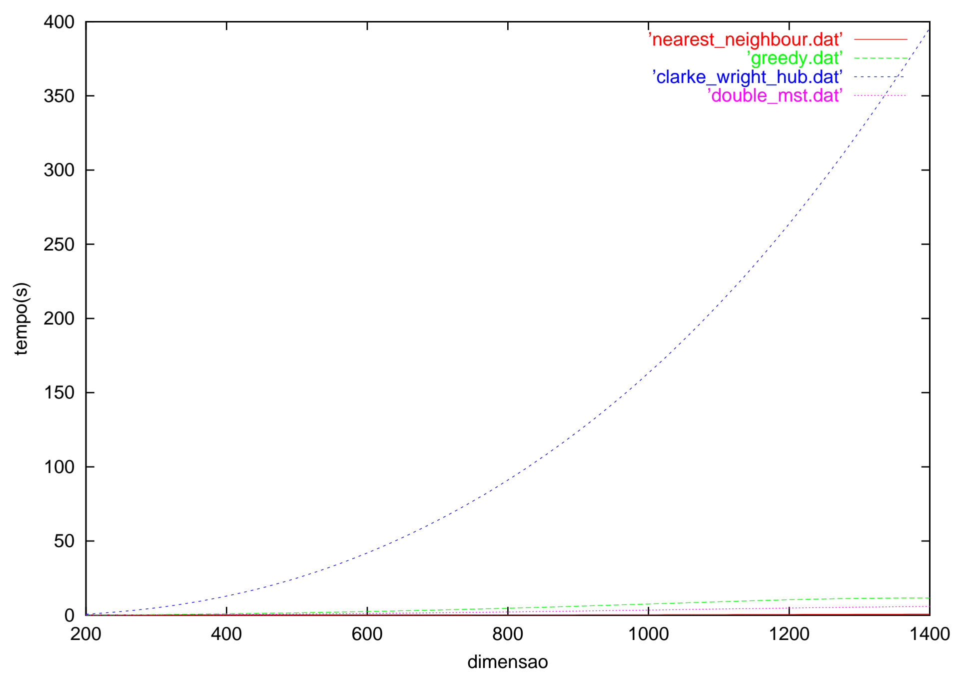

Figura 6.2: Comparação da heurística Clarke-Wright com as outras três.

6.2.2 Heurísticas de Busca Local

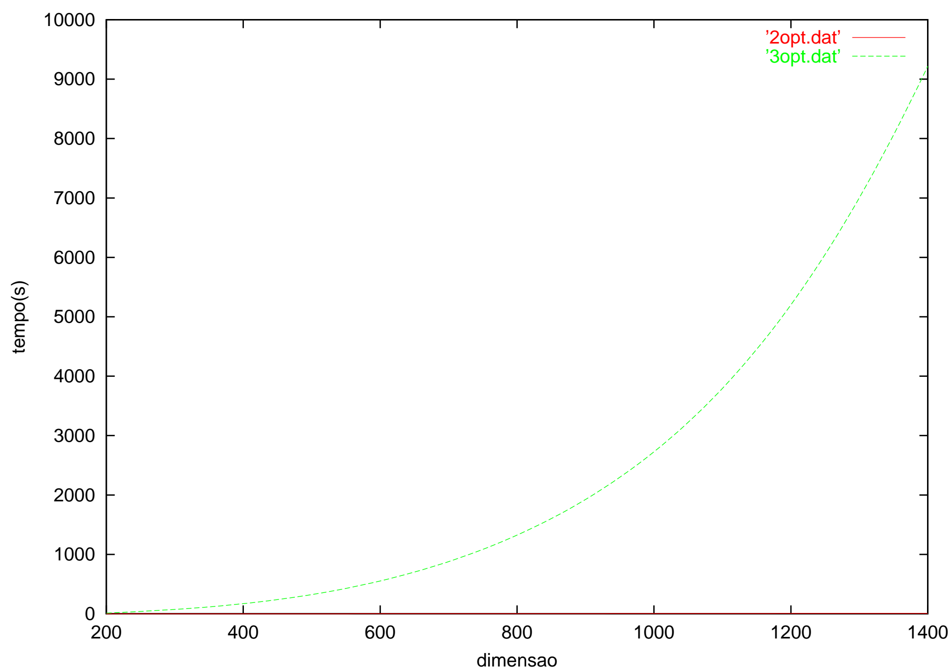

Figura 6.3: Comparação entre as heurísticas 2-Opt e 3-Opt.

6.2.3 Heurística GRASP

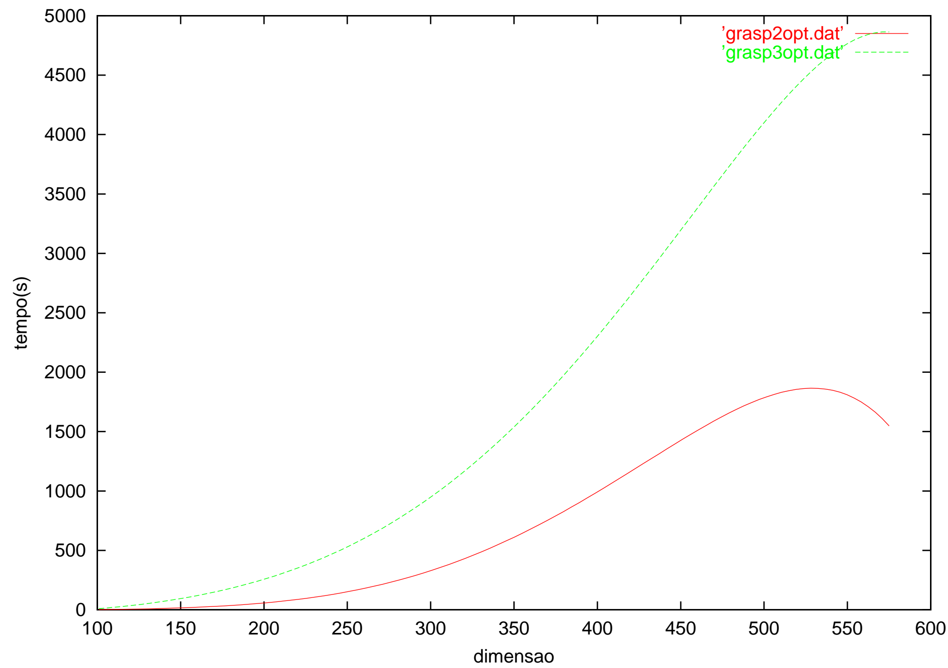

Figura 6.4: Comparação entre a heurística GRASP utilizando busca local 2-Opt e 3-Opt.

6.3 Avaliação da Qualidade da Solução

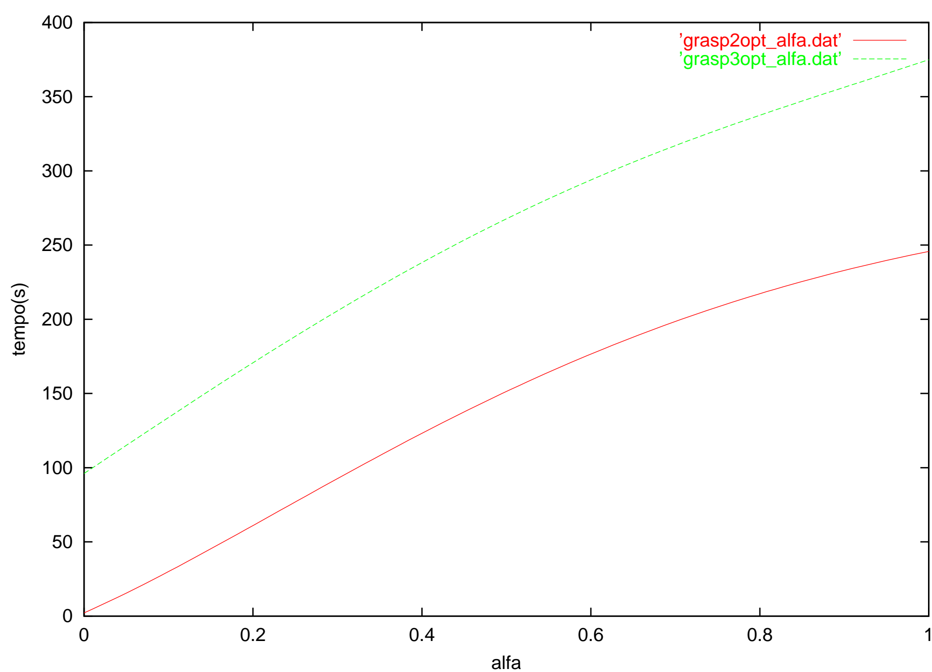

Figura 6.5: Comparação entre a heurística GRASP variando o fator de qualidade utilizando a instância kroA200.

Capítulo 7

Conclusão

Neste trabalho apresentamos uma série de heurísticas para o Problema do Caixeiro Viajante. As que proporcionaram os melhores resultados utilizam, de alguma forma, busca local. Em compensação estas heurísticas consomem muito tempo de processamento e por isso devemos nos preocupar durante a implementação, para minimizá-lo. Acreditamos que uma atenção especial deve ser dada à heurística GRASP, que apesar de consumir um tempo maior de processamento, encontra as melhores soluções.

Finalizando, gostaríamos de ressaltar a liberdade na criação de heurísticas para o TSP já que os resultados práticos superam as expectativas geradas pelas análises teóricas.

Bibliografia

[GJ79] GAREY, M.R.; JOHNSON, D.S. *Strong NP-completeness results: Motivations, examples and implications.* Journal of the ACM, 25:499-508, 1978.

[BM76] BONDY, J.A.; MURTY, U.S.R. *Graph Theory with Applications.* Macmillan/Elsevier, 1976.

[CLR92] CORMEN, T.H.; LEISERSON, C.E.; RIVEST, R.L. *Introduction to Algorithms.* MIT Press, 1992.

[LP86] LOVÁSZ, L. *Semidefinitive programs and combinatorial optimization.* In Brazilian Summer School on Combinatories and Algorithms. CIMPA, 2001.

[CRO58] CROES, G.A. *A method for solving travelling salesman problems.* Operations Res. 6, 1958.

[FLO56] FLOOD, M.M. *The travelling-salesman problem* Operations Res. 4, 1956.

[BOC58] BOCK, F. *An algorithm for solving "travelling-salesman" and related network optimization problems*, 14th ORSA National Meeting, 1958.

[PS78] PAPADIMITRIOU, C.H.; STEIGLITZ, H. *Some examples of difficult travelling salesman problem.* Operations Res. 26, 1978.

[LUE76] LUEKER, G. manuscript, Princeton University, 1976.

[CKT94] CHANDRA, B.; KARLOFF, H.; TOVEY, C. *New results on the old k-opt algorithm for the TSP*, in Proceedings 5th ACM-SIAM Symp. on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, 1994.

[JPY88] JOHNSON, D. S.; PAPADIMITRIOU, C. H.; YANNAKAKIS, M. *How easy is local search?* J. Comput. System Sci. 37. 1988

[REI91] REINELT, G. *TSPLIB, A travelling salesman problem library.* ORSA J. Comput. 3, 1991.