FACULDADE DE TECNOLOGIA DE SÃO PAULO

Guilherme Bohnstedt

Técnica *Branch and Bound* com heurísticas para resolver o Problema do Caixeiro Viajante

SÃO PAULO

2017

FACULDADE DE TECNOLOGIA DE SÃO PAULO

GUILHERME BOHNSTEDT

Técnica *Branch and Bound* com heurísticas para resolver o Problema do Caixeiro Viajante

Trabalho submetido como exigência parcial
para a obtenção do Grau de Tecnólogo em
Análise e Desenvolvimento de Sistemas
Orientador: Prof. Dr. Silvio do Lago Pereira

SÃO PAULO
2017

FACULDADE DE TECNOLOGIA DE SÃO PAULO

GUILHERME BOHNSTEDT

Algoritmo de Busca em Profundidade aplicando Heurísticas para o problema do Caixeiro Viajante

Trabalho submetido como exigência parcial para a obtenção do Grau de Tecnólogo em Análise e Desenvolvimento de Sistemas.

Parecer do Professor Orientador:_____

_____

_____

Conceito/Nota Final: _____

Orientador: Prof. Dr. Silvio do Lago Pereira

SÃO PAULO, ____ de _____ de 2017

Agradecimentos

Agradeço a Jeová, meu Deus, o Autor da Bíblia de onde extraí conselhos sábios que me ajudaram nessa etapa da minha vida. Por me mostrar que Ele jamais desampara quem o procura e dá prioridade ao Seu Reino.

Agradeço aos meus pais, Roberto e Rose, pelo apoio e sustento que me deram desde minha entrada na Faculdade. Por me darem conselhos e fazerem sacrifícios para que eu pudesse levar essa etapa da vida de maneira mais suave.

Agradeço à minha namorada, Beatriz, que me ajudou a não desistir e me ajudou a ter o foco correto do que eu precisava concluir.

Agradeço ao meu orientador, Professor Silvio, que me deu o suporte necessário para que eu abordasse o assunto deste trabalho.

Agradeço aos amigos e colegas que estiveram comigo durante os anos de graduação. Pelos momentos que tivemos juntos e pela troca de conhecimento.

Agradeço a você, leitor, que agora participa conosco nessa jornada científica e compreender que ainda há muito o que estudar.

Resumo

Desde os primórdios da computação os algoritmos de busca tem sido base para muitos estudos. Durante muitas décadas, centenas de maneiras para solucionar esses problemas foram criadas. Um dos mais conhecidos é o Problema do Caixeiro Viajante. A modelagem desse problema permite resolver outras centenas de problemas. Neste trabalho será apresentado uma abordagem exata para solucionar o problema do Caixeiro Viajante através da técnica *Branch and Bound*. Serão propostas maneiras de minimizar o tempo de execução das buscas por meio de heurísticas analisando a eficiência de sua aplicação.

Palavras-chaves: Caixeiro viajante, simétrico, algoritmos de busca, poda, otimização, heurísticas, *Branch and Bound.*

Abstract

Since the early days of computing search algorithms have been the basis for many studies. For many decades, hundreds of ways to solve these problems have been created. One of the best known is the Traveling Salesman Problem. Modeling this problem allow solve hundreds of other problems. In this paper an exact approach will be presented to solve the problem of the Traveling Salesman through the Branch and Bound technique. It will be proposed ways to minimize the execution time of the searches through heuristics, analyzing the efficiency of their application.

Keywords: Traveler salesman, symmetric, search algorithms, pruning, optimization, heuristics, Branch and Bound.

SUMÁRIO

1. INTRODUÇÃO 9

1.1. Objetivo 10

1.2. Organização 10

2. FUNDAMENTOS TEÓRICOS 11

2.1. Tipos de Problemas 11

2.1.1. Problemas de Decisão 11

2.1.2. Problemas de Busca 11

2.2. Classe de Complexidade 12

2.2.3. Classe P 14

2.2.4. Classe NP 14

2.2.5. Classe NP-Completo 15

2.3. Grafos 16

2.3.1. Grafo Orientado 16

2.3.2. Grafo Não Orientado 17

2.3.3. Adjacência 17

2.3.4. Caminho em um grafo 17

2.3.5. Tipos de grafos 18

2.4. Heurística 18

2.4.1. Metaheurística 18

2.5. O Problema do Caixeiro Viajante 19

2.5.1. Aplicabilidade 20

2.5.2. Algoritmos para o TSP 22

2.5.2.1. Abordagem de soluções exatas 23

2.5.2.2. Abordagem de soluções aproximadas 24

2.5.2.2.1. Algoritmos para construção de circuitos 24

2.5.2.2.2. Algoritmos para melhoria de circuitos 25

3. ALGORITMOS IMPLEMENTADOS 27

3.1. Representação de grafos 27

3.1.1. Estrutura de um grafo para TSP 28

3.2. Técnica Branch & Bound 31

3.2.1. Função limitante 32

3.2.2. Estratégia de seleção 32

3.2.3. Regra de ramificação 36

3.2.4. Produção da solução inicial 36

3.3. Heurísticas 38

3.3.1. Árvore geradora mínima 38

3.3.2. Intersecção de arestas 42

4. RESULTADOS EXPERIMENTAIS 48

4.1. Tipos de instâncias 48

4.2. Ambiente de testes 48

4.3. Resultados 49

5. CONCLUSÕES 51

5.1. Considerações **51**

5.2. Trabalhos futuros **51**

5.3. Considerações pessoais **52**

REFERÊNCIA BIBLIOGRÁFICA **53**

APÊNDICE A - Código fonte **61**

1. INTRODUÇÃO

O problema do Caixeiro Viajante (*Travelling Salesman Problem* - TSP) é um problema clássico de otimização combinatória, que consiste em encontrar um caminho de comprimento mínimo para visitar uma série de cidades, passando uma única vez em cada uma delas e retornado à cidade de partida. O problema foi estudado por diversos pesquisadores, entre eles, Menger, Whitney, Mahalanobis, Jessen e Flood (SCHRIJVER, 2005). Este problema tem atraído ainda mais intensamente a pesquisa de novas soluções, devido a sua grande quantidade de aplicações práticas, relação com outros problemas e grande dificuldade de encontrar uma solução exata em um tempo computacional aceitável. Algumas aplicações práticas são sequenciamento das operações de máquinas em manufatura, otimização de perfuração de placas de circuitos impressos e a maioria dos problemas de roteamento de veículos (NASCIMENTO, 2004).

O objetivo dos problemas de otimização é maximizar ou minimizar uma função objetivo dado um conjunto de variáveis de decisão atendendo a um determinado conjunto de restrições (ROTHLAUF, 2011). Uma forma imediata de resolver esse problema é enumerar todas as possíveis soluções e analisar uma a uma até encontrar o conjunto ou a solução que atende à necessidade. No entanto, torna-se inviável quando a quantidade de possíveis soluções é muito grande. Um supercomputador poderia demorar até milhares de horas para processar todas as soluções.

Existem diversos métodos que tornam o processo de busca pela solução ótima. Os algoritmos de *Branch & Bound* (LAND e DOIG, 1960) melhoram muito o tempo de busca, mas não são tão eficientes quando o problema é muito complexo, ou seja, o número de cidades é muito grande (CONWAY, 2003). Algoritmos probabilísticos determinam rapidamente uma solução, mas nem sempre é a ótima (Vide Seção 2.6.2).

Há dois tipos de problemas do Caixeiro Viajante: o simétrico e assimétrico. A diferenciação está intimamente ligada ao sentido que podemos percorrer uma determinada ligação entre duas cidades (ou aresta entre dois vértices de um grafo representando o mapa de cidades). Caso importe o sentido, dizemos que este é um

TSP assimétrico, ou seja, a ligação é unidirecional. Caso o sentido não importa, dizemos que este é um TSP simétrico, ou seja, a ligação é bidirecional. Neste trabalho focaremos no TSP simétrico.

Neste trabalho, buscamos desenvolver um algoritmo preciso, ou seja, que garanta a melhor solução e que, ao mesmo tempo, apresente um custo computacional aceitável para instâncias razoavelmente grandes, mas de tamanho ainda bastante limitado.

1.1. Objetivo

O principal objetivo deste trabalho é estudar a implementação da técnica de *Branch & Bound* de forma mais eficiente, em termos de custo computacional, para melhoria do processo de busca de soluções para o problema do Caixeiro Viajante.

Especificamente, compreender os conceitos relacionados ao problema e diagnosticar as dificuldades em encontrar uma solução. Depois, analisar algoritmos existentes e propor uma solução melhorada da técnica *Branch & Bound*. Finalmente, apresentar os resultados e, por fim, analisar se o processo foi melhorado.

1.2. Organização

O restante desta monografia é organizado da seguinte forma:

O Capítulo 2 introduz os fundamentos teóricos necessários para a compreensão do algoritmo implementado e analisado neste trabalho.

O Capítulo 3 discute os detalhes de implementação do algoritmo proposto neste trabalho para solução do problema do Caixeiro Viajante.

O Capítulo 4 apresenta os resultados empíricos, obtidos com a execução do algoritmo implementado, para diferentes instâncias do problema do Caixeiro Viajante.

O Capítulo 5 apresenta as conclusões finais do trabalho, com base na análise dos resultados empíricos obtidos com o algoritmo implementado.

2. FUNDAMENTOS TEÓRICOS

2.1. Tipos de Problemas

Para entender melhor o objetivo deste trabalho e qual é a sua dificuldade, vamos abordar a complexidade computacional intrínseca de problemas.

Existem dois tipos fundamentais de problemas computacionais: os problemas de decisão e os problemas de busca.

2.1.1. Problemas de Decisão

Os problemas de decisão surgem apenas se uma pessoa ou um grupo de pessoas - ambos referidos como "o ator" na metodologia de decisão - possui uma idéia consciente de um estado desejável. Este estado-alvo é quase sempre diferente da situação atual ou pode se tornar diferente no futuro. O ator deve, portanto, agir: deve tentar minimizar a discrepância entre a situação atual e a situação-alvo (SANDERS, 1999, p. 7, tradução nossa).

Um problema de decisão consiste na especificação de um subconjunto de possíveis instâncias. Dada uma instância, determinar se a instância está no conjunto especificado, ou seja, retornar uma resposta *verdadeiro* ou *falso*. Por exemplo, considerando o problema onde é dado um número natural e é perguntado para determinar se o número é ou não é primo (GOLDREICH, 2006). A definição formal de solução de um problema de decisão é:

Sendo $S \subseteq \{0,1\}^*$. *Uma função* $f : \{0,1\}^* \to \{0,1\}$ *soluciona um problema de decisão de* S *(ou decide ser um membro em* S*) se para todo* x *considera-se que* $f(x) = 1$ *se e somente se* $x \in S$ (GOLDREICH, 2006, v. 2, p19, tradução nossa).

Um problema de decisão pode ter diversos níveis de complexidade. O número de critérios de decisão distingue os diferentes tipos de problema (RÜHLI, 1988, p. 186).

2.1.2. Problemas de Busca

No cotidiano, é muito comum nos depararmos com problemas de busca. Apesar de não ser a maioria, os problemas de busca costumam ser mais complicados de serem resolvidos. Por exemplo, suponhamos que você tenha várias

atividades para serem feitas. Na sua agenda, é necessário organizar essas atividades de forma que você consiga realizar todas. No entanto, cada atividade tem um horário, dia da semana e pessoas que precisam estar presentes para concluir a atividade. Dependendo do número de atividades, da quantidade de horas disponível que você possui e da quantidade de pessoas envolvidas isso se torna muito complexo de ser organizado. Outro exemplo: suponhamos que você comece a fazer uma dieta. Nesta dieta são requisitados alguns vitaminas e minerais. No entanto, não existe um único produto que possui todas essas exigências. Então será necessário fazer grupos de produtos que supram essas necessidades. O problema está em organizar produtos com todas as vitaminas e minerais necessários e então formular uma receita para, no final, obter o resultado esperado.

O problema de busca consiste na especificação de um conjunto de soluções válidas (possivelmente até um conjunto vazio) para cada possível instância. Isto é, dada uma instância, encontrar uma solução correspondente (ou determinar que essa solução não existe). Por exemplo, considerando o problema onde é dado um sistema de equações e é perguntado para encontrar uma solução válida (GOLDREICH, 2006). A definição formal de solução de um problema de busca é:

Sendo $R \subseteq \{0,1\}^* x \{0,1\}^*$. *Uma função* $f : \{0,1\}^* \to \{0,1\} \cup \{\perp\}$ *soluciona um problema de busca de* R *se para todo* x *considera-se que* $(x, f(x)) \in R$ *se e somente se* $R(x) \equiv \{y : (x,y) \in R\}$ *não for vazio* (GOLDREICH, 2006, v. 2, p. 18, tradução nossa).

Dentro dos problemas de busca existem os problemas de otimização, quando o objetivo é não apenas procurar um conjunto de soluções, mas que este seja ótimo.

2.2. Classe de Complexidade

As classes de complexidade de problemas estão relacionadas com a complexidade temporal de um problema, assumindo que esse problema é resolvido por um algoritmo. Intuitivamente, a complexidade temporal de tal problema é definida como a complexidade temporal do algoritmo mais rápido que o resolve (GOLDREICH, 2006, v. 2, p.32, tradução nossa). A complexidade temporal (ou complexidade de tempo) é o custo de tempo necessário para processar determinado algoritmo. De forma simplista, o tempo é calculado pela quantidade de passos

necessários para processar uma determinada entrada para o problema. Isso depende de inúmeros parâmetros da entrada em particular. Por exemplo, se a entrada for um grafo, o número de passos pode depender do número de nós, do número de arestas, e o grau máximo do grafo, ou uma combinação de todos esses fatores (SIPSER, 2006, v. 2, p. 248). Por exemplo, um algoritmo pode ter uma função temporal $f(n) = 2n^2 + 2n + 6$, sendo n um parâmetro que varia de acordo com a entrada. Como a função temporal de um algoritmo é complexa de ser calculada, por convenção se faz uma análise assintótica. A análise assintótica nos permite estimar a complexidade temporal de um algoritmo, e leva em consideração que os parâmetros de entrada tendem ao infinito. A notação usada é chamada de *big-O notation* e é simbolizada por O ("O" maiúsculo). Essa notação faz a relação $f(n) = O(n)$ e considera apenas o termo de ordem maior. Nela são desconsiderados todos os termos menores e coeficientes da função. Exemplificando com o caso já citado acima: O algoritmo que tem função $f(n) = 2n^2 + 2n + 6$ na notação é apenas representado por $O(n^2)$. Essa é uma estimativa para todos as possíveis entradas para o algoritmo (SIPSER, 2006, v. 2, p. 248).

A Tabela 1 mostra a diferença de tempo de processamento entre diversas complexidades para uma determinada entrada de um algoritmo. Nesse sistema computacional cada operação básica, ou seja caso passo do algoritmo, demora um microsegundo.

Tabela 1 . Tempo de processamento por função de complexidade

	$n = 100$	$n = 1000$	$n = 1000000$
$lg\ n$	2 microsegundos	3 microsegundos	6 microsegundos
$\sqrt{n}$	10 microsegundos	$10\sqrt{10}$ microsegundos	1 milisegundo
n	100 microsegundos	1 milisegundo	1 segundo
n^2	10 milisegundos	1 segundo	~11 dias
n^3	1 segundo	~16 minutos	~10 séculos
2^n	$\sim 4x10^{13}$ milênios	$\sim 4x10^{285}$ milênios	$\sim 5x10^{301036}$ milênios
$n!$	$\sim 3x10^{142}$ milênios	$\sim 1x10^{2552}$ milênios	$\sim 2,6x10^{5565693}$ milênios

Fonte: Elaborado pelo autor

2.2.3. Classe P

Um problema pertence à classe P (*Deterministic Polynomial time*) quando possui um algoritmo que determina uma solução em tempo polinomial, ou seja, que ao processar uma instância de tamanho *n* demora $O(n^k)$ unidades de tempo, sendo *k* uma constante qualquer. Esses são tipos de problemas fáceis de serem resolvidos e são considerados como problemas tratáveis.

Para a maioria dos propósitos, um algoritmo que resolve um problema em tempo polinomial é considerado rápido. Vamos analisar uma diferenciação entre funções polinomiais e funções exponenciais.

Note a drástica diferença de crescimento entre um polinômio bem usual como n^3 e uma função exponencial, também bem comum, como 2^n. Por exemplo, dado $n = 1000$, uma entrada de tamanho razoável para um algoritmo. Neste caso, a função polinomial terá valor de 1 bilhão. Apesar de ser um número muito grande, é relativamente fácil de um computador administrar isso. Em contrapartida, a função exponencial admite um valor muito maior que o número de átomos do universo (SIPSER, 2006, v. 2, p.256).

2.2.4. Classe NP

Um problema pertence à classe NP (*Non-Deterministic Polynomial time*) quando possui um algoritmo que decide em tempo polinomial se uma determinada possível solução pertence ao conjunto de soluções do problema, ou seja, dada uma instância *L* do problema *X,* verificar em tempo polinomial se *S* é solução para *L.*

Em certos problemas, incluindo alguns problemas bem interessantes ou bem úteis, ainda não foram encontrados algoritmos que os resolvam em tempo polinomial. Qual a dificuldade de encontrar tais algoritmos? Ainda não há uma resposta definitiva para essa pergunta. Pode ser que que ainda não foi encontrado um princípio que ajude a solucionar o problema de forma mais estratégica. Ou pode ser que realmente não haja solução em tempo polinomial (SIPSER, 2006, v. 2, p.264).

Um problema simples, mas bem conhecido é o problema do número composto. Esse problema pergunta se, dado um número inteiro positivo N existem positivos inteiros m e n tal que $N = m.n$, sendo $m, n > 1$. A grande dificuldade desse

problema é quando N é primo. Um algoritmo trivial que resolve esse problema, tem complexidade $O(\sqrt{2}^{\log n})$. Essa é uma função exponencial o que torna o problema difícil de ser resolvido. No entanto, esse é um problema bem fácil de ser verificado em tempo polinomial. Se forem dados os valores de m ou n fica fácil verificar qual o termo restante, basta aplicar a divisão. Em 2002, Manindra Agrawal, Neeraj Kayal e Nitin Saxena desenvolveram um algoritmo capaz de decidir em tempo polinomial o problema. O algoritmo ficou conhecido como Teste da primalidade AKS (AGRAWAL, et al., 2004,v. 2, p. 781-793).

Os conceitos acima são importantes para definir a classe de problemas NP. Os diferentes problemas estão em NP, mas existem alguns problemas que podem ser resolvidos em tempo polinomial. Assim, é trivial deduzir que $P \in NP$. Como já observado acima, ainda não podemos concluir que os problemas que ainda não possuem resolução em tempo polinomial não possua tais algoritmos. A questão sobre se $P = NP$ é hoje um dos maiores problemas em aberto da ciência da computação teórica. A maioria das pesquisas indicam que são classes diferentes, já que foram investidos muitos recursos para desenvolver algoritmos eficientes, mas sem sucesso (SIPSER, 2006, v. 2, p.270).

2.2.5. Classe NP-Completo

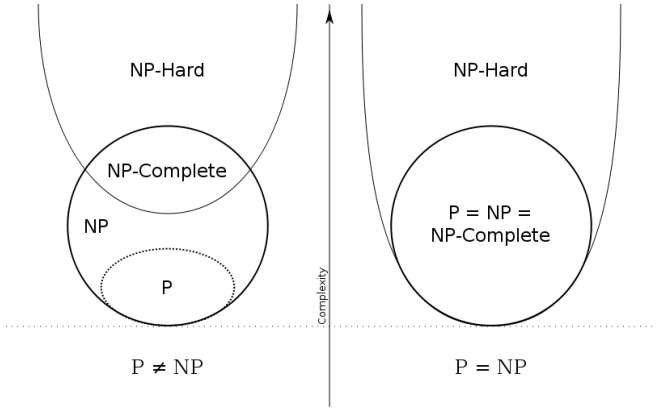
Envolvendo a questão abordada na Seção 2.2.4, P *versus* NP, um trabalho feito por Stephen Cook e Leonid Levin, mostrou que existem alguns problemas que se forem resolvidos com algoritmos em tempo polinomial, então **todos** os problemas de NP também poderão ser resolvidos com essa complexidade. Isso por causa de um fenômeno chamado de ***redução***. Em termos simples, redução é uma maneira de converter um problema para outro problema de tal forma que a solução para o segundo problema pode ser usada para solucionar o primeiro problema (SIPSER, 2006, v. 2, p.187, tradução nossa).

Um problema pertence à NP-Completo (entre outras variações, NP-Completude) quando um problema X pertence tanto à classe NP quanto à classe NP-Difícil. Um problema NP-Difícil é aquele que todos os problemas pertencentes à NP não são mais difíceis do que X. Os problemas NP-completos são

muito fáceis de serem verificados em tempo polinomial, mas não existe nenhum algoritmo conhecido que os resolvam em tempo polinomial.

Abaixo a Figura 1 ilustra os conjuntos de classe.

Figura 1 . Diagrama de Conjunto de Classes de Complexidade de Problemas



Fonte:Wikipédia[1].

Neste trabalho vamos abordar e estudar um problema bem típico da classe NP-Difícil: o Problema do Caixeiro Viajante (*Travelling salesman problem*). Importante salientar que o objetivo do trabalho não é provar que $P = NP$ ou que $P \neq NP$. O objetivo é analisar as dificuldades de resolver o problema e apresentar uma forma de reduzir o custo computacional para determinar uma solução ótima.

2.3. Grafos

As definições a seguir são essenciais para o entendimento da implementação que vamos analisar à frente.

2.3.1. Grafo Orientado

Um grafo orientado (ou digrafo) G é um par (V, E), onde V é um conjunto finito e E é uma relação binária de V . O conjunto V é chamado de **conjunto de vértices** de G, e os elementos são chamados de **vértices**. O conjunto E é chamado de conjunto de arestas de G, e os elementos são chamados arestas.

Neste tipo de grafo é possível *loops* no mesmo vértice, ou seja, uma aresta de um vértice para ele mesmo (CORMEN, 2002, v. 2, p.1080, tradução nossa).

[1] Disponível em:<https://pt.wikipedia.org/wiki/NP-difícil> Acesso em nov. 2016.

2.3.2. Grafo Não Orientado

Em um grafo não orientado $G = (V, E)$, o conjunto de arestas E é constituído de pares não ordenados de vértices, em vez de pares ordenados. Sendo assim, uma aresta é um conjunto $\{u, v\}$, onde $u, v \in V$ e $u \neq v$. Por convenção, é usada a notação (u, v) para uma aresta, ao invés da notação $\{u, v\}$, e (u, v) e (v, u) são considerados a mesma aresta. Em grafos não orientados não existem arestas no próprio vértice, então toda aresta consiste em exatamente dois vértices diferentes (CORMEN, 2002, v. 2, p.1080, tradução nossa).

2.3.3. Adjacência

Se (u, v) é uma aresta no grafo $G = (V, E)$, então o vértice v é **adjacente** ao vértice u. Quando o grafo é não orientado, a relação de adjacência é simétrica. Quando o grafo é orientado, a relação de adjacência não necessariamente é simétrica (CORMEN, 2002, v. 2, p.1081, tradução nossa).

2.3.4. Caminho em um grafo

O caminho k em um grafo do vértice u ao vértice u' em um grafo $G = (V, E)$ é uma sequência $\{v_0, v_1, v_2, \ldots, v_k\}$ de vértices, tal que $u = v_0$, $u' = v_k$, e $(v_{i-1}, v_i) \in E$ para $i = 1, 2, 3, ..., k$. O comprimento do caminho é a quantidade de arestas nele. Caso seja um grafo **ponderado**, ou seja, quando as arestas possuem valores diferentes entre si, o comprimento do caminho é a soma dos valores das arestas (CORMEN, 2002, v. 2, p.1081, tradução nossa).

Um caminho é **simples** quando todos os vértices desse caminho são distintos (CORMEN, 2002, v. 2, p.1081, tradução nossa).

Um **subcaminho** de um caminho $p = \{v_0, v_1, \ldots, v_k\}$ é uma subsequência contínua desses vértices. Isto é, para qualquer $0 \leq i \leq j \leq k$, a subsequência de vértices $\{v_i, v_{i+1}, \ldots, v_j\}$ é um subcaminho de p (CORMEN, 2002, v. 2, p.1081, tradução nossa).

Em um grafo não orientado, um caminho $\{v_0, v_1, v_2, \ldots, v_k\}$ forma um (**simples**) **ciclo** se $k \geq 3$, $v_0 = v_k$, e $v_1, v_2, \ldots, v_k$ são distintos. Um grafo que não possui ciclos é chamado acíclico (CORMEN, 2002, v. 2, p.1082, tradução nossa).

Um grafo não orientado é **conexo** se cada par de vértice é conectado por um caminho. Caso contrário ele é **desconexo** (CORMEN, 2002, v. 2, p.1082).

2.3.5. Tipos de grafos

Existem diversos tipos de grafos. Esses tipos, ou variações, dependem da relação de conectividade entre os vértices e de características citadas acima.

Um **grafo completo** é um grafo simples, não orientado, no qual cada par de vértice é adjacente (CORMEN, 2002, v. 2, p.1083, tradução nossa).

Um **grafo bipartido** é um grafo não orientado $G = (V, E)$, no qual V pode ser particionado em dois conjuntos V_1 e V_2 de tal modo que $(u, v) \in E$ implica em $u \in V_1$ e $v \in V_2$ ou $u \in V_2$ e $v \in V_1$. Isto é, todas as arestas vão de um conjunto ao outro (CORMEN, 2002, v. 2, p.1083, tradução nossa).

Um grafo acíclico, não orientado, é uma **floresta**, e um grafo conexo, acíclico e não orientado é uma **árvore** (CORMEN, 2002, v. 2, p.1083, tradução nossa).

2.4. Heurística

Heurísticas são “procedimentos e normas usados em pesquisa feita por meio da quantificação de proximidade a um determinado objetivo.” (MICHAELIS, 2015).

Uma heurística é uma técnica projetada para resolver um problema mais rapidamente quando os métodos clássicos são muito lentos ou para encontrar uma solução aproximada quando os métodos clássicos não conseguem encontrar uma solução exata.

No nosso dia a dia usamos algumas processos heurísticos para resolver problemas. Por exemplo, se não conseguimos entender um problema, podemos tentar desenhá-lo. Se não encontramos soluções para um problema específico, podemos assumir que temos uma boa resposta e tentamos derivar um novo conhecimento sobre a dificuldade em resolvê-lo. Ao tentar resolver problemas complexos, talvez possamos começar resolvendo problemas mais simples antes (PÓLYA ,1945).

2.4.1. Metaheurística

Metaheurística é um procedimento ou heurística de nível superior projetada para encontrar, gerar ou selecionar uma heurística (algoritmo de busca parcial) que

possa fornecer uma solução suficientemente boa para um problema de otimização, especialmente com informações incompletas ou imperfeitas ou capacidade de computação limitada (DORIGO; GAMBARDELLA, 2009; BLUM; ROLI, 2003).

2.5. O Problema do Caixeiro Viajante

O Problema do Caixeiro Viajante (*Travelling salesman problem ou TSP*) é um problema de otimização combinatória. É dado um conjunto $\{c_1, c_2, \ldots, c_N\}$ de *cidades* e para cada par $\{c_i, c_j\}$ de distintas cidades a distância $d(c_i, c_j)$. O objetivo é encontrar uma ordem π de cidades que minimize a quantidade.

$$\sum_{i=1}^{N-1} d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi(N)}, c_{\pi(1)})$$

Essa quantidade se refere ao comprimento do circuito, uma vez que o comprimento é o circuito que um vendedor fará quando visitar todas as cidades na ordem especificada pela permutação, retornando ao final para à cidade inicial (JOHNSON; MCGEOCH 1995, p. 3, tradução nossa). Esse problema também é conhecido como Ciclo Hamiltoniano que tem a seguinte definição: “O grafo... é Hamiltoniano se possuir um Ciclo Hamiltoniano (circuito), isto é, se admitir um caminho indireto (direto) fechado que passe em cada vértice exatamente uma vez.” (CAMPELLO; MACULAN,1994)

Podemos dizer em outras palavras que o problema, basicamente, se resume em uma pergunta : “Dada uma lista de cidades e as distâncias entre cada par de cidades, qual a menor distância para se percorrer visitando apenas uma vez cada cidade e retornando à cidade de origem?”. Apesar de parecer um problema simples ele é extremamente complexo do ponto de vista computacional.

Inicialmente podemos pensar em reduzir o problema de otimização em um problema de enumeração, ou seja, construímos todos os circuitos possíveis retornando a menor distância calculada. Existem várias formas de implementar isso. Boas implementações desse processo tem complexidade proporcional ao total de circuitos. Calcular o número de circuitos para uma instância n de cidades para o TSP é fácil. Dada uma lista de cidades, escolhemos qualquer cidade como ponto de partida. A partir desse começo teremos $n-1$ escolhas para a segunda cidade, $n-2$

escolhas para a terceira cidade, e assim por diante. Multiplicando todas as possibilidades de escolha teremos o total de circuitos possíveis igual a:

$$(n-1)! = (n-1)x(n-2)x(n-3)...3x2x1$$ [2]

Então esse método gastaria um tempo proporcional a $(n-1)!$ para resolver o problema para n cidades (APPLEGATE, 2006, p.45, tradução nossa).

A Tabela 2 utiliza um computador capaz de processar 1 bilhão de adições por segundo. Para calcular cada circuito é necessário fazer n adições. Então esse computador tem a capacidade de processar 1 bilhão dividido por n circuitos por segundo. Para cada instância do TSP com n cidades é possível $(n-1)!$ circuitos. Portanto, o tempo total para calcular todas as rotas é a relação entre a quantidade total de circuitos por segundo que esse computador consegue processar e a quantidade total de possíveis circuitos.

Note que o tempo necessário para calcular todas os circuitos com 20 cidades é quase a média do ciclo de vida de uma pessoa.

Tabela 2 - Estimativa de tempo de processamento do TSP para uma instância de tamanho n.

n	circuitos por segundo	$(n-1)!$	Tempo de cálculo total
5	200 milhões	24	insignificante
10	100 milhões	362 880	0.003 seg
15	67 milhões	87 bilhões	20 min
20	50 milhões	1.2×10^{17}	73 anos
25	40 milhões	6.2×10^{23}	470 milhões de anos

Fonte: Elaborado pelo autor.

2.5.1. Aplicabilidade

A aplicação direta do TSP está no problema de **perfuração de placas de circuito impresso (PCBs)** (GRÖTSCHEL; HOLLAND, 1991). Para conectar um condutor em uma camada com um condutor em outra camada, ou para posicionar os pinos de circuitos integrados, furos devem ser feitos através da placa. Os furos

[2] Esse total é para o TSP assimétrico. Para o simétrico, o total é proporcional a metade desse valor. Por exemplo, para um TSP simétrico de 3 cidades existe apenas um circuito.

podem ser de tamanhos diferentes. Para perfurar dois furos de diâmetros diferentes consecutivamente, a cabeça da máquina tem que se mover para uma caixa de ferramentas e trocar o equipamento de perfuração. Isso é muito demorado. Assim, é claro que é preciso escolher algum diâmetro, perfurar todos os furos do mesmo diâmetro, mudar a broca, perfurar os furos do próximo diâmetro, etc. Assim, este problema de perfuração pode ser visto como uma série de TSPs, um para cada diâmetro do furo, onde as "cidades" são a posição inicial e o conjunto de todos os furos que podem ser perfurados com uma mesma broca. A "distância" entre duas cidades é dada pelo tempo que leva para mover a cabeça de perfuração de uma posição para a outra. O objetivo é minimizar o tempo de viagem da cabeça da máquina.

Outra aplicação é no problema da **revisão de motores de turbinas a gás** (PLANTE et al., 1987). Esse problema ocorre quando os motores de turbinas a gás de aeronaves precisam ser revisados. Para garantir um fluxo uniforme de gás através das turbinas, existem conjuntos de palhetas guia do bocal localizados em cada fase de turbina. Uma montagem consiste basicamente num número de palhetas guia fixadas em torno da sua circunferência. Todas estas palhetas têm características individuais e a colocação correta das palhetas pode resultar em benefícios substanciais (reduzindo a vibração, aumentando a uniformidade do fluxo, reduzindo o consumo de combustível). O problema de colocar as palhetas da melhor maneira possível pode ser modelado como um TSP com uma função objetiva especial.

A **análise da estrutura de cristais,** ou **cristalografia por raios X,** (BLAND & SHALLCROSS, 1989; DREISSIG & UEBACH, 1990) é uma aplicação importante do TSP. Aqui é utilizado um difratômetro de raios X para obter informação sobre a estrutura do material cristalino. Para este fim, um detector mede a intensidade dos reflexos de raios X do cristal a partir de várias posições. Considerando que a medição em si pode ser realizada muito rapidamente, há uma considerável sobrecarga no tempo de posicionamento, visto que é necessário centenas de milhares de posições para algumas análises. O posicionamento envolve mover quatro motores. O tempo necessário para mover um motor de uma posição para a outra pode ser calculado com muita precisão. O resultado da análise não depende

da sequência das posições em que as medições são feitas. No entanto, o tempo total necessário para a análise depende da sequência. Portanto, o problema consiste em encontrar uma sequência que minimize o tempo total de posicionamento.

O problema do **despacho de pedidos** está associado ao manuseio de materiais em um armazém (RATLIFF; ROSENTHAL 1983). Suponha que chegue a um depósito uma ordem para um determinado subconjunto dos itens armazenados no depósito. Algum veículo tem que recolher todos os objetos deste pedido para enviá-los ao cliente. A relação com o TSP é imediatamente vista. Os locais de armazenamento dos itens correspondem aos nós do grafo. A distância entre dois nós é dada pelo tempo necessário para mover o veículo de um local para o outro. O problema de encontrar uma rota mais curta para o veículo pode ser resolvido como um TSP.

Suponha que, em uma cidade, as caixas postais devem ser esvaziadas todos os dias dentro de um certo período de tempo, digamos 1 hora. O problema, conhecido como **roteamento de veículos**, é encontrar o número mínimo de caminhões para fazer isso e o menor tempo para fazer as coletas. Como outro exemplo, suponha que n clientes exigem certas quantidades de alguns produtos e um fornecedor tem de satisfazer todas as demandas com uma frota de caminhões. O problema é especificar o cliente e um cronograma de entrega para cada caminhão, não excedendo a capacidade de carga de cada caminhão e a distância total de viagem seja mínima. Existem muitas variações desses dois problemas, onde as restrições de tempo e capacidade são combinadas, são comuns em muitas aplicações do mundo real (LENSTRA, 1974).

Existem outras diversas aplicações que grande parte da literatura não consegue abordar todas. Outros exemplos são: Logistica, Sequenciamento do Genoma, Telescópios Guiados, *Data Clustering*, Roteamento de ônibus escolar, Sistema de Navegação por Satélite, Planejamento de missões do Exército (APPLEGATE, 2006, p. 59-78; DAVENDRA, 2010, p. 3-6).

2.5.2. Algoritmos para o TSP

Visto a grande quantidade de aplicações que podemos dar a esse tipo de otimização e visto que cada vez mais os processos são tratados de forma

computacional, o interesse em algoritmos otimizados se tornou o foco de muitos pesquisadores e até de empresas.

As primeiras formulações matemáticas para o TSP são formulações para programação linear (DANTZIG, 1954). A formulação mais comum para o TSP simétrico apresentada é a seguinte (DAVENDRA, 2010, p. 6-7):

Minimizar:

$$\sum_{i<j} c_{ij} x_{ij} \quad (1)$$

Sujeito a:

$$\sum_{i<k} x_{ik} + \sum_{j>k} x_{kj} = 2 \ \ (k \in V) \quad (2)$$

$$\sum_{i,j \in S} x_{ij} \leq |S| - 1 \ (S \subset V, 3 \leq |S| \leq n - 3) \quad (3)$$

$$x_{ij} = 0 \ ou \ 1 \ (i,j) \in E \quad (4)$$

Nesta formulação, as restrições (2), (3) e (4) se referem ao grau das restrições, restrição de eliminação de subcircuitos e restrições de integridade, respectivamente. Na presença da restrição (2), (3) é algebricamente equivalente a restrição de conectividade.

$$\sum_{i \in S, , j \in V, , j \in S} x_{ij} \geq 2 \ (S \subset V, 3 \leq |S| \leq n - 3) \quad (5)$$

Os algoritmos de solução para programação linear do TSP são baseados nessas formulações. A resolução através de programação linear gera soluções ótimas, mas , como já contextualizado, conforme a quantidade de vértices aumenta mais complexo de resolver. Em vista disso, foram desenvolvidas diversas técnicas heurísticas que abordam soluções aproximadas, mas com custo computacional menor. Essas técnicas se resumem em dois métodos: método de construção de circuitos e método de melhoria de circuitos.

2.5.2.1. Abordagem de soluções exatas

A abordagem exata busca sempre encontrar a melhor solução possível. Quando Dantzig introduziu a primeira formulação, o método simplex estava incompleto e não existiam algoritmos disponíveis para resolver problemas de programação linear inteira. As primeiras estratégias de resolução consistem no

relaxamento inicial das restrições e nos requisitos de integridade, que foram aprimorados depois do problema de relaxação (MARTIN, 1966). Na investida de melhorar o desempenho dos algoritmos, foram aplicados métodos tais como o ‘Algoritmo Euclidiano Acelerado‘ (GOMORY, 1963; SEDJELMACI, 2004). Outros métodos como *Branch and Bound* (MILIOTIS, 1978), eliminação de restrições de subcircuitos (LAND, 1979), árvores de clique com inequações (CHV’ATAL, 1973; GRÖTSCHEL; PULLEYBLANK, 1986) e caminhos de inequações (CORNU’EJOLS et al., 1985). Mais recentemente, o algoritmo exato mais eficiente é o Algoritmo Dantzig–Fulkerson–Johnson (APPLEGATE, 2003).

2.5.2.2. Abordagem de soluções aproximadas

Essa abordagem é utilizada quando a busca de solução não precisa ser exata e com baixo custo computacional. Para isso são implementadas heurísticas. Existem basicamente dois tipos de heurísticas para o TSP: a de construção e a de melhoria.

A heurística de construção consiste em buscar localmente uma nova rota, ou seja, a cada passo é analisado somente resultados locais (CAMPELLO, MACULAN, 1994). A heurística de melhoria, ou melhoramento, como o próprio nome indica, consiste em encontrar uma solução inicial e depois tenta melhorar o circuitos através de técnicas de trocas (*swap*) para encontrar soluções de melhor qualidade.

2.5.2.2.1. Algoritmos para construção de circuitos

- Heurística do vizinho mais próximo, que é caracterizado pela escolha da cidade mais próxima, sempre que o caixeiro se desloque, até que todas as cidades sejam visitadas (SOLOMON,1987);
- Heurística de inserção, que se descreve pela inclusão de cidades, uma a uma, atendendo a um determinado critério de proximidade, por exemplo, a cidade mais distante, partindo de um circuito inicial com duas cidades. Aquando da inserção, a escolha deve ser analisada entre cada par de cidades do circuito parcial, até que todas estejam inseridas (DAVENDRA, 2010);
- Heurística da cobertura mínima, onde é elaborada uma árvore que contenha todas as cidades, com a finalidade de se encontrar uma árvore de cobertura com a menor soma de distâncias possível (GABOW, 1986);

- Heurística das economias, ou heurística de Clarke e Wright, consiste no agrupamento sequencial de cidades, com base numa ordem decrescente de economias decorrentes da sua inclusão, isto é, considerando o impacto, da junção do nó no circuito, nas economias agregadas da distância entre nós e da distância de cada um dos nós ao nó inicial (CLARKE; WRIGHT, 1964).

2.5.2.2.2. Algoritmos para melhoria de circuitos

- k-opt, nesta proposta, *k* arestas são substituídas, no circuito, por outras *k* arestas com o objetivo de diminuir a distância total percorrida. Quanto maior for o valor de *k*, melhor é a precisão do método, mas maior é o esforço computacional (LINS,1973).
- Arrefecimento simulado, ou *Simulated Annealing,* é uma metaheurística para otimização que consiste numa técnica de busca local probabilística, e se fundamenta numa analogia com a termodinâmica. Substitui a solução atual por uma solução próxima (i.e., na sua vizinhança no espaço de soluções), escolhida de acordo com uma função objetivo e com uma variável T (dita *Temperatura*, por analogia). Quanto maior for T, maior a componente aleatória que será incluída na próxima solução escolhida. À medida que o algoritmo progride, o valor de T é decrementado, começando o algoritmo a convergir para uma solução ótima, necessariamente local (KIRKPATRICK, 1983).
- Busca Tabu, é um procedimento adaptativo auxiliar, que guia um algoritmo de busca local na exploração contínua dentro de um espaço de busca. A partir de uma solução inicial, tenta avançar para uma outra solução (melhor que a anterior) na sua vizinhança até que se satisfaça um determinado critério de parada (GLOVER; LAGUNA, 1997).
- GRASP (*Greedy Randomized Adaptive Search Procedure*), consiste em criar uma solução inicial e depois efetuar uma busca local para melhorar a qualidade da solução. Seu diferencial para outros métodos está na geração dessa solução inicial, baseada nas três primeiras iniciais de sua sigla em inglês: gulosa (*Greedy*), aleatória (*Randomized*) e adaptativa (*Adaptive*) (FEO; RESENDE, 1995).

Outros exemplos de algoritmos e métodos utilizados para otimizar circuitos:

- Algoritmos genéticos, tem por base um processo iterativo sobre uma determinada população fixa, denominados por indivíduos, que representam as várias soluções do problema (GOLDBERG; HOLLAND 1988).
- Redes Neurais, caracterizada por uma equação diferencial, que engloba uma função do tipo sigmóide. Os vários parâmetros incluídos nas equações afetam a convergência da rede, pois tratam-se de fatores penalizantes pelas violações às restrições do problema, de controle para a minimização da função objetivo, entre outros. Esta formulação possui ainda um termo que avalia as violações às restrições do problema dando a indicação do cumprimento das mesmas, após um certo número de iterações. Depois de encontrados tais parâmetros, aplica-se o princípio WTA (SIQUEIRA, 2006).
- Colônia de formigas (ACO), onde as formigas são simples agentes que, no caso do TSP, constroem circuitos através do movimento entre cidades no grafo do problema. A solução construída pelas formigas é elaborada por trilhos de feromonas (artificiais) e pela disponibilidade de informação heurística, à priori. Quando o algoritmo ACO é aplicado, é associada uma força da feromona , onde $\tau_{ij}(t) = \tau_{ji}(t)$ é uma informação numérica que é modificada durante o algoritmo, e *t* é o contador das iterações (DORIGO; GAMBARDELLA, 1996).

Existem outras heurísticas e formas de resolver o TSP (MARTIN; REINELT, 2011, p. 17-40).

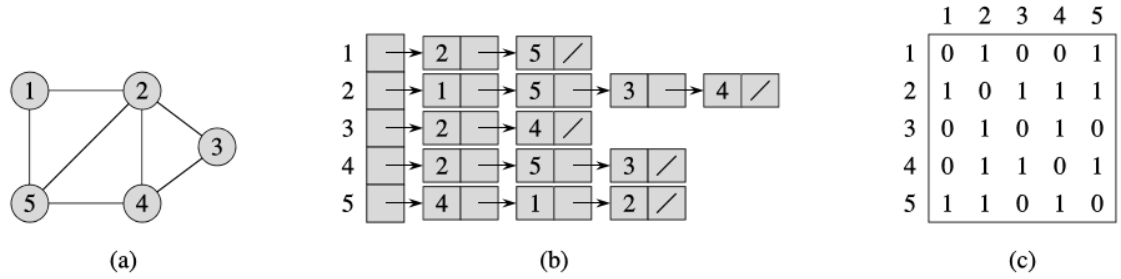
3. ALGORITMOS IMPLEMENTADOS

Vamos abordar neste Capítulo o modo como resolvemos o TSP Simétrico[3]. Demonstrar a estrutura do grafo da instância e a implementação das heurísticas utilizadas para minimizar o tempo de busca. Toda a implementação foi desenvolvida em linguagem C, por ser uma linguagem de alto desempenho, alto poder de manipulação de memória e compilação consistente.

3.1. Representação de grafos

Existem dois padrões para representar um grafo $G = (V, E)$: como uma coleção de listas de adjacência ou como matriz de adjacência. Qualquer que seja a forma ela é aplicada a grafos orientados e não orientados. A representação em forma de listas de adjacência é comumente utilizada, visto que ela provê uma representação compacta para grafos esparsos - aqueles onde $|E|$ é muito menor que $|V|^2$. A representação através de matriz de adjacência é mais utilizada, quando o grafo é denso - $|E|$ é muito próximo a $|V|^2$ - ou quando é necessário verificar rapidamente se existe uma aresta conectando dois vértices (CORMEN, 2002, p. 527, tradução nossa).

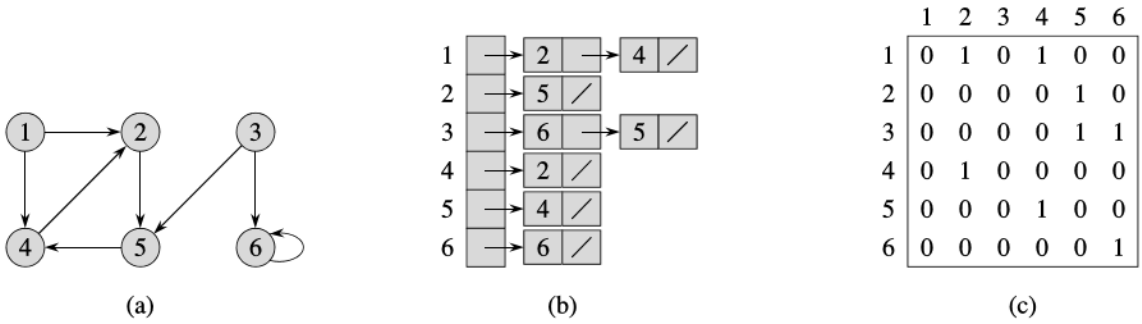
Figura 2 . Representação de grafos não orientado. (a) Um grafo não orientado G com 5 vértices e 7 arestas. (b) Uma representação de listas adjacentes de G. (c) Uma representação de matriz de adjacência de G.



Fonte: CORMEN, 2002, p.528

[3] Para um melhor desempenho do algoritmo, as heurísticas utilizadas se baseiam em instâncias que possuem as coordenadas dos vértices. Caso a instância não as possua o algoritmo funcionará com mais tempo de processamento.

Figura 3 . Representação de grafos orientados. (a) Um grafo orientado G com 6 vértices e 8 arestas. (b) Uma representação de listas adjacentes de G. (c) Uma representação de matriz de adjacência de G.



Fonte: CORMEN, 2002, p.528

A representação como listas de adjacência do grafo $G = (V, E)$ consiste em um vetor Adj de $|V|$ listas, uma para cada vértice V. Para cada $u \in V$, a lista adjacente $Adj[u]$ contém todos os vértices v que possuem uma aresta $(u, v) \in E$. Isto é, $Adj[u]$ consiste em todos os vértices adjacente a u em G. Os vértices em cada lista adjacente é organizado em uma ordem aleatória (CORMEN, 2002, p. 528, tradução nossa).

A representação de matriz de adjacência de um grafo $G = (V, E)$, assume que os vértices são numerados 1,2…, $|V|$ de forma arbitrária. Assim, a matriz de adjacência de um grafo G consiste em uma matriz $|V| \, x \, |V|$, $A = (a_{ij})$ onde $a_{ij} = \{1 \; se \; (i,j) \in E, 0 \; caso \; contrário.\}$ (CORMEN, 2002, p. 528, tradução nossa).

As duas representações podem ser adaptadas para grafos ponderados, ou seja, quando as arestas possuem pesos (CORMEN, 2002).

3.1.1. Estrutura de um grafo para TSP

Neste trabalho optamos por utilizar a representação de listas de adjacência. O grafo de representação é estruturado como um vetor de dados com os seguintes campos:

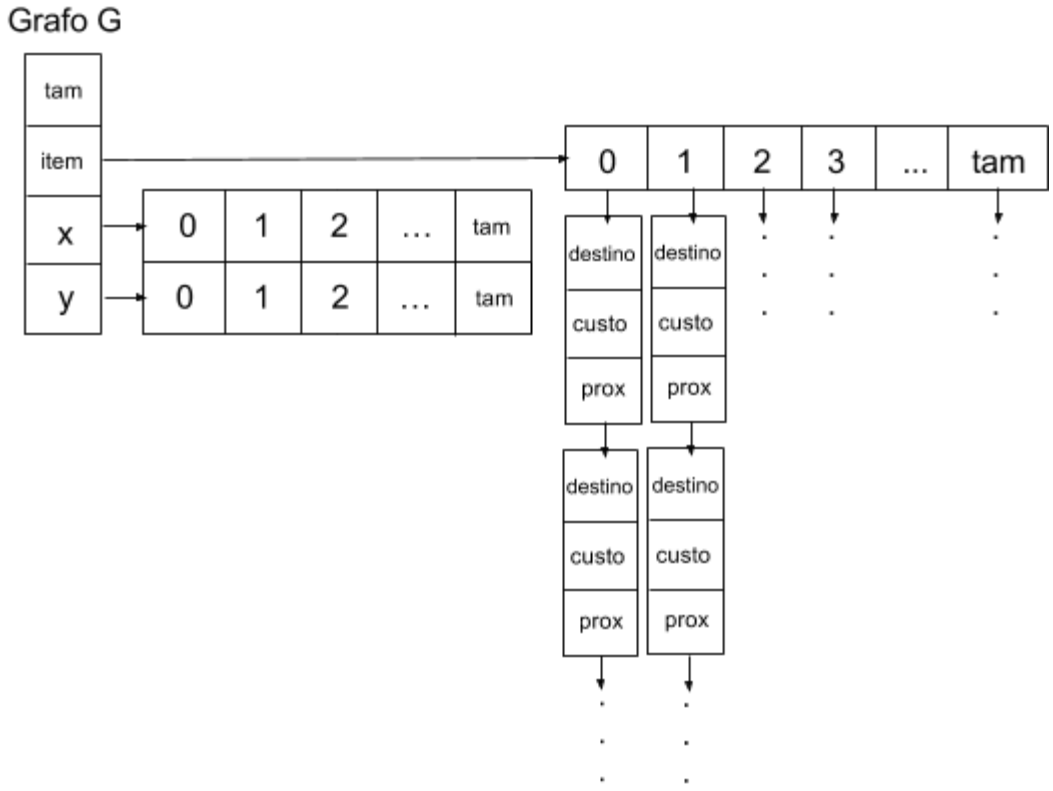
- *tam* - quantidade de cidades do problema;
- *item* - listas de adjacências;
- *x* - vetor com os valores do eixo x de cada vértice;
- *y* - vetor com os valores do eixo y de cada vértice.

O campo *item* é um vetor onde cada posição é uma lista de adjacência conforme conceito introduzido na Seção 3.1:

- *destino* - vértice de destino a partir do vértice atual;
- *custo* - custo para se deslocar até o vértice de destino a partir do atual;
- *prox* - endereço de memória para acessar o próximo campo do vetor.

A Figura 4 é um representação gráfica da ideia apresentada acima.

Figura 4 . Representação gráfica da estrutura de um grafo conexo ponderado.



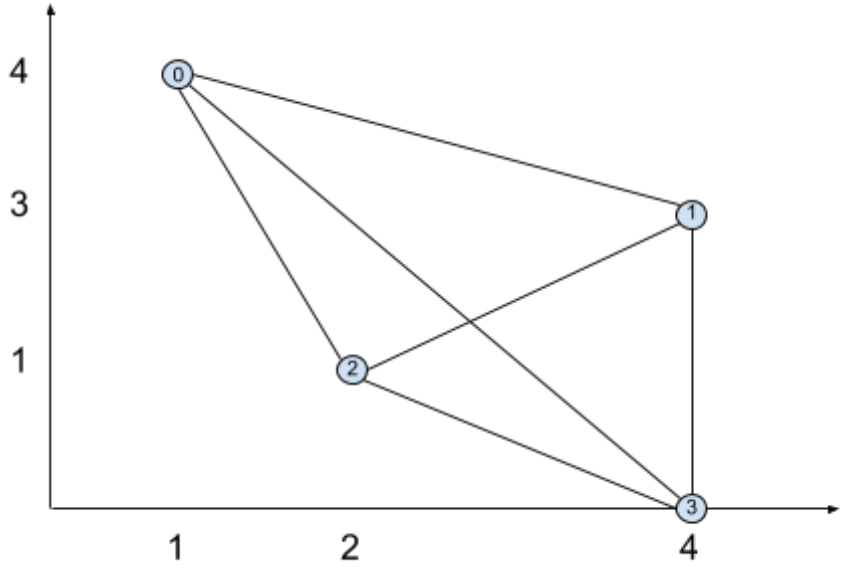
Fonte: Elaborado pelo autor

Cada $item[v]$, para $v = 0, 1, 2, .., tam$, encontramos uma lista de vértices destino d e, para cada vértice, o respectivo custo da aresta $c = (v, d)$. A ordem de apresentação dos vértices d é crescente, ou seja, o primeiro vértice d a partir de v é o que apresenta a aresta de menor custo c. Essa forma de organização aumentará a eficiência dos algoritmos que analisaremos nas Seções a seguir.

3.1.1.1. Exemplo de grafo dado uma instância

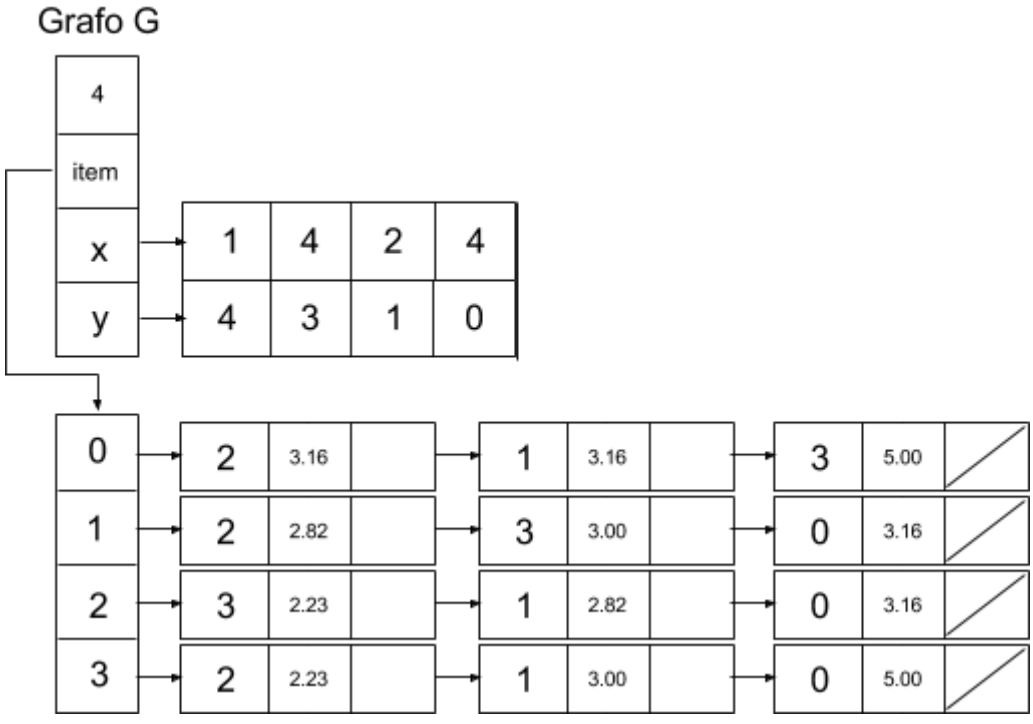
A Figura 5 é uma instância em um eixo de coordenadas cartesiano. A Figura 6 representa o modelo dessa instância graficamente. Os vértices são representados por números e o custo das arestas são calculados através da aplicação do teorema de Pitágoras.

Figura 5 . Exemplo de instância para o TSP simétrico.



Fonte:Elaborado pelo autor

Figura 6 . Representação gráfica da estrutura de um grafo conexo ponderado a partir de uma instância.



Fonte: Elaborado pelo autor

A instância deve ser fornecida em forma de matriz de adjacência com custos. Caso não seja fornecida uma fonte de dados específica, ou seja, um arquivo com a quantidade de vértices e as coordenadas dos vértices, então pode ser gerado uma instância aleatória.

Nas Seções a seguir será analisado as técnicas utilizadas para minimizar o tempo de busca da solução.

3.2. Técnica *Branch & Bound*

A técnica do *Branch & Bound* (B&B) foi proposta inicialmente por A. H. Land e A. G. Doig (LAND; DOIG, 1960) para resolver problemas de programação discreta, mas se tornou uma ferramenta muito utilizada para resolver problemas de otimização combinatória como o abordado neste trabalho (CLAUSEN , 1999; LITTLE, 1963).

O algoritmo consiste em fazer uma busca completa pelo espaço de soluções a fim de encontrar a melhor solução. No entanto, uma enumeração explícita é normalmente impossível devido ao crescimento exponencial de potenciais soluções. Então utilizando *bounds* (limitantes) e combinando com uma solução temporária, tratada como ótima, o algoritmo faz uma busca em apenas algumas partes do espaço de solução de forma implícita (CLAUSEN , 1999, p. 2, tradução nossa). Divide-se o problema em subproblemas menores onde é mais fácil de encontrar soluções. Para cada iteração um subproblema é analisado podendo ser dividido em outro subproblema ou sendo descartado caso ele extrapole um limitante.

O algoritmo possui três características principais:

1. uma *função limitante (bound*) que proporciona um limite inferior ao subproblema analisado,
2. uma *estratégia de seleção* para qual subproblema será analisado, e
3. uma regra de *ramificação (branch*) que diz se um determinado subproblema não pode ser descartado, ditando a divisão deste subproblema em tantos outros quanto necessário para as próximas iterações (CLAUSEN , 1999, p. 12, tradução nossa).

Adicionalmente, é relevante a produção de uma boa solução inicial utilizando uma heurística para facilitar a definição das características acima.

3.2.1. Função limitante

A função limitante é um fator chave para qualquer algoritmo de B&B no sentido que funções limitantes ruins não são compensadas por escolhas de estratégias de ramificação ou seleção. Idealmente, o valor limitante para um determinado subproblema deve ser igual ao valor da solução mais viável. Visto que encontrar esse valor talvez seja tão difícil quanto resolver o problema, busca-se chegar o mais próximo possível utilizando algoritmos polinomiais (CLAUSEN , 1999, p. 13, tradução nossa).

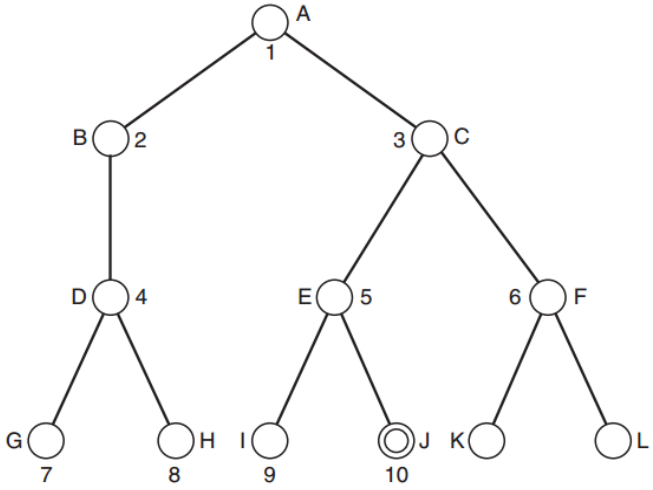
Neste trabalho a função limitante é o próprio valor da melhor solução encontrada até o momento. Para cada nova melhor solução encontrada durante a busca, o valor da função limitante é igual ao valor dessa solução.

3.2.2. Estratégia de seleção

A estratégia para selecionar o próximo subproblema a ser analisado varia de acordo com o tipo de problema a ser resolvido e a capacidade computacional (capacidade de processamento, memória disponível, etc). As duas estratégias mais comuns são a busca em largura e a busca em profundidade.

A busca em largura (*Breadth-First Search - BFS*), como o nome sugere, examina todos os nós de um nível (às vezes chamado de camada) passando para os próximos níveis (COPPIN, 2004). A Figura 7 ilustra o conceito desse tipo de busca.

Figura 7. Representação do BFS. Os números indicam a ordem em que os nós são examinados.

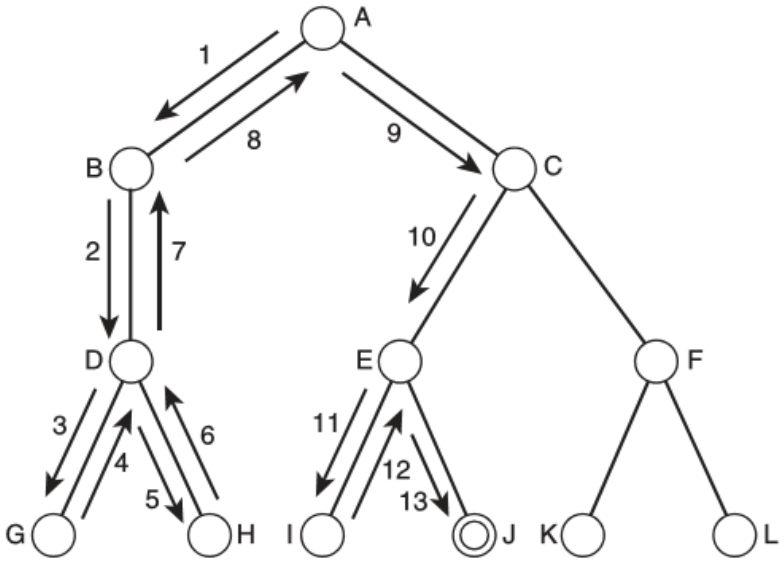


Fonte: COPPIN, 2004, p. 76

Partindo de um nó raiz, são expandidos todos os nós possíveis. Cada nó daquele nível é verificado e caso seja encontrado uma solução a busca é finalizada. Caso não seja, todos os nós da camada atual são expandidos. A busca continua sucessivamente para cada camada até encontrar uma solução

Por sua vez, a busca em profundidade (*Depth-First Search - DFS*) explora cada caminho o mais profundo possível antes de explorar outro caminho (COPPIN, 2004). A Figura 8 ilustra o princípio do DFS.

Figura 8. Representação do DFS. Os números indicam a ordem em que os nós são examinados.



Fonte: COPPIN, 2004, p. 75

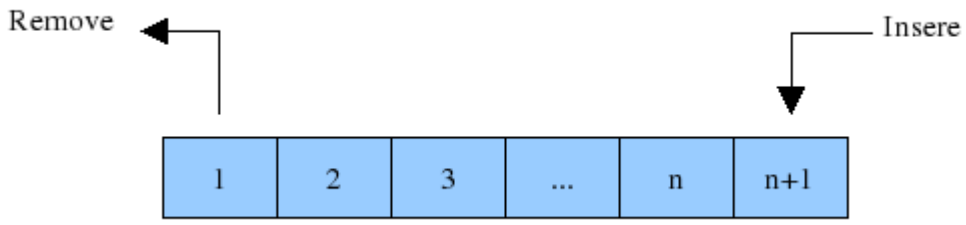
Partindo de um nó raiz, e assumindo que a busca será feita pela esquerda da árvore, examina-se todo o caminho para baixo tanto quanto for possível. Caso o nó final for uma solução, a busca é finalizada. Caso não seja, o método realiza um *backtracking* (retrocesso) e analisa as outras ramificações ainda não exploradas do nó imediatamente anterior (CLAUSEN , 1999, p. 19, tradução nossa).

Comumente essas duas estratégias são representadas por estruturas de dados chamadas de fila e pilha, para BFS e DFS, respectivamente.

As filas são utilizadas quando é necessário resgatar dados na ordem *first in, first out* (FIFO), ou seja, o primeiro dado que foi colocado na fila, será o primeiro a ser retirado (SKIENA, 2008, p. 71). Esse conceito pode ser claramente visto, por exemplo, ao esperar na fila para entrar em um restaurante. O primeiro a chegar será o primeiro a receber um local para sentar.

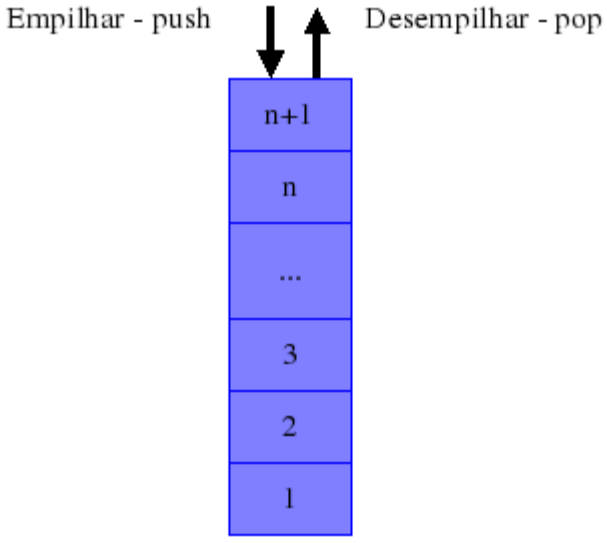
As pilhas são utilizadas quando é necessário resgatar dados na ordem *last in, first out* (LIFO), ou seja, o último dado a ser colocado na pilha será o primeiro a ser retirado (SKIENA, 2008, p. 71). Esse conceito pode ser observado, ao arrumar uma mala com roupas para viagem. As últimas roupas a serem colocadas na mala serão as primeiras a serem retiradas.

Figura 9. Representação da estrutura de filas - FIFO



Fonte : FARIAS, 2009[4]

Figura 10. Representação da estrutura de pilhas - LIFO



Fonte : FARIAS, 2009[5]

Neste trabalho a estratégia de seleção utilizada é o DFS. No início da busca, o algoritmo cria uma árvore com apenas um vértice (nó raiz). O vértice seguinte a ser analisado é o que está imediatamente mais próximo ao vértice atual. Caso o vértice analisado não faça parte do circuito que está sendo construído e as futuras

[4] Disponível em : <http://www.cos.ufrj.br/~rfarias/cos121/filas.html>. Acesso em maio de 2017.
[5] Disponível em : <http://www.cos.ufrj.br/~rfarias/cos121/pilhas.html>. Acesso em maio de 2017.

ramificações deste vértice não extrapolam a função limitante, então o vértice é inserido na solução. Nesse aspecto é aplicada a regra de ramificação que será trata a seguir.

Algoritmo 1. Busca B&B com estratégia DFS

```
busca(Lista Caminho, inteiro comprimento, Grafo G)
1.  se tam(Caminho) = 1 então
2.      custoMinimo ← ∞
3.  fim se
4.  se tam(Caminho) = tam(G) então
5.      se tam(Caminho) < custoMinimo então
6.          exibeCaminho(Caminho)
7.          custoMinimo ← custoAtual
8.          retorna
9.      fim se
10. fim se
11. para cada vertice destino d do ultimo vertice v de Caminho faça
12.     se Heurísticas então
13.         c ←criaNovoNo(destino(d), custo(d) + custo(Caminho), Caminho)
14.         busca(c, tam(Caminho)+1, G)
15.     fim se
16. fim para
```

A função *busca* é a rotina principal do algoritmo. Como parâmetros para a função são passados uma lista que contém os vértices do circuito construído. Essa estrutura irá armazenar a solução em construção durante o processamento do algoritmo. São passados também o valor do comprimento atual da solução em construção e uma representação do grafo da instância. Inicialmente é verificado se o tamanho do circuito atual, ou seja, do circuito armazenado dentro da lista é igual a um. Isso significa que na árvore de busca consta apenas um vértice, o nó raiz. Assim, qualquer circuito encontrado será utilizado como função limitante inicial. Se o circuito construído até então tiver a mesma quantidade de vértices que o grafo do problema e o custo do circuito construído for menor que o valor da função limitante, então o custo do circuito atual se torna o valor da função limitante.

A função *busca* é recursiva, ou seja, ela é chamada dentro dela mesma. Esse recurso é utilizado quando é necessário criar o efeito de pilha e, consequentemente, o recurso da busca em profundidade. Para o último vértice que está em *Caminho* é verificado os vértices adjacentes. Caso seja decidido que este vértice adjacente cumpre as regras de ramificação ele é inserido na solução parcial.

Note, portanto, que dependendo do resultado lógico de **Heurísticas** o algoritmo inserirá um vértice na solução parcial. As heurísticas implementadas constituem a regra de ramificação.

3.2.3. Regra de ramificação

No contexto do algoritmo B&B todas as regras de ramificação podem ser tidas como uma divisão de uma parte do espaço de busca com a adição de restrições. Quando um subproblema é dividido em dois, chamamos de ramificação dicotômica. Ao ser dividido em mais subproblemas chamamos de ramificação politômica (COPPIN, 2004, p. 19).

Para o caso do TSP, por exemplo, podemos ter um circuito incompleto. Cada circuito incompleto pode ser tratado como um subproblema para o TSP. Ramifica-se esse subproblema para outros subproblemas ao incluir novos vértices nesse circuito e/ou alterando partes do circuito, respeitando a função limitante e as regras de ramificação. Dessa forma teremos mais subproblemas para serem analisados. Caso esse subproblema gere um circuito completo, temos uma solução que será tratada como ótima. Caso gere um circuito parcial, outras ramificações poderão ser feitas.

Neste trabalho a regra de ramificação inclui algumas heurísticas que determinam se um nó continuará ou não a ser expandido. Dessa forma, permitimos que o algoritmo faça “podas” na árvore busca, ou seja, que ele descarte ramificações que não levarão a soluções ótimas. As heurísticas utilizadas para a regra de ramificação são discutidas na Seção 3.3.

3.2.4. Produção da solução inicial

Como abordado na Seção 3.2 foi mencionado a necessidade da produção de uma solução inicial de qualidade para a técnica B&B. Dependendo da qualidade da solução inicial, a busca encontra a solução ótima mais rapidamente, devido a quantidade menor de ramificações que deverão ser feitas.

Visto que o algoritmo inicia em um nó raiz, nesse caso, um vértice inicial, utilizamos o algoritmo do vizinho mais próximo para buscar um vértice que tende a construir um circuito ótimo.

O algoritmo do vizinho mais próximo é a mais simples e direta heurística para o TSP. A chave para esse algoritmo é sempre visitar o vértice mais próximo. Os passos base desse algoritmo são:

1. Selecionar um vértice qualquer;
2. Selecionar o vértice não visitado mais próximo;
3. Existem vértices ainda não visitado? Se sim, repetir passo 2;
4. Retornar ao vértice inicial (DAVENDRA, 2010, p. 13).

Para um problema com n vértices, o algoritmo terá que verificar todos os vértices vizinhos para decidir qual o mais próximo. O algoritmo fará $n(n-1)$ passos, ou seja, $n^2 - n$. Logo, a complexidade de tempo associado a esse método é $O(n^2)$ (DAVENDRA, 2010, p. 13). No entanto, como discutido na Seção 3.1.1. , a representação do grafo está organizada com os custos c na ordem crescente de cada vértice n para cada vértice d. Desta forma, sempre que o algoritmo encontrar um vértice destino d que ainda não foi visitado, este sempre será o mais próximo.

Algoritmo 2. Produção da solução inicial através do heurística do vizinho mais próximo

buscaMelhorVertice (Grafo G)
1. *melhorVertice* ← 0
2. *melhorCircuito* ← ∞
3. **para** *v* **de** *0* **até** *tam(G)* **faça**
4. *novoCaminho* ← *custoCaminho(v,v,G)*
5. **se** *novoCaminho* < *melhorCaminho* **então**
6. *melhorVertice* ← *v*
7. *melhorCaminho* = *novoCaminho*
8. **fim se**
9. **fim para**

devolva *melhorVertice*

A função *buscaMelhorVertice* recebe uma representação do instância e retorna um vértice. Para cada vértice é aplicado o algoritmo do vizinho mais próximo. O vértice que produzir um circuito de menor custo, é utilizado como nó raiz para iniciar a árvore de busca da técnica B&B.

Algoritmo 3. Cálculo do custo de subcircuito

custoCaminho (*verticeInicial, verticePartida, Grafo G*)
1. *custo* ← 0
2. *continua* ← verdadeiro
3. **enquanto** *continua* **faça**
4. **para** *cada vérticeDestino a partir de verticePartida* **faça**

```
5.        se vérticeDestino ainda não visitado então
6.            custo = custo + custo(vérticeDestino)
7.            verticePartida = vérticeDestino
8.            vérticeDestino é um vértice visitado
9.            finaliza laço enquanto
10.       senão
11.           continua = falso
12.       fim se
13.   fim para
14. fim enquanto
devolva custo
```

A função *custoCaminho* é utilizado pela função *buscaMelhorVertice.* Essa função é a implementação do vizinho mais próximo. Dado um vértice inicial, ela retorna o custo do circuito produzido.

3.3. Heurísticas

Antes da aplicação de heurísticas é necessário considerar as restrições inatas do problema. Para cada vértice d analisado, deve-se verificar se ele já está contido na solução parcial.

*se(**Não d** ⊂ **Caminho**) então*

A primeira heurística implementada é verificar se a soma do custo para ir do último vértice v da solução parcial até o vértice destino d com o custo total da solução construída é menor que o valor da função limitante. Se a soma for maior, o caminho que está sendo construído não irá gerar um circuito ótimo.

se*(Não* d ⊂ *Caminho*

E *custo(v,d) + custo(Caminho) < custoMinimo)* então

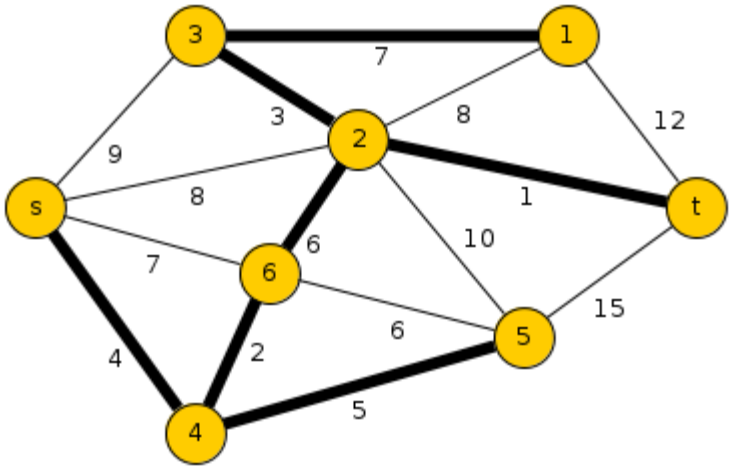
3.3.1. Árvore geradora mínima

Na criação dos circuitos eletrônicos, muitas vezes é necessário fazer com que os terminais de vários componentes sejam conectados eletricamente. Para interligar um conjunto de n terminais, é necessário um arranjo de $n-1$ conexões, onde cada uma conecta dois terminais. De todos esses arranjos, o que utiliza quantidade menor de material para as conexões geralmente é o mais desejável.

Esse problema pode ser modelado como um grafo conexo e não orientado $G=(V,E)$, onde V é o conjunto de terminais, E é o conjunto de possível conexões elétricas entre pares de terminais, e para cada aresta $(u,v) \in E$, um peso $w(u,v)$

especificando o custo (quantidade de material) para conectar u e v. Busca-se um subconjunto acíclico $T \subseteq E$ que conecte todos os vértices de tal maneira que o custo total $w(T) = \sum_{(u,v)\in T} w(u,v)$ seja mínimo. Se T é acíclico e conecta todos os vértices, formando uma árvore, então T é chamado árvore geradora, uma vez que "abrange" todo o grafo G (CORMEN, 2002, p. 561, tradução nossa). Esse problema também é conhecido como árvore de extensão mínima, árvore de extensão de peso mínimo e *minimum spanning tree* (MST). A Figura 11 mostra uma árvore geradora mínima de um grafo ponderado.

Figura 11. As arestas em destaque representa a MST.



Fonte: MEIDANIS, 2012[6]

Visto que a MST determina o custo mínimo para acessar todos os vértices de um dado grafo, ao tentar expandir um nó é verificado se a soma da MST dos vértices ainda não pertencentes a solução com o custo da solução parcial não extrapola a função limitante. Isso garante que, até o momento, a solução não excederá um custo ótimo conseguido até então.

Existem três algoritmos clássicos eficientes para a construção de MST's são os de Kruskal, Prim e Boruvka (SKIENA, 2008; PETTIE; RAMACHANDRAN, 2002).

O Algoritmo de Prim inicia-se por um vértice arbitrário v e repetidamente busca uma aresta de custo mínimo que conecta um novo vértice a essa árvore (SKIENA, 2008, p. 485, tradução nossa).

[6] Disponível em <http://www.ic.unicamp.br/~meidanis/courses/mc558/2012s2/P2>. Acesso em maio de 2017

O Algoritmo de Boruvka consiste em que cada aresta de custo mínimo conectado em a vértice deve estar na árvore geradora mínima. A união dessas arestas resultará em uma floresta geradora de no máximo $n/2$ árvores. Para cada uma dessas T árvores, seleciona-se uma aresta (x,y) de custo mínimo, tal que $x \in T$ e $y \notin T$. Cada uma dessas arestas deve estar novamente em uma MST, e a união resulta em uma floresta geradora com pelo menos metade de árvores (SKIENA, 2008, p. 486, tradução nossa).

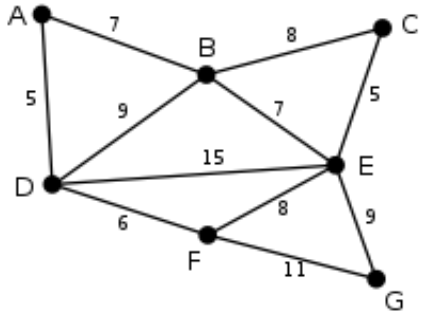
O Algoritmo de Kruskal inicia com cada vértice sendo uma árvore distinta. Essas árvores são unificadas por repetidamente verificar uma aresta de custo mínimo que conecte duas árvores diferentes sem formar ciclos (SKIENA, 2008, p. 485, tradução nossa).

Os passos base do Algoritmo de Kruskal são:

1. Criar uma floresta F , onde cada vértice é um árvore;
2. Criar um conjunto S contendo as arestas do grafo;
3. Enquanto S for não-vazio, faça:
    a. Escolher uma aresta de peso mínimo de S;
    b. Se essa aresta conecta duas árvores diferentes, unifica as duas árvores;
    c. Repete passo 3.

A Figura 12 é um grafo conexo, não orientado e ponderado onde deseja-se encontrar uma árvore geradora mínima. A Figura 13 mostra a construção da MST para cada iteração do Algoritmo de Kruskal.

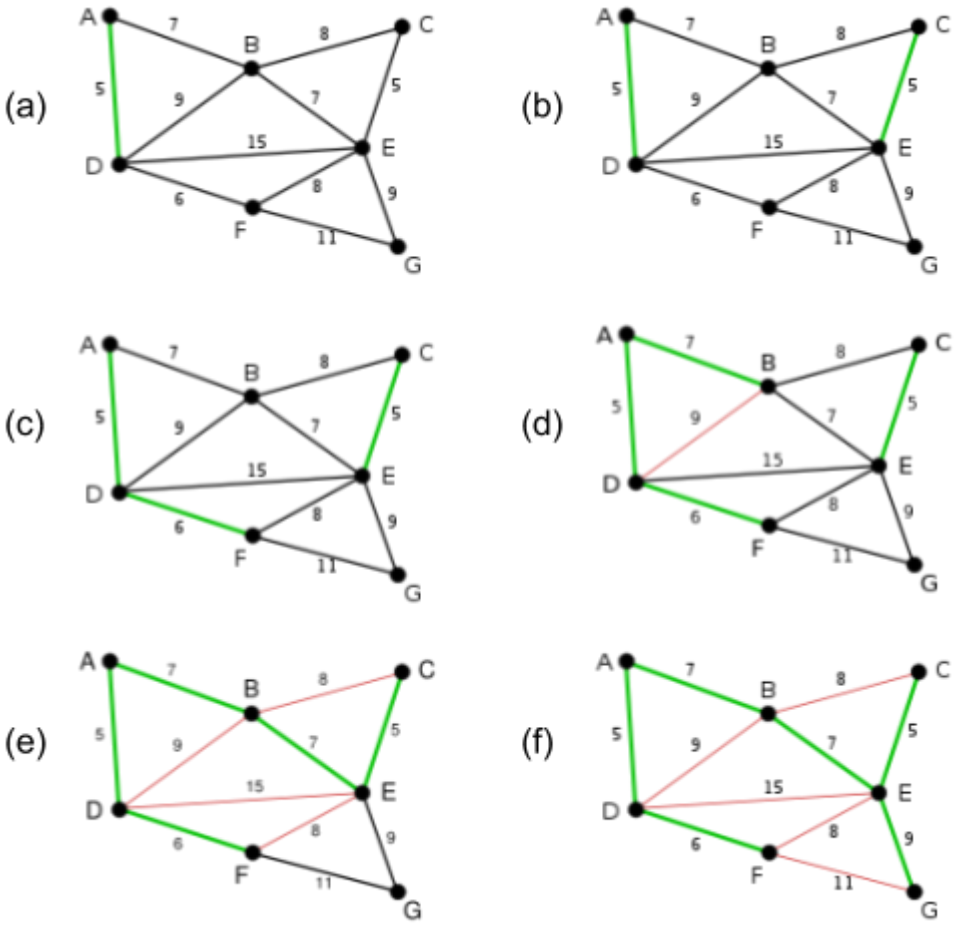
Figura 12. Representação de grafo ponderado para ser processado pelo algoritmo de Kruskal.



Fonte: Wikipedia[7]

[7] Disponível em <https://pt.wikipedia.org/wiki/Algoritmo_de_Kruskal>. Acesso em maio de 2017

Figura 13. Representação do processamento do algoritmo de Kruskal.



Fonte: Wikipedia[8]

Para este trabalho foi escolhido o Algoritmo de Kruskal numa implementação que consome V^3, no máximo, unidades de tempo, sendo V o número de vértices. Utilizando outras estruturas, o algoritmo consegue calcular em até $E \log V$ unidades de tempo (CORMEN, 2002, p. 561).

Algoritmo 4. Cálculo da MST utilizando o algoritmo de Kruskal

algKruskal(Grafo G, Lista Caminho)
1. arv[1...tam(G)] ← *0*
2. **para** *v* **de** 0 até tam(*G)* faça
3. **se** *v não estiver em Caminho* **então**
4. *orig* ← *v*
5. **fim se**
6. **fim para**
7. *pai[orig]* ← *orig*

[8] Disponível em <https://pt.wikipedia.org/wiki/Algoritmo_de_Kruskal>. Acesso em maio de 2017

```
8.  enquanto verdadeiro faça
9.      para v de 0 até tam(G) faça
10.         para cada d adjacente a v faça
11.             se arv[d] diferente de arv[d] E menorPeso > custo(d) então
12.                 menorPeso = custo(d)
13.                 orig = i
14.                 dest = d
15.                 finaliza laço para
16.             fim se
17.         fim para
18.     fim para
19.     para v de 0 até tam(G) faça
20.         se arv[v] = arv[dest] então
21.             arv[v] ← arv[orig]
22.         fim se
23.     fim para
24. fim enquanto
25. total ← calculaCustoMST()
devolva total
```

A função *algKruskal* recebe uma Grafo G conexo com os custo. No início do algoritmo cria-se uma floresta, ou seja, cada vértice é uma árvore independente e seleciona um vértice para iniciar a geração da árvore. A partir do vértice inicial é feita uma busca gulosa, ou seja, verifica-se o vértice mais próximo que ainda não está na árvore. Este vértice mais próximo é conectado ao vértice inicial e passam a ser uma árvore. Caso o vértice conectado já faça parte de uma árvore com outros vértices, todos os vértices passam a fazer parte da nova árvore. O algoritmo termina quando todos os vértices estiverem na mesma árvore. O vetor “pai”, ou em algumas implementações “chefe” é utilizado para definir quem é o vértice original da árvore, ou o vértice que conecta uma árvore a outra. A rotina calculaCustoMST() calcula o custo total da MST.

se*(Não* $d \subset$ *Caminho*

E *custo(v,d) + custo(Caminho) < custoMinimo*

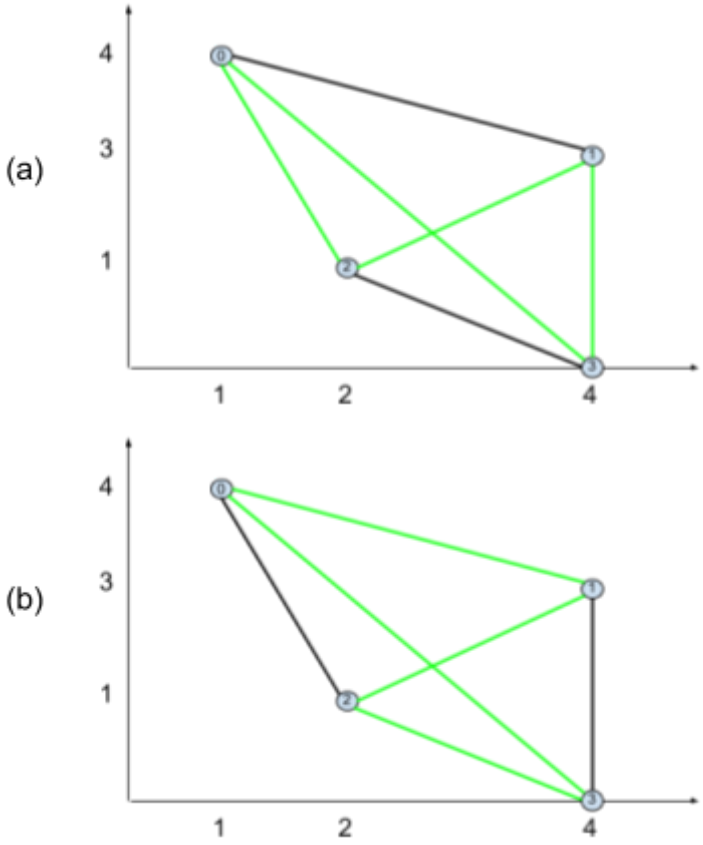
E *algKruskal(G,Caminho) + custo(Caminho) < custoMinimo*) então

3.3.2. Intersecção de arestas

Outra heurística é verificar se a adição de um novo vértice na solução não provocará intersecção entre arestas. Intersecções provocam circuitos maiores. Por

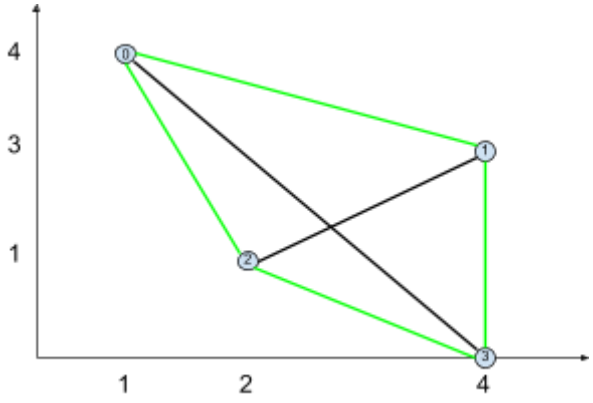
exemplo, a Figura 14 utiliza o grafo dado na Seção 3.1.1.1 mostrando o custo de circuitos com intersecção de arestas. A Figura 15 mostra um circuito sem intersecção de arestas. O custo do circuito sem o intersecção é menor.

Figura 14. Intersecção de arestas. (a) Custo do circuito é 13,98. (b) Custo do circuito é 13,21



Fonte: Elaborado pelo autor

Figura 15. Sem Intersecção de arestas. O circuito tem custo 11,55.

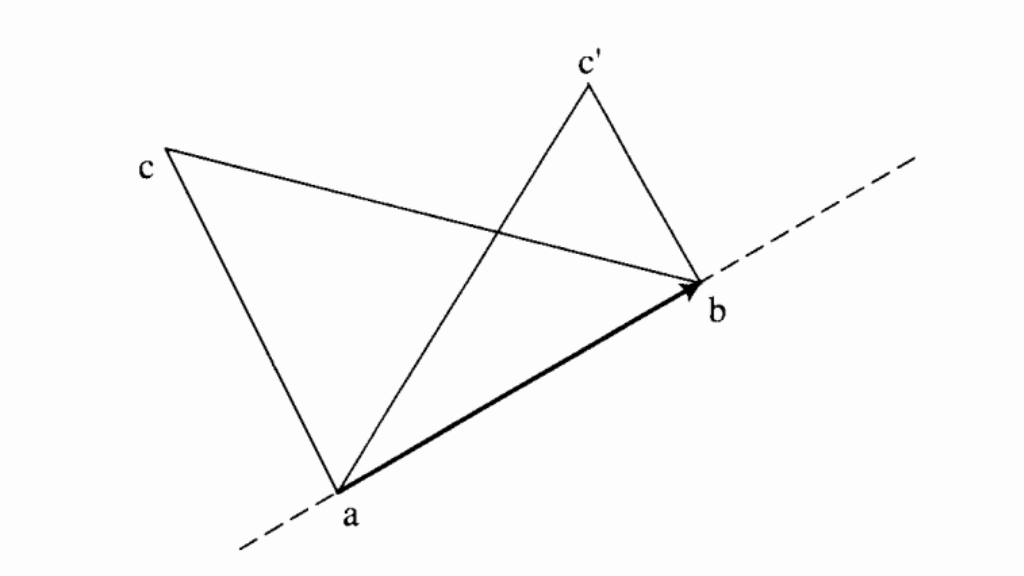


Fonte: Elaborado pelo autor

A existência de intersecção entre dois segmentos pode ser decidida usando o primitiva *Esquerda*, que determina se um ponto está ou não a esquerda de um segmento (O'ROURKE, 2004).

Um segmento é determinado por dois pontos dados em uma ordem particular (a, b). Se o ponto c está a esquerda de um segmento determinado por (a, b), então a tripla (a, b, c) forma um circuito anti-horário (O'ROURKE, 2004, p. 28, tradução nossa).

Figura 16. O ponto c à esquerda do segmento ab.



Fonte: O'rourke, 2004, p. 29

Dadas as coordenadas dos vértices, a área do triângulo $\triangle abc$ é positiva, visto que forma um circuito anti-horário. Se o vértice c estiver a direita de (a, b) então a área será negativa (O'ROURKE, 2004, p. 28).

Se dois segmentos ab e cd se interceptam, então c e d são divididos por uma linha L_1 contendo ab, isto é c está de um lado da linha e d está de outro. Da mesma forma, a e b são divididos por L_2, contendo cd. Apenas uma dessas condições não é o suficiente para garantir a intersecção entre os segmentos, mas a é suficientemente garantido com a combinação delas (O'ROURKE, 2004, p. 28). Então a primitiva Esquerda é combinada de forma que seja possível decidir se há ou não intersecção entre dois segmentos. A equação abaixo representa essa combinação:

$$S = (esquerda(a,b,c) \oplus esquerda(a,b,d)) \wedge (esquerda(c,d,a) \oplus esquerda(c,d,b))$$

A Tabela 3 mostra o resultado lógico da equação. Considerar $esquerda(a,b,c) = m$, $esquerda(a,b,d) = n$, $esquerda(c,d,a) = o$ e $esquerda(c,d,b) = p$.

Tabela 3. Tabela verdade da equação S.

m	n	o	p	$m \oplus n = q$	$o \oplus p = r$	$q \wedge r$	S
0	0	0	0	0	0	0	0
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	0
0	0	1	1	0	0	0	0
0	1	0	0	1	0	0	0
0	1	0	1	1	1	1	1
0	1	1	0	1	1	1	1
0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	0
1	0	0	1	1	1	1	1
1	0	1	0	1	1	1	1
1	0	1	1	1	0	0	0
1	1	0	0	0	0	0	0
1	1	0	1	0	1	0	0
1	1	1	0	0	1	0	0
1	1	1	1	0	0	0	0

Fonte: Elaborado pelo autor.

Algoritmo 5. Verificador de intersecção entre dois segmentos

verificaIntersecção(*Lista Caminho,Grafo g, vertice*)

1. *ultimo_Vertice ← ultimoVertice(Caminho)*
2. *xA ← coordenadaX(g, ultimo_Vertice)*
3. *yA ← coordenadaY(g, ultimo_Vertice)*
4. *xB ← coordenadaX(g, vertice)*

```
    5.  yB ← coordenadaY(g, vertice)
    6.  se tam(Caminho) < 3 então devolva falso
    7.  para todos os vértices v de Caminho faça
    8.      xC ← coordenadaX(g, v)
    9.      yC ← coordenadaY(g, v)
    10.     xD ← coordenadaX(g, verticeApos(v, Caminho))
    11.     yD ← coordenadaX(g, verticeApos(v, Caminho))
    12.     se intersecção(xA,yA,xB,yB,xC,yC,xD,yD) então
    13.         devolva verdadeiro
    14.     fim se
    15. fim para
devolva falso
```

A função *verificaInterceptacao* inicia armazenando as coordenadas dos vértices. Depois verifica se existem apenas dois vértices, se sim com certeza os vértices não se interceptam. Depois, cada aresta em que consiste o caminho é analisada em relação ao último vértice do caminho e o vértice candidato.

Algoritmo 6. Verifica intersecção entre dois dados segmentos

```
intersecção(xA,yA,xB,yB,xC,yC,xD,yD)
        1. devolva esquerda(xC,yC,xD,yD,xA,yA) XOU
           esquerda(xC,yC,xD,yD,xB,yB)) E (esquerda(xA,yA,xB,yB,xC,yC)
           XOU esquerda(xA,yA,xB,yB,xD,yD))
```

A função *intersecção* verifica se um dado segmento AB e um segmento CD, onde se A estiver de um lado de CD e B do outro, e C estiver de um lado de AB e D do outro há interceptação

Algoritmo 7. Primitiva *Esquerda*

```
esquerda(x1, y1, x2, y2, x3, y3)
        1. devolva (((x2 - x1) * (y3 - y1) - (y2 - y1) * (x3 - x1))>=0)
```

A função *esquerda* é a primitiva que decide se dadas as coordenadas dos vértices, o vértice (x3,y3) está a esquerda do segmento([x1,y1][x2,y2]).

A função *verificaInterceptacao* devolve *falso* caso não exista interceptação.

se*(Não* d ⊂ *Caminho*

E *custo(v,d) + custo(Caminho) < custoMinimo*

E *algKruskal(G,Caminho) + custo(Caminho) < custoMinimo*

E *Não verificaInterceptacao(Caminho, G, d)*) então

Assim, caso o resultado lógico acima seja verdadeiro, o vértice é inserido à solução. Para mais informações sobre a implementação vide Apêndice A.

4. RESULTADOS EXPERIMENTAIS

Para testes foram utilizadas as instâncias da Biblioteca TSPLIB que pode ser encontrada no endereço http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/. No mesmo endereço podem ser encontrados os custos mínimos para cada instância.

4.1. Tipos de instâncias

No documento *TSPLIB 95 (Gerhard Reinelt, Universit¨at Heidelberg)* encontrado no endereço acima especificado, apresenta o padrão dos arquivos para leitura do algoritmo. Os tipos de instâncias são:

- EXPLICIT, onde os custos são listados de forma explícita;
- EUC_2D, custos são distâncias euclidianas em 2D;
- EUC_3D, custos são distâncias euclidianas em 3D;
- MAX_2D, custos são distâncias máximas em 2D;
- MAX_3D, custos são distâncias euclidianas em 3D;
- MAN_2D, custos são distâncias Manhattan em 2D;
- MAN_3D, custos são distâncias Manhattan em 3D;
- CEIL_2D, custos são distâncias euclidianas arredondadas para cima em 2D;
- GEO, custos são distâncias geográficas;
- ATT, função especial de distâncias para problemas att48 e att532;
- XRAY1, função especial de distâncias para problemas cristalografia
- XRAY2, função especial de distâncias para problemas cristalografia
- SPECIAL Função especial de distância especificada em outro documento.

Para o algoritmo implementado ter um desempenho otimizado são aceitos apenas os tipos EUC_2D e GEO onde são dadas as coordenadas dos vértices.

As funções que calculam os custos das arestas foram implementadas de acordo com o documento acima citado(Vide Tópico *2. The distance functions*).

4.2. Ambiente de testes

Para o teste do algoritmo foi utilizado o seguinte ambiente computacional:

- Plataforma Windows 10 Home 64 bits
- Processador AMD A6 Quad Core 1.5 Ghz
- Memória: 6GB RAM

4.3. Resultados

A Tabela 4 mostra a quantidade de circuitos verificados com a aplicação das diferentes heurísticas analisadas neste trabalho. Para cada heurística implementada um número menor de circuitos são analisados, ou seja, são feitas podas mais efetivas na árvore de busca. A primeira heurística implementada é verificar se o custo da solução parcial com o vértice em análise não extrapola a função limitante. Chamaremos de S. A segunda heurística é verificar se a MST dos vértices que ainda não foram analisados mais o custo da solução parcial não extrapola a função limitante. Chamaremos de $sMST$. E a última heurística é verificar se há intersecção entre arestas com a inclusão de um novo vértice. Chamaremos de iV.

Tabela 4. Relação entre a aplicação de heurísticas e a quantidade circuitos verificados para uma instância de tamanho n.

		Quantidade de circuitos verificados		
Instância	n	S	$S + sMST$	$S + sMST + iV$
burma14	14	6 642	2 157	591
ulysses16	16	356 181	62 271	3 081
ulysses22	22	66 128 974	1 798 341	62 812
eil51	51	-	-	12 652

Fonte: Elaborado pelo autor

Tabela 5. Relação entre a aplicação de heurísticas e o tempo necessário de processamento para uma instância de tamanho n.

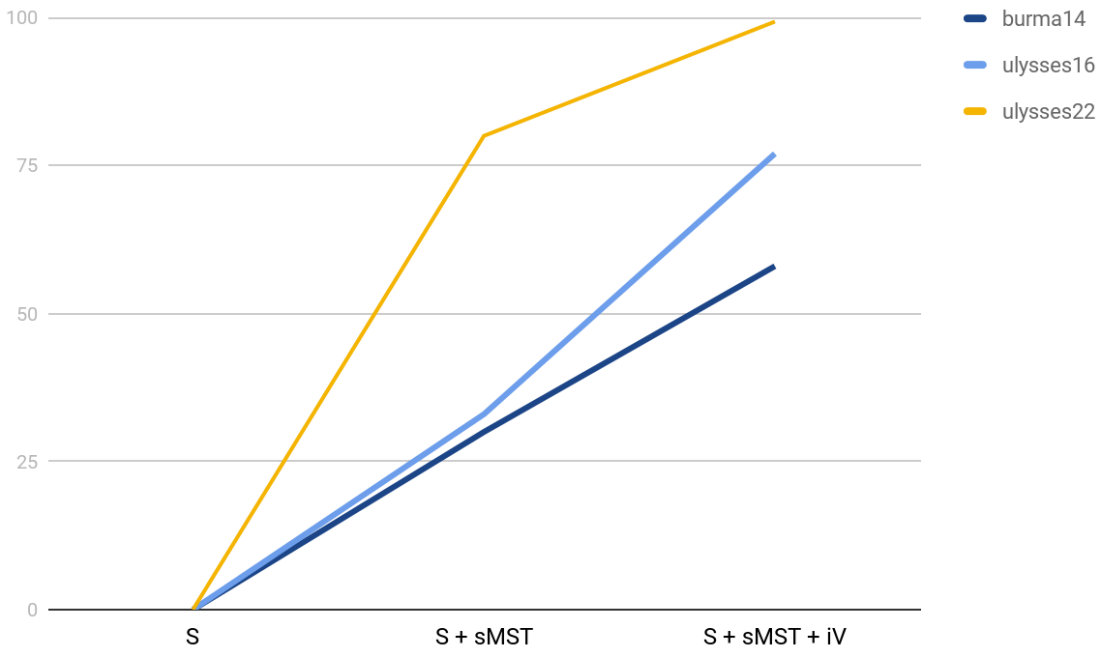
		Tempo de busca		
Instância	n	S	$S + sMST$	$S + sMST + iV$
burma14	14	3s	2s	0.7s
ulysses16	16	7s	5s	3s
ulysses22	22	3h 26min	40min	1min 30s
eil51	51	-	-	10 dias

Fonte: Elaborado pelo autor

Os resultados apresentados nas tabelas acima consideram a execução completa do algoritmo, ou seja, até que todo o espaço de soluções seja analisado. Isso garante que a resposta final do algoritmo é a resposta ótima para a instância. Em alguns testes, a resposta ótima foi encontrada em pouco tempo de execução. Então pode ser vantajoso a parada da execução quando uma resposta boa seja encontrada. Dependendo do problema, pode ser criada uma série de critérios que identifique se uma solução boa já foi encontrada, ou pode ser parado depois de um determinado tempo de execução.

Como mostra a Tabela 5 o tempo para a análise total do espaço de soluções é consideravelmente reduzido. O Gráfico 1 mostra a melhora (em relação ao tempo) com a implementação de cada heurística. O tempo de execução do algoritmo apenas com a heurística S é a base para analisar a melhora. Por exemplo, a instância ulysses22 apresentou um tempo 80% melhor com a implementação de $S + sMST$ em relação a S. E 99,3% mais rápido com a implementação de $S + sMST + iV$ em relação a S. Note que quanto maior a complexidade da instância, mais efetivamente as heurísticas farão diferença no tempo de busca total.

Gráfico 1. Representação do percentual de melhora das heurísticas implementadas.



Fonte: Elaborado pelo autor.

5. CONCLUSÕES

Esta seção tem por objetivo dar ao leitor uma visão do que podemos esperar do estudo de otimização, mostrar estudos futuros com base no algoritmo tratado nas Seções anteriores e incentivá-lo a contribuir com pesquisas na área.

5.1. Considerações

Fica claro que o algoritmo implementado não resolve a questão $P = NP?$ e nem é este o objetivo do trabalho. Mas podemos ver que chegamos a soluções ótimas com um tempo de busca muito inferior ao da força bruta. Apesar de inviável com instâncias muito grandes, como no caso do *eil51,* o tempo se tornou aceitável em comparação com algoritmos clássicos de busca exaustiva (sem o uso de heurísticas). Para instâncias muito maiores que isso o algoritmo se torna incapaz de resolver em tempo aceitável. Para propósito de pesquisa e conceituação o algoritmo se tornou aceitável. Isso demonstra que alguns problemas podem ser minimizados utilizando o poder de heurísticas efetivas e bem estudadas. Deve ser levado em consideração que computadores com maior capacidade de processamento levam um tempo menor para processar o algoritmo. No entanto, essa diminuição do tempo é relativa a apenas uma constante, a classe de complexidade do problema não é alterada

Para métodos que buscam sempre a solução ótima para o problema em discussão, a grande massa de autores, especialistas e estudiosos acreditam veementemente que não há algoritmos que resolvam em tempo polinomial. Em contrapartida a maioria dos algoritmos que hoje resolvem de maneira consistente e eficiente instâncias com mais de 1000 vértices são algoritmos probabilísticos que nem sempre gerarão soluções ótimas. Só o tempo dirá se é possível soluções ótimas em tempo polinomial.

5.2. Trabalhos futuros

Os trabalhos futuros, baseados nesse trabalho de graduação, consistem em aprimorar o que foi aqui abordado. Uma melhora que pode ser feita para maximizar o poder de solução em tempo aceitável é implementar um algoritmo que calcule uma MST em tempo muito inferior. Como abordado na Seção 3.5.1 o código

implementado gasta V^3 unidades de tempo para ser processado e, como mencionado nessa mesma Seção, é possível reduzir para $E \log V$ unidades de tempo (CORMEN, 2002, p. 561, tradução nossa).

Também podem ser implementadas outras heurísticas de poda como, por exemplo, um mecanismo que detecta prematuramente que uma determinada ramificação irá gerar um circuito já analisado. Desta forma, podemos reduzir pela metade a quantidade de circuitos analisados.

5.3. Considerações pessoais

Durante a pesquisa para a elaboração desta monografia, tive a oportunidade de estudar métodos de resolução do dado problema de diversas formas. Alguns autores perceberam que é viável observar a natureza e tentar aplicar, em grau relativo, sua sabedoria. Isso acontece nos Algoritmos genéticos (HOLLAND , 1975), no Algoritmo da Colônia de Formiga (DORIGO; GAMBARDELLA, 1996), no Arrefecimento Simulado (KIRKPATRICK, 1983) e Computadores de DNA (ADLEMAN , 1994). Tanto Algoritmos Genéticos como os Computadores de DNA têm como base a incrível capacidade que o DNA dos seres vivos possuem de adaptabilidade. O Algoritmo da Colônia de Formiga se baseia num processo natural instintivo das formigas para realizar seu trabalho. O Arrefecimento Simulado se baseia em um processo físico laboratorial, entretanto natural. Muitos autores têm estudado mais profundamente processos naturais ainda mais complexos e aplicando os resultados a Computação Biológica. Concluímos, logicamente, que os humanos precisam estudar ainda mais os complexos processos naturais. Neste aspecto, fica fácil concluir a mesma coisa que Salomão concluiu ao escrever : “Dediquei-me a adquirir sabedoria e a observar toda a atividade realizada na terra, e até mesmo fiquei sem dormir, dia e noite. Então eu considerei todo o trabalho do verdadeiro Deus [Jeová][9] e percebi que a humanidade não é capaz de compreender o que acontece debaixo do sol. Por mais que os homens tentem, não podem compreender. Mesmo que digam que são bastante sábios para entender, não são capazes de compreender.” (BÍBLIA, Eclesiastes, 8, 16-17).

[9] Não consta no texto original. Foi adicionado pelo autor deste trabalho.

REFERÊNCIA BIBLIOGRÁFICA

[ED] Aula 112 - Grafo: Árvore Geradora Mínima. Disponível em <https://www.youtube.com/watch?v=eHC2tjQPX3A>. Acesso em 23 de novembro de 2016.

ADLEMAN, Leonard M. Molecular computation of solutions to combinatorial problems. **Nature**, v. 369, p. 40, 1994.

AGRAWAL, Manindra; KAYAL, Neeraj; SAXENA, Nitin. Annals of Mathematics. **PRIMES is in P**, v. 160, p. 781-798, 2004.

Algoritmo do vizinho mais próximo. Disponivel em<https://pt.wikipedia.org/wiki/Algoritmo_do_vizinho_mais_próximo>. Acesso em 23 de novembro de 2016.

APPLEGATE, David, et. al. **The Traveling Salesman Problem: A Computational Study**, 2006.

APPLEGATE, D.L., BIXBY, R.E., CHV'ATAL, V. & COOK, W.J. **Implementing the Dantzig– Fulkerson–Johnson algorithm for large scale traveling salesman problems**. Math Program Ser B Vol. 97, pp. 91–153, 2003.

BIANCHI, Leonora; DORIGO, Marco; GAMBARDELLA, Luca Maria; GUTJAHR, Walter J. **A survey on metaheuristics for stochastic combinatorial optimization**. 2009.

BÍBLIA. A. T. Eclesiastes. In: BÍBLIA. Português. **Tradução do Novo Mundo da Bíblia Sagrada**. Associação Torre de Vigia de Bíblias e Tratados. São Paulo, 2015. p. 949-950.

BLAND, Robert G.; SHALLCROSS, David F. Large travelling salesman problems arising from experiments in X-ray crystallography: a preliminary report on computation. **Operations Research Letters**, v. 8, n. 3, p. 125-128, 1989.

BLUM, C.; ROLI, A. **Metaheuristics in combinatorial optimization: Overview and conceptual comparison**. 2003.

Busca em profundidade. Disponivel em <https://pt.wikipedia.org/wiki/Busca_em_ profundidade>. Acesso em 23 de novembro de 2016.

CASTONGUAY, Diane. **Tempo Polinomial: Verificação de tempo polinomial**. Disponível em:<http://www.inf.ufg.br/~diane/PAA/2006/AULA2006/VerificacaoPolinomial.pdf>. Acesso em: 16 maio 2017.

CHV'ATAL, V. **Edmonds polytopes and weakly Hamiltonian graphs**. Mathematical Programming, Vol. 5, pp. 29–40, 1973.

CLARKE, G. & WRIGHT, J., **Scheduling of vehicles from a central depot to a number of delivery points.** Operations Research. 1964.

CLAUSEN, Jens. **Branch and bound algorithms-principles and examples**. Department of Computer Science, University of Copenhagen, p. 1-30, 1999.

CONWAY, Richard Walter; MAXWELL, William L.; MILLER, Louis W. **Theory of scheduling**. Edição de Courier Dover Publications, 2003.

COMU'EJOLS, G.; FONLUPT, J.; NADDEF, D. **The travelling salesman problem on a graph and some related integer polyhedra.** Mathematical Programming, Vol. 33, pp. 1–27, 1985.

COPPIN, Ben. **Artificial intelligence illuminated**. 1. ed. Londres: Jones & Bartlett, 2004. 768 p.

CORBERÁN, A; et. al. **Heuristic Solutions to the Problem of Routing School Buses with Multiple Objectives**.

CORNUÉJOLS, Gérard; FONLUPT, Jean; NADDEF, Denis. **The traveling salesman problem on a graph and some related integer polyhedra**. Mathematical programming, v. 33, n. 1, p. 1-27, 1985.

CORMEN, Thomas H. **Introduction to Algorithms**. MIT Press, 2002.

DANTZIG, G.B.; FULKERSON, D.R. & JOHNSON, S.M. (1954). **Solution of a large-scale traveling salesman problem**. Operations Research, Vol. 2, pp.393–410, 1954.

DAVENDRA, Donald. Traveling Salesman Problem. **Theory and Applications, URL: http://www. intechopen. com**, 2010.

DREISSIG, W.; UEBAEH, W. **Personal communication. Linear programming with pattern constraints**. 1990. Tese de Doutorado. PhD thesis, Department of Economics, Harvard University, Cambridge, MA.

DORIGO, M.; GAMBARDELLA, L. M. **Ant colonies for the travelling salesman problem**. Elsevier Science Ireland Ltd, 1996.

FEO, Thomas A.; RESENDE, Mauricio G. C. Greedy randomized adaptive search procedures. **Journal of global optimization**, v. 6, n. 2, p. 109-133, 1995.

FEOFILOFF , Paulo. **Complexidade Computacional e Problemas NP-Completos**. Disponível em <https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/NPcompleto2.html> Acesso em 23 de novembro de 2016.

FREITAS, **Operações Primitivas.** Disponivel em<https://www.ime.usp.br/~freitas/gc/primitivas.html> Acesso em 23 de novembro de 2016.

FREITAS, **Intersecção entre dois segmentos.** Disponível em<https://www.ime.usp.br/~freitas/gc/intersec2.html>. Acesso em 23 de novembro de 2016.

GABOW, H. N.; et. al. **Efficient algorithms for finding minimum spanning trees in undirected and directed graphs**. 1986.

GALBIER, F.; LORENA, L. A. N. **Uma Heurística Construtiva aplicada a Problemas de Roteamento de Veículos**. 2004.

GILBERT, E.N. , POLLAK H.O. **Steiner minimal trees.** *SIAM J. Appl. Math.* , 1968.

GLOVER, F.; LAGUNA, M. **Tabu Search Kluwer**. Boston, MA, 1997.

GOMORY, R.E. **An algorithm for integer solutions to linear programs**. In: Graves RL & Wolfe P (eds). Recent Advances in Mathematical Programming. McGraw-Hill: New York, pp. 269–302, 1963.

GOLDBERG, David E.; HOLLAND, John H. Genetic algorithms and machine learning. **Machine learning**, v. 3, n. 2, p. 95-99, 1988.

GRÖTSCHEL, M., PULLEYBLANK, W.R. **Clique tree inequalities and the symmetric Travelling Salesman Problem**. Mathematics of Operations Research, Vol. 11, No. 4, (November, 1986), pp. 537–569, 1986.

GRÖTSCHEL, M. & HOLLAND, O. **Solution of Large-scale Symmetric Traveling Salesman Problems. Mathematical Programming**, Vol. 51, pp.141-202, 1991.

GUEDES, Allison da Costa Neves; LEITE, Jéssica Neiva de Figueiredo; ALOISE, Dário José. Um algoritmo genético com infecção viral para o problema do caixeiro viajante. **Revista Online de Iniciação Científica**, Natal, 2005.

HEURÍSTICA. In: **MICHAELIS**, Dicionário Michaelis da Língua Portuguesa, São Paulo, Editora Melhoramentos Ltda, 2015.

JOHNSON, David S.; MCGEOCH, Lyle A. The traveling salesman problem: A case study in local optimization. **Local search in combinatorial optimization**, v. 1, p. 215-310, 1997.

KIRKPATRICK, Scott et al. **Optimization by simulated annealing**. science, v. 220, n. 4598, p. 671-680, 1983.

LAND, A. H.; DOIG, A. G. **An automatic method of solving discrete programming problems**. Econometrica 28 (3): pp. 497-520, 1960.

LAND, A.H. **The solution of some 100-city traveling salesman problems**. Technical report, London School of Economics, London, 1979.

LEISERSON, Charles E.; SCHARDL, Tao B. **A Work-Efficient Parallel Breadth-First Search Algorithm (or How to Cope with the Nondeterminism of Reducers)**. ACM Symp. on Parallelism in Algorithms and Architectures, Cambridge, MA, 2010

LENSTRA, J.K,. & RINNOOY Kan, A.H.G. **Some simple applications of the traveling salesman problem**. Operational Research Quarterly, Vol. 26, pp. 717–33, 1975.

LINS, S. & B. W. Kernigham. **An Effective Heuristic Algorithm for the Traveling Salesman Problem, Operations Research**, v.21, p.498-516, 1973.

LITTLE, John D. C.; MURTY, Katta G.; SWEENEW, Dura W.; KAREL, Caroline. **An algorithm for the traveling salesman problem**. Operations Research, 1963.

LIU, Fuh-Hwa; SHEN, Sheng-Yuan. A method for vehicle routing problem with multiple vehicle types and time windows. **PROCEEDINGS-NATIONAL SCIENCE COUNCIL REPUBLIC OF CHINA PART A PHYSICAL SCIENCE AND ENGINEERING**, v. 23, p. 526-536, 1999.

MACULAN, N.; CAMPELLO, R. E. **Algoritmos e Heurísticas: Desenvolvimento e Avaliação de Performance**.[Sl]: Niterói. 1994.

MAR'I, R.; REINELT, G. **The Linear Ordering Problem, Exact and Heuristic Methods in Combinatorial Optimization**.

MARTIN, G.T. **Solving the traveling salesman problem by integer programming**. Working Paper, CEIR, New York, 1966.

MENGER, Karl. **Ergebnisse eines mathematischen Kolloquiums**. 3. ed. [S.l.]: Deuticke, 1932. 468 p.

MILIOTIS, P. Using cutting planes to solve the symmetric travelling salesman problem. **Mathematical programming**, v. 15, n. 1, p. 177-188, 1978.

NP-completeness. Disponivel em <https://en.wikipedia.org/wiki/NP-completeness> Acesso em 23 de novembro de 2016.

O'ROURKE , Joseph. **Computational Geometry in C** Cambridge University Press, 1993.

P versus NP problem. Disponivel em <https://en.wikipedia.org/wiki/P_versus_NP _problem>. Acesso em 23 de novembro de 2016.

PETTIE, Seth; RAMACHANDRAN, Vijaya. An optimal minimum spanning tree algorithm. **Journal of the ACM (JACM)**, v. 49, n. 1, p. 16-34, 2002.

PLANTE, Robert D.; LOWE, Timothy J.; CHANDRASEKARAN, R. The product matrix traveling salesman problem: an application and solution heuristic. **Operations Research**, v. 35, n. 5, p. 772-783, 1987.

PÓLYA, George, **How to Solve It: A New Aspect of Mathematical Method**, Princeton, NJ: Princeton University Press, 1945.

PRESTES, Edson . **Complexidade de Algoritmos**. Disponivel em <http://www.inf.ufrgs.br/~prestes/Courses/Complexity/aula27.pdf> Acesso em 23 de novembro de 2016.

RATLIFF, H. Donald; ROSENTHAL, Arnon S. Order-picking in a rectangular warehouse: a solvable case of the traveling salesman problem. **Operations Research**, v. 31, n. 3, p. 507-521, 1983.

ROTHLAUF, Franz. **Design of modern heuristics: principles and application**. Springer Science & Business Media, 2011.

SCHRIJVER, Alexander. On the history of combinatorial optimization (till 1960). **Handbooks in operations research and management science**, v. 12, p. 1-68, 2005.

SEDJELMACI ,Sidi Mohamed. **The Accelerated Euclidean Algorithm**. Eds. 2004, University of Cantabria, Santander, Spain, pp.283-287, 2004.

SIQUEIRA, Paulo Henrique; et.al.. **A new approach to solve the traveling salesman problem**. 2006.

SKIENA, Steven S. **The algorithm design manual: Text**. Springer Science & Business Media, 2008.

SOLOMON, M.M. **Algorithms for the Vehicle Routing and Scheduling Problems with Time Window Constraints**. Operations Research, 35(2): p. 254-265. 1987.

Travelling salesman problem. Disponivel em <https://en.wikipedia.org/wiki/Travelling_salesman_problem>. Acesso em 23 de novembro de 2016.

TSPLIB is a library of sample instances for the TSP. Disponivel em <http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/> . Acesso em 23 novembro 2016.

APÊNDICE A - Código fonte

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <limits.h>
#include <float.h>
#include <time.h>
#include <conio.h>
#include <string.h>

#define PRIMEIRO_VERTICE 0

#define R 6378.388
#define TO_RAD (3.1415926536 / 180)

int verticePartida=0;

typedef struct no { int destino; double custo; struct no *prox; }
*Lista;
typedef struct grafo { int tam;
                       Lista *item;
                       double *x;
                       double *y;
                     } *Grafo;

// Gera matriz de custos, cada um deles entre 1 e 1000

double *gera_matriz(int total) {
   int n = sqrt(total);
   double *g = malloc(total*sizeof(double));
   for(int i=0; i<n; i++)
      for(int j=i; j<n; j++)
         g[i*n+j] = g[j*n+i] = ( i==j ) ? 0 : 1+rand()%999;
   return g;
}

Lista no(int d, double c, Lista p) {
   Lista n = malloc(sizeof(struct no));
   n->destino = d;
```

```
   n->custo = c;
   n->prox = p;
   return n;
}

void insere(int d, double c, Lista *L) {
   if( *L==NULL || c<=(*L)->custo ) *L = no(d,c,*L);
   else insere(d,c,&(*L)->prox);
}

void exibe_vizinhos(Lista L) {
   printf("[");
   while( L ) {
      printf("(%d,%lf)",L->destino+1,L->custo);
      if( L->prox ) printf(",");
      L = L->prox;
   }
   printf("]");
}

void exibe_inversa(Lista L) {
   if( L==NULL ) return;
   exibe_inversa(L->prox);
   if( L->prox ) printf(",");
   printf("%d",L->destino+1);
}

double custo_ultima_aresta(Lista L, Grafo G) {
   int ultimo_vertice = L->destino;
   for(Lista p=G->item[ultimo_vertice]; p; p = p->prox)
      if( p->destino == verticePartida)
         return p->custo;
   return -1; // este comando nunca deve ser executado
}


void exibeCaminho(Lista L, Grafo G) {
   printf("\n[");
   exibe_inversa(L);
   printf("] : %lf",L->custo+custo_ultima_aresta(L,G));
}
```

```
//Apartir de um vértice dado, verifica o custo usando o algoritmo
do vizinho mais próximo
double custoCaminho(int vertice_inicio, int vertice_anterior, int
vertice_partida, int *vertices, Grafo g){
	double custo=0;
	int continua=1;
	for(Lista p = g->item[vertice_partida]; p; p = p->prox)
		if(p->destino == vertice_anterior) custo+=p->custo;
	while(continua){
		for(Lista p = g->item[vertice_partida]; p; p = p->prox){
			if(!vertices[p->destino]){
				custo+=p->custo;
				vertice_partida=p->destino;
				vertices[p->destino]=1;
				continua=1;
				break;
			}else{
				continua=0;
			}
		}
	}
	for(Lista p = g->item[vertice_partida]; p; p = p->prox)
		if(p->destino == vertice_inicio) custo+=p->custo;
	return custo;
}

//Procura melhor vértice para começar a busca;
//Dado um grafo procura o menor caminho a partir de cada vertice,
produzindo um único caminho por vertice
int buscaMelhorVertice(Grafo G){
	int melhorVertice=0;
	double melhorCaminho =DBL_MAX;
	int *vertices = malloc(G->tam*sizeof(int));
	//for(Lista p = c; p; p = p->prox) vertices[p->destino]=1;
	for(int i=0;i<G->tam;i++){
		for(int i=0;i<G->tam;i++) vertices[i]=0;
		vertices[i]=1;
		double novoCaminho = custoCaminho(i,-1,i,vertices,G);
		if(novoCaminho<melhorCaminho){
			melhorVertice=i;
```

```
                melhorCaminho = novoCaminho;
            }
        }
        free(vertices);
        return melhorVertice;
}

//Busca a arvore geradora mínima utilizando o algoritmo de Kruskal

double algKruskal(Grafo g, int *pai){
        int i,j, orig, dest, NV=g->tam;
        double total=0;
        int *arv = (int *) malloc(NV*sizeof(int));
        for(i=0;i<NV;i++){
            arv[i] = i;// Cada vértice sem pai está na sua própria
árvore
            if(pai[i]!=-2) orig=i;
        }
        pai[orig] = orig;
        while(1){
            double menorPeso = DBL_MAX;
            for(i=0;i<NV;i++){//percorre os vértices
                if(pai[i] == -2) continue; // Pula os vértices que
não participam na MST
                for(Lista p = g->item[i]; p; p =
p->prox){//arestas

                    if(arv[i] != arv[p->destino] && menorPeso >
p->custo && pai[p->destino]!= -2){
                        menorPeso = p->custo;
                        orig = i;
                        dest=p->destino;
                        break;// Como as arestas já estão
organizadas em ordem crescente, a primeira vez que essa condição é
executada, tem-se certeza que é o mínimo custo
                    }
                }
            }
            if(menorPeso == DBL_MAX) break;
            if(pai[orig] == -1) pai[orig]=dest;
            else pai[dest]=orig;
```

```
            for(i=0;i<NV;i++){
                  if(arv[i] == arv[dest]){
                        arv[i] = arv[orig];
                  }
            }
      }
      //Calcula o custo da MST
      for(int v=0;v<NV;v++){
            if(pai[v]==-2) continue;
            for(Lista p = g->item[v]; p; p = p->prox){
                  if(p->destino == pai[v] &&
pai[pai[v]]!=v)total+=p->custo;
                  else if(p->destino == pai[v] &&
v<pai[v])total+=p->custo;//Verificação para não somar arestas já
somadas
            }
      }
      free(arv);
      for(i=0;i<NV;i++)if(pai[i]!=-2)pai[i]=-1;
      return total;
}

//Dada a coordenada de dois pontos que formam segmento e de um
ponto isolado, retorna se o ponto está a esquerda do segmento.
int esquerda(double x1,double y1, double x2, double y2, double x3,
double y3){
      return (((x2 - x1) * (y3 - y1) - (y2 - y1) * (x3 - x1))>=0);
}

//Dadas as coordenadas dos pontos que formam dois segmento, retorna
se os segmentos se cruzam.
int InterceptaSegmentos(double xA,double yA,double xB, double
yB,double xC, double yC,double xD, double yD){
      return (esquerda(xC,yC,xD,yD,xA,yA) ^
esquerda(xC,yC,xD,yD,xB,yB)) & (esquerda(xA,yA,xB,yB,xC,yC) ^
esquerda(xA,yA,xB,yB,xD,yD));
}

//Dado um caminho percorrido, verifica se o vértice analisado forma
um segmento que sobrepõe uma aresta
int verificaInterceptacao(Lista Caminho,Grafo g,int vert, int
```

```
comprimento){
      int ultimo_Vertice = Caminho->destino;
      Lista c = Caminho->prox;
      double xA = g->x[ultimo_Vertice];
      double yA = g->y[ultimo_Vertice];
      double xB = g->x[vert];
      double yB = g->y[vert];
      if(comprimento<3) return 0;
      for(Lista p = c; p && p->prox ; p = p->prox){
            double xC = g->x[p->destino];
            double yC = g->y[p->destino];
            double xD = g->x[p->prox->destino];
            double yD = g->y[p->prox->destino];
            if(InterceptaSegmentos(xA,yA,xB,yB,xC,yC,xD,yD)) return
1;
      }
      return 0;
}

//Calculo para coordenadas Geográficas - Usando a Documentação de
cálculo do TSPLIB
double geo(double th1, double ph1, double th2, double ph2)
{

      double PI = 3.141592;
      int deg = th1;
      double min = th1 - deg;
      double latitude_x1 = PI * (deg + 5.0 * min / 3.0 ) / 180.0;
      deg = ph1;
      min = ph1 - deg;
      double longitude_y1 = PI * (deg + 5.0 * min / 3.0 ) / 180.0;

      deg = th2;
      min = th2 - deg;
      double latitude_x2 = PI * (deg + 5.0 * min / 3.0 ) / 180.0;
      deg = ph2;
      min = ph2 - deg;
      double longitude_y2 = PI * (deg + 5.0 * min / 3.0 ) / 180.0;

      double RRR = 6378.388;
      double q1 = cos( longitude_y1 - longitude_y2 );
```

```
      double q2 = cos( latitude_x1 - latitude_x2 );
      double q3 = cos( latitude_x1 + latitude_x2 );
      return RRR * acos( 0.5*((1.0+q1)*q2 - (1.0-q1)*q3) ) + 1.0;
}

int distanciaEuclideana(double x1, double y1, double x2, double
y2){
      double xd = x1 - x2;
      double yd = y1 - y2;
      return sqrt( xd*xd + yd*yd);
}


// O parâmetro n é a ordem da matriz (isto é, o número de vértices
no grafo)
// Se g é NULL, gera um grafo G a partir de uma matriz aleatória de
custos g
// Senão, gera um grafo G a partir dos valores que estão na matriz
de custos dados g

Grafo grafo(double *g, int n) {
   int aleatorio = 0;
   if( g==NULL ) {
      g = gera_matriz(n*n);
      aleatorio = 1;
   }
   Grafo G = malloc(sizeof(struct grafo));
   G->tam = n;
   G->item = malloc(n*sizeof(Lista));
   for(int i=0; i<n; i++) {
      G->item[i] = NULL;
      for(int j=0; j<n; j++)
         if( i!=j )
            insere(j,g[i*n+j],&G->item[i]);
   }
   if( aleatorio ) free(g);
   return G;
}

int pertence(int d,Lista L) {
   if( L==NULL ) return 0;
```

```
   if( d == L->destino ) return 1;
   return pertence(d,L->prox);
}

void exibeg(Grafo G) {
   for(int i=0; i<G->tam; i++) {
      printf("\n%d: ",i+1);
      exibe_vizinhos(G->item[i]);
   }
}

void busca(Lista Caminho, int comprimento, Grafo G, int *pai) {
   static double custo_minimo;
   static int quantidade=0;
   int *paiCopia = (int *) malloc(G->tam*sizeof(int));
   for(int i=0;i<G->tam;i++)paiCopia[i]=pai[i];
   if( comprimento == 1 ) custo_minimo = DBL_MAX;
   if( comprimento == G->tam ) {
        quantidade++;
        double custo_atual =
Caminho->custo+custo_ultima_aresta(Caminho,G);
      if( custo_atual<custo_minimo ) {
            exibeCaminho(Caminho,G);
            printf("\nTotal de caminhos percorridos para encontrar
a resposta: %d",quantidade);
            printf("\nHorario: %s\n",__TIME__);
         custo_minimo = custo_atual;
         return;
      }
   }
   int corrente = Caminho->destino;
   for(Lista p = G->item[corrente]; p; p = p->prox){
      if( !pertence(p->destino,Caminho)
             && p->custo+Caminho->custo < custo_minimo
             && algKruskal(G,paiCopia)+Caminho->custo <
custo_minimo
             &&
!verificaInterceptacao(Caminho,G,p->destino,comprimento)) {
         Lista c = no(p->destino,p->custo+Caminho->custo,Caminho);
            paiCopia[p->destino] =-2;
         busca(c,comprimento+1,G,paiCopia);
```

```
            free(c);
        }
        }
        free(paiCopia);
}

int main(void) {
        Grafo G;
        clock_t inicio;
        srand(time(NULL)); // inicia gerador de números aleatórios

        FILE *fd;

        char arquivo[20];
      char buf[20];
      char comment[50];
      char ewformat[20];
      char ewtype[20];
      int i, j, N;
      double aux,x,y;

      /* Parser do arquivo de entrada.
       * Interpreta dois tipos de arquivo: EXPLICIT/UPPER_ROW e
arquivo de coordenadas euclideanas EUC_2D
       * Considera invalido se o arquivo nao iniciar com a palavra
NAME */
        printf("Digite o nome do arquivo:");
        gets(arquivo);
      fd=fopen(arquivo,"r");
      fscanf(fd,"%s",buf);
      if(strcmp(buf,"NAME")!=0&&strcmp(buf,"NAME:")!=0){
          printf("O arquivo de entrada nao eh um arquivo TSP
valido\n");
          exit(0);
      }

      /* Parser
       * O laco while termina quando acabar de ler o cabecalho */
      while(1){
            fscanf(fd,"%s",buf);
            if(!strcmp(buf,"COMMENT:"))
```

```
            fgets(comment, 100, fd);
          else if(!strcmp(buf,"DIMENSION:")){
            fscanf(fd," %d",&N);
          }else if(!strcmp(buf,"COMMENT")){
            fscanf(fd," : ");
           fgets(comment, 100, fd);
          }
          else if(!strcmp(buf,"DIMENSION")){
            fscanf(fd," : %d",&N);
          }
          else if(!strcmp(buf,"EDGE_WEIGHT_FORMAT:")){
            fscanf(fd," %s",ewformat);
          }
          else if(!strcmp(buf,"EDGE_WEIGHT_FORMAT")){
            fscanf(fd," : %s",ewformat);
          }
          else if(!strcmp(buf,"EDGE_WEIGHT_TYPE:")){
            fscanf(fd," %s",ewtype);
          }
          else if(!strcmp(buf,"EDGE_WEIGHT_SECTION"))
            break;
          else if(!strcmp(buf,"NODE_COORD_SECTION"))
            break;
          else if(!strcmp(buf,"EOF")){
            printf("Erro no reconhecimento do arquivo\n");
            printf("O campo EDGE_WEIGHT_FORMAT ou
NODE_COORD_SECTION nao existe\n");
           exit(0);
          }
    }

    /* Imprime a linha de comentario extraida do arquivo de entrada
     * A expressao simbolica serve para eliminar um espaco em
branco deixado pelo parser no inicio */
    printf("\n%s",(comment[0]==' ')?comment+1:comment);
     printf("\n%s",ewtype);

     //Aloca matriz de custos
     double *m = malloc((N*N)*sizeof(double));

    /* Aloca memoria para os vetores e matrizes */
```

```
    double *xVert = malloc(sizeof(double) * N);
     double *yVert = malloc(sizeof(double) * N);
    if(!strcmp(ewtype,"GEO")){
           /* Arquivo de entrada tipo GEOGRAPHIC
           * obtem as distancias utilizando a latitute e longitude
fornecidas */
           for(i=0;i<N;i++)
           fscanf(fd,"%d %lf %lf\n",&aux,&(xVert[i]),&(yVert[i]));

           for(int i=0; i<N; i++){
                 for(int j=i; j<N; j++){
                       if(i==j){
                             m[i*N+j] = m[j*N+i]=0;
                       }else{
                             double valor =
geo(xVert[i],yVert[i],xVert[j],yVert[j]);
                             m[i*N+j] = m[j*N+i] = floor(valor);
                       }
                 }
           }
    }else if(!strcmp(ewtype,"EUC_2D")){
           /* Arquivo de entrada tipo EUC_2D
           * obtem as distancias utilizando a x e y fornecidas */
           for(i=0;i<N;i++)
           fscanf(fd,"%d %lf %lf\n",&aux,&(xVert[i]),&(yVert[i]));

           for(int i=0; i<N; i++){
                 for(int j=i; j<N; j++){
                       if(i==j){
                             m[i*N+j] = m[j*N+i]=0;
                       }else{
                             double valor =
distanciaEuclideana(xVert[i],yVert[i],xVert[j],yVert[j]);
                             m[i*N+j] = m[j*N+i] = floor(valor);
                       }
                 }
           }
    }
    printf("Distancias lidas com sucesso!\n");

     G = grafo(m,N);
```

```
      G->x=xVert; G->y=yVert;
      exibeg(G);
      inicio = clock();
      verticePartida = buscaMelhorVertice(G);
      int pais[N];
      for(i=0;i<=N;i++)pais[i]=-1;
      printf("\n\nCaminhos encontrados (o ultimo tem custo
minimo):\n");
      busca(no(verticePartida,0,NULL),1,G,pais);
      printf("\n\nTempo de busca: %.1f\n",
1.0*(clock()-inicio)/CLOCKS_PER_SEC);
      printf("\n\nPressione <enter>");
      getchar();
      _clrscr();

      return 0;
}
```