UNIOESTE - Universidade Estadual do Oeste do Paraná

CENTRO DE EXATAS E TECNOLÓGICAS

Colegiado de Ciência da Computação

Curso de Bacharelado em Ciência da Computação

Docente: Adriana Postal

Inteligência Artificial

Relatório de *Inteligência Artificial - Trabalho Busca*

Problema do Caixeiro Viajante

Busca em Profundidade

Subindo o Morro

Gabriel Santos da Silva
Leonardo Bednarczuk Balan de Oliveira

Cascavel
2023

unioeste
Universidade Estadual do Oeste do Paraná

I - Definição do problema escolhido

O problema escolhido para a implementação do trabalho é o **caixeiro viajante**. O problema do caixeiro viajante consiste em encontrar o caminho mais curto que consegue visitar cada um dos vértices exatamente uma vez e retornar ao vértice de origem. Para ser calculado/computado é necessário um grafo completo (ou possível de cálculo), com o custo associado a cada aresta, representando a distância entre os vértices.

O caixeiro viajante é considerado um problema NP-difícil, ou seja, significa que não há algoritmo eficiente conhecido que resolva todas as instâncias do problema em tempo polinomial. Isso implica que, à medida que o número de vértices aumenta, o tempo necessário para encontrar a solução ideal cresce de forma exponencial.

Além disso, o problema pode não ser passível de cálculo se o grafo for incompleto (não contendo todas as arestas possíveis entre os vértices) ou se houver custos negativos nas arestas, o que invalida a interpretação física do problema. Portanto, ao abordar esse problema, é fundamental garantir que as premissas do grafo estejam alinhadas com as técnicas de otimização escolhidas, e que elas sejam apropriadas para a resolução eficiente das instâncias consideradas. Podemos ver as restrições escolhidas pela nossa dupla para a implementação da resolução do problema no tópico abaixo ‘Restrições no Problema Implementado’.

II - Restrições no Problema Implementado

- Grafo completo ou passível de resolução: O grafo é completo quando há uma aresta conectada a cada par de vértices, ou seja, todos os vértices têm caminhos possíveis entre eles. Todavia, não é realmente necessário que o grafo seja completo, grafos não completos específicos também conseguem lidar com o problema do caixeiro viajante, dependendo do vértice de origem. Dessa forma, como em nosso algoritmo sempre começamos com o primeiro vértice lido do arquivo (explicação das escolhas de projeto no tópico abaixo) temos uma base de quais grafos podem ser resolvidos por ele. *Assim não especificamos que o grafo tem que ser completo, mas sim passível de cálculo.*
- Simetria das Distâncias/Grafo não direcionado: *Assumimos que as distâncias entre as cidades são simétricas/não direcionadas*, ou seja, a distância de A para B é a mesma que de B para A. Um requisito importante para o problema clássico do caixeiro viajante, que definimos em nosso projeto.

- Função Objetivo: *O objetivo é minimizar (menor caminho) a soma total das distâncias percorridas*. Expressamos isso como uma função objetivo a ser otimizada.
- Restrição de Visitas Únicas: *Cada cidade deve ser visitada exatamente uma vez durante o percurso*, assim como define o problema clássico do Caixeiro Viajante.
- Ciclo Hamiltoniano: A solução feita pelo algoritmo deve *formar um ciclo hamiltoniano*, ou seja, *um ciclo que visita cada vértice exatamente uma vez e volta ao vértice de origem*.
- Restrição do Nó/Vértice Inicial e Final: Especificamos no algoritmo sempre o vértice inicial, logo, *a solução encontrada deve voltar a esse nó inicial ao fim da execução (tornando-o o nó final também)*, fechando o ciclo e resolvendo o problema.
- Natureza NP-Completa: O Problema do Caixeiro Viajante é conhecido por ser um *problema NP-completo*, o que significa que *não há algoritmo eficiente conhecido para resolvê-lo em tempo polinomial para todas as instâncias do problema*. Dessa forma, tentamos otimizar ao máximo nossos algoritmos, mesmo sabendo dessa impossibilidade de se ter um algoritmo realmente eficiente para o problema.

III - Decisões de Projeto

- Entrada de vértices e arestas por arquivo: Definimos que os dados sobre os vértices e arestas são passados por 2 arquivos diferentes, um arquivo de vértices contendo um vértice por linha do arquivo, exemplo: ‘A’, e um arquivo de arestas contendo por linha ‘Vértice_Origem Vértice_Destino Custo’, exemplo: ‘A B 10’ (Do vértice A para o B, ou de B para A, o custo é 10).

- Grafos possíveis de cálculo passados por parte do usuário: Como a entrada dos dados é por arquivo, cabe ao usuário informar dois arquivos que juntos formarão um grafo possível de ser calculado pelo Problema do Caixeiro Viajante. Focamos em deixar nossos algoritmos com o princípio de responsabilidade única, ou seja, eles resolvem o problema com a Busca em Profundidade ou a Heurística Subindo o Morro e nada além disso, assim, tornando o algoritmo implementado puro e com um tempo de execução mais preciso. Caso sejam passados vértices e arestas impossíveis de cálculo, o algoritmo informa que não foi possível achar solução para o problema. Mas não precisa estresse, vários arquivos de vértices e arestas serão disponibilizados na pasta de nosso projeto, facilitando a execução dele.

- Vértice Inicial/Final: Escolhemos o vértice inicial/final sendo o primeiro do arquivo de vértices, por exemplo: o primeiro vértice/primeira linha do arquivo de vértices é ‘A’, o algoritmo vai resolver o Problema do Caixeiro Viajante iniciando e voltando para esse nó. Caso querer trocar o vértice de início, só inverter com algum outro do arquivo. Também, como os grafos de nosso problema não precisam ser necessariamente completos, escolhendo sempre um como inicial, não caímos na tribulação de sortear um vértice aleatoriamente e esse vértice não ser passível de cálculo do problema a partir dele.

- Sem Interface Gráfica: Nossas iterações, resultados finais e prints são todos exibidos no terminal intuitivamente. Escolhemos não utilizar interface gráfica devido a complexidade de implementação, e pela análise da saída do programa no terminal, onde notamos que ela é fácil de ser compreendida.

- Estrutura dos 2 Algoritmos Padronizada: Implementamos os 2 algoritmos com uma estrutura bem similar, buscando facilitar o entendimento dos 2 em conjunto. Todavia, como dividimos o foco da dupla entre os 2 algoritmos, cada código ficou um pouco diferente (o de Busca em Profundidade tem uma classe que engloba todas as funções; O Subindo o Morro tem as funções definidas sem uma classe as englobar). Mas isso, olhando na prática, não afetou praticamente em nada a estrutura dos códigos.

- Conceito de Iteração nos 2 algoritmos: No Busca em Profundidade temos cada iteração incrementando a cada visita de vértice novo na descida do algoritmo, ou seja, a medida que o caminho vai sendo montado, de vértice em vértice ‘iteração += 1’. Já no Subindo o Morro, cada iteração é uma permutação de caminho diferente gerada pelo algoritmo a partir do vértice inicial, assim, no funcionamento do código, cada caminho permutado gerado recebe ‘iteração += 1’.

- Linguagem Python Escolhida: Implementamos os 2 códigos em Python, visando a simplicidade e otimização.

IV - Algoritmos de busca

IV.1 - busca cega:

Como algoritmo de busca cega escolhemos o **algoritmo de profundidade**, esse algoritmo de busca é amplamente utilizado em inteligência artificial e

otimização de algoritmos. A exploração do espaço de busca começa a partir do nó raiz do problema, e a cada passo, o algoritmo se move para um nó filho não visitado. Seu diferencial é a priorização da exploração de um ramo completo antes de retroceder para explorar outros ramos. Isso significa que a busca irá terminar a execução o mais profundamente possível antes de iniciar uma nova verificação.

Porém a busca cega em profundidade pode ser ineficiente em cenários nos quais o caminho mais curto para a solução está localizado em um nível mais superficial da árvore de busca.

Seu Pseudocódigo é de simples entendimento. Seja ele:

```
##############################################
Função buscaEmProfundidade(estado):
   Se estado é o objetivo:
      Retorne sucesso
   Para cada ação possível a partir do estado:
      PróximoEstado = aplicarAção(estado, ação)

#recursiva:
      Se buscaEmProfundidade(PróximoEstado) == sucesso:
         Retorne sucesso
   Retorne falha
##############################################
```

Ao qual “estado” é o Estado atual, “objetivo” é onde desejamos chegar, “PróximoEstado” representa a ação de trocar de vértice e a verificação recursiva em que retorna ‘sucesso’ se o objetivo foi possível ou ‘falha’ se não for possível.

IV.2 - Busca Heurística:

Como algoritmo de busca Heurística escolhemos o **algoritmo subindo o morro**. Esse algoritmo é um algoritmo de otimização que visa encontrar a solução ótima movendo-se sempre em direção ao ponto mais alto/menos custoso que encontrar em um espaço de estados. Primeiramente o algoritmo parte de um estado qualquer, e a cada iteração, escolhe o melhor vizinho para a heurística mais alta, sempre considerando o vizinho escolhido como o novo estado atual. Esse processo é repetido até que não haja mais nenhum vizinho com um valor mais alto, indicando que o algoritmo atingiu um máximo local.

Porém a busca heurística subindo o morro pode ficar presa em máximos locais e não alcançar o máximo global, algumas técnicas podem ser utilizadas para tentar resolver isso, como a reinicialização aleatória do algoritmo para superar essa

limitação. Contudo, a eficácia desse algoritmo depende diretamente da heurística e do espaço de busca definido para o problema.

Seja seu Pseudocódigo:

```
##########################################
Função buscaHeuristicaSubindoOMorro(estadoInicial):
   estadoAtual = estadoInicial

Enquanto não atingir um máximo local:
      vizinhos = gerarVizinhos(estadoAtual)
      melhorVizinho = selecionarMelhorVizinho(vizinhos)

      Se valorHeuristico(melhorVizinho) <= valorHeuristico(estadoAtual):
         Retorne estadoAtual  // Atingiu um máximo local

      estadoAtual = melhorVizinho
   Retorne estadoAtual  // Atingiu o máximo local
##########################################
```

Ao qual o “estadoInicial” é estado em que desejamos iniciar a busca heurística, “gerarVizinhos” e “selecionarMelhorVizinho” geram os vizinhos possiveis do estado atual e escolhem o melhor entre eles. Por fim, uma verificação se o melhor vizinho é melhor que o estado atual é feita, se o estado atual possuir valor heurístico melhor então o estado atual é um melhor local.

V - Comparação entre os dois algoritmos

V.1 - parte prática

Arquivo ‘v1.txt’ com 5 vértices;
Arquivo ‘a1.txt’ com 10 arestas, formando um grafo completo.

Busca em Profundidade: Nos prints abaixo podemos ver a execução total do Busca em Profundidade para o problema do Caixeiro Viajante especificado pelos arquivos *‘v1.txt’* e *‘a1.txt’* (esses arquivos estão na pasta do projeto entregue pela dupla). Nota-se que colocamos os prints da execução total do algoritmo, pois nesse caso o problema ficou ‘pequeno’, todavia, para arquivos mais grandes, a geração do Busca em Profundidade fica bem extensa, assim não colocamos mais prints da execução total, mas sim dos resultados nos casos após esse.

unioeste
Universidade Estadual do Oeste do Paraná

Execução Total do Busca em Profundidade do problema acima:

```
>_ Console  Shell  +
Run
Visitando Vértice A - Caminho: [] - Custo: 0
Visitando Vértice B - Caminho: [0] - Custo: 10
Visitando Vértice C - Caminho: [0, 1] - Custo: 45
Visitando Vértice D - Caminho: [0, 1, 2] - Custo: 90
Visitando Vértice E - Caminho: [0, 1, 2, 3] - Custo: 150
Visitando Vértice E - Caminho: [0, 1, 2] - Custo: 95
Visitando Vértice D - Caminho: [0, 1, 2, 4] - Custo: 155
Visitando Vértice D - Caminho: [0, 1] - Custo: 40
Visitando Vértice C - Caminho: [0, 1, 3] - Custo: 85
Visitando Vértice E - Caminho: [0, 1, 3, 2] - Custo: 135
Visitando Vértice E - Caminho: [0, 1, 3] - Custo: 100
Visitando Vértice C - Caminho: [0, 1, 3, 4] - Custo: 150
Visitando Vértice E - Caminho: [0, 1] - Custo: 50
Visitando Vértice C - Caminho: [0, 1, 4] - Custo: 100
Visitando Vértice D - Caminho: [0, 1, 4, 2] - Custo: 145
Visitando Vértice D - Caminho: [0, 1, 4] - Custo: 110
Visitando Vértice C - Caminho: [0, 1, 4, 3] - Custo: 155
Visitando Vértice C - Caminho: [0] - Custo: 15
Visitando Vértice B - Caminho: [0, 2] - Custo: 50
Visitando Vértice D - Caminho: [0, 2, 1] - Custo: 80
Visitando Vértice E - Caminho: [0, 2, 1, 3] - Custo: 140
Visitando Vértice E - Caminho: [0, 2, 1] - Custo: 90
Visitando Vértice D - Caminho: [0, 2, 1, 4] - Custo: 150
Visitando Vértice D - Caminho: [0, 2] - Custo: 60
Visitando Vértice B - Caminho: [0, 2, 3] - Custo: 90
Visitando Vértice E - Caminho: [0, 2, 3, 1] - Custo: 130
Visitando Vértice E - Caminho: [0, 2, 3] - Custo: 120
Visitando Vértice B - Caminho: [0, 2, 3, 4] - Custo: 160
Visitando Vértice E - Caminho: [0, 2] - Custo: 65
Visitando Vértice B - Caminho: [0, 2, 4] - Custo: 105
Visitando Vértice D - Caminho: [0, 2, 4, 1] - Custo: 135
Visitando Vértice D - Caminho: [0, 2, 4] - Custo: 125
Visitando Vértice B - Caminho: [0, 2, 4, 3] - Custo: 155
Visitando Vértice D - Caminho: [0] - Custo: 20
Visitando Vértice B - Caminho: [0, 3] - Custo: 50
```

```
>_ Console  Shell  +
Run
Visitando Vértice B - Caminho: [0, 3] - Custo: 50
Visitando Vértice C - Caminho: [0, 3, 1] - Custo: 85
Visitando Vértice E - Caminho: [0, 3, 1, 2] - Custo: 135
Visitando Vértice E - Caminho: [0, 3, 1] - Custo: 90
Visitando Vértice C - Caminho: [0, 3, 1, 4] - Custo: 140
Visitando Vértice C - Caminho: [0, 3] - Custo: 65
Visitando Vértice B - Caminho: [0, 3, 2] - Custo: 100
Visitando Vértice E - Caminho: [0, 3, 2, 1] - Custo: 140
Visitando Vértice E - Caminho: [0, 3, 2] - Custo: 115
Visitando Vértice B - Caminho: [0, 3, 2, 4] - Custo: 155
Visitando Vértice E - Caminho: [0, 3] - Custo: 80
Visitando Vértice B - Caminho: [0, 3, 4] - Custo: 120
Visitando Vértice C - Caminho: [0, 3, 4, 1] - Custo: 155
Visitando Vértice C - Caminho: [0, 3, 4] - Custo: 130
Visitando Vértice B - Caminho: [0, 3, 4, 2] - Custo: 165
Visitando Vértice E - Caminho: [0] - Custo: 25
Visitando Vértice B - Caminho: [0, 4] - Custo: 65
Visitando Vértice C - Caminho: [0, 4, 1] - Custo: 100
Visitando Vértice D - Caminho: [0, 4, 1, 2] - Custo: 145
Visitando Vértice D - Caminho: [0, 4, 1] - Custo: 95
Visitando Vértice C - Caminho: [0, 4, 1, 3] - Custo: 140
Visitando Vértice C - Caminho: [0, 4] - Custo: 75
Visitando Vértice B - Caminho: [0, 4, 2] - Custo: 110
Visitando Vértice D - Caminho: [0, 4, 2, 1] - Custo: 140
Visitando Vértice D - Caminho: [0, 4, 2] - Custo: 120
Visitando Vértice B - Caminho: [0, 4, 2, 3] - Custo: 150
Visitando Vértice D - Caminho: [0, 4] - Custo: 85
Visitando Vértice B - Caminho: [0, 4, 3] - Custo: 115
Visitando Vértice C - Caminho: [0, 4, 3, 1] - Custo: 150
Visitando Vértice C - Caminho: [0, 4, 3] - Custo: 130
Visitando Vértice B - Caminho: [0, 4, 3, 2] - Custo: 165
```

Notamos que o algoritmo seguindo a abordagem de Busca em Profundidade vai visitando todos os caminhos possíveis, dessa forma, conseguindo chegar no melhor resultado:

```
====================================================

Tempo de Execução: 0.0017817020416259766 segundos
Número total de Iterações: 65

Menor caminho: ['A', 'C', 'D', 'B', 'E', 'A']
Custo do Caminho: 155

====================================================
```

Acima vemos o tempo de execução da busca, o número total de iterações que foi necessário para explorar todos os caminhos possíveis e encontrar o melhor resultado, e por fim o menor caminho com o seu custo.

Subindo o Morro:

```
Iteração 1: Melhor rota atual: ['A', 'B', 'C', 'D', 'E'],
Melhor custo atual: 175
Iteração 3: Melhor rota atual: ['A', 'B', 'D', 'C', 'E'],
Melhor custo atual: 160
Iteração 9: Melhor rota atual: ['A', 'C', 'D', 'B', 'E'],
Melhor custo atual: 155

====================================================

Tempo de execução: 0.00022840499877929688 segundos
Número total de iterações: 24

Menor Caminho: ['A', 'C', 'D', 'B', 'E', 'A']
Custo do Caminho: 155

====================================================
```

Na imagem acima podemos ver a execução completa do Subindo o Morro, juntamente com o print dos resultados finais que ficou com a estrutura padronizada entre os dois algoritmos.

Notamos o funcionamento da Heurística subindo o morro printando sempre quando acha uma permutação/caminho melhor do que o anteriormente melhor salvo, também com o número da iteração na qual esse caminho foi encontrado. Nesse caso podemos notar que o algoritmo não caiu em nenhum máximo local e conseguiu encontrar a melhor solução igual o Busca em Profundidade. Por suas características próprias, diferentemente do Busca em Profundidade, o Subindo o

Morro consegue ser mais rápido e encontrar o melhor caminho com menos iterações.

No caso abaixo podemos ver o Subindo o Morro caindo em um máximo local, pois, limitando o número máximo de iterações para 8 ele não consegue achar o melhor caminho que foi encontrado apenas na iteração 9. Percebemos então que o número de iterações é fundamental para ele.

```
Digite o Nº máximo de iterações desejado: 8


Iteração 1: Melhor rota atual: ['A', 'B', 'C', 'D', 'E'],
Melhor custo atual: 175
Iteração 3: Melhor rota atual: ['A', 'B', 'D', 'C', 'E'],
Melhor custo atual: 160

Atingido o número máximo de iterações (8). Abortando a busca.

=====================================================

Tempo de execução: 0.0003170967102050781 segundos
Número total de iterações: 8

Menor Caminho: ['A', 'B', 'D', 'C', 'E', 'A']
Custo do Caminho: 160

=====================================================
```

Comparando o Busca em Profundidade e o Subindo o Morro com as entradas 'v1.txt' e 'a1.txt': O subindo o morro se demonstrou mais rápido e eficiente na resolução do problema especificado.

Arquivo 'v2.txt' com 10 vértices;
Arquivo 'a2.txt' com 20 arestas;
Com no máximo 10000 iterações.

Busca em Profundidade:

```
================================================

Tempo de Execução: 1.0921571254730225 segundos
Número total de Iterações: 1655

Menor caminho: ['A', 'B', 'E', 'D', 'G', 'J', 'H', 'I', 'F', 'C', 'A']
Custo do Caminho: 515

================================================
```

Subindo o Morro:

```
Atingido o número máximo de iterações (10000). Abortando a busca.

==================================================

Tempo de execução: 0.020954370498657227 segundos
Número total de iterações: 10000

Menor Caminho: ['A', 'B', 'C', 'F', 'E', 'H', 'I', 'J', 'G', 'D', 'A']
Custo do Caminho: 520

==================================================
```

Notamos nas execuções acima, que limitando o número de iterações para um problema maior, o Busca em Profundidade conseguiu acessar todos os caminhos possíveis e chegar no melhor resultado. Já o Subindo o Morro, provavelmente sempre caindo em máximos locais, não conseguiu gerar permutações de caminhos ótimos, se estabilizando em uma solução um pouco pior do que a de seu rival. Busca em Profundidade ganhou encontrando a melhor solução por força bruta, todavia, sempre demora mais para executar.

Arquivo ‘v3.txt’ com 12 vértices;
Arquivo ‘a3.txt’ com 66 arestas;
Com no máximo 100000 iterações.

Busca em Profundidade:

```
==================================================

Tempo de Execução: 93.73550820350647 segundos
Número total de Iterações: 100000

Menor caminho: ['A', 'B', 'C', 'D', 'E', 'L', 'I', 'J', 'F', 'G', 'H', 'K', 'A']
Custo do Caminho: 33

==================================================
```

Subindo o Morro:

```
Atingido o número máximo de iterações (100000). Abortando a busca.

==================================================

Tempo de execução: 1.1635220050811768 segundos
Número total de iterações: 100000

Menor Caminho: ['A', 'B', 'C', 'F', 'G', 'J', 'I', 'L', 'D', 'E', 'H', 'K', 'A']
Custo do Caminho: 29

==================================================
```

Acima, notamos que, para um problema com muitas arestas e com um número razoável de vértices, encontramos problemas no Busca em Profundidade. Esse problema se dá pelo fato dele gerar todas as combinações possíveis de caminho, e aqui, mesmo com um máximo de cem mil de iterações ele não conseguiu gerar todos e finalizou com uma resposta não ótima, sem contar o tempo de execução, que extrapola em relação ao Subindo o Morro.

Neste caso o Subindo o Morro ganha em todos os quesitos, mesmo parando com o número máximo de iterações (talvez poderia até encontrar outro caminho melhor).

Arquivo ‘v3.txt’ com 12 vértices;
Arquivo ‘a3.txt’ com 66 arestas;
Com no máximo 5000000 iterações.

Busca em Profundidade: O algoritmo executou indefinidamente por mais de uma hora, então paramos a execução. Acreditamos que demoraria bem mais tempo ainda para ele conseguir explorar todo o espaço de busca, devido a natureza dele ser NP-difícil ((O(N!)) em relação a complexidade de tempo.

Subindo o Morro:

```
Atingido o número máximo de iterações (5000000). Abortando a busca.

==================================================

Tempo de execução: 52.3247389793396 segundos
Número total de iterações: 5000000

Menor Caminho: ['A', 'B', 'C', 'L', 'I', 'J', 'F', 'G', 'E', 'D', 'K', 'H', 'A']
Custo do Caminho: 27

==================================================
```

Geral: para problemas/grafos pequenos os 2 acham a melhor solução. Já para problemas grandes, tem-se aí uma dificuldade.

V.2 - parte teórica

Busca em Profundidade:

- Completa e Ótima: Pode garantir completude se o grafo for finito, pois ela explora todo o espaço de busca. Não é garantido ser ótimo para o Problema do Caixeiro Viajante, principalmente com espaços de busca gigantes que podem levar a uma execução quase infinita do algoritmo.
- Complexidade de Tempo: A complexidade de tempo é razoável (O(N!)), mas o Problema do Caixeiro Viajante é conhecido por ser NP-difícil, então a B. em P. pode não ser prática para instâncias grandes como falado acima.
- Memória: Utiliza uma quantidade de memória proporcional à profundidade máxima do espaço de busca. Pode levar a um uso grande de memória, especialmente para grafos grandes.
- Heurística: Não utiliza heurísticas específicas para otimização e, portanto, pode percorrer caminhos desnecessários.

Subindo o Morro (Hill Climbing)

- Heurística: O Subindo o Morro é uma busca heurística que segue a abordagem "melhoria local". Ele avalia as soluções vizinhas e move-se em direção à solução que melhora o valor da função de avaliação (no caso do Caixeiro Viajante, a distância total percorrida). Pode ficar preso em ótimos locais, não garantindo a otimalidade global.
- Complexidade de Tempo: O tempo de execução depende da qualidade da heurística utilizada. Em casos ideais, a complexidade pode ser mais baixa do que a Busca em Profundidade (como implementamos em nosso algoritmo, executa em bem menos tempo que a B. em P.), mas ainda assim pode ser significativa.
- Memória: Geralmente utiliza menos memória do que a Busca em Profundidade, pois avalia soluções vizinhas em vez de manter todo o espaço de busca.
- Ótimo Local vs. Ótimo Global: O Subindo o Morro foca em encontrar um ótimo local, mas nem sempre garante o ótimo global. Pode ser sensível à escolha inicial (em nosso caso, o vértice inicial é sempre o primeiro do arquivo, e a partir dele são geradas as permutações).

São adequados para o Problema do Caixeiro Viajante?

Busca em Profundidade: Pode ser aplicada ao problema para encontrar soluções, mas a complexidade exponencial (O(N!)) torna impraticável para instâncias/grafos grandes.
Subindo o Morro: Pode ser utilizado, mas é uma heurística de melhoria local, o que significa que pode ficar preso em ótimos locais e não garante a solução globalmente ótima.

Concluindo, ambos os algoritmos podem ser aplicados ao problema, mas são mais adequados para instâncias pequenas ou para explorar soluções iniciais. Para problemas maiores e mais complexos, são recomendados algoritmos heurísticos mais avançados, como Algoritmos Genéticos, Simulated Annealing, ou heurísticas específicas para o Caixeiro Viajante, que têm maior probabilidade de encontrar soluções próximas da ótima global de maneira mais eficiente.

V - Considerações finais

Com base no problema escolhido e nos algoritmos de busca selecionados podemos observar que a solução para o problema, dado certas arestas e vértices, pode demorar, solucionar ou parar em resultados diferentes, isso devido às características de cada algoritmo. O algoritmo de Busca em Profundidade tem características marcantes como o de explorar todo o espaço de busca, já o algoritmo heurístico Subindo o Morro segue a abordagem de melhoria local, fazendo assim cada algoritmo ter uma forma de resolver o problema do caixeiro viajante.

O problema do caixeiro viajante deve ser analisado com muita cautela, devido a necessidade do programa em encontrar um caminho válido de custo menor, assim sendo necessário uma análise no grafo de busca para ajudar a escolher qual técnica será mais vantajosa, sendo a técnica de Busca em Profundidade mais recomendada para grafos menores e a heurística Subindo o Morro ser usada com cautela para que não caia em um máximo local.

Por fim, concluímos que o auxílio dos algoritmos de busca foi de extrema ajuda na hora de resolver o problema do caixeiro viajante, porém devemos ter uma análise minuciosa dos resultados e da execução do algoritmos para que o resultado final seja satisfatório.

unioeste
Universidade Estadual do Oeste do Paraná

VI - Referências bibliográficas

CAMBUIM, Prof. Lucas. **Introdução a Inteligência Artificial: tipos de busca (busca cega)**. Pernambuco, 2020. 36 slides, color. Disponível em: https://www.cin.ufpe.br/~lfsc/cursos/introducaoainteligenciaartificial/IA-Aula4-BuscaCega.pdf. Acesso em: 28 nov. 2023.

JÜNGER, Michael; REINELT, Gerhard; RINALDI, Giovanni. **The traveling salesman problem**. Handbooks in operations research and management science, v. 7, p. 225-330, 1995.

LIMA, Acervo. **INTRODUÇÃO AO HILL CLIMBING | INTELIGÊNCIA ARTIFICIAL**. 2023. Disponível em: https://acervolima.com/introducao-ao-hill-climbing-inteligencia-artificial/. Acesso em: 28 nov. 2023.

OLIVEIRA, André Filipe Maurício de Araújo. **Extensões do problema do caixeiro viajante**. 2015. Dissertação de Mestrado.