TP1 - Algoritmo de Busca para Menor Caminho

Lorenzo Carneiro Magalhães - 2021031505
Universidade Federal de Minas Gerais

1 Introdução

Nesse trabalho prático foi desenvolvido alguns algoritmos de busca em um mapa pré-definido. Esse mapa conta com terrenos específicos que possuem custos associados e é dado um ponto de início e fim.

Então basta utilizar do algoritmo especificado para realizar a busca. É importante ressaltar que nem sempre a busca é ótima dependendo do algoritmo escolhido e do mapa fornecido.

2 Algoritmos

Foi necessário implementar os algoritmos: Busca em Largura (BFS), Aprofundamento Iterativo (IDS) e Busca de Custo Uniforme (UCS), busca Gulosa e A*.

Perceba, que a busca Gulosa e o A* dependem de heurísticas. A heurística implementada consistiu em somar a distância Manhattan até o ponto final ao custo real.

2.1 Implementação e Execução

Todas as implementações consideraram os eixos naturais das matrizes, portanto a altura da matriz é o eixo X e o comprimento da matriz é o eixo Y. Isso foi feito pela facilidade da codificação. Para isso funcionar, primeiro foi necessário inverter os X's e os Y's recebidos, assim como inverter os eixos no caminho retornado pelas funções de busca ao final.

Em todas as implementações, recebo a matriz de custos, que é basicamente a matriz do tabuleiro, o ponto inicial e ponto final.

Para execução do script principal, basta utilizar o comando:

python pathfinder.py [ARQUIVO] [ALGORITMO] [PONTOS]

2.2 Greedy

O algoritmo Greedy foi o primeiro que realizei, pois ele é fácil e rápido de ser implementado.

Inicialmente havia implementado errado o Greedy, pois não tinha levado em consideração a frase "Três são sem informação: Busca em Largura (BFS), Aprofundamento Iterativo (IDS) e Busca de Custo Uniforme (UCS). Os outros são o busca Gulosa e A*" no enunciado. Por essa frase, entende-se que o algoritmo deve ter a informação do labirinto, mas da maneira que fiz inicialmente ele apenas pegava o vizinho mais próximo. Nesse sentido, com essa ideia em mente, fiz a correção mudando a heurística para selecionar o vizinho que minimiza a distância manhattan para o nó final.

No algoritmo, não há algo muito complexo, apenas via os vizinhos não visitados e selecionava aquele que casava melhor com a heurística somado com o valor de custo da célula. O vetor de visitados é uma lista de tuplas.

Perceba então que o algoritmo não garante achar uma solução para o problema. O enunciado pede que se utilize uma heurística válida, mas não necessariamente completa.

2.3 BFS

Foi realizada uma simples BFS sem nenhuma estrutura adicional. Defini um dicionário para registrar o caminho de cada nó, uma matriz de custos e também uma lista para funcionar como a fila da BFS. A partir deste ponto, comecei a explorar os vizinhos, e caso o caminho para esse vizinho fosse menor do que o menor caminho atual, então apenas o defino como o novo.

Apesar de não precisar em uma BFS tradicional, também defini um vetor de visitados para checar nós visitados mais rapidamente e não levar em conta os pesos ponderados, já que numa implementação sem lista de visitados a BFS adiciona o vizinho caso o custo dele seja menor que o atual, de maneira que no caso de arestas ponderadas o algoritmo possa redefinir um custo já definido anteriormente, o que não aconteceria em um grafo com arestas com custos uniformes.

Logo, é perceptível que esse algoritmo não obtém a resposta ótima, conforme previsto. Isso ocorre porque ele leva em consideração apenas o número de passos mínimos até alcançar o ponto final, o que não necessariamente implica em um caminho ótimo quando tem-se arestas com custos variados.

Além disso, o algoritmo expande muitos nós de maneira desnecessária, já que não evita passar por lugares que não parecem bons caminhos.

2.4 Dijkstra/UCS

O Dijkstra é um algoritmo que consegue encontrar o custo mínimo em casos como o do labirinto em que o grafo não possui pesos negativos. Assim, os casos dele são totalmente determinísticos e puderam ser testados com os arquivos de teste fornecidos pelo professor.

A implementação foi simples:

Definições:

- matriz de custos utilizando o numpy (talvez fosse interessante fazer um dicionário do python devido à performance, mas utilizei minha base do BFS)
- dicionário para armazenar o caminho
- lista para funcionar como um min heap (como auxílio da biblioteca heapq)

Passos do algoritmo:

- Enquanto o heap não está vazio:
- pega o min cost (x, y) do heap
- se for o objetivo, então para o algoritmo e reconstroi o caminho
- para cada vizinho (nx, ny) de (x, y):
- se o custo de (nx, ny) vindo de (x, y) for menor que o cadastrado, então:
- adicione (new_cost, nx, ny) ao heap
- defina o caminho de (nx, ny) como (x, y)
- defina o novo custo

Por fim, o algoritmo realiza o processamento rapidamente e consegue encontrar o caminho ótimo, caso tenha.

2.5 IDS

Esse algoritmo em específico eu não conhecia antes das aulas do curso, então realizei algumas pesquisas para realizar a implementação e mesmo assim tive algumas dificuldades com a minha implementação. A parte recursiva do código me deu um nó na cabeça, mas por fim entendi como o algoritmo funciona e como a implementação deve ser feita.

Confesso que minha implementação provavelmente não é a melhor, pois achei confusa, mas funciona.

O algoritmo em si não é tão útil em um cenário de achar caminhos em mapas e principalmente quando os mapas têm rotas ponderadas, pois ele não acha o ótimo, devido às células possuírem custo e devido à alta sensibilidade no hiperparâmetro de profundidade, e também não é muito rápido, pois visita muitas vezes os mesmos nós do mapa.

Essa visita de nós repetidamente ocorre porque não se mantém um grupo de nós visitados globais, de modo que todos os nós são revisitados a cada nova profundidade. Seria similar ao

cálculo do fatorial de um número da maneira ingênua em que não é salvo os resultados anteriores. Além disso, pode ocorrer do algoritmo ficar rodando muito tempo e simplesmente parar sem de fato encontrar a solução pois a profundidade inserida foi atingida.

Para piorar a performance, a implementação usa recursividade, o que sabidamente interfere negativamente na performance.

2.6 A*

Com o melhor desempenho, o algoritmo A* performa muito bem. Utilizando de uma heurística consistente, tem-se que a execução fica muito rápida e eficiente. Durante a execução o algoritmo busca por caminhos muito bons e que fazem sentido.

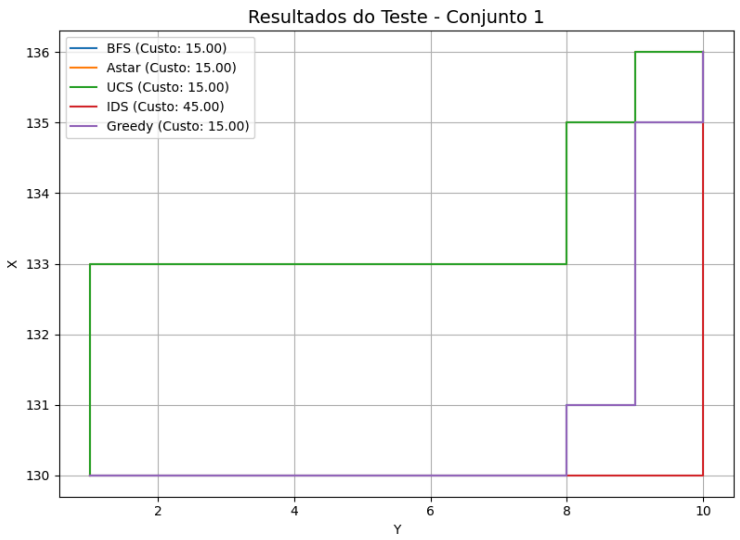
Isso ocorre pois ele considera o fator de custo vindo do dijkstra juntamente com a característica da implementação gulosa de sempre querer ficar mais perto do ponto final. Isso confere ao algoritmo uma eficiência na busca.

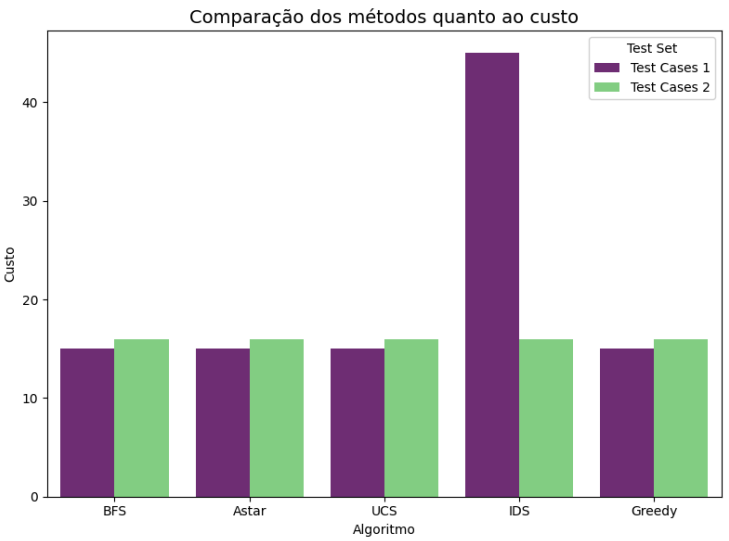
É importante ressaltar que a heurística utilizada foi a mesma da gulosa: a distância Manhattan. Foi utilizada essa heurística pois casa bem com o problema, é fácil de implementar e também porque ela está presente nos slides da disciplina.

2 Resultados Práticos

Os resultados obtidos a partir das experimentações baseadas nos casos de teste refletem o que foi discutido nos tópicos anteriores.

As experimentações foram divididas em 2 conjuntos de teste, nos quais realizei uma busca por um caminho de maneira que os nós de início e final foram escolhidos aleatoriamente, mas manualmente.

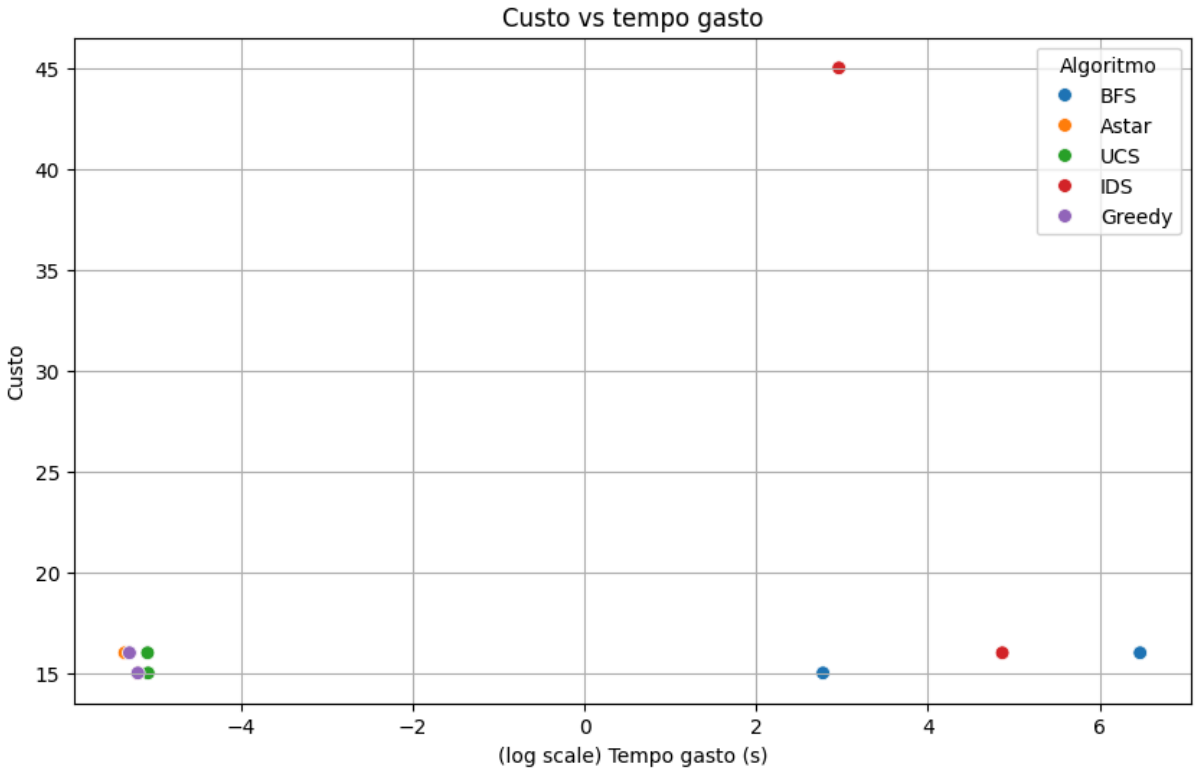




Essa escolha manual foi realizada a fim de evitar caminhos muito longos e demorados. Nessa figura é apresentado os diferentes caminhos escolhidos pelos algoritmos, assim como o custo associado a cada um deles nos testes.

No primeiro gráfico, é mostrado os caminhos escolhidos pelo algoritmo em questão até encontrar o nó final no caso de teste 1. Dessa maneira, é perceptível a existência de 2 caminhos ótimos, os quais todos os algoritmos exceto o IDS selecionaram.

Nota-se, portanto, que o algoritmo IDS não performou bem nesse caso. Os algoritmos BFS e Greedy poderiam também não achar o ótimo, conforme a teoria, mas nesse caso específico eles foram capazes de encontrá-lo.



Essa outra figura demonstra uma comparação entre tempo e custo encontrado pelos algoritmos.

Como esperado e citado anteriormente, o BFS e o IDS demoram bastante para completarem o processamento e não necessariamente acham o valor ótimo de custo. Assim, esses algoritmos não são muito úteis para solucionar o problema em questão.

Em contrapartida, os algoritmos UCS e A* apresentaram desempenho muito bom, já que foram capazes de achar o valor ótimo em tempo mínimo. É perceptível através do gráfico afirmar que o A* executa mais rapidamente do que o UCS, já que em ambos os casos de teste o A* completou o processamento mais rapidamente.

Adicionalmente, o algoritmo Guloso apresentou nesses casos um desempenho extraordinário, já que foi capaz de dizer corretamente o caminho de custo mínimo muito rapidamente. No entanto, não há nenhuma garantia que essa performance e corretude se mantenha em outros testes, portanto pode-se dizer que essa solução não é confiável apesar de sua velocidade.

Observação:

Tentei buscar manualmente por entradas que demonstraram que o IDS, BFS e Greedy não achavam soluções ótimas simultaneamente, mas não houve tempo para tal, já que o IDS e o BFS demoravam muito.