Trabalho Prático 2

Algoritmos de Busca em Grafos

João Vitor Soares Santos

Matrícula: 2023002138

21 de Julho de 2024

Introdução

O objetivo deste trabalho é auxiliar o herói Linque, que está perdido na Floresta da Neblina, a encontrar um caminho seguro para escapar de sua perigosa situação. Para resolver esse problema, serão utilizados dois algoritmos de busca fundamentais: Dijkstra e A*. Linque enfrenta uma floresta composta por clareiras e trilhas unidirecionais, com a adição de portais mágicos que não consomem energia. A tarefa é encontrar um caminho do ponto de partida até a saída, respeitando as restrições de energia e o número máximo de portais que podem ser utilizados. Caso o caminho não seja encontrado, Linque ficará preso na floresta para todo o sempre.

A implementação deste problema envolve o uso de várias estruturas de dados essenciais:

- **Heap (Fila de Prioridade):** Utilizado para a gestão eficiente das prioridades durante a execução dos algoritmos de Dijkstra e A*. A heap permite que as operações de inserção e extração do menor elemento sejam realizadas de forma eficiente, o que é crucial para a performance dos algoritmos de busca em grafos.
- **Listas de Adjacência:** Utilizadas para representar o grafo de forma eficiente em termos de espaço. Cada vértice mantém uma lista de suas conexões (arestas) com outros vértices, permitindo uma fácil navegação e atualização das arestas durante a execução dos algoritmos.
- **Matriz de Adjacência:** Oferece uma representação alternativa do grafo, onde a presença de uma aresta entre dois vértices é indicada em uma matriz. Embora ocupe mais espaço em comparação com as listas de adjacência, a matriz pode facilitar certas operações e análises, especialmente em grafos densos.

O trabalho incluirá a implementação dessas estruturas de dados e a aplicação dos algoritmos de Dijkstra e A* para encontrar o caminho mais seguro e eficiente para Linque. A análise comparativa dos algoritmos, juntamente com a avaliação do desempenho das estruturas de dados, permitirá uma compreensão mais profunda das suas características e adequações para resolver problemas de otimização em grafos.

O código desenvolvido para este trabalho está disponível no link a seguir:

The Lost Woods

Implementação

Arquitetura do projeto

A arquitetura é definida em quatro partes principais:

- ./src : O diretório que contém os arquivos fonte C++
- ./include : O diretório contendo os cabeçalhos necessários C++
- ./python/ : O diretório contendo geradores de caso teste, de visualização de grafos e de reconstrução de caminhos encontrados
- Makefile: arquivo que contém diretivas usadas para automatizar a compilação

Existem dois comandos principais no arquivo Makefile

- make all: compila todo o código fonte em uma saída a.out, localizada na pasta ./bin
- make clean: limpa todos os arquivos de saída e objeto nos diretórios ./bin e ./obj

Além disso o make file possui uma variável *TEST*, que pode possuir quatro valores

- make: o programa é compilado em sua forma padrão
- make TEST=alg: o programa é compilado para testar a eficiência dos algoritmos Dijkstra e A*
- make TEST=list-space: o programa é compilado para testar a complexidade espacial da estrutura de dados Listas de Adjacência
- make TEST=matriz-space: o programa é compilado para testar a complexidade espacial da estrutura de dados Matriz de Adjacência

Compilação do projeto

1. Abra o terminal e navegue até a pasta base do projeto.
2. Certifique-se de que o arquivo Makefile está presente na pasta base do projeto.
3. Execute o seguinte comando no terminal para compilar o projeto:

   usuario@pc:~$ make

Programa principal

Forma padrão

O programa ao ser executado receberá algumas variáveis:

A primeira linha consiste em três inteiros n, m e k que representam respectivamente quantos vértices, trilhas e portais a floresta possui. As próximas n linhas possuem dois racionais x e y, descrevendo as coordenadas de cada clareira em um plano. Depois, as próximas m linhas possuem dois inteiros, indicando que existe uma trilha entre as duas clareiras, que só pode ser percorrida começando em u e terminando em v. Por fim as próximas k linhas possuem também dois inteiros u e v, indicando quais clareiras são conectadas por portais (os portais também só podem ser atravessados começando em u e terminando em v). A última linha contém um racional s indicando a quantidade de energia que Linque possui para caminhar e um inteiro q indicando quantos portais podem ser utilizados.

Caso algum parâmetro seja passado incorretamente, exceções serão lançadas e a função *tutorial(),* que detalha a entrada esperada, será executada. Resumidamente, a entrada principal detalha a estrutura de um grafo simples direcionado, que é um grafo direcionado que não tem laços nem arestas múltiplas, já a última linha indica parâmetros para os algoritmos de busca em grafos, especificando o limite de energia e de portais que podem ser utilizados.

Já a saída consiste em duas linhas, que indicam se os algoritmos Dijkstra e A* conseguiram encontrar um caminho para o Linque dentro das restrições, 0 representa impossível, 1 possível.

TEST=alg

Ao ser compilado com intuito de analisar a eficiência dos algoritmos de busca em grafos, o programa recebe as mesmas entradas do programa em sua forma padrão, a única diferença sendo que ele não recebe o valor s, que indica a energia que Linque possui.

Já em sua saída, o programa imprime os resultados ***tempo de execução, tamanho do caminho encontrado*** e ***caminho encontrado (***É impresso a sequência de vértices do caminho), para os seguintes algoritmos: Dijkstra com listas de adjacência, A* com listas de adjacência, Dijkstra com Matriz de adjacência e A* com Matriz de adjacência.

TEST=matriz-space ou TEST=list-space

Ao ser compilado com intuito de analisar a complexidade espacial das estruturas de dados Matriz de adjacência e Listas de adjacência, o programa recebe as mesmas entradas do programa em sua forma padrão, as diferenças sendo que na primeira linha ele recebe um novo valor *dijkOrAstar,* que representa o que ele quer testar, *0 r*epresenta a estrutura de dados a Matriz ou as listas, *1* representa o dijkstra, *2* representa A*, além disso o programa não recebe o valor s, que indica a energia que Linque possui.

O programa nesse estado não apresenta nenhuma saída, seu principal foco é ser utilizado junto ao *valgrind,* para que possa ser analisado quantos bytes foram alocados.

Programas auxiliares

generator.py

O *generator.py* gera grafos aleatórios para esse trabalho. Ele tem dois modos e cinco entradas na linha de código

```
usuario@pc:~$ python3 ./python/generator.py <mode = i/d> <vertices> <edges> <portals> <seed>
```

Caso o modo escolhido seja *i,* a entrada será composta do número de vértices, o número de arestas, e o número portais, e por final a *seed* que será usada para fins de aleatoriedade. Caso o modo escolhido seja *d,* os valores de arestas e portais serão representados em meios de densidade (porcentagem). A saída desse programa é um arquivo contendo as entradas que representam o grafo.

creategraph.py

O *creategraph.py* gera uma visualização para os grafos gerados pelo *generator.py*. Ele recebe uma entrada pela linha de código o caminho do arquivo que representa o grafo

```
usuario@pc:~$ python3 ./python/creategraph.py <filename>
```

A saída desse programa é um arquivo *.png* que contém a representação visual do grafo.

recoverpath.py

O *recoverpath.py* gera uma visualização para os grafos gerados pelo *generator.py* com um adendo, uma linha adicional é adicionada no final, contendo uma sequência de vértices, que representam o caminho que se deseja recuperar. Ele recebe uma entrada pela linha de código o caminho do arquivo que representa o grafo

```
usuario@pc:~$ python3 ./python/recoverpath.py <filename>
```

A saída desse programa é um arquivo *.png* que contém a representação visual do caminho recuperado.

Estratégias de Robustez

Como estratégias de robustez, foram implementadas diversas verificações e manuseio de exceções utilizando diretivas de tratamento de erros como try, catch e throw, além disso foram utilizadas classes de erro padrão da biblioteca padrão C++. Essas estratégias visam garantir que entradas inválidas sejam tratadas adequadamente e que o programa se comporte de maneira previsível em todas as situações.

Números Negativos: O programa não aceita números negativos para o número de vértices, arestas, portais, energia de Linque e quantidade de portais que podem ser usados. Se qualquer um desses valores for negativo, uma exceção é lançada e uma mensagem de erro é exibida

Vértices Duplicados: Não é permitido inserir o mesmo vértice mais de uma vez. Se um vértice duplicado for detectado, uma exceção é lançada

Arestas Duplicadas: Não é permitido inserir a mesma aresta mais de uma vez. Se uma aresta duplicada for detectada, uma exceção é lançada

Portais Duplicados: Não é permitido inserir o mesmo portal mais de uma vez. Se um portal duplicado for detectado, uma exceção é lançada.

Execução do programa

Após a compilação bem-sucedida, execute o seguinte comando no terminal para rodar o programa:

```
usuario@pc:~$ ./bin/a.out
```

Configuração usada

Os seguintes código e testes foram feitos nas seguintes configurações:

- Sistema Operacional : Linux (Ubuntu 23.10) 64 bits
- Linguagem de programação implementada: C++
- Compilador utilizado: g++
- Processador : Ryzen 5
- Quantidade de memória RAM : 8 Gigabytes

Estruturas de dados

Tuple

A estrutura de dados auxiliar tupla foi criada para auxiliar nesse projeto, sendo basicamente um tipo de dados que armazena dois outros tipos de dados, para isso essa classe foi implementada com templates.

Indexed Min Heap (Indexed Priority Queue)

Para este trabalho a estrutura de dados *heap* indexado foi implementada, sua principal diferença comparado com a sua versão padrão é o fato de que é possível verificar se determinado elemento está no *heap,* retornando um ponteiro para o elemento, em tempo constante **O(1)**, graças a isso a operação de atualizar determinado item do *heap* através de sua chave é facilitada**.** Para isso ser possível um array auxiliar que guarda que elemento está em determinada posição do *heap* é mantido.

Principais métodos:

- void insert(Key key, Value value) : **O(log(heapsize))**
- Tuple<Key, Value>* contains(Key key) : **O(1)**
- void update(Key key, Value value): **O(log(heapsize))**
- Tuple<Key, Value> remove() : **O(log(heapsize))**

Com o objetivo de manter o heap ordenado, as funções auxiliares ***heapifyDown,*** que ordena o heap por de baixo para cima, e ***heapifyUp***, que ordena o heap de cima para baixo, são usadas.

Ademais, a estrutura de dados foi implementada usando templates com o intuito de ser útil tanto para o algoritmo dijkstra e para o A*, dessa forma, em sua inicialização, deve ser passado como parâmetro o *tamanho máximo do heap,* uma *classe de comparação*, que é usada para definir as prioridades dentro do heap, e uma *classe de hash,* que determina o id dentro do vetor auxiliar de indexes através da chave do elemento. Por último, o tipo de chave e o valor do item são definidos no template.

Singly Linked List

A estrutura de dados lista simplesmente encadeada é a mesma passada durante as aulas de estrutura de dados, porém ela foi implementada com templates para fins de generalidade, ela possuiu diversos métodos e funcionalidades porém os utilizados neste trabalho são :

- void insertEnd(T item) : **O(1)**
- int search(T item) : **O(listsize)**

Além disso, um integrador customizado foi implementado para facilitar percorrer a lista encadeada.

Graph with Adjacency Lists

Para criar as listas de adjacência, a estrutura de dados *Singly Linked List* é utilizada, vale ressaltar que nessa implementação todo caminho de tamanho zero é considerado um portal, por isso não é permitido inserir o mesmo vértice duas vezes (único caso que poderia acontecer de um

caminho ter comprimento zero), além disso o número máximo de vértices e o número de vértices inseridos é mantido, tal qual um vetor que mantém todos os vértices. Tal estrutura de dados, para fins de generalidade, foi implementada utilizando templates, e seus principais métodos são:

- void insertVertice(Vert item) : **O(1)**
- int verticeExist(Vert item) : **O(maxvertices)**
- Vert getVertice(int index) : **O(1)**
- SinglyLinkedListUnordered<Tuple<ll, double>>* getNeighboors(int v) : **O(1)**
- void insertEdge(int v1, int v2, double w) : **O(1)**
- bool edgeExist(int v1, int v2) : **O(maxvertices)**
- bool portalExist(int v1, int v2) : **O(maxvertices)**

Sendo a principal função dessa estrutura a *getNeighboors(int v1),* que retorna uma *lista encadeada* contendo os vértices adjacentes a v1.

Graph with Adjacency Matriz

Nesta estrutura de dados a matriz de adjacência é definida através de uma matriz de tupla contendo um double que representa o possível tamanho de um caminho e um booleano que indica que há um portal conectando os dois vértices, além disso o número máximo de vértices e o número de vértices inseridos é mantido, tal qual um vetor que mantém todos os vértices. Tal estrutura de dados, para fins de generalidade, foi implementada utilizando templates, e seus principais métodos são:

- void insertVertice(Vert item) : **O(1)**
- int verticeExist(Vert item) : **O(maxvertices)**
- Vert getVertice(int index) : **O(1)**
- Tuple<double, bool>* getNeighboors(int v) : **O(1)**
- void insertEdge(int v1, int v2, double w) : **O(1)**
- void insertPortal(int v1, int v2, double w) : **O(1)**
- bool edgeExist(int v1, int v2) : **O(1)**
- bool portalExist(int v1, int v2) : **O(1)**

Sendo a principal função dessa estrutura a *getNeighboors(int v1),* que retorna um vetor de tamanho **maxvertices**, contendo diversos tipos de valores **double,** caso o valor na posição *i* for negativo, os dois vértices não são adjacentes, caso o valor seja 0 os dois vértices são ligados direcionalmente por um portal, e valores positivos representam o tamanho do caminho entre os dois vértices

Análise de Complexidade

Tendo em vista os conceitos aprendidos em sala de aula, será feita agora a análise assintótica temporal dos algoritmos e estruturas de dados, iremos considerar |V| o número de vértices e |E| o número de arestas do grafo.

Complexidade temporal:

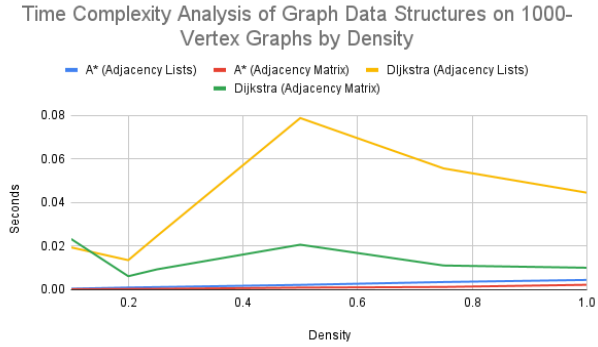
- Dijkstra: **O((E| + |V|) log |V|)**
- A* : **O((E| + |V|) log |V|)**

Complexidade espacial:

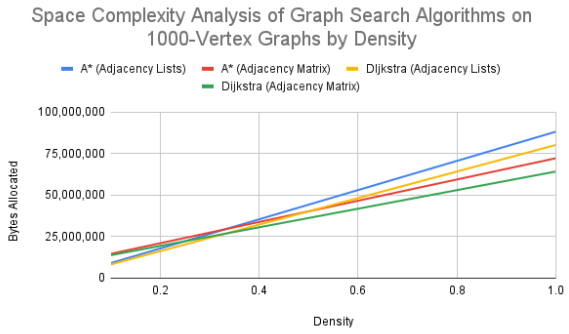
- Graph with Adjacency Lists: **O(|V| + |E|)**
- Graph with Adjacency Matriz: **O(|V| ²)**

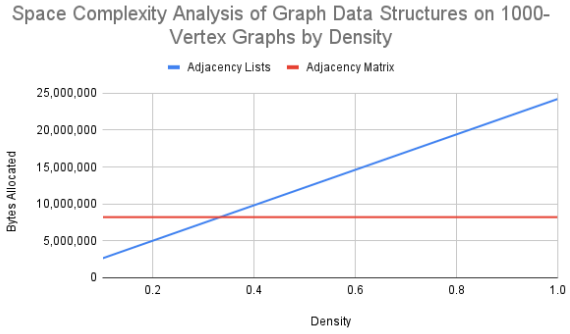
Análise Experimental

Como primeiro caso, iremos comparar as estruturas de dados Listas de Adjacência e Matriz de Adjacência, os testes foram feitos em um mesmo grafo de mil vértices com diferentes densidades contendo 0.1% de portais e podendo usar no máximo 1% deles :



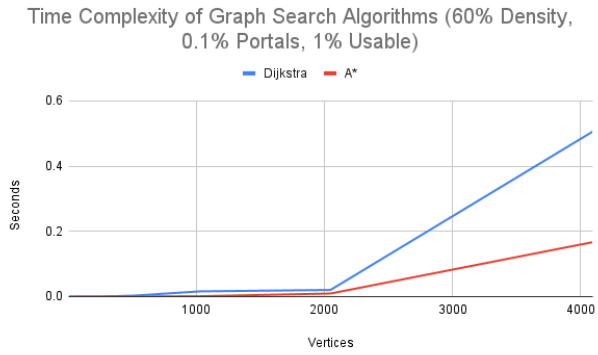
Como pode ser visto no gráfico apresentado, as versões do *Dijkstra* e da *A** usando matriz de adjacência são evidentemente superiores em questão de tempo de execução do que suas versões com matrizes de adjacência, o que pode ser explicado em e questões de localidade e referência, afinal em uma matriz os blocos estão agrupados em porções contíguas, então percorrer pelos vértices adjacentes a outro é rápido, enquanto na listas de adjacência os blocos de cada item estão em localizações completamente diferentes da memória, o que prejudica no princípio de localidade espacial.

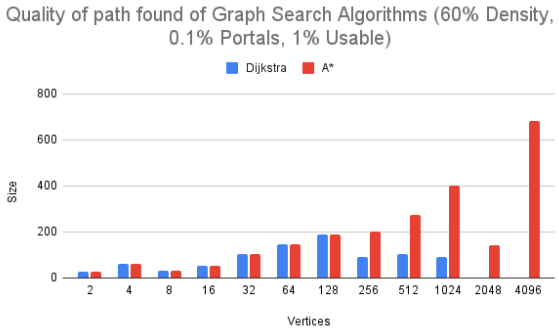




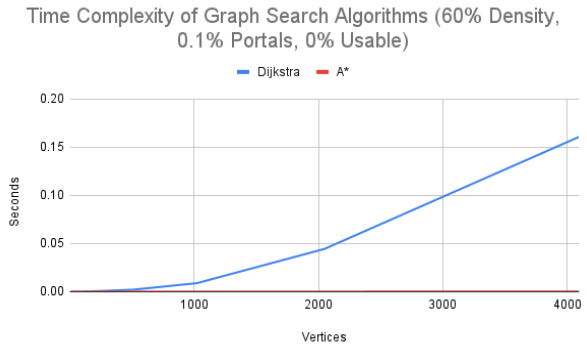
Em termos de desempenho espacial fica claro que, em grafos mais esparsos as listas de adjacência são mais eficientes, afinal elas armazenam apenas o essencial para determinar que o vértice *v1* é adjacente ao *v2*, não deixando nenhum byte inutilizado, o que não é o caso da matriz de adjacência que independentemente da esparsidade do grafo aloca a mesma quantidade de bytes. Porém, o fato da matriz de adjacência ser estática em questões de espaço acaba sendo uma vantagem em grafos mais densos, é evidente nos gráficos que a partir do momento que o grafo tem sua densidade em 25%, é mais vantajoso o uso de matrizes, afinal as listas requerem tamanho extra (ponteiros e etc).

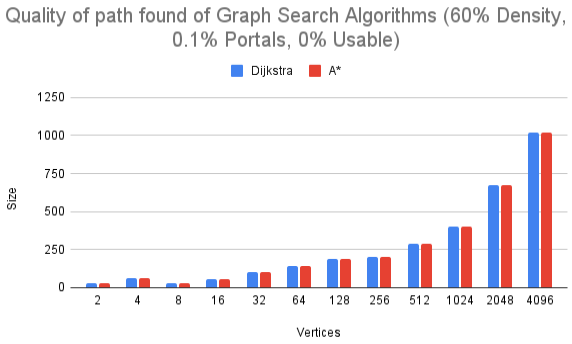
A partir dessas descobertas, a análise a posteriori levou em consideração os algoritmos implementados com a matriz de adjacência. Sendo assim iremos analisar a eficiência dos algoritmos Dijkstra e A*, valendo ressaltar que a heurística escolhida para guiar o A* foi a distância euclidiana do vértice atual até o objetivo final. Dessa forma, os algoritmos foram testados em grafos aleatórios com número de arestas variadas com densidade de 60% contendo 0.1% de portais e podendo usar no máximo 1% deles:



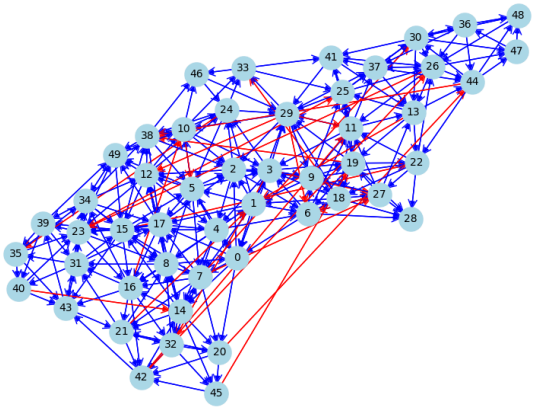


Através dos gráficos fica claro que a maior eficiência do algoritmo A* em comparação com o Dijkstra vem com uma perda significativa na qualidade do caminho encontrado quando o número de vértices do grafo aumenta, isso se deve ao fato da heurística escolhida (distância euclidiana até o objetivo) não ser muito compatível com um detalhe importante do nosso problema, os portais. Vale relembrar que o *Dijkstra*, é um algoritmo ótimo, ele sempre encontra o caminho mínimo até seu objetivo, porém isso vem com o preço de calcular a distância mínima de todos os vértices vizinhos do seu objetivo. Dessa forma, os algoritmos foram testados novamente no mesmo grafo, mas com um porém, o uso de portais agora está restrito a zero.

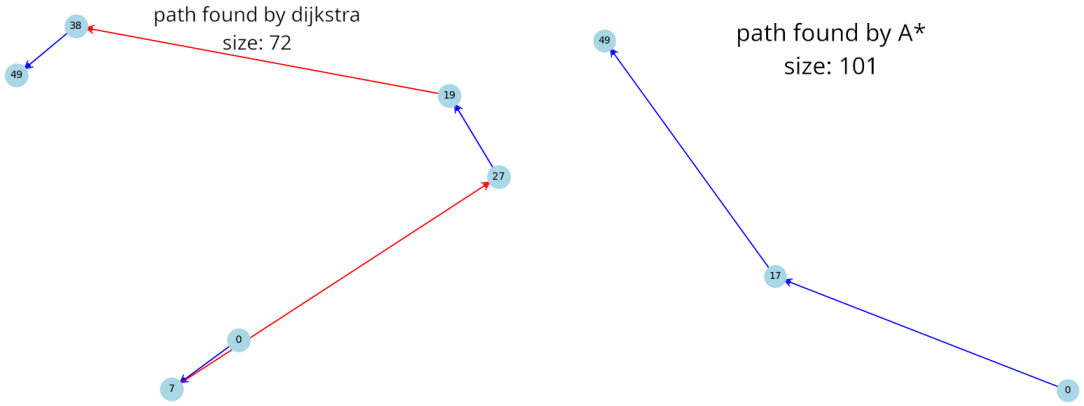




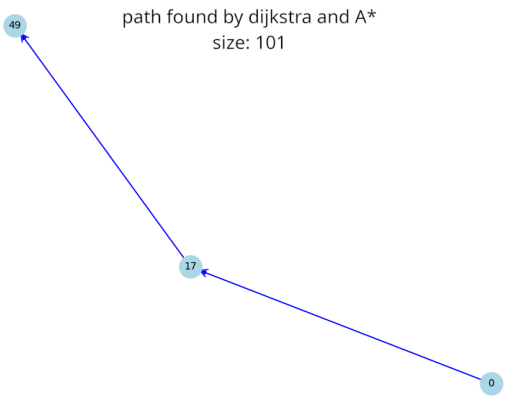
Nesse contexto, é notável a diferença de qualidade que vimos anteriormente no A*, agora que o fator imprevisível dos portais foi deixado de lado, a heurística escolhida para guiar o algoritmo brilha em sua mais bela forma. A discrepância de qualidade vem do fato da distância euclidiana não conseguir tirar proveito de todos os portais, afinal tal heurística sempre guia o algoritmo para mais perto de seu objetivo, alheio a quaisquer portais secretos que levem a um caminho mais curto. Para uma análise da qualidade do caminho encontrado pelos algoritmos, o grafo abaixo foi gerado, usando 50 vértices, 10% de densidade e 1% de portais. Para fins de transparência, as arestas azuis representam passagens e as vermelhas portais.



Ao limitar o número de portais que podem ser usados a 10%, os algoritmos Dijkstra e A* encontram caminhos discrepantes, isso acontece que o caminho mínimo que tira o proveito máximo dos portais não pode ser encontrado pela heurística do A*. Além disso, de maneira criativa, o Dijkstra encontra um caminho eficiente ao dar um passo para trás, para dar dois para frente



Como apontado anteriormente, ao reduzir o número de portais que podem ser usados para zero, a heurística do A* ganha mais poder, tornando suas respostas mais precisas e até ótimas. De maneira interessante o caminho mínimo encontrado tanto pelo Dijkstra quanto pelo A* ao limitar o número de portais é o mesmo encontrado pelo A* anteriormente.



Conclusão

Concluímos que, quando se trata da representação de grafos através de listas de adjacência e de matriz de adjacência alguns pontos devem ser considerados, sendo o mais importante deles a densidade do grafo. Em grafos menos densos é ultra vantajoso o uso de listas de adjacência, dado que a economia de espaço é muito grande ao se comparar com a matriz de adjacência que em sua base é estática e sempre gastará a mesma quantidade de bytes, porém em grafos mais densos o uso de matrizes de adjacência se torna obrigatório, afinal as listas de adjacência consomem memória extra para ponteiros, e em grafos de grande densidade elas acabam sendo mais caras que as matrizes, além disso, graças aos princípios de localidade e referência o uso de matrizes é mais eficiente por que os bytes são alocados de forma contígua.

Ademais, vimos que a eficiência ganhada através da heurística no algoritmo A* É considerável comparado ao Dijkstra, porém, em grafos muitos portais e como limite de uso grande, esse algoritmo pode não ser tão útil a Linque, que possivelmente está desgastado de suas batalhas e procura um caminho curto para escapar da floresta da neblina. Entretanto, tal fato só acontece por que a heurística escolhida (distância euclidiana até o objetivo) não consegue tirar proveito máximo dos portais, tal fato poderia ser resolvido caso Linque conhecesse a localização de todos os portais e para onde eles o levam. Uma heurística alternativa que sugiro seria considerar não apenas a distância euclidiana de um vértice até o objetivo, mas sim o mínimo entre a distância euclidiana de um vértice até o objetivo e a distância até o objetivo de outros vértices que são ligado através de portais, dessa forma o algoritmo A* conseguiria tirar um proveito maior dos portais.

Por meio da realização deste trabalho, foi possível praticar os conceitos relacionados a algoritmos de busca em grafos, especificamente Dijkstra e A*. Em suma, o trabalho proporcionou uma experiência prática valiosa na aplicação dos princípios teóricos de ciência da computação a problemas reais, fortalecendo as habilidades analíticas e de programação necessárias para o desenvolvimento de soluções eficientes, e além de tudo foi extremamente gratificante ajudar o herói Linque em sua jornada.

Bibliografia

A* search. In *Wikipedia, The Free Encyclopedia*. Disponivel em https://en.wikipedia.org/wiki/A*_search_algorithm

Dijkstra's algorithm. In *Wikipedia, The Free Encyclopedia*. Disponivel em https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

Adjacency list. In *Wikipedia, The Free Encyclopedia*. Disponivel em https://en.wikipedia.org/wiki/Adjacency_list

Adjacency matrix. In *Wikipedia, The Free Encyclopedia*. Disponivel em https://en.wikipedia.org/wiki/Adjacency_matrix