TRABALHO PRÁTICO DE AEDS 3
EU, ROBÔ

Introdução

Neste trabalho prático, foi desenvolvido um software que realiza o caminhamento em uma arena hipotética de robôs, nas quais cada célula da matriz tem um peso de deslocamento, tanto para entrar nela, quanto para sair. Desse modo, o algoritmo realiza um caminhamento por meio de um mapeamento em forma de grafos duplamente direcionados, indicando a casa que você está atualmente e o possível peso do caminho para se chegar à outra casa. No entanto, para a otimização desse algoritmo, o caminho terá que interligar a origem ao destino desejado apenas pelo menos caminho possível, tendo que contar com obstáculos e atalhos espalhados aleatoriamente diante de cada arena.

Desse modo, a solução encontrada para esse problema foi guardar as informações da arena desejada para o caminhamento dentro de uma estrutura de dados, que posteriormente, possibilitaria a criação de uma estrutura de dados do tipo Grafo, usando uma matriz de adjacência. Portanto, após a montagem do grafo duplamente direcionado, utiliza-se do algoritmo de Dijkstra para encontrar o menor caminho possível entre dois pontos, sendo esses a origem e o termino.

Solução do Problema

A solução do problema se deu em relação à implementação de um algoritmo para cálculo do caminho mínimo entre dois pontos. Para isso, foi implementado uma estrutura de dados do tipo TGrafo, para guardar as informações mais relevantes que serão necessárias para a execução do problema em apenas uma estrutura de dados.

A estrutura do TAD é a seguinte:

```
typedef struct MatrizEntrada

{

    int peso; // Inteiro para guardar o peso de cada célula da matriz

    int id; // Inteiro para guardar a numeração das casas da matriz da entrada

} MatrizEntrada;


typedef struct TGrafo

{

    int **Matriz; // Grafo do estilo de matriz de adjacência

    MatrizEntrada **Mapa; // Matriz de pesos e id's da matriz da entrada (definida no struct MatrizEntrada acima)
```

```
    int NVertices; // Numero de vértices totais do grafo
    int Origem; // ID do vértice origem na matriz mapa
    int Termino; // ID do vértice termino na matriz mapa
} TGrafo;
```

Desse modo, todas as informações relevantes estarão contidas nessa estrutura de dados, como já foi citado anteriormente.

Para a montagem do grafo, foi utilizada uma estratégia de leitura da matriz e registro de caminhos interligados, a partir da montagem do grafo de tamanho N^2, sendo N o número de elementos contidos no mapa da arena. A lógica pensada foi pegar cada casa da arena e fazer o caminhamento de tal modo que fosse considerada o duplo caminhamento (grafo não dirigido) interligando todos os vértices que fossem possíveis de serem alcançados. Desse modo, será gerada uma matriz N x N que conterá em suas casas o peso de interligação de cada vértice identificado no eixo x a cada vértice identificado no eixo y. Para isso, foram utilizadas as estratégias de:

- Caso seja possível o caminhamento, o grafo recebe o valor do peso interligando as duas células;
- Caso a célula seja um atalho, é buscado no mapa da arena outros atalhos, para interligá-los, de modo que o grafo receba a ligação entre esses dois vértices com peso 0 para o seu caminhamento.
- Caso não seja possível o caminhamento (por estar fora dos caminhamentos de dx ou dy ou algum dos dois vértices (ou ambos) ser (em) obstáculos), o grafo recebe o valor do peso de INT_MAX, que contém o valor máximo que pode ser alcançado por um inteiro com sinal e será convencionado como Infinito em toda a execução do programa.

Logo, têm-se um grafo do tipo Matriz de Adjacência que contém todas as possíveis ligações entre os vértices, de modo a representa-los com seus respectivos pesos de caminhamento.

Já para o caminhamento mínimo, foi utilizado uma função baseada no algoritmo de Dijsktra, que tem como seu objetivo encontrar o peso mínimo para ir de um vértice em um grafo para outro. O pseudocódigo utilizado para a construção do algoritmo foi:

Enquanto VérticeAtual != VérticeDestino

Menor_distância ← infinito;

Distancia_Parcial ← Distancia_Atual;

Para cada vértice do grafo

Se ele nunca foi visitado, calcule a distância mínima para o menor caminho

Se algum dos movimentos for impedido, atribua nova_distancia ← Infinito

Se o movimento for válido, nova_distancia ← distancia_parcial + peso da célula atual no mapa da arena

Atribua a posição do vetor distancia essa nova distância mínima

Se essa distância mínima for menor que a menor_distancia, substitua o valor de menor_distancia pelo valor da distância mínima calculada

Fim para

Se o vértice atual for igual ao vértice passado, o caminhamento não será mais possível a partir daquele ponto, pois ele não tem mais pra onde caminhar.

Fim enquanto

Desse modo, o menor caminho entre dois vértices será encontrado e armazenado no vetor distancia, sendo que a posição n, sendo n igual ao vértice destino, terá em si a menor distância de caminhamento possível.

Análise de Complexidade

Considerando N como o número de elementos dados na matriz de entrada (mapa da arena), sendo que N = dimx * dimy, foram calculadas as complexidades a seguir:

TGrafo *alocaGrafo(int dimx, int dimy)

Essa função é regida pela complexidade de $O(N^2)$ em seu tempo, pelo fato da alocação de uma matriz de dimensões N x N para representar o grafo do mapa da arena. Já a complexidade do espaço será $O(6+N+N^2)$, pelo fato das alocações da estrutura de dados, da matriz que representará o mapa da arena e pela matriz de adjacência que representará o grafo.

TGrafo *desalocaGrafo(TGrafo *Grafo, int dimx, int dimy)

Essa função é regida pela complexidade de $O(N^2)$ em seu tempo, pelo fato da desalocação de uma matriz de dimensões N x N previamente alocada. Já a complexidade do espaço será O(1), uma vez que nenhum espaço é utilizado para nenhuma atribuição relevante nessa função.

TGrafo *montaMatrizAdj(TGrafo *Grafo, int dimx, int dimy, int dx, int dy)

Essa função é regida pela complexidade de $O(N + N^2 + N^2)$ em seu tempo, uma vez que vários loops aninhados são utilizados para atribuir a matriz de adjacência os respectivos cálculos dos pesos de caminhamentos que serão possíveis de serem analisados. Já a complexidade do espaço será O(1) no melhor caso, pois nenhuma operação relevante será realizada para gastar algum espaço que já não esteja alocado previamente. No pior caso, será O(2N), pois terão 2 variáveis para cada elemento na matriz do mapa da arena.

int *alocaVetor(int tam)

Essa função realiza a alocação de um vetor simples para ser utilizado na função Dijsktra. Portanto, essa função terá uma complexidade de tempo igual a O(1), pois somente uma operação relevante é realizada, que é a alocação dinâmica com a operação calloc. Já em relação a complexidade do espaço, ela é O(N), uma vez que o espaço que será alocado terá o tamanho do número de vértices da matriz da arena.

int *desalocaVetor(int *vetor)

Essa função terá complexidade de espaço e tempo iguais à O(1), pois não é gasto nenhum espaço para atribuição para realizar a operação desejada e a operação só é realizada 1 vez apenas.

void Dijkstra(TGrafo *Grafo)

Essa função terá complexidade de tempo igual a $O(N + N^2)$, no pior caso, uma vez que ela possui dois loops que andarão desde o vértice da origem ao vértice do destino, sendo que o segundo faz um caminhamento de cálculo do caminho mínimo para cada vértice. Já em relação à complexidade do espaço, O(2 + N + N) é o limite superior, pois são utilizadas 2 variáveis de controle auxiliares e dois vetores de tamanho N para o cálculo da menor distância entre dois pontos do grafo.

int main(int argc, char *argv[])

Essa função terá complexidade de tempo igual à $O(N^2)$, uma vez que ela é regida pela função de maior complexidade de tempo de execução. Já a complexidade de espaço será O(12 + N), uma vez que são utilizadas 12 variáveis auxiliares para a execução da função principal do programa e uma leitura de todos os elementos da matriz da arena (N elementos).

Avaliação Experimental

Para esse trabalho prático, foram desenvolvidos 2 tipos de avaliações experimentais. Para tais tempos de execução, foram utilizados o comando time ./tp2.exe entrada.txt [...]. Os tempos de execução e as suas respectivas discussão estão contidas abaixo:

1. Casos de teste gerador por mim:
    a. Uma matriz 100 x 100
        dx = dy = 0

        real *0m0.613s*

        user *0m0.532s*

        sys *0m0.080s*

        dx = dy = 1

        real *0m0.335s*

        user *0m0.272s*

        sys *0m0.060s*

        dx = dy = 2

        real *0m0.339s*

        user *0m0.288s*

        sys *0m0.048s*

        dx = dy = 3

real *0m0.336s*

user *0m0.260s*

sys *0m0.076s*

b. Uma matriz 200 x 200
dx = dy = 0

real 0m0.551s

user 0m0.468s

sys 0m0.080s

dx = dy = 1

real 0m0.337s

user 0m0.256s

sys 0m0.076s

dx = dy = 2

real 0m0.335s

user 0m0.248s

sys 0m0.084s

dx = dy = 3

real 0m0.334s

user 0m0.268s

sys 0m0.064s

c. Uma matriz 300 x 300

dx = dy = 0

real 0m9.026s

user 0m7.560s

sys 0m1.436s

dx = dy = 1

real 0m5.754s

user 0m4.240s

sys 0m1.512s

dx = dy = 2

real 0m5.718s

user 0m4.132s

sys 0m1.584s

dx = dy = 3

real 0m5.706s

user 0m4.188s

sys 0m1.516s

2. Casos de teste dados no trabalho prático (Toy's)
   time ./tp2.exe toyA.txt 0 0 1 2 0 0

   real 0m0.001s

   user 0m0.000s

   sys 0m0.000s

   time ./tp2.exe toyA.txt 0 0 1 2 1 3

   real 0m0.000s

   user 0m0.000s

   sys 0m0.000s

   time ./tp2.exe toyA.txt 0 1 4 3 2 1

   real 0m0.000s

   user 0m0.000s

   sys 0m0.000s

   time ./tp2.exe toyB.txt 0 1 0 9 0 0

   real 0m0.001s

   user 0m0.000s

   sys 0m0.000s

   time ./tp2.exe toyB.txt 7 7 2 6 1 2

   real 0m0.001s

   user 0m0.000s

   sys 0m0.000s

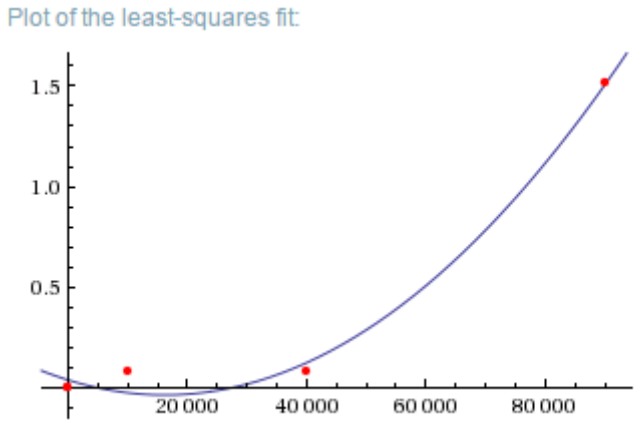
   time ./tp2.exe toyB.txt 7 7 0 0 1 1

   real 0m0.001s

   user 0m0.000s

   sys 0m0.000s

Desse modo, foi possível gerar um gráfico com regressão quadrática, pelo fato da complexidade de tempo do programa ser $O(N^2)$:



Também foi gerada função quadrática que rege o comportamento do programa em relação ao tempo:

$$2.83236 \times 10^{-10}\, x^2 - 9.17474 \times 10^{-6}\, x + 0.0364161$$

Portanto, pode-se concluir que ao longo que a entrada vai aumentando, a complexidade do tempo vai aumentando quadraticamente, o que comprova a complexidade de tempo calculada na seção “Análise de Complexidade” ter sido $O(N^2)$ para o programa.