TP2: Eu, robô

Alef Henrique de Castro Monteiro

31 de maio de 2016

1 Introdução

A tarefa do Trabalho Prático 2 é a simulação de uma competição de robótica. Mais precisamente, o trabalho tem por finalidade verificar se é possível ou não o deslocamento de um robô em uma arena de competição. Cada robô começa em uma determinada posição e deve ser capaz de navegar até uma posição final, de forma que seu deslocamento desvie dos obstáculos presentes na arena de competição e obedeça às restrições de movimento impostas a ele.

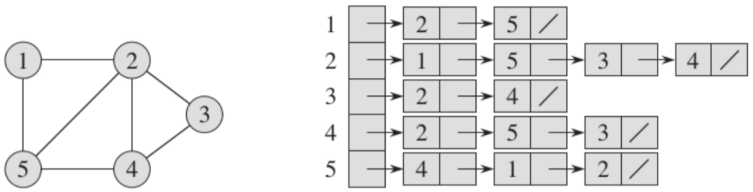
Para a realização da tarefa, o problema é modelado em um grafo. Cada vértice do grafo representa uma das coordenadas(posições) da arena de competição. As arestas, por sua vez, representam os movimentos possíveis de um robô, de acordo com as condições do terreno e de acordo com as restrições do movimento. Como a dificuldade de cada área da arena pode variar devido às condições do terreno, a dificuldade relativa à cada coordenada é representada pelo peso de cada aresta. Neste trabalho, o objetivo é verificar a menor custo possível para o deslocamento do robô entre a posição inicial I e a posição final F. Para cumprir a tarefa, o será implementado e discutido o algoritmo de Dijkstra, que calcula o caminho de menor custo entre dois vértices de um grafo G. Além da implementação do problema modelado em forma de um grafo, o trabalho discute também a eficiência do algoritmo utilizado para a resolução do problema e, ainda, a influência que as restrições de movimento e os obstáculos podem causar no algoritmo.

2 Solução do problema

Nesta seção do trabalho, serão apresentados os detalhes da implementação do programa utilizado no trabalho. A solução do problema é iniciada com a verificação dos parâmetros passados no momento da execução do programa, pelo terminal. Nesse momento, são obtidas as posições inicial e final do robô, ou seja, é informado ao programa o vértice em que se inicia e o vértice em que deve ser finalizado o deslocamento no grafo. Além disso, são passados como parâmetro também as restrições de movimento do robô e o arquivo em que contém o mapa da arena de competição. Ao ser executado, o programa verifica o mapa da arena de competição no arquivo de entrada. O mapa é fornecido em forma de uma matriz NxM, onde o valor de cada célula corresponde à dificuldade do robô para passar por aquela posição.

Inicialmente, o programa coleta todo esse conteúdo e coloca em uma matriz com N linhas e M colunas. Depois de armazenar o mapa em uma matriz, o programa cria um grafo G com NxM vértices, no qual cada um de seus vértices corresponde a uma célula da matriz, ou seja, cada vértice corresponde a uma coordenada da arena de competição. A implementação do grafo é feita por meio de uma lista de adjacências, conforme **figura 1**. A escolha pela implementação do grafo como uma lista de adjacência se deve ao fato de esta implementação permitir ao algoritmo economizar o espaço ocupado pelo grafo na memória, já que o programa não tem que reservar espaço para arestas não existentes. Outro grande motivo da escolha pela implementação do grafo como lista de adjacência é que, mesmo quando não há restrições de movimento para o deslocamento do robô, o número máximo de vértices do grafo não chega a ser V^2 (V é o número de vértices), pois cada vértice, se não for uma atalho, tem no máximo 4 adjacentes. Dessa forma, como a implementação do grafo como uma lista de adjacência é recomendada para grafos esparsos, ou seja, grafos em que o número A de arestas é bem menor que o V^2, a escolha pela lista de adjacência mostra-se mais eficaz que uma eventual escolha pela matriz de adjacência.

Figura 1: Representação de um grafo como lista de adjacência



Para adicionar as arestas do grafo G, é utilizada a função 'Mapa_grafo'. Inicialmente, a função percorre todas as células do mapa e chama a função 'Insere_aresta_movimento' para inserir as 4 arestas que podem sair desse vértice, cada uma no sentido de uma das diagonais da célula. Conforme mostrado na **figura 3**, cada uma dessas arestas representa o movimento em **L** feito pelo robô, de acordo com as restrições de movimento. No entanto, para cada um dos 4 vértices alcançáveis a partir dessa célula, existem duas opções de caminho (**figura 2**).

	0	1	2	3	4	5
0	198	0	0	35	20	14
1	232	0	0	14	21	13
2	147	72	61	66	33	16
3	87	78	81	141	130	40
4	38	45	56	139	83	97

Figura 2: Dois caminhos possíveis entre (2,2) e (0,4), com restrição de movimento $d_x = 2$ e $d_y = 2$

	0	1	2	3	4	5
0	198	0	0	35	20	14
1	232	0	0	14	21	13
2	147	72	61	66	33	16
3	87	78	81	141	130	40
4	38	45	56	139	83	97

Figura 3: 4 arestas possíveis a partir de (2,2), com restrição de movimento $d_x = 2$ e $d_y = 2$

A função 'Insere_aresta_movimento' é que faz a verificação de qual dos dois caminhos possíveis até cada uma das outras 4 células tem o menor peso e insere a aresta até o vértice com peso igual ao do caminho 'mais leve', caso o caminho pertença ao mapa e caso não haja obstáculos no caminho. Para calcular o peso dos caminhos, são utilizadas as funções 'Calcula_peso_eixo_x' e 'Calcula_peso_eixo_y', que visitam cada uma das células presentes no caminho e armazenam seu peso, além de verificar se a célula representa um obstáculo ou não. Além disso, a função 'Insere_aresta_movimento' verifica se há algum atalho que possa ser utilizado pelo robô no momento em que ele finaliza ou está prestes a iniciar um movimento. Se houver algum atalho que possa ser utilizado, a função armazena as coordenadas da célula representada pelo atalho em um vetor. Dessa forma, a função 'Mapa_grafo' poderá, posteriormente, inserir arestas que conectam todos esses atalhos com todos os outros atalhos do mapa. O funcionamento das funções 'Mapa_grafo' e 'Insere_aresta_movimento' pode

ser detalhado a partir dos pseudo-códigos a seguir.

Algoritmo 1: Mapa_grafo

Entrada: Grafo G, Matriz *mapa*, M, N, restricaoX, restricaoY, Posição inicial I, Posição final F

```
Cria vetor atalhos para armazenar as posições dos atalhos do mapa
for i ← 1 to N do
    for j ← 1 to M do
        Insere_aresta_movimento(G, mapa, atalhos, M, N, i, j, Num_atalhos, restricaoX, restricaoY)
    end
end
for i ← 1 to Num_atalhos do
    for j ← 1 to Num_atalhos do
        Peso ⟵ 0
        if atalhos[i] ≠ atalhos[j] then
            Insere_aresta(G, atalhos[i], atalhos[j], Peso)
        end
    end
end
```

Algoritmo 2: Insere_aresta_movimento

Entrada: Grafo G, Matriz *mapa*, Vetor *atalhos*, M, N, i, j, $Num\_atalhos$, restricaoX, restricaoY

```
if mapa[i][j] = 0 then
    if restricaoX = 0 e restricaoY = 0 then
        Insere_aresta_sem_restricoes()
    end
    else
        if (j + restricaoX) < M e (i + restricaoY) < N then
            //Calcula o peso e insere aresta que representa o movimento no sentido direita-baixo
            Insere_aresta_direita_baixo()
        end
        if (j + restricaoX) < M e (i − restricaoY) ≥ 0) then
            // Calcula o peso e insere aresta que representa o movimento no sentido direita-cima
            Insere_aresta_direita_cima()
        end
        if (j − restricaoX) ≥ 0 e (i + restricaoY) < N) then
            // Calcula o peso e insere aresta que representa o movimento no sentido esquerda-baixo
            Insere_aresta_esquerda_baixo()
        end
        if (j − restricao_x) ≥ 0 e (i − restricao_y) ≥ 0) then
            // Calcula o peso e insere aresta que representa o movimento no sentido esquerda-cima
            Insere_aresta_esquerda_cima()
        end
    end
end
k ⟵ 0
else if mapa[i][j] = −1 then
    atalhos[k] ⟵ (i, j)
end
```

Depois de criar o grafo, conforme todas as especificações passadas, o programa realiza a execução do algoritmo de Dijkstra, que atribui a cada vértice do grafo G o custo mínimo para se chegar a ele, a partir de um vértice inicial I do grafo. Para a resolução do problema, poderia ser utilizado também o algoritmo de Bellman-Ford. A escolha pelo algoritmo de Dijkstra em detrimento do de Bellman-Ford se deve simplesmente ao fato de a complexidade do primeiro ser menor. Seja A o número de arestas do grafo e V o número de vértices. O algoritmo de Bellman-Ford tem complexidade $O(V.A)$, enquanto o algoritmo utilizado pelo programa tem complexidade

$O(A.\log(V))$. O funcionamento do algoritmo de Dijkstra pode ser verificado a partir do pseudo-código a seguir.

Algoritmo 3: Djikstra [1]

```
   Entrada: Grafo G, Vertice_origem, Num_vertices
 1 for i ← 1 to Num_vertices do
 2 |  G[i].peso_origem ⟵ infinito
 3 |  G[i].antecessor ⟵ NULL
 4 end
 5 Vertice_origem.peso_origem ⟵ 0
 6 S ⟵ Conjunto vazio
 7 Heap ⟵ Vertices de G
 8 while Heap nao esta vazio do
 9 |  Retira do Heap o vertice u de menor Peso_origem
10 |  S ⟵ S + {u}
11 |  for cada vertice v adjacente de u do
12 |  |  if u pertence ao conjunto S then
13 |  |  |  Relax(u, v, Peso_aresta(u, v))
14 |  |  end
15 |  end
16 end
```

Algoritmo 4: Relax [1]

```
   Entrada: Vertice v1, Vertice v2, Peso_aresta(v1, v2)
 1 if v2.peso_origem > (v1.peso_origem + Peso_aresta(v1, v2)) then
 2 |  v2.peso_origem ⟵ v1.peso_origem + Peso_aresta(v1, v2)
 3 |  v2.antecessor ⟵ v1
 4 end
```

Após a execução do algoritmo de Dijkstra no grafo G, fica armazenado em cada um dos vértices o custo mínimo para se chegar a ele a partir do vertice inicial I. No entanto, o programa busca o custo mínimo para se chegar em uma posição F específica do mapa, ou seja, o custo mínimo para ir de I até F. Dessa forma, o programa utiliza a função 'Calcula_peso_minimo' para verificar se o caminho de I até F é possível, dadas as restrições de movimento do robô e os obstáculos do mapa. Os detalhes da função 'Calcula_peso_minimo' podem ser observados a partir do pseudo-código a seguir.

Algoritmo 5: Calcula_peso_minimo

```
   Entrada: Grafo G, Matriz mapa, M, N, restricaoX, restricaoY, Posição inicial I, Posição final F
 1 if G[F].peso_origem = infinito then
 2 |  Imprime − 1
 3 end
 4 else
 5 |  ImprimeG[F].peso_origem
 6 end
```

Após a execução da função 'Calcula_peso_minimo', o programa imprime o valor do custo mínimo total do deslocamento. Caso seja possível o deslocamento do robô entre I e F, a função imprime o custo mínimo do caminho $I - F$. Caso contrário, o programa imprime o valor -1 para indicar que o deslocamento até F é impossível. Após a impressão da saída, o programa é finalizado.

3 Análise de Complexidade

O trabalho é composto do programa principal tp2.c e mais 2 tipos abstratos de dados (TADs): Heap e Grafo. Nesta seção, cada uma das principais funções será analisada separadamente e, por fim, será analisado o programa principal. Para as análises a seguir, será utilizada a seguinte notação: M é o número de colunas do mapa fornecido na entrada, N é o número de linhas do mapa fornecido na entrada, $V = M * N$ é o número de vértices do grafo gerado pelo programa, A é o número de arestas do grafo e, por fim, Rx e Ry representam as

restrições de movimento nos eixos x e y, respectivamente.

3.1 Análise de complexidade das principais funções

- **void Cria_grafo_lista()** - A função cria um novo grafo e inicializa os valores de suas varáveis. Para isso, a função precisa realizar operações para cada um dos vértices do grafo e, portanto, a função é $O(V)$.
- **void Dijkstra()** - A função, que está representada no **Algoritmo 3**, é composta pelo algoritmo de Djikstra e, portanto, tem complexidade $O(A*\log V)$[1], já que utiliza um heap para armazenar os vértices.
- **void Mapa_matriz()** - A função faz a leitura de cada uma das células do mapa contido no arquivo de entrada e insere o conteúdo da célula em uma matriz. Como cada célula do mapa representará um vértice do grafo criado pelo programa, a complexidade da função é $O(V)$.
- **int Calcula_peso_eixo_x()** - Para calcular o peso de um movimento, essa função visita cada uma das células percorridas pelo robô no eixo x em um mesmo movimento. Além das operações de custo $O(1)$, a função sempre realiza um loop, que depende da restrição de movimento Rx no eixo x. Dentro desse loop, todas as operações tem custo constante. Dessa forma, a complexidade final da função é $O(Rx)$.
- **int Calcula_peso_eixo_y()** - Essa função é semelhante à função anterior. No entanto, dessa vez a função visita as células do movimento do robô que estão no eixo y. Portanto, a complexidade da função é $O(Ry)$.
- **void Insere_aresta_direita_baixo()** - A função insere a aresta que representa um possível movimento do robô em diagonal, no sentido $direita - baixo$. No pior caso, a função chama 2 vezes a função 'Calcula_peso_eixo_x' - $O(Rx)$, 2 vezes a função 'Calcula_peso_eixo_y' - $O(Ry)$ e uma vez a função 'Insere_aresta', que tem custo O(1) para inserir uma nova aresta no grafo. Dessa forma, a função tem complexidade $O(Rx + Ry)$.
- **void Insere_aresta_direita_cima()** - A única diferença entre esta função e a anterior é que esta função insere a aresta que representa o movimento do robô no sentido $direita - cima$. Assim como a anterior, no pior caso, a função chama 2 vezes a função 'Calcula_peso_eixo_x' - $O(Rx)$, 2 vezes a função 'Calcula_peso_eixo_y' - $O(Ry)$ e uma vez a função 'Insere_aresta', que tem custo O(1) para inserir uma nova aresta no grafo. Dessa forma, a função tem complexidade $O(Rx + Ry)$.
- **void Insere_aresta_esquerda_baixo()** - A função é semelhante às duas anteriores, porém a aresta inserida representa o movimento do robô no sentido $esquerda - baixo$. Portanto, a complexidade da função é $O(Rx + Ry)$.
- **void Insere_aresta_esquerda_cima()** - Assim como as anteriores, essa função insere uma aresta que representa um possível movimento do robô. No entanto, essa função é utilizada para inserir uma aresta que representa o movimento no sentido $esquerda - cima$. Portanto, a complexidade da função é $O(Rx + Ry)$.
- **void Insere_aresta_sem_restricoes()** - Essa função é utilizada para inserir as 4 arestas que representam os possíveis movimentos do robô a partir de uma célula. No pior caso, a função chama 2 vezes a função 'Calcula_peso_eixo_x' - $O(Rx)$, 2 vezes a função 'Calcula_peso_eixo_y' - $O(Ry)$ e 4 vezes a função 'Insere_aresta', que tem custo O(1) para inserir uma nova aresta no grafo. Dessa forma, a função tem complexidade $O(Rx + Ry)$.
- **void Insere_aresta_movimento()** - A função, representada pelo **Algoritmo 2**, pode chamar a função 'Insere_aresta_sem_restricoes'- $O(Rx + Ry)$ uma vez (linha 3), caso não haja restrições ao movimento do robô. Se houver restrições, a função pode inserir até 4 arestas (linhas 6-21). Cada uma das funções chamadas para inserir as arestas tem complexidade $O(Rx + Ry)$. Portanto, a complexidade da função também é $O(Rx + Ry)$.
- **void Mapa_grafo()** - A função, representada pelo **Algoritmo 1**, é composta por dois loops principais (linhas 2 e 7). O primeiro loop (linha 2) chama a função 'Insere_aresta_movimento'- $O(Rx + Ry)$ para cada um dos vértices do grafo. Portanto, o primeiro loop é $O(V * (Rx + Ry))$. O segundo loop(linha 7), por sua vez, chama a função 'Insere_aresta'- $O(1)$ para cada uma das células do mapa que representam um atalho. Como o número de atalhos do mapa é no máximo igual ao número de vértices, o loop tem complexidade $O(V)$. Por fim, a complexidade final da função é $O(V * (Rx + Ry))$ nos casos em que há restrição ao movimento do robô e $O(V)$ quando não há restrição.
- **void Calcula_peso_minimo()** - A função, representada pelo **algoritmo 5**, verifica o peso do caminho mínimo entre os vértices após a execução do algoritmo de Djikstra. Essa função contém apenas operações $O(1)$ e, portanto, tem complexidade $O(1)$.

- **void Libera_memoria_grafo()** - A função é utilizada no fim do programa para liberar a memória alocada para o grafo. Essa função realiza a operação 'free' para todos os elementos da lista de adjacência de cada um dos vértices. Dessa forma, a complexidade da função é $O(A)$, já que o número de elementos das listas de adjacência dos vértice corresponde ao número de arestas do grafo.

3.2 Análise de complexidade do programa Principal

No pior caso, o programa principal chama uma vez as funções:

- 'Mapa_matriz' - $O(V)$
- 'Cria_grafo_lista' - $O(V)$
- 'Mapa_grafo' - $O(V * (Rx + Ry))$
- 'Djikstra' - $O(A * \log(V))$
- 'Calcula_peso_minimo' - $O(1)$
- 'Libera_memoria_grafo' - $O(A)$

Além disso, o programa realiza algumas outras operações $O(1)$. Dessa forma, o custo final do programa é $O(V + V + V * (Rx + Ry) + A * \log V + A) = O(V * (Rx + Ry) + A * \log V)$. No entanto, A pode ficar bem próximo de $V * (V - 1) = V^2 - V$, caso o número de atalhos no mapa seja bem próximo do número de células do mapa, e, assim, A seria $O(V^2)$. Como a soma das restrições $(Rx + Ry)$ nunca é maior que V, $O(A * \log V)$ domina assintoticamente $O(V * (Rx + Ry))$. Dessa forma, pode-se afirmar que a complexidade do programa fica $O(A * \log V)$.

3.3 Análise de Complexidade do Espaço

Durante o programa, é necessária a alocação de memória para que as células do mapa passado pela entrada formem uma matriz. Como o número de células do mapa é igual ao número de vértices do grafo criado pelo programa posteriormente, a complexidade espacial dessa matriz é $O(V)$. Além disso, o grafo G armazena a lista dos vértices e suas respectivas listas de adjacência, o que tem custo $O(V + A)$. O vetor que representa a lista de prioridades (heap) no algoritmo de Djiskstra é do tamanho do número de vértices do grafo. Dessa forma, esse vetor tem custo de armazenamento $O(V)$. Assim, a complexidade de espaço total do programa é $O(V + (V + A) + V) = O(V + A)$.

4 Análise de Experimentos

4.1 Metodologia

Para a realização do estudo experimental, foram realizados 14 testes. Os testes utilizados no experimento foram feitos com a finalidade de avaliar a influência das restrições de movimento Rx e Ry, do número de obstáculos e do número de atalhos no tempo de execução do programa. Em todos os testes, foram gerados valores entre 1 e 1000 de forma aleatória para os pesos. Nos testes em que foram utilizados atalhos ou obstáculos, o número destes é fixo, porém a posição dos atalhos ou dos obstáculos foi gerada aleatoriamente. Os testes utilizados estão detalhados nas **tabelas 1, 2 e 3**.

	M	N	Restrição no eixo x (Rx)	Restrição no eixo y (Ry)	Posição Inicial do robô (I)	Posição Final do robô (F)	Número de Obstáculos	Quantidade de Atalhos	Número de Vértices (V)
Teste 1	501	501	25	25	0,0	(500,500)	0	0	251001
Teste 2	501	501	50	50	0,0	(500,500)	0	0	251001
Teste 3	501	501	100	100	0,0	(500,500)	0	0	251001
Teste 4	501	501	250	250	0,0	(500,500)	0	0	251001
Teste 5	501	501	500	500	0,0	(500,500)	0	0	251001

Tabela 1: Testes com variação das restrições de movimento(Rx e Ry)

	M	N	Restrição no eixo x (Rx)	Restrição no eixo y (Ry)	Posição Inicial do robô (I)	Posição Final do robô (F)	Quantidade de Obstáculos	Quantidade de Atalhos	Número de Vértices (V)
Teste 6	501	501	0	0	0,0	(500,500)	20%	0	251001
Teste 7	501	501	0	0	0,0	(500,500)	40%	0	251001
Teste 8	501	501	0	0	0,0	(500,500)	60%	0	251001
Teste 9	501	501	0	0	0,0	(500,500)	80%	0	251001
Teste 10	501	501	0	0	0,0	(500,500)	100%	0	251001

Tabela 2: Testes com variação do número de obstáculos

	M	N	Restrição no eixo x (Rx)	Restrição no eixo y (Ry)	Posição Inicial do robô (I)	Posição Final do robô (F)	Quantidade de Obstáculos	Quantidade de Atalhos	Número de Vértices (V)
Teste 11	51	51	0	0	0,0	(50,50)	0	20%	2601
Teste 12	51	51	0	0	0,0	(50,50)	0	40%	2601
Teste 13	51	51	0	0	0,0	(50,50)	0	60%	2601
Teste 14	51	51	0	0	0,0	(50,50)	0	80%	2601

Tabela 3: Testes com variação do número de atalhos

Para a execução dos testes, foi acrescentada ao programa uma função para verificar o número de arestas inseridas no grafo criado. Dessa forma, pode-se realizar também um estudo para verificar a interferência das variáveis na quantidade A de arestas.

A implementação dos experimentos foi feita com a utilização do compilador GCC, utilizando as flags $-O3$, $-g$ e $-Wall$. Os testes foram executados em uma máquina com 8GB de memória e com processador Intel i5 2.7 GHz.

4.2 Resultado dos experimentos

Após a execução, foi verificado tempo de execução de cada um dos testes do experimento. Além do tempo de execução, foi verificado também o número de arestas criadas no grafo gerado pelo programa de cada teste. Os resultados dos experimentos com a variação das restrições de movimento Rx e Ry, do número de obstáculos e do número de atalhos estão detalhados nas **tabelas 4, 5 e 6**, respectivamente. Após cada uma das tabelas, está um gráfico da variável em função do tempo.

Tabela 4: Resultado dos testes com variação das restrições de movimento(Rx e Ry)

	M	N	Restrição no eixo x (Rx)	Restrição no eixo y (Ry)	Número de Obstáculos	Quantidade de Atalhos	Número de Vértices (V)	Número de Arestas (A)	Tempo
Teste 1	501	501	25	25	0	0	251001	906304	0m0.028s
Teste 2	501	501	50	50	0	0	251001	813604	0m0.026s
Teste 3	501	501	100	100	0	0	251001	643204	0m0.024s
Teste 4	501	501	250	250	0	0	251001	252004	0m0.015s
Teste 5	501	501	500	500	0	0	251001	4	0m0.008s

Gráfico 1: Restrição de movimento $Rx = Ry$ x tempo

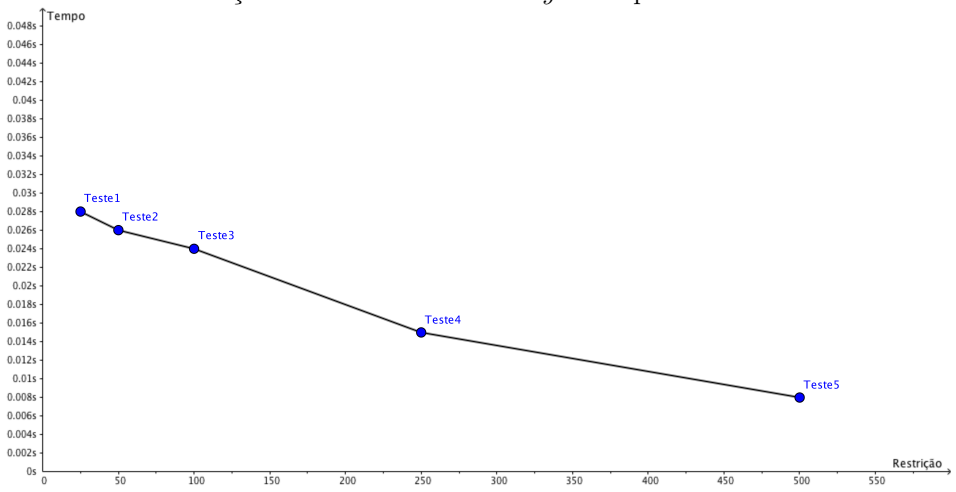


Tabela 5: Resultado dos testes com variação do número de obstáculos

	M	N	Restrição no eixo x (Rx)	Restrição no eixo y (Ry)	Quantidade de Obstáculos	Quantidade de Atalhos	Número de Vértices (V)	Número de Arestas (A)	Tempo
Teste 6	501	501	0	0	20%	0	251001	622444	0m2.440s
Teste 7	501	501	0	0	40%	0	251001	282149	0m0.014s
Teste 8	501	501	0	0	60%	0	251001	81946	0m0.009s
Teste 9	501	501	0	0	80%	0	251001	21848	0m0.009s
Teste 10	501	501	0	0	100%	0	251001	0	0m0.009s

Gráfico 2: Quantidade de obstáculos x tempo

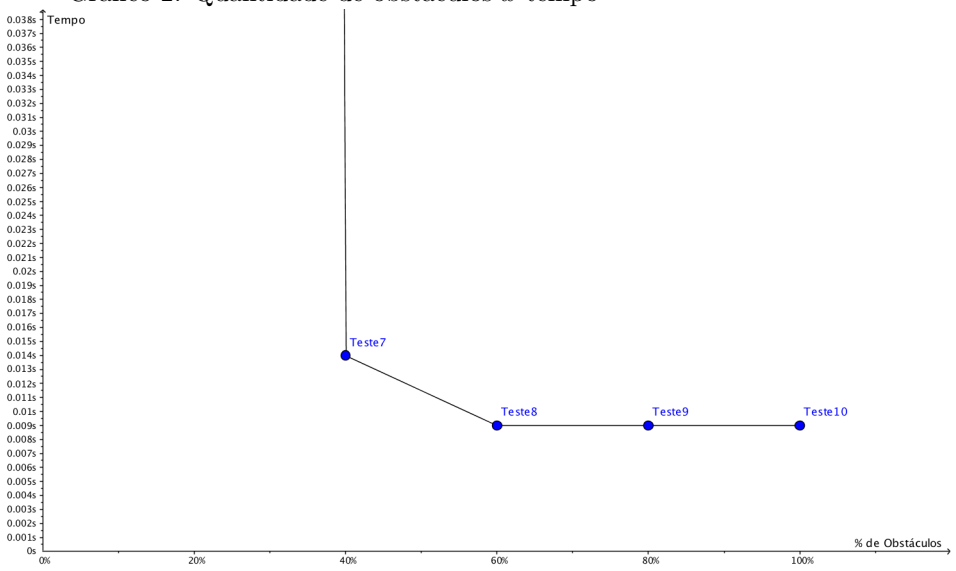
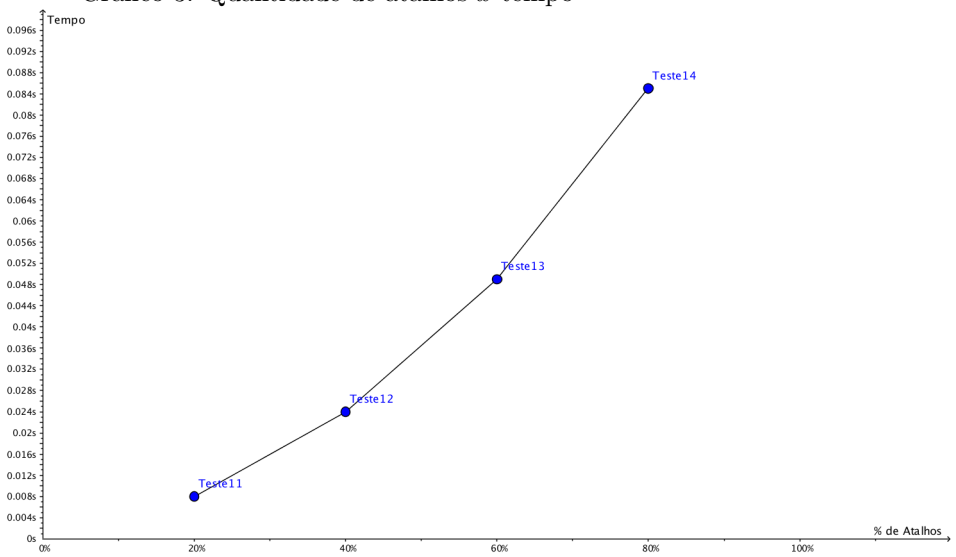


Tabela 6: Resultado dos testes com variação do número de atalhos

	M	N	Restrição no eixo x (Rx)	Restrição no eixo y (Ry)	Quantidade de Obstáculos	Quantidade de Atalhos	Número de Vértices (V)	Número de Arestas (A)	Tempo
Teste 11	51	51	0	0	0	20%	2601	268058	0m0.008s
Teste 12	51	51	0	0	0	40%	2601	1041538	0m0.024s
Teste 13	51	51	0	0	0	60%	2601	2361038	0m0.049s
Teste 14	51	51	0	0	0	80%	2601	4176558	0m0.085s

Gráfico 3: Quantidade de atalhos x tempo



4.3 Análise dos Resultados

4.3.1 Influência do tamanhos das restrições Rx e Ry

Quanto maior são as restrições, menor é o número de arestas criadas (**Tabela 4**). A explicação disso é verificada ao analisar o funcionamento do programa (**Seção 2**), que cria apenas uma aresta para cada movimento possível do robô. Como o número de movimentos possíveis do robô fica mais restrito, ele diminui com o aumento das restrições e, consequentemente, faz com que o número de arestas criadas diminua. Como não há atalhos e nem obstáculos nos **Testes 1-5**, as únicas variáveis que modificam o número de arestas são as restrições de movimento Rx e Ry.

Na análise de complexidade realizada na **Seção 3.2**, verificou-se que a complexidade do programa é $O(A * \log V)$. Como o tamanho das restrições está diretamente ligado ao número A de arestas, o tamanho das restrições deve influenciar diretamente no tempo de execução do programa.

Ao modificar apenas o valor das restrições de movimento Rx e Ry nos **Testes 1-5**, pode-se verificar a partir do **Gráfico 1** que o tempo decresce com o aumento das restrições de movimento. De acordo com o gráfico obtido, o comportamento da curva do tempo do tempo em função das restrições tende a ser linear. Portanto, além de confirmar o resultado da análise de complexidade, o gráfico ainda confirma que o tamanho das restrições Rx e Ry é inversamente proporcional ao número de arestas e ao tempo, quando não existem atalhos e obstáculos no mapa.

4.3.2 Influência do número de obstáculos do mapa

Ao analisar o comportamento do tempo de execução a partir da interferência causada pelo número de obstáculos do mapa, verifica-se que o tempo diminui com o aumento da quantidade de obstáculos. Além disso, pode ser observado também que o número de arestas diminui com o aumento dos obstáculos. De acordo com o **Gráfico 2** e com a **Tabela 5**, o comportamento do tempo não é linear, pois a taxa de decrescimento do tempo diminui com o aumento do número de obstáculos.

Ao criar obstáculos, as possibilidades de movimento do robô diminuem bastante e, portanto, o número de arestas também diminui. No entanto, a interferência dos obstáculos no número de arestas é maior quando as restrições de movimento são maiores. Quando a restrição é maior, por exemplo, cada movimento do robô passa por mais células e, consequentemente, cada célula estará contida em um número maior de caminhos. Assim, com o aumento das restrições, o obstáculo irá obstruir um maior número de arestas. Como os **Testes 6-10** foram feitos sem restrições, a interferência dos obstáculos na quantidade de arestas é menor. Mesmo assim, pode-se verificar que o número de obstáculos interfere bastante no número de arestas e, portanto, no tempo. Assim, o aumento do número de obstáculos do mapa faz com que o tempo de execução diminua exatamente porque o número de arestas do grafo criado diminui.

4.3.3 Influência do número de atalhos no mapa

A influência da quantidade de atalhos do mapa no tempo é verificada a partir da **Tabela 6** e do **Gráfico 3**. A tabela e o gráfico mostram que com a variação do número de atalhos, o tempo cresce mais que linearmente. Além disso, pode ser verificado que a quantidade de atalhos influencia bastante o número de arestas. Na verdade, além das arestas 'normais', relacionadas ao movimento do robô entre células vizinhas do mapa, o programa cria arestas entre todos os atalhos. Assim, quando o número de atalhos no mapa é X, são criadas $X * (X - 1) = X^2 - X$ arestas além das arestas 'normais'. Portanto, o número de atalhos aumenta de forma considerável a quantidade de arestas, sendo esse aumento cada vez maior quando o número de arestas 'normais' diminui, ou seja, quando as restrições de movimento Rx e Ry são maiores. Quando não existem restrições de movimento, o número de arestas normais é $O(4 * V)$, pois cada vértice, com exceção das extremidades do mapa, tem 4 arestas que saem do vértice. Como os **Testes 11-14** foram realizados sem a presença de restrições de movimento e sem obstáculos no mapa, a influência dos atalhos no tempo é quadrática, já que o número de arestas é $O(X^2 - X) + O(4 * V) = O(X^2)$. Essa influência quadrática pode ser observada a partir do **Gráfico 3**, que mostra um crescimento mais que linear do tempo em função da quantidade de atalhos no mapa. Portanto, diferentemente do tamanho das restrições Rx e Ry, o número de atalhos tem influência quadrática no tempo e no número de arestas.

5 Conclusão

O Trabalho Prático 2 abordou a modelagem de um problema em forma de grafo. No caso deste trabalho, o problema era verificar o caminho com o menor custo para que um robô se deslocasse de uma posição I até uma posição F de uma arena de competição de robótica. Na modelagem do problema, cada posição do mapa foi transformado em um vértice do grafo e as arestas do grafo representaram os movimentos possíveis do robô no mapa. Essa transformação do problema em um grafo é uma tarefa complicada e foi o grande desafio deste trabalho. Essa dificuldade é encontrada principalmente por causa dos atalhos e dos obstáculos, já que eles criam situações específicas e devem ser tratados de maneira diferente das células normais. Além de modelar o problema em grafo e resolvê-lo, foi feita uma aprofundada análise de complexidade do programa utilizado no trabalho. Além disso, o trabalho realizou diversos experimentos para analisar a influência das variáveis do problema no tempo de execução do programa. Após o estudo experimental, o trabalho estabeleceu relações entre esta análise de complexidade e os experimentos. Nestas relações, os testes executados comprovaram a complexidade do programa pré-estabelecida pela análise de complexidade e, ainda, mostraram o grau de influência proporcionado por cada variável no tempo de execução.

Finalmente, a partir do grafo criado, o uso do algoritmo de Djikstra mostrou-se bem eficiente para resolver o problema proposto. Vale ressaltar que a grande eficiência do programa está na criação do grafo, onde o número de arestas criadas pode variar conforme os diversos modos de implementação. Neste trabalho, foi proposta uma solução com o objetivo de reduzir ao máximo o número de arestas criadas. Dessa forma, o tempo de execução do programa ficou bastante reduzido, já que as arestas influenciam muito na complexidade temporal do programa (**Seção 3.2**). Entretanto, o programa implementado poderia ter um melhor funcionamento para as operações que envolvem o grafo, caso a implementação do grafo fosse feita com a utilização de Hash para representar as arestas, uma vez que o acesso às arestas seria $O(1)$. Por fim, como o trabalho visa a prática de algoritmos e estruturas de dados trabalhados em sala de aula, optou-se por utilizar as listas de adjacência para representar o grafo.

Referências

[1] Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009.