TP2: Eu, robô

Vitor Cláudio Chaves de Aguiar

2 de junho de 2016

1 Introdução

O objetivo desse trabalho prático é mostrar o caminho menos custoso para que um robô dado uma posição inicial chegue até uma posição final indicada. O robô se encontra em um mapa onde para acessar cada coordenada dele há uma dificuldade específica e ao longo do caminho é possível que encontre obstáculos e atalhos. Obstáculos são coordenadas aonde não se é possível passar, ou seja, nenhuma rota pode ser traçada passando por um obstáculo. Através de um atalho é possível ir para qualquer outro atalho localizado no mapa sem nenhum custo adicional.

As entradas de dados consiste em um arquivo contendo o mapa onde o robô se encontra com os custos de acesso de cada coordenada, a posição x e y inicial do robô, a posição x e y final onde se deve chegar e as restrições de movimentação x e y.

As restrições de movimentação faz com que o robô tenha um número fixo de células que deve percorrer tanto no eixo X quanto no eixo Y por vez. Porém essa movimentação pode ser feita em todas as direções e pode começar tanto pelo eixo X quanto no eixo Y. Um atalho só pode ser acessado se o robô parar nele ao final da sua movimentação. Uma movimentação que passa por um obstáculo não é válida.

A solução apresentada consiste modelar a entrada de dados com a finalidade de representar o problema em forma de um grafo e utilizar de algoritmos vistos em sala de aula para resolver a problemática de achar o custo do caminho de menor custo.

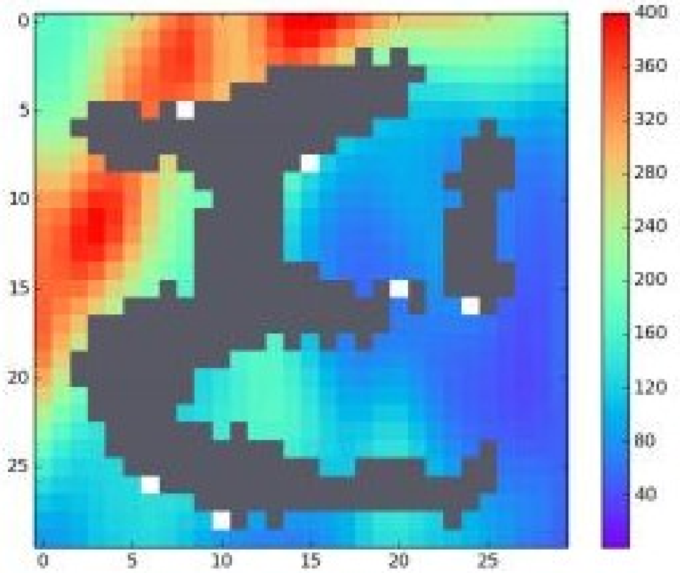

Figura 1: Figura retirada da especificação exemplificando um mapa com os custos de cada célula, obstáculos e atalhos.

2 Solução do problema

Inicialmente é construido uma matriz de inteiros contendo o custo de cada coordenada do mapa. Desta forma é possível visualizar melhor o problema e ter acesso O(1) do custo de acesso de cada coordenada. Para modelar o problema em grafo foi definido a utilização das listas de adjacência, desta forma é possível representar todas as arestas que um vértice possui de uma forma dinâmica.

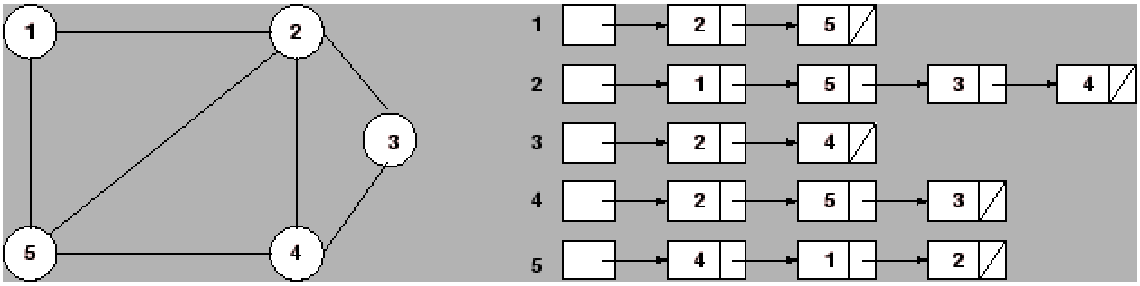

Figura 2: Figura exemplificando uma lista de adjacência.

O fluxo do programa para adicionar todas as arestas é diferente se há ou não restrições para movimentação.

Algorithm 1: Inserir todas as arestas (mapa, comprimento, altura, grafo)

```
for celula ∈ mapa do
    if celula = −1 then
        insere posição no vetor de atalhos
    end
    if restriçãoX = 0 && restriçãoY = 0 then
        Inserir sem restrição (linha, coluna, comprimento, altura, mapa, grafo)
    end
    else
        Inserir com restrição (linha, coluna, comprimento, altura, mapa, grafo)
    end
end
insere aresta entre todos os atalhos
```

Algorithm 2: Inserir com restrição (mapa, grafo, restriçãoX, restriçãoY)

```
M = conjunto de todos os destinos formados pelas movimentações no eixo X e eixo Y
for destino ∈ M do
    if destino ∈ mapa then
        calcula peso movimentando primeiro pelo eixo X
        calcula outro peso movimentando primeiro pelo eixo Y
        verifica qual peso é menor e insere a aresta da direção verificada
    end
end
```

Algorithm 3: Inserir sem restrição (linha, coluna, comprimento, altura, mapa, grafo)

```
// Verifica se pode adicionar aresta à esquerda da célula
if coluna > 0 then
    elemento ⟵ mapa[linha][coluna]
    destino ⟵ mapa[linha][coluna − 1]
    if destino ≠ 0 then
        calcula peso da aresta entre o elemento e o destino
        adiciona aresta (elemento, destino)
    end
end
// Verifica se pode adicionar aresta à direita da célula
if coluna < comprimento − 1 then
    elemento ⟵ mapa[linha][coluna]
    destino ⟵ mapa[linha][coluna + 1]
    if destino ≠ 0 then
        calcula peso da aresta entre o elemento e o destino
        adiciona aresta (elemento, destino)
    end
end
// Verifica se pode adicionar aresta em cima da célula
if linha > 0 then
    elemento ⟵ mapa[linha][coluna]
    destino ⟵ mapa[linha − 1][coluna]
    if destino ≠ 0 then
        calcula peso da aresta entre o elemento e o destino
        adiciona aresta (elemento, destino)
    end
end
// Verifica se pode adicionar aresta embaixo da célula
if linha < altura − 1 then
    elemento ⟵ mapa[linha][coluna]
    destino ⟵ mapa[linha + 1][coluna]
    if destino ≠ 0 then
        calcula peso da aresta entre o elemento e o destino
        adiciona aresta (elemento, destino)
    end
end
```

Os pseudocódigos citados acima se diz respeito a inserção das arestas no grafo. É verificado para cada par de coordenadas no mapa todas as possíveis arestas. Se não houver restrição de movimentação, as possíveis arestas de um par de coordenadas será aresta com o elemento à sua direita, na sua esquerda, em cima e embaixo. Porém se houver restrições, 8 possíveis caminhamentos podem originar no máximo 4 destinos válidos.

A imagem acima retrata uma situação aonde todos os 8 caminhamentos são válidos com restriçãoX = 2 e restriçãoY = 2 criando 4 destinos. Diante disso, a implementação verifica se os 4 destinos possíveis são válidos e, devido a um possível destino poder ser acessado por dois caminhos distintos, é calculado o custo desses dois caminhos e se adiciona a aresta com o custo do caminho menos custoso. Um destino válido é aquele que está presente no mapa e não é um obstáculo, já um caminho válido é aquele que está presente no mapa e não passa por nenhum obstáculo.

Cada elemento do mapa que é analisado para inserir aresta é verificado também se ele possui valor igual a -1. Se isso ocorrer, a sua posição no grafo é salva em um vetor de atalhos e não insere aresta para ele nesse momento. Ao final das inserções de aresta em elementos maiores que 0, é utilizado o vetor de posições dos atalhos para inserir arestas entre eles.

Após a inserção de todas as arestas no grafo de acordo com o mapa de entrada é possível utilizar de um algoritmo visto em sala de aula chamado `Dijkstra`. Esse algoritmo calcula os menores caminhos dado um ponto inicial. Porém os pesos das arestas não podem ter valores negativos para garantir que ele funcione. O algoritmo utiliza da técnica de relaxamento que mantém o atributo d[v] que é um limite superior sobre o peso de um caminho mais curto desde uma origem s até v.

Algorithm 4: Relax (u, v, peso, d, pi)

if $d[v]$ *maior* $d[u] + peso$ **then**
    $d[v] \longleftarrow d[u] + peso$
    $pi[v] \longleftarrow u$
end

Algorithm 5: Inicializar (inicio, numeroVertices, d, pi)

$i \longleftarrow 1$
for 1 **to** $numeroVertices$ **do**
    $d[i] \longleftarrow INTMAX$
    $pi[i] \longleftarrow 0$
end

O algoritmo `Inicializar` recebe `d` que é o vetor que armazena o peso do caminho mais curto até aquele índice e recebe `pi` que é o vetor de predecessores. Diante disso todas as posições de `d` é inicializada com o maior valor possível, no caso da implementação foi usado INTMAX que é o maior valor inteiro. Quanto aos valores de `pi` foi utilizado 0, pois é um valor inválido de vértice na implementação.

Algorithm 6: Dijkstra (G, s, size, f)

Aloca, preenche e constroi um heap mínimo
Inicializar (s, size, d, pi)
for $tamanhoHeap$ **to** 1 **do**
    $primeiroElemento \longleftarrow ExtrairMinimoHeap()$
    for $aresta \in primeiroElemento$ **do**
        Relax(primeiroElemento, aresta, peso, d, pi)
    end
end

O algoritmo `Dijkstra` necessita de uma estrutura que retorne para ele a aresta com menor custo partindo de um vértice específico. Para isso foi implementado o Heap (fila de prioridades) mínima. Quando construída o elemento de menor valor sempre fica na primeira posição do vetor, permitindo sempre localizar a aresta com menor peso para utilizar no algoritmo.

Após a execução do `Dijkstra` cada índice do vetor 'd' possui o valor do custo do caminho mínimo do ponto inicial dado até aquele vértice do índice. Diante disso, ao final do algoritmo é impresso na tela o valor de `d[posição final]` se foi encontrado um caminho ou -1 se não existe um caminho.

3 Análise de complexidade

3.1 Tempo

- A função `insertEdge` insere a aresta porém de forma ordenada ao vértice de destino. No pior caso, para inserir na posição correta, a execução vai percorrer toda a lista de adjacência do vértice origem e essa lista pode conter no pior caso V-1 arestas considerando que todos os vértices são atalhos e irão ter aresta com todos menos com si mesmo. Diante disso a complexidade dessa função será **O(V)** (V = width * height).

- A função `insertNoRestriction` representada pelo pseudocódigo do Algoritmo 3, faz diversas verificações e validações com custo **O(1)**. Apoś essas validações ele chama `insertEdge` e com isso torna a complexidade dessa função igual a **O(V)** (V = width * height).

- A função `insertWithRestriction` representada pelo pseudocódigo do Algoritmo 2, faz diversas verificações e validações com custo **O(1)**. Porém antes de inserir a aresta, o custo é calculado de uma forma diferente. Os valores somados ao custo da aresta são calculados à partir das restrições de movimentação X e Y. Diante disso, no pior caso onde a restrição X é igual ao comprimento do mapa e a restrição Y é igual a altura do mapa a complexidade dessa função seria O(width + height) + O(V) vindo do custo de inserir aresta, ficando portanto **O(V)**(V = width * height).

- A função `insertAllEdges` representada pelo pseudocódigo do Algoritmo 1, executa para cada elemento do mapa que tem valor maior que 0 uma ação de inserir aresta. Diante do fato da função `insertWithRestriction` e da função `insertNoRestriction` terem a mesma complexidade de O(V) a complexidade de inserção das arestas de cada elemento com valor maior que 0 é de O(4*V) ou $O(V)$. No final é feito a inserção de arestas dos atalhos que no pior dos casos, todos os vértices são atalhos possuindo uma complexidade de $O(V*(V-1))$. A complexidade total dessa função será então $O(V^2)$ (V = width * height).

- A função `dijkstra` representada pelo pseudocódigo do Algoritmo 6, possui um tempo de execução dependente da manutenção da fila de prioridades utilizada pois as outras operações tem custo O(1). Diante disso, é possível observar que a operação `heapDecreaseKey` que tem custo O(log(v)) é chamada **E** vezes, sendo **E** o número de arestas e a operação para extrair o mínimo do heap e construí-lo novamente tem custo O(log(v)) é chamada **V** vezes, sendo **v** o número de vértices total. Portanto a complexidade final desse algoritmo fica sendo **O(log(V)*(V + E))**.

- Diante do programa executar apenas uma vez a função `insertAllEdges` e a função `dijkstra`, para saber qual função domina é necessário colocar o **E** da complexidade do `dijkstra` em função do pior caso de arestas descrita na função `insertAllEdges`. Ou seja, **E** = V^2, substituindo fica log(V)*(V + [V^2 - V]) = log(V)*V^2. Portando a complexidade final do programa é definida pela complexidade do `dijkstra` sendo log(V)*V^2.

3.2 Espaço

A complexidade de espaço do programa pode ser determinada à partir da alocação inicial do mapa em uma matriz e a alocação da lista de adjacência.

A alocação da matriz possui complexidade de espaço **O(V)** sendo V o número total de vértices devido do fato que ela aloca memória para cada coordenada da matriz. Já a lista de adjacência possui complexidade de espaço **O(V + A)** pois ela aloca espaço para todos os vértices e todas as arestas ao final da inserção das arestas.

Como reflexo disso, a complexidade de espaço total do programa é definida por **O(V + A)**.

4 Análise Experimental

4.1 Metodologia

Para realizar a análise experimental cada instância foi testada 3 vezes em uma máquina Intel Core i5 2,67GHz de 4GB de memória RAM. Para medição do tempo de execução foi utilizado o comando `time` no linux e o cálculo final do tempo foi obtido pela média dos 3 testes.

4.2 Análise de performance

Teste 1
Variáveis estáticas: mapa sem obstáculos ou atalhos
Variáveis a serem analisadas: Dx e Dy

Número	Obstáculos	Atalhos	Comprimento	Altura	Dx	Dy	Vértice inicio	Vértice fim	Tempo
1	0	0	300	300	0	0	0	45000	31,058s
2	0	0	300	300	1	1	0	45000	23,724s
3	0	0	300	300	5	5	0	45000	1,337s
4	0	0	300	300	10	10	0	45000	0,559s
5	0	0	300	300	15	15	0	45000	0,504s

A tabela acima testa o tempo de execução considerando que apenas os valores de restrição de movimento se altere e com o mapa ausente de obstáculos ou atalhos. Diante disso é possível observar que quanto maior for a restrição de movimento, menor será o tempo de execução. Esse comportamento acontece pois quanto maior for a restrição menos caminhos possíveis podem ser formados e o número de arestas de cada célula do mapa é reduzida. Como a quantidade de arestas influencia diretamente na complexidade do programa esse tempo menor é justificado.

Teste 2
Variáveis estáticas: mapa sem obstáculos ou atalhos.
Variáveis a serem analisadas: Obstáculos

Número	Obstáculos	Atalhos	Comprimento	Altura	Dx	Dy	Vértice inicio	Vértice fim	Tempo
1	5,00%	0	300	300	1	1	0	45000	3,213s
2	10,00%	0	300	300	1	1	0	45000	1,049s
3	20,00%	0	300	300	1	1	0	45000	0,474s
4	33,00%	0	300	300	1	1	0	45000	0,343s
5	50,00%	0	300	300	1	1	0	45000	0,073s

A tabela acima testa o tempo de execução considerando que apenas os valores da porcentagem de obstáculos se altere. Diante disso é possível observar que quanto maior for a quantidade de obstáculos menor será o tempo de execução. Esse comportamento pode ser observado pois quanto mais obstáculos no mapa os caminhos possíveis diminuem cada vez mais e, com isso, menos arestas vão ser formadas. Como a complexidade do programa depende do número de arestas, o número de obstáculos interfere diretamente nesse valor.

Teste 3
Variáveis estáticas: Restrições, obstáculos, atalhos.
Variáveis a serem analisadas: Comprimento, altura

Número	Obstáculos	Atalhos	Comprimento	Altura	Dx	Dy	Vértice inicio	Vértice fim	Tempo
1	0	0	300	300	1	1	0	90000	0,067s
2	0	0	600	600	1	1	0	360000	0,136s
3	0	0	700	700	1	1	0	490000	0,170s
4	0	0	900	900	1	1	0	810000	0,202s
5	0	0	1500	1500	1	1	0	2250000	0,355s

A tabela acima testa o tempo de execução considerando apenas a mudança dos valores totais de vértice, ou seja, mudando o valor de altura e comprimento da entrada. Para que isso seja possível, a posição inicial e final foram colocadas sendo o primeiro e o ultimo vértice com restrição de movimento máxima. Desta forma, mesmo mudando os valores de comprimento e altura do mapa de entrada, apenas duas arestas são formadas. Observado a tabela foi possível observar que quanto maior o tamanho do mapa, maior vai ser o tempo de execução, pois a complexidade é influenciada também pelo número total de vértices, porém não na mesma proporção que o número de arestas influenciam.

5 Conclusão

Neste trabalho foi modelado um problema utilizando grafo. Tendo um grafo do mapa onde o robô se encontra foi possível identificar qual seria o melhor caminho para ele percorrer diante de qualquer posição inicial/final, qualquer custo positivo de acesso às coordenadas, qualquer obstáculo e qualquer atalho. Para indicar o custo desse menor caminho que foi pedido na especificação, foi utilizado o algoritmo de `Dijkstra`.