TP2: Eu, robô

Giovani Amaral Barcelos

1 de junho de 2016

1 Introdução

O objetivo desse trabalho prático é, a partir da leitura de um mapa do terreno de uma arena de competição entre robôs, descobrir a distância mais curta entre dois pontos.

Nessa arena, temos caminhos com diferentes pesos/dificuldades, além de atalhos, que são pontos onde uma vez que o robô se encontra nele, ele pode sair em qualquer outro atalho. Ademais, temos obstáculos, que são pontos por onde o robô não pode passar, ou seja, ao se encontrar com um obstáculo o robô deve procurar por outro caminho.

Para finalizar, podemos ter uma restrição que nos limita ao movimentar pelos eixos X e Y, ou seja, numa restrição X=2 e Y=1 ou o contrário, nos movimentamos igual um cavalo no jogo de xadrez.

2 Solução do problema

Para solucionar este problema, foi usado um grafo não direcionado, onde cada posição no mapa foi representado por um vértice. Desta forma, os vértices foram conectados com outros respeitando a restrição, de forma que no exemplo da restrição X=2 e Y=1 **o vértice (0,0) foi conectado diretamente com o (2,1)**, onde o peso da aresta era o peso total do caminho.

Entretanto, de (0,0) para (2,1) temos dois possíveis caminhos: passando primeiro por X e passando primeiro por Y. Foram avaliados os dois caminhos e nossa aresta tem sempre o min(peso_x, peso_y);

Função de caminhamento e validação no eixo X:

Algorithm 1: Pseudo-código do caminhamento e validação no eixo X

```
Data: intx, inty, intrestricao_x
Result: int peso
for i ← x to (x + restricao_x) do
    if peso[V(i,y)] == 0 then
        return -1
    end
    if i == x OR i == restricao_x then
        peso += peso[V(i,y)] // Se estamos nas extremidade do caminho, consideramos
            o peso do vértice duas vezes, pois ele entra no vértice e depois sai
    end
    else
        peso += 2*peso[V(i,y)]
    end
end
return peso
```

2.1 Conexão de arestas

Como temos uma restrição em X e em Y, cada vértice pode movimentar para no máximo 4 outros vértices, excluindo quando o vértice é um atalho. No caso de ele não ser um atalho, os movimentos poderão variar de acordo com os sinais de X e Y, ou seja, teremos os movimentos (+X, +Y), (+X, -Y),(-X, +Y),(-X, -Y). Ainda, para cada destino, temos dois caminhos possíveis, dependendo da nossa escolha, como citado acima.

No trabalho cada um desses movimento foi chamado de quadrante, exatamente como nos gŕaficos cartesianos. Para cada um dos quadrantes, checamos se o vértice destino está dentro dos limites do grafo e não é um obstáculo.

Caso seja um vértice válido, caminhamos até ele pelos dois caminhos e adicionamos a aresta entre origem e destino com a menor distância válida.

Esse fluxo pode ser explicado pelo pseudo código referente à um quadrante, onde x e y são as coordenadas do vértice origem(vértice onde estamos adicionando as arestas no momento), x2 e y2 são as coordenadas do vértice destino, M é o numero de colunas do nosso mapa e N o numero de linhas.

Algorithm 2: Pseudo-código da adição de arestas para um quadrante

```
1  valido1 = 1
2  valido2 = 1
3  if x2 < M AND y2 ≥ 0 AND peso[V(x2, y2)] ≠ 0 then
4  |  peso_1x ← valida_caminho_x(x, y, x2)
5  |  peso_1y ← valida_caminho_y(x2, y, y2)
6  |  peso_2y ← valida_caminho_y(x, y, y2)
7  |  peso_2x ← valida_caminho_y(x, y2, x2)
8  end
9  if peso_1x == −1 OR peso_1y == −1 then
10 |  valido1 ⇐ 0
11 end
12 if peso_2x == −1 OR peso_2y == −1 then
13 |  valido2 = 0
14 end
15 if valido1 && valido2 then
16 |  insere_aresta(V(x,y), V(x2,y2), min(pesototal_1, pesototal_2))
17 end
18 if valido1 && !valido2 then
19 |  insere_aresta(V(x,y), V(x2,y2), pesototal_1)
20 end
21 if !valido1 && valido2 then
22 |  insere_aresta(V(x,y), V(x2,y2), pesototal_2)
23 end
```

Essa função deverá ser chamada para cada quadrante, do 1 ao 4, variando o sinal de x2 e y2, adicionando assim no máximo 4 arestas;

3 Implementação

Abaixo vemos o fluxo principal do programa

Algorithm 3: Pseudo-código do fluxo principal

```
1 mapa ← le_entrada()
2 transforma_mapa_em_grafo()
3 conecta_atalhos()
4 Djkistra(vertice_inicial)
5 print D[vertice_final]
```

3.1 Leitura da entrada

Essa função é responsável por ler todas as informações de entrada e converte-la para uma matriz de inteiros. Ele faz 2 leituras para ler os índices M e N e depois M*N leituras, sendo cada uma delas uma célula do mapa;

3.2 Transformação do mapa em grafo

Essa função é responsável tão somente por pegar cada célula da matriz de pesos lida e transformá-la num vértice, além de adiciona as arestas aos vértices adjacentes respeitando a restrição. Isso pode ser explicado pelos pesudocódigos 1 e 2, abordados na seção de solução do problema..

3.3 Conecta atalhos

Função responsável para criar uma aresta entre cada atalhos e todos os outros atalhos.

Algorithm 4: Pseudo-código da conexao de atalhos

```
1 for indice_atalho in lista_atalho do
2 |   for indice_atalho2 in lista_atalho do
3 |   |   if indice_atalho ≠ indice_atalho2 then
4 |   |   |   adiciona_aresta(indice_atalho, indice_atalho2)
5 |   |   end
6 |   end
7 end
```

3.4 Djkistra

Algoritmo de caminho mais curto Djkistra, que é abordado no [1]. Os tres pseudo codigos abaixo são baseados na obra citada acima e compõe o algoritmo de Djkistra.

Algorithm 5: Pseudo-código inicializa fonte de unica

```
Data: GrafoG, ints
for vertice in G do
    D[vertice] ← ∞
end
D[s] ← 0
```

Algorithm 6: Pseudo-código relax

```
Data: intu, intv
Result: int peso
if D[v] > D[u] + peso(u, v) then
    D[v] ← D[u] + peso(u, v)
end
```

Algorithm 7: Pseudo-código Djkistra

```
Data: Grafog, intv
inicializa_fonte_unica(g, v)
// S é o conjunto de todos os vértices que já foram visitados
S ← 0
// E Q representa nossa fila de prioridades, que é V - S
Q ← V
while Q ≠ 0 do
    // Pegamos o proximo elemento da lista
    u ← extract_min(Q)
    S ← S + u
    for vertice in adj[v] do
        relax(u,v)
    end
end
```

4 Análise teórica do custo assintótico de tempo

Nas analises seguintes, consideraremos M o numero de colunas N o numero de linhas, T o tamanho do grafo, A o número de areatas e V o numero de vértices. dx é a restrição em X e dy a restrição em Y. **extract_min** Extrai o primeiro elemento e remonta o heap $O(logV)$ **heap_decrease_key** Diminui a chave de um elemento do heap e remonta-o. $O(logV)$ **Djkistra** - A complexidade desse algoritmo já é bastante conhecida e é explicada e elaborada em [1]. Chamamos extract_min V vezes, e decrease_key A vezes, logo **O(A * log(V))**.

conecta_atalhos - Conecta um atalhos a todos os outros, então sua complexidade é $Atalhos*(Atalhos-1)$. O numero de atalhos no pior caso é V, logo a complexidade é $O(V^2)$.

insere_aresta Percorre a lista de adjacentes de um vértice até o final. No pior caso, $O(V-1)$ ou $O(V)$

validar_caminho_x Essa função percorre restrição_x células, logo **O(d_x) validar_caminho_y** Essa função percorre restrição_y células, logo **O(d_y)**

mapa_para_grafo Para cada vértice, chamamos validar_caminho_x 8 vezes e validar_caminho_y 8 vezes, logo para essa parte temos $8dx+8dy$. Inserimos no máximo 4 arestas, então temos $4*O(V)$. Logo a complexidade total da função é V * (4V + 8dx + 8dy), ou $O(V*(V+dx+dy))$

peso Percorre os adjacentes de um vértice V buscando uma aresta específica e retorna seu peso. Pior caso $O(V)$

adj Percorre os adjacentes de um vértice V buscando por um determinado adjacente e retorna 1 ou 0. Pior caso $O(V)$

Portanto, a complexidade total do programa é $A*log(V)+V^2+V^2+8dx+8dy$ ou $(A*log(V))+V^22+dx+dy$

4.1 Análise teórica do custo assintótico de espaço

Para essa análise vamos considerar o grafo, a fila de prioridades e o mapa de pesos.

A estrutura vértice possui 6 inteiros($6 * 4 = 24bytes$) e uma lista encadeada para outros vértices que só possuirão os inteiros. Logo cada vértice ocupa $24 + ((V - 1) * 24)$ bytes, ou $24 + 24V - 24$ ou 24V bytes. Nosso grafo então ocupa $24V * V = 24V^2$ bytes. $O(V^2)$

Nossa fila de prioridade é somente uma estrutura com um int e um array de V elementos, onde cada elemento possui dois inteiros. Logo $4 + 8V$ bytes. $O(V)$

Nosso mapa contém N linhas e M colunas, onde cada célula é um inteiro. Como $N * M = V$, temos V * 4 bytes; $O(V)$

A complexidade total de espaço é $O(V^2)$

5 Análise de experimentos

Para realizar a análise experimental, foi implementado um gerador de testes em ruby que gera grafos baseados em parametros como restrição e obstáculos. Para medir o tempo de execução do código, foi utilizada o comando time do Ubuntu. Cada teste foi rodado 5 vezes e foi feita uma média com os tempos. Para realizar os testes foi utilizada uma máquina com Linux Ubuntu 15, processador I3 e 6GB de RAM.

5.1 Restrição

Fica claro perceber a forma como o número da restrições impacta no tempo de execução do programa.

Isso se deve pois, com restrição maiores, temos bem menos vértices para serem conectado entre si, ou seja, temos mais caminhos entre quaisquer dois vértices que não sejam obstáculos. Na situação da restrição 0 0, ou seja, pode movimentar para todos os lados uma vez, nós conseguimos a partir de um vértice X alcançar todos os outros vértices, a menos que sejam bloqueados completamente por obstáculos.

Isso faz com que nosso Djkistra precise percorrer menos caminhos, pois existem menos vértices conectados entre si.

Ou seja, restrição maior diminui a conexão entre os vértices e portanto o numero de arestas, e como a complexidade do nosso Djkistra depende de A, a execução fica mais rápida.

O dx sempre é igual o dy para facilitar a análise e vizualização.

Grafo 200 25
5000 vértices
Sair do 0 0 e chegar no 195 19
Eixo X : tamanho da restrição
Eixo Y : tempo de execução em ms

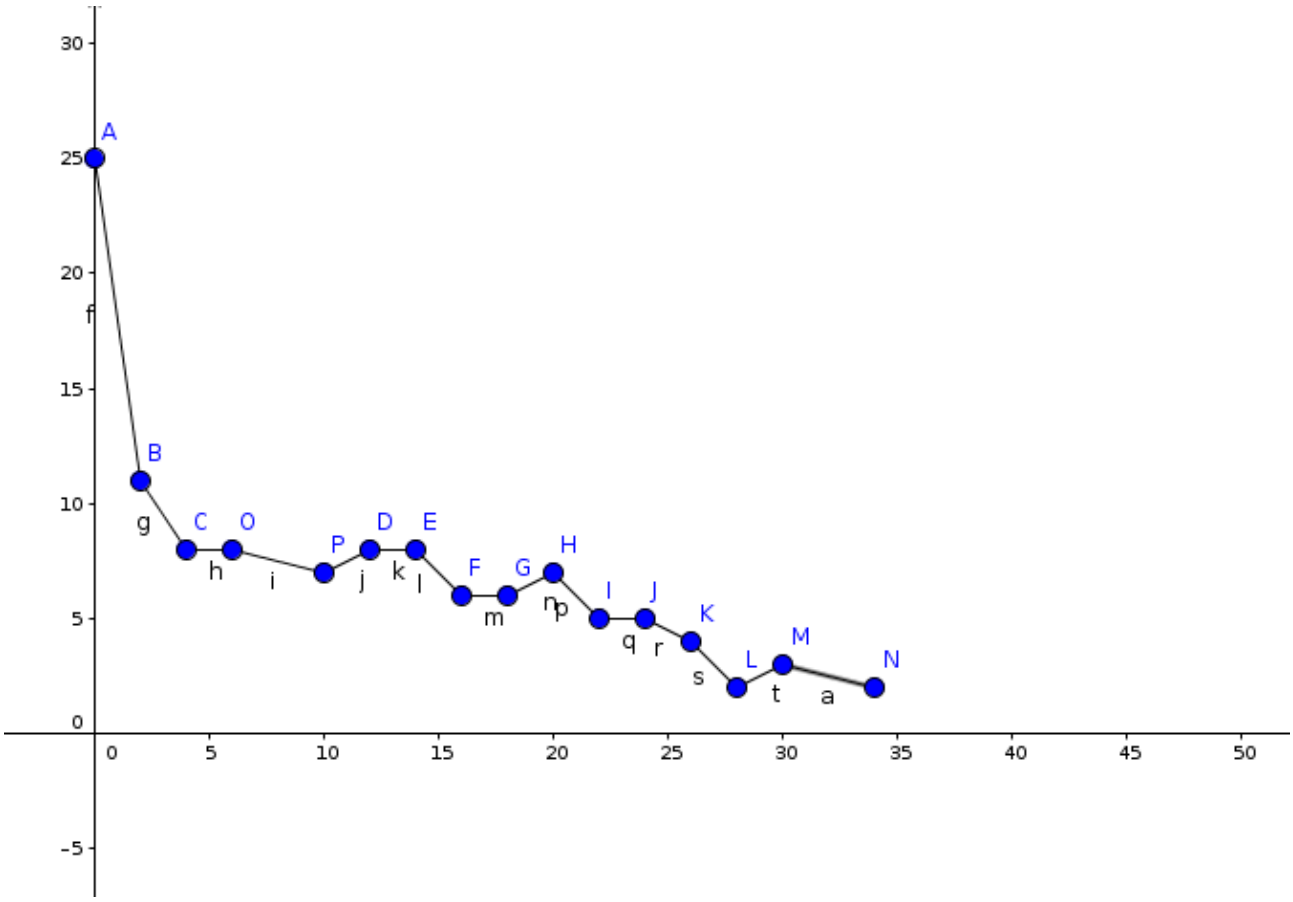

Figura 1: Grafico que mostra o impacto de restrições no tempo de execução.

5.2 Obstáculos

Grafo 200 25
5000 vértices
Sair do 0 0 e chegar no 195 19
Eixo X : número de obstáculos divido por 10, para facilitar vizualização
Eixo Y : tempo de execução em ms

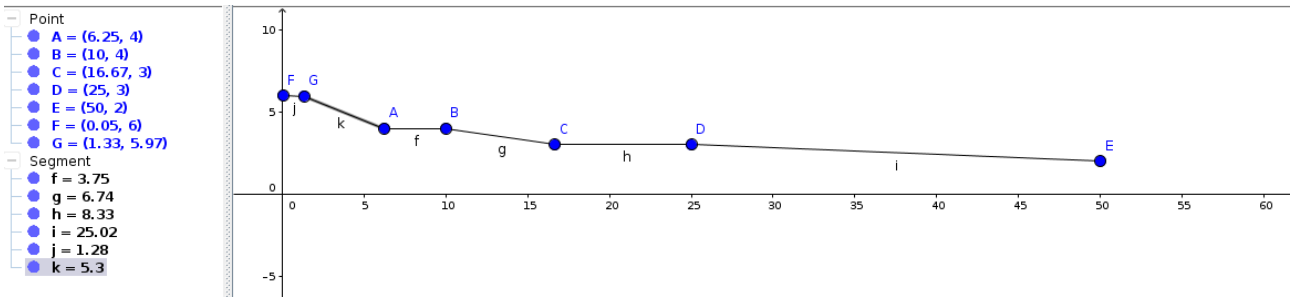

Figura 2: Grafico que mostra o impacto do numero de obstaculos no tempo de execução.

Devemos ressaltar que na segunda figura o número de obstáculos está dividido por 10. Isso foi feito para facilitar a vizualização de dados, uma vez que aumentos muito grandes no número de obstáculo mudou pouco o tempo de execução, portanto é quase desprezível para a análise. No ponto E, onde tinhamos 5000 obstáculos, a execução foi pouco mais rápida do que a situação onde tinhamos 1000.

6 Bibliografia

Referências

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