Universidade Federal de São João del Rei
Departamento de Ciência da Computação
Curso de Ciência da Computação

Trabalho Prático 2: uma resolução por Programação Dinâmica e uma por Busca em Largura para um problema de caminho mínimo

Rodrigo José Zonzin Esteves

´

1 Introdução

Resumo: *Dado uma matriz $A_{n \times m}$, representando um caminho, determinar qual é a sequência de posições $(a_{ij}, a_{i'j'}, ..., a_{i''j''})$ menos custosa partindo de a_{11} até a_{nm}.*

Para cada sequência, só é permitido andar para a direita ou para baixo. Exemplo: A sequência $(a_{ij}, a_{i'j'})$ só é válida se $i' = i + 1$ ou se $j' = j + 1$.

A solução para esse trabalho está vastamente documentada. Em comparação com a posposta apresentada, a única diferença se refere aos personagens. Usualmente trata-se de um cavaleiro atravessando uma masmorra para salvar uma princesa. As restrições de locomoção são as mesmas (andar somente para a direita ou para baixo) e a vida do cavaleiro deve sempre ser maior do que 1. LeetCode (2023)

Para a solução do trabalho, optamos por seguir uma abordagem inspirada por Programação Dinâmica e outra pelo algoritmo de Busca em Largura. Inicialmente também tentamos uma solução por recursão, mas a abandonamos para dar preferência às que já estavam documentadas.

2 Abordagem por Programação Dinâmica

Para calcular o melhor caminho que Harry terá de tomar, podemos inverter a lógica de caminhamento do personagem. Em vez de analisarmos o melhor caminho partindo do ponto $(1, 1)$ ou $(0, 0)$, podemos analisar qual o caminho minimiza a vida necessária para o trajeto partindo de (R, C) ou $(R - 1, C - 1)$.

Isso nos permite armazenar informações em uma tabela e usá-la para construir soluções mais gerais. Nossa tabela será a matriz $V_{R \times C}$, definida a seguir.

2.1 Detalhes da abordagem

Seja $M_{R \times C}$ a matriz pela qual Harry deve caminhar. Seja $V_{R \times C}$ uma matriz que armazenará os valores intermediários da abordagem.

$$M = \begin{bmatrix} m_{11} & ... & m_{1C} \\ ... & ... & ... \\ m_{R1} & ... & m_{RC} \end{bmatrix}$$

Calculamos a energia mínima para que Harry esteja vivo em m_{RC}. Se houver um monstro em (R, C) que retire m_{RC} unidades de vida, Harry deve possuir pelo menos $|m_{RC}| + 1$ unidades de vida para sobreviver ao monstro. Se houver uma poção (ou nada), precisamos garantir que Harry tenha pelo menos 1 unidade de vida.

$$V_{RC} = \begin{cases} 1 & m_{RC} \geq 0 \\ -m_{RC} + 1 & m_{RC} < 0 \end{cases}$$

A partir dessa informação, calculamos os elementos da linha inferior e os elementos da última coluna da tabela. Para cada elemento V_{Rj} (última linha), tomamos a vida necessária para que Harry caminhe da esquerda para a direita, sobrevivendo ao monstro vizinho V_{Rj-11} ou, pelo menos, mantendo-se com 1 unidade de vida.

Se existir um monstro em uma célula mais à esquerda de onde Harry está, ele precisará de compensar o quanto de vida ele perde com o tanto de vida que ele tem até aquele ponto. Se não houver monstro, ele precisa garantir pelo menos 1 vida.

$$V_{Rj} = \begin{cases} 1 & m_{Rj} \geq 0 \\ V_{Rj-1} - m_{Rj} & m - Rj < 0 \end{cases} \; para \; j = 1, 2, ..., C-1$$

De forma geral, basta calcularmos o máximo entre a subtração da vida atual com a vida consumida pelo monstro (ou adquirida pela poção) e a vida mínima, 1.

$$V_{Rj} = max(V_{Rj-1} - m_{Rj}, 1)$$

Da mesma maneira, construímos a matriz para os elementos da última coluna.

$$V_{iC} = max(V_{i-1C} - m_{iC}, 1), \; para \; i = R, R-1, ..., 1$$

Até esse ponto, temos a seguinte tabela.

$$V = \begin{bmatrix} & & & max(*, 1) \\ & & & ... \\ & & & max(*, 1) \\ max(., 1) & ... & max(., 1) & 1 - m_{RC} \end{bmatrix}^{1}$$

A partir desses dados, podemos completar a tabela tratando cada "quina"de dados faltante.

Para isso, tomamos a menor energia possível que Harry tomaria vindo dos dois sentidos possíveis: direita-esquerda e baixo-cima.

$$V = \begin{bmatrix} & & & min(., *) & max(*, 1) \\ & & & min(., *) & ... \\ min(., *) & ... & min(., *) & min(., *) & max(*, 1) \\ max(., 1) & ... & max(., 1) & max(., 1) & 1 - m_{RC} \end{bmatrix}$$

Aplicamos esse procedimento até que sobre somente uma quina. Tal quina coincidirá com o elemento da posição V_{11}.

$$V = \begin{bmatrix} \mathbf{min}(V_{21}, V_{12}) & ... & min(., *) & min(., *) & max(*, 1) \\ ... & ... & ... & ... & ... \\ min(., *) & ... & min(., *) & min(., *) & max(*, 1) \\ max(., 1) & ... & max(., 1) & max(., 1) & 1 - m_{RC} \end{bmatrix}$$

V_{11} é nossa resposta final. Através dessa análise, caminhamos por todos os possíveis pontos ótimos da nossa solução.

3 Abordagem pelo Algoritmo de Busca em Largura

A *matriz* por onde Harry caminha pode ser compreendido como um grafo, onde cada nodo é representado pela posição ij do caminho de Harry.

Todos os nodos estão conectados por arestas (é um grafo denso), mas Harry só pode se movimentar pelas arestas adjacentes de um determinado nodo (ou seja, para a direita ou para baixo).

O algoritmo de Busca em Largura (BFS) pode ser utilizado para obter o caminho mais curto de um nodo u até outro nodo v [2] - $u, v \in M$.

[1] . e * para simplificar a notação

[2] Seja M a matriz de adjacência que representa o grafo por onde Harry caminha

O procedimento de visitação do algoritmo constrói uma árvore de busca em largura que armazena o caminho mais curto entre o vértice de origem e outro vértice qualquer do grafo.

Dessa maneira, podemos usar o BFS para percorrer o caminho de Harry de forma sistemática e encontrar a quantidade mínima de unidades de vida necessárias para mantê-lo vivo do início ao fim.

3.1 Detalhes da abordagem

O algoritmo começa na posição M_{00} e mantém uma heap para explorar posições adjacentes a esse vértice. Cada elemento da heap contém as coordenadas ij da posição, a soma atual de unidades de vida e a soma mínima de unidades de vida encontrada ao longo do caminho até aquela posição.

Enquanto a heap não estiver vazia, o algoritmo retira o elemento com a maior soma de vida de vida da heap e atualiza uma matriz $somaMaxVida$ com o valor máximo de vida encontrado até aquela posição.

Logo após isso, ele verifica se chegou à posição $(R-1, C-1)$ e retorna a quantidade mínima de pontos de vida necessários $(1 - somaMinima)$.

Caso ele não tenha chegado, ele continua caminhando as posições adjacentes e colocando-as na heap com as somas atualizadas de pontos de vida. Esse procedimento continua até que todas as posições sejam exploradas ou até que a posição final seja alcançada.

O pseudo código a seguir apresenta representa a abordagem escolhida.

```
function bfs(Grafo g, int R, int C):
  Matriz somaMaxVida[R][C];

  for m_ij ∈ M:
    m_ij = −∞

  Heap H = heap_vazia(100)
  enfilera(Q, elemento_zero)

  while not vazia(H):
    i = desinfilera(H)

    atualiza(somaMaxVida, i_soma)

    if i_x == R-1 and i_y == R-1:
      return max(i_somaMin, 1)

    for d_i = direita, b_i = baixo:
      if d_i == valido and b_i == valido
        enfilera([d_i, b_i])

  return 1
```

4 Análise de complexidade e testes realizados

4.1 Análise teórica

Analisaremos a complexidade de tempo. Dessa maneira, tomamos o número de comparações como a variável de interesse para o desenvolvimento da análise da complexidade do código para o pior caso.

4.1.1 Programação dinâmica

O código da função $pd()$ começa alocando uma matriz de dimensões $R \times C$. A operação de alocar a tabela de programação dinâmica aloca R linhas e itera sobre cada uma delas alocando um vetor de C colunas. Sua complexidade final é de $O(R \cdot C)$

Em seguida, preenchemos os valores da última linha e da última coluna da tabela. Para isso, realizações uma operação aritmética (tempo constante) para cada posição $(i, R-1)$ e para cada posição $(C-1, j)$. Como iteramos por $R-1$ valores para preencher a última linha e mais $C-1$ valores para a última coluna, temos $O(R+C)$ para esse trecho de código.

Dois loops aninhados são usados para preencher as *células-quina* da tabela. Para cada poisção (i, j), realizamos duas operações aritméticas (tempo constante) para calcular os valores de ir para baixo ou para a direita. Em seguida, realizamos uma operação lógica (tempo constante) pra atribuir o mínimo dos 2 valores e colocá-lo na posição corrente. Temos então a seguinte complexidade $O(R-1 \cdot C-1) = O(R \cdot C)$.

Por fim, após todos os valores estarem devidamente preenchidos, retornamos o primeiro valor da tabela de programação dinâmica. Essa operação tem tempo constante $O(1)$.

$$O_{PG} = O(R \cdot C) + O(R+C) + O(R \cdot C) + O(1)$$
$$O_{PG} = O(R \cdot C)$$

4.1.2 BFS

O código inicia alocando uma matriz $somaMaxVida$ de dimensão $R \times C$. Como analisado na função anterior, essa função tem complexidade $O(R \times C)$.

Em seguida, alocamos uma estrutura de dados $Heap$ de tamanho inicial $R + C$. Essa operação tem complexidade constante, uma vez que ela não itera sobre nenhum dos elementos, mas apenas reserva blocos de dados na memória principal. Tempos, portanto, $O(1)$.

O loop *while* é executado até que a $Heap$ esteja vazia. No pior caso, esse laço será executado $R \cdot C$ vezes (uma posição da matriz só pode ser visitada uma única vez). Dentro desse loop são executadas operações lógico-aritméticas de tempo constante (comparação e soma de inteiros). Os casos de retorno (estar na posição $(R-1, C-1)$, por exemplo) também têm tempo constante. Há ainda um for que executa no máximo 2 vezes para verificar se as células vizinhas são válidas. Temos portanto $O_{while} = O(O(R \cdot C) + O(2)) = O(R \cdot C)$.

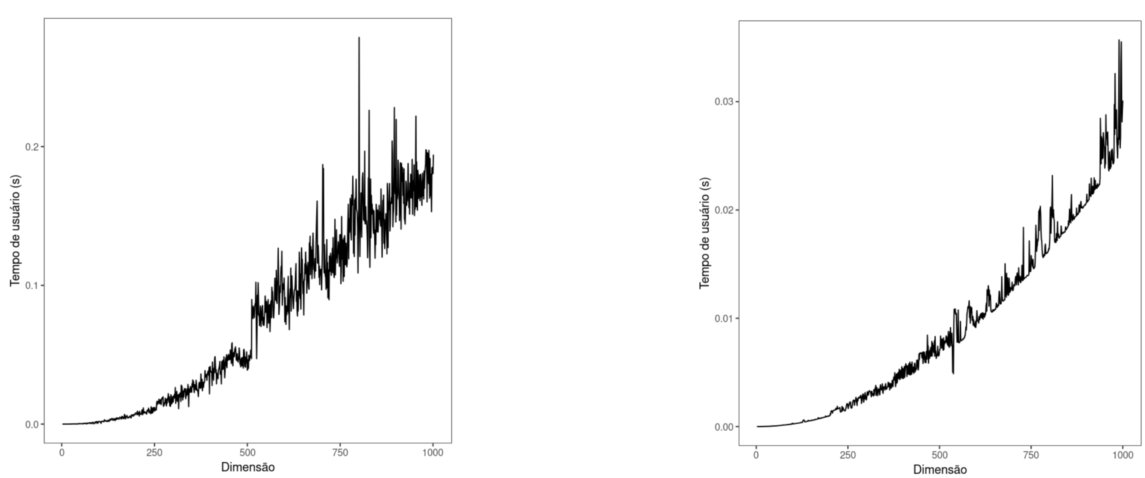
pela propriedade da soma, temos a seguinte complexidade:

$$O_{BFS} = O(R \cdot C) + O(1) + O(R \cdot C)$$
$$O_{BFS} = O(R \cdot C)$$

4.2 Análise empírica

Para compararmos o desempenho dos nossos algoritmos, tomamos um caso de teste de 1000×1000 linhas e colunas e mensuramos os tempos de usuário e sistema necessários para execuções que variaram de $2 \times 2, 3 \times 3, ..., 1000 \times 1000$.

A figura abaixo demonstra o resultado obtido.



Programação Dinâmica

BFS

Figura 1: Tempo vs Dimensão

Ambos os códigos têm desempenho quadrático. Uma vez que a sequência adotada tem número de linhas e colunas igual, tempos complexidade $O(R\ cdotR) = O(R^2) = O(C^2)$.

Estimamos a função de complexidade para ambos os casos.

$$\hat{f}_{PG}(C) = 0.0731836 + 1.9129838 \cdot C + 0.2751489 \cdot C^2$$

$$\hat{f}_{BFS}(C) = 0.009223 + 0.2464 \cdot C + 0.06044 \cdot C^2$$

com $R^2_{PG} = 0.9459$ e $R^2_{PBFS} = 0.9829$ e significância para todos os parâmetros do modelo.

A diferença de eficiência pode ser atribuída ao menor número de operações aritméticas que o BFS realiza. Na abordagem por programação dinâmica, os primeiros dois loops "for"iteram sobre os índices i e j, indo de $m-2$ e $n-2$ até 0. Dentro desses loops, cada iteração faz duas comparações, uma para a variável "baixo"e outra para a variável "direita". [3]No total, temos $(R-1)\cdot(C-1) = (R-1)(R-1) = 2(R-1)^2$ comparações.

Para o BFS, no entanto, o *while* é executado $R = C$ vezes, com duas comparações cada para verificar as condições de término ou *continue*Temos $2R$ comparações até essa parte. Em seguida, há ainda um *for* aninhado que realiza duas comparações a cada iteração, ou seja, $2R$ comparações. Somando essas comparações, temos $4R$ comparações para o BFS.

5 Leaks de memória

Para cada execução do programa pelo método de programação dinâmica, alocamos T matrizes de $R \times C$ elementos do tipo int. Quando executamos o programa pela solução do BFS fazemos ainda o 1 alloc da estrutura *Heap* por chamada, contendo pelo menos 500 elementos do tipo *Item* (4 inteiros).

A seguir, apresentamos o resultado gerado pelo utilitário Valgrind.

Realizamos 4 testes com dimensões

```
zonzin@rodrigo:~/Documentos/Faculdade/AEDSIII/tp2/tp2_final$ valgrind ./tp2 1 entrada.txt
==164498== Memcheck, a memory error detector
==164498== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==164498== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==164498== Command: ./tp2 1 entrada.txt
==164498==
==164498==
==164498== HEAP SUMMARY:
==164498==     in use at exit: 0 bytes in 0 blocks
==164498==   total heap usage: 26 allocs, 26 frees, 15,892 bytes allocated
==164498==
==164498== All heap blocks were freed -- no leaks are possible
==164498==
==164498== For lists of detected and suppressed errors, rerun with: -s
==164498== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

(a) Programação dinâmica

```
zonzin@rodrigo:~/Documentos/Faculdade/AEDSIII/tp2/tp2_final$ valgrind ./tp2 2 entrada.txt
==164619== Memcheck, a memory error detector
==164619== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==164619== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==164619== Command: ./tp2 2 entrada.txt
==164619==
==164619==
==164619== HEAP SUMMARY:
==164619==     in use at exit: 0 bytes in 0 blocks
==164619==   total heap usage: 18 allocs, 18 frees, 9,364 bytes allocated
==164619==
==164619== All heap blocks were freed -- no leaks are possible
==164619==
==164619== For lists of detected and suppressed errors, rerun with: -s
==164619== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

(b) BFS

Figura 2: Sumário da Heap

[3] $R = C$

6 Conclusão

Durante o trabalho, pudemos observar a importância dos Tipos Abstratos de Dados (TAD) como base para soluções mais complexas. A utilização do TAD "Matriz"permitiu o encapsulamento de código e operações, sendo aplicado em ambas as abordagens. Em particular, o TAD *Heap*, apresentado em disciplinas anteriores, demonstrou o quão efetivo ele pode ser em sua gama de possíveis aplicações no mundo real.

Além disso, o uso do algoritmo BFS destacou a eficiência das estruturas que podem ser modeladas como grafos ($Matriz = Grafo \Leftrightarrow Grafo = Matriz$). A facilidade da implementação adaptada do algoritmo de Busca em Largura também foi notável.

Vale ressaltar a diferença de desempenho entre os dois algoritmos. Embora as variáveis de confusão não tenham sido tratadas adequadamente (o que compromete a robustez da afirmação), foi interessante constatar que o comportamento quadrático observado nos testes dos dois algoritmos estava em conformidade com suas características teóricas, como aquelas apresentadas em Cormen et al. (2009). Um algoritmo é mais rápido que o outro, mas ambos apresentam o mesmo comportamento assintótico.

Ademais, as limitações deste trabalho podem ser contornadas em trabalhos futuros, tomando métricas mais rigorosas para a mensuração de desempenho de algoritmos. Espera-se que, mesmo com essas melhorias, os resultados sejam consistentes com os encontrados na literatura teórica.

Referências

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Algorithms*. The MIT Press, Cambridge, MA, 3rd edition.

LeetCode (2023). *Dungeon Game*. https://leetcode.com/problems/dungeon-game/ [Accessado: maio de 2023].