ADLER GUILHERME FURTADO FARIA
2023028501

TP 3 ALG I

UFMG Belo Horizonte, 2024

INTRODUÇÃO

Neste trabalho prático, o desafio proposto envolve um tabuleiro com uma rainha e diversos peões, onde o objetivo é capturar todos os peões no menor número possível de movimentos. O tabuleiro contém espaços livres, bloqueios e peões dispostos em posições específicas. O movimento da rainha segue as regras do xadrez tradicional: pode se mover em qualquer direção reta (horizontal, vertical ou diagonal), até encontrar um obstáculo ou o fim do tabuleiro.

Foram implementadas duas soluções com abordagens distintas, uma exata, que garante a menor quantidade possível de movimentos ao custo de maior tempo de execução; e uma aproximada, que usa um algoritmo guloso para obter uma resposta eficiente e rápida, mesmo que não seja sempre a ótima.

Ambas as soluções partem da modelagem do problema como um grafo, em que os vértices representam a rainha e os peões, e as arestas representam o custo (distância mínima) entre as posições, considerando as regras de movimentação e obstáculos no tabuleiro.

MODELAGEM

O tabuleiro é representado por uma matriz de caracteres de tamanho N x M. Cada célula da matriz representa um tipo de elemento:

- . (ponto) : célula vazia, por onde a rainha pode se mover livremente;
- - (hífen) : célula bloqueada, intransponível para a rainha;
- P : célula que contém um peão a ser capturado;
- R : posição inicial da rainha.

Para representar posições no tabuleiro, utilizamos a estrutura *Ponto* (que possui as variáveis x e y correspondendo às coordenadas do mesmo). Já a estrutura Tabuleiro armazena a matriz do tabuleiro, as dimensões, a posição da rainha e um vetor com todos os peões presentes.

A movimentação da rainha é simulada através de uma busca em largura (BFS) adaptada para seus movimentos característicos: ela pode andar em qualquer uma das 8 direções (horizontal, vertical e diagonal) até atingir um obstáculo ou o final do tabuleiro.

Durante a execução da BFS, é construída uma matriz de distâncias que registra o menor número de movimentos necessários para ir de um ponto a outro, respeitando os limites do tabuleiro e os obstáculos. Essa matriz é utilizada por ambas as soluções (exata e gulosa), permitindo rápida consulta do custo entre duas posições.

Esse pré-processamento permite transformar o problema original em um grafo ponderado e completo, pois todos os vértices são ligados entre si. Assim, os vértices representam a rainha e os peões, as arestas os custos (número mínimo de movimentos entre cada par de posições) e os pesos são definidos pelas distâncias calculadas com BFS.Os **vértices** representam a rainha e os peões.

Dessa forma, o problema se assemelha a uma variante do problema do Caixeiro Viajante, em que os vértices devem ser percorridos uma única vez, e o objetivo é minimizar o custo total.

SOLUÇÃO

Assim como solicitado, este trabalho implementa duas abordagens para resolver o problema: uma solução aproximada e uma exata.

SOLUÇÃO APROXIMADA

A solução aproximada se baseia em pegar os peões um de cada vez, sempre escolhendo aquele que for mais perto da posição atual da rainha. A cada rodada, o algoritmo calcula as distâncias entre a posição atual da rainha e as casas do tabuleiro que podem ser alcançadas, respeitando os obstáculos. Essas distâncias são obtidas por uma busca em largura, implementada na função bfs, que simula os movimentos da rainha expandindo em todas as direções possíveis, parando apenas quando encontra obstáculos ou peões.

Depois de calcular essas distâncias, a função *resolverAproximado* percorre a lista de peões ainda não capturados (armazenados no vetor peoes do TAD *tabuleiro*) e identifica qual deles está mais perto da rainha — ou seja, qual tem a menor distância no momento. Esse peão é então entendido como o próximo alvo da rainha. Após a captura, a posição atual da rainha é atualizada para a do peão recém-pego, e o processo se repete até que todos os peões tenham sido capturados ou que algum peão não possa ser alcançado. Caso algum peão esteja inacessível, a execução é interrompida e a resposta é considerada inválida (retornando -1).

Esse método garante que, a cada parte do processo, a rainha faz uma escolha localmente ótima (algoritmo guloso), ou seja, escolhe o melhor próximo passo com base na sua posição atual, sem considerar o impacto dessa escolha nas etapas futuras.

A estrutura principal envolvida aqui é o TAD *tabuleiro*, definido em *tabuleiro.hpp*, que representa o estado do tabuleiro, armazena a matriz com os símbolos (matriz), a posição da rainha (rainha) e as posições dos peões (peoes). Ele também oferece métodos como *lerEntrada* (para leitura do tabuleiro), *dentroDosLimites* (valida posições) e *celulaLivre* (verifica se uma célula pode ser ocupada).

SOLUÇÃO EXATA

A solução exata encontra o caminho com o menor número possível de ações necessárias para capturar todos os peões, independentemente da ordem em que isso aconteça. Para fazer isso, o programa primeiro calcula todas as distâncias entre a rainha e cada peão, e também entre cada par de peões, simulando os movimentos da rainha a partir de cada posição. Isso é feito com a função *calcularDistanciasBFS*, que é parecida com a bfs da solução aproximada, mas usada exclusivamente para gerar a matriz de distâncias reutilizável na solução exata.

Com essas distâncias disponíveis, a função *resolverCaminhoOtimo* começa a considerar todas as possíveis sequências em que os peões podem ser capturados. Para evitar recalcular as mesmas várias vezes, ela usa programação dinâmica com bitmask, armazenando os menores custos já obtidos em uma matriz chamada dp, onde cada entrada representa um subconjunto de peões capturados e a posição atual da rainha.

Assim, cada estado da matriz dp corresponde a uma combinação de peões já capturados (representados por bitmask, ou seja 1 se já tiverem sido capturados e 0 caso contrário) e uma posição final da rainha (um índice que se refere a alguma das posições na lista de posições).

O algoritmo tenta "expandir" esse estado, ou seja, simula a possibilidade de capturar mais um peão a partir dali, e atualiza o custo total para esse novo estado, se for menor que o que já estava registrado. Esse processo continua até que todas as combinações possíveis sejam consideradas.

Ao final, o algoritmo examina todos os estados onde todos os peões foram capturados, e identifica o de menor custo total. Assim, ele garante que a rainha siga a ordem de capturas que realmente resulta no menor número de movimentos no total — mesmo que, em certos momentos, ela precise se afastar de um peão mais próximo em nome de uma estratégia global melhor.

Essa solução também depende do TAD *Tabuleiro*, da estrutura *Ponto* (usada para representar coordenadas no tabuleiro), e utiliza vetores auxiliares para representar as distâncias (*dist*) e os estados da dinâmica (*dp*). A matriz *matrizDistancias* é usada para armazenar os resultados da BFS entre as posições relevantes.

ANÁLISE DE COMPLEXIDADE

SOLUÇÃO APROXIMADA

Tem complexidade de tempo assintótica de $O(K \cdot N \cdot M)$, onde K é o número de peões e $N \times M$ representa o tamanho do tabuleiro. Isso ocorre porque, a cada rodada, o algoritmo realiza uma busca em largura (BFS) a partir da posição atual da rainha, o que custa $O(N \cdot M)$, e em seguida percorre a lista de peões ainda não capturados para encontrar o mais próximo.

Como esse processo se repete até que todos os K peões sejam capturados, o custo total de tempo fica proporcional a K vezes o custo de cada busca. Em relação ao espaço, a solução utiliza uma matriz de distâncias com $O(N \cdot M)$ posições e um vetor de controle de peões capturados com $O(K)$, resultando em uma complexidade de memória de $O(N \cdot M)$.

SOLUÇÃO EXATA

Apresenta uma complexidade de tempo de $O(K^2 \cdot 2^K)$, visto que explora todas as combinações possíveis de captura dos peões utilizando programação dinâmica com bitmask. Para isso, ela mantém uma matriz *dp* onde cada estado representa um subconjunto de peões já capturados (com 2^K possíveis máscaras) e uma posição final da rainha (até $K + 1$ opções).

Em cada estado, o algoritmo tenta transições para peões ainda não capturados, o que adiciona um fator K ao custo. Além disso, no início da execução, são realizadas $K + 1$ buscas em largura, uma a partir de cada posição relevante (rainha + peões), o que tem custo total adicional de $O(K \cdot N \cdot M)$, mas que é desprezível por conta do crescimento exponencial da parte principal.

Quanto à memória, o principal consumo vem da matriz *dp*, com $O(2^K \cdot K)$ posições, e da matriz de distâncias entre as posições, que ocupa $O(K^2)$. Logo, a complexidade de memória da solução exata é $O(2^K \cdot K)$.

DISCUSSÃO DOS ALGORITMOS

Peões	Tempo algoritmo exato (ms)	Tempo algoritmo aproximado (ms)	Resultado exato	Resultado aproximado
3	0	0	8	8
10	36	30	34	36
15	142	72	43	50
16	218	77	57	68
17	367	89	53	53
18	721	90	51	58

Essa tabela apresenta os tempos e resultados empíricos retirados dos casos de teste e gravados via biblioteca *chrono.*

A solução exata apresentou um crescimento quase exponencial no tempo de execução, visto que pequenas alterações no número de peões alterou significativamente o tempo de execução, sendo coerente com a complexidade discutida acima.

A solução aproximada tem um crescimento mais suave e compatível com a complexidade discutida. Mesmo com o número de peões dobrando (de 9 para 18), o tempo de execução cresce de forma quase linear.

Apesar de ser mais rápida, a solução aproximada em muitos casos retorna um resultado diferente da solução exata (por exemplo, com 10, 15, 16 e 18 peões). No entanto, em alguns casos, como com 17 peões, a solução gulosa encontrou o mesmo valor que a ótima, o que acredito ter sido sorte pois o menor caminho acabou casando com o caminho percorrido, sendo notável uma perda de exatidão na solução aproximada em detrimento a um ganho de velocidade e menor gasto de memória.

Assim, cada algoritmo acaba resolvendo o problema de forma eficaz para diferentes situações, o algoritmo aproximado para problemas com uma entrada grande e algoritmo exato para quando a exatidão da resposta é extremamente necessária.

REFERÊNCIAS

Slides da disciplina.

ANDRETTA, André Luiz. Aula sobre Problema do Caixeiro Viajante. Disponível em: https://sites.icmc.usp.br/andretta/ensino/aulas/sme0241-1-19/aula3-TSP.pdf. Acesso em: 22 jun. 2025.