Trabalho Prático 3 - Captura dos peões

Luiz Felipe Gondim Araujo - 2023028188

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

luizfelipegondimaraujo@ufmg.br

1 Introdução

Esta documentação apresenta a solução desenvolvida para determinar o número mínimo de movimentos necessários para que uma rainha capture todos os peões em um tabuleiro personalizado. Foram implementadas duas abordagens: uma exata, utilizando programação dinâmica com estados representando subconjuntos de peões capturados, e uma aproximada, baseada em uma estratégia gulosa que sempre move a rainha até o peão mais próximo. Ambas compartilham a mesma modelagem do problema e são comparadas em termos de precisão e desempenho computacional.

2 Implementação

O programa foi desenvolvido em linguagem C++, utilizando o compilador G++ da GNU Compiler Collection. Em relação ao ambiente de desenvolvimento, consistiu em uma máquina rodando o sistema operacional Linux Mint 21, equipada com 16 GB de memória RAM. A escolha do Linux Mint foi pensada visando a execução do código na avaliação efetuada pelos monitores. Além disso, ainda foram usados o valgrind e o gdb para capturar possíveis erros e vazamentos em relação ao uso de memória.

3 Modelagem

O problema foi modelado como um grafo implícito sobre um tabuleiro bidimensional, onde cada célula do tabuleiro representa um vértice e as movimentações possíveis da rainha definem as arestas entre os vértices. A rainha pode se mover em qualquer uma das oito direções (horizontal, vertical e diagonal), com alcance ilimitado até encontrar um obstáculo ou o limite do tabuleiro, o que define a conectividade entre os vértices.

Ademais, cada posição acessível do tabuleiro pode ser alcançada por meio de sequências válidas de movimentos da rainha, desde que não atravessem células marcadas como intransponíveis. Sendo assim, a movimentação é simulada dinamicamente durante a execução da busca em largura (BFS), sem a necessidade de construir explicitamente o grafo completo, o que evita o custo exponencial de armazenar todas as conexões possíveis.

Por fim, as posições de interesse (casa inicial da rainha e posições dos peões) são tratadas como pontos especiais no tabuleiro. A distância mínima entre cada par de pontos é calculada com chamadas independentes à BFS. Com base nessa matriz de distâncias, o problema de capturar

todos os peões é modelado como uma variante do problema do Caixeiro Viajante (TSP), onde a rainha deve visitar todos os peões com o menor custo possível, partindo de sua posição inicial. Para a solução exata, foi utilizada programação dinâmica com bitmask para simular todas as ordens possíveis de visitação dos peões, enquanto para a solução aproximada, foi aplicada uma solução gulosa baseada na escolha do peão mais próximo a cada passo.

4 Solução

Para resolver o problema proposto, foi desenvolvida a classe `Solver`, responsável por centralizar todas as funcionalidades necessárias para o cálculo do número mínimo de movimentos da rainha até capturar todos os peões. Foram implementadas duas abordagens distintas: uma exata utilizando programação dinâmica com o algoritmo de Held-Karp, e outra aproximada baseada em uma heurística gulosa de vizinho mais próximo.

4.1 Leitura e Processamento da Entrada

Na função `main`, a entrada é lida diretamente da entrada padrão. O tabuleiro é armazenado em um vetor de strings, e durante a leitura cada célula é inspecionada para identificar a posição inicial da rainha (marcada com 'R') e as posições dos peões (marcadas com 'P'). Essas informações são armazenadas em estruturas `Position` para posterior uso pelos algoritmos de resolução.

4.2 Cálculo da Distância com BFS

A função `bfs` implementa uma busca em largura modificada para simular o movimento da rainha em até 8 direções. Sabendo disso, a cada iteração, a rainha avança em linha reta até encontrar um obstáculo ou a borda do tabuleiro. O algoritmo retorna a menor quantidade de movimentos para ir de um ponto de origem até um ponto de destino, sendo utilizado como sub-rotina tanto na versão exata quanto na aproximada.

4.3 Construção da Matriz de Distâncias

A função `computeDistanceMatrix` é responsável por calcular previamente a distância entre todos os pares de pontos relevantes do tabuleiro, ou seja, a posição da rainha e todas as posições dos peões. Para cada par de pontos, a função chama o método `bfs` e armazena o resultado em uma matriz de distâncias, que serve de base para os dois algoritmos de resolução.

4.4 Solução Exata com Held-Karp

A função `exactSolver` resolve o problema de forma exata utilizando o algoritmo de Held-Karp, uma abordagem de programação dinâmica eficiente para instâncias pequenas do Problema do Caixeiro Viajante (TSP). O algoritmo utiliza uma estrutura de estados `dp[mask][u]`, onde `mask` é uma máscara de bits que representa o conjunto de peões já capturados (cada bit indica se o peão correspondente foi visitado) e `u` é o último peão visitado.

Inicialmente, o algoritmo define os estados base considerando os movimentos da rainha a partir de sua posição inicial até cada um dos peões. Em seguida, para cada subconjunto de peões já visitados (`mask`) e para cada peão atual (`u`), tenta-se estender o caminho para um novo peão ainda não visitado. A matriz de distâncias pré-calculada (`computeDistanceMatrix`) é utilizada para acessar os custos entre pares de posições. Ao final, dentre todos os caminhos possíveis que visitam

todos os peões, o algoritmo retorna o de menor custo total. Essa abordagem garante exatidão ao custo de complexidade exponencial, sendo viável apenas para um número moderado de peões.

4.5 Solução Aproximada com Heurística Gulosa

A função `approximateSolver` resolve o problema de forma aproximada por meio de uma heurística gulosa baseada na ideia do *vizinho mais próximo*. A cada iteração, a rainha parte de sua posição atual e avalia a distância mínima até cada um dos peões ainda não capturados, utilizando a função `bfs` para esse cálculo. Em seguida, ela se move até o peão mais próximo (em número de movimentos), atualizando sua posição para refletir a nova captura.

Esse processo é repetido até que todos os peões tenham sido visitados. Em termos de estratégia, a decisão é sempre local e imediata, sem considerar o impacto de cada movimento sobre o custo total do percurso. Isso faz com que a solução possa ser significativamente mais rápida que o algoritmo exato, especialmente quando o número de peões é grande, já que evita explorar todas as permutações possíveis. Mas como dito anteriormente, essa solução não garante resposta ótima.

4.6 Medição de Tempo com `chrono`

Para fins de comparação entre as abordagens, foram incluídas medições de tempo com a biblioteca `chrono` da STL. O tempo de execução de cada algoritmo é calculado em milissegundos e impresso junto da respectiva resposta, permitindo avaliar o custo computacional de cada estratégia frente à qualidade da solução obtida.

5 Instruções de Compilação e Execução

Para executar o software, efetue os seguintes passos:

- Acesse a raiz do projeto;
- Utilizando um terminal, execute o comando 'make' para compilar o código;
- Ainda na raiz do projeto, digite: ./tp3.out;

6 Análise de Complexidade

A análise de complexidade será feita, principalmente, sobre os métodos principais da classe Solver e a main, uma vez que estes permitem uma visão geral da complexidade que o sistema carrega. Os demais métodos possuem custo constante, não impactando significamente a aplicação.

6.1 bfs

Em cada direção (8 possíveis), a rainha se move até encontrar um obstáculo ou o limite do tabuleiro, o que pode levar até $O(\max(N, M))$ passos por direção. Como o algoritmo utiliza uma fila para expandir os estados e cada célula é visitada no máximo uma vez por direção, o tempo total é $O(N \cdot M)$ no pior caso. O espaço utilizado é $O(N \cdot M)$ para armazenar a matriz de distâncias e a fila de posições a serem processadas. Portanto, a complexidade final de tempo e espaço é $O(N \cdot M)$.

6.2 computeDistanceMatrix

A função calcula a distância entre todos os pares de pontos de interesse (rainha e peões), totalizando $O(K^2)$ pares, onde K é o número de peões mais a rainha. Para cada par, é chamada a função `bfs`, com tempo $O(N \cdot M)$. Assim, a complexidade total de tempo é $O(K^2 \cdot N \cdot M)$. O espaço utilizado é $O(K^2)$ para armazenar a matriz de distâncias entre os pontos, além do espaço temporário de $O(N \cdot M)$ reutilizado a cada chamada de `bfs`. Portanto, a complexidade final de tempo é $O(K^2 \cdot N \cdot M)$ e de espaço é $O(K^2 + N \cdot M)$.

6.3 exactSolver

A complexidade da função é dominada pelo pré-processamento da matriz de distâncias entre todos os pontos (rainha e peões), seguido pela execução do algoritmo de Held-Karp. A matriz de distâncias é construída com $O(K^2)$ chamadas à função `bfs`, cada uma com complexidade $O(N \cdot M)$, resultando em tempo $O(K^2 \cdot N \cdot M)$. A etapa de programação dinâmica mantém uma tabela `dp[mask][u]` com $O(2^K \cdot K)$ estados, e para cada estado, realiza até $O(K)$ transições, levando a um tempo de $O(2^K \cdot K^2)$. O espaço utilizado é $O(K^2 + 2^K \cdot K)$ para armazenar a matriz de distâncias e a tabela de DP. Portanto, a complexidade total de tempo é $O(K^2 \cdot N \cdot M + 2^K \cdot K^2)$ e a complexidade espacial é $O(2^K \cdot K + K^2)$.

6.4 approximateSolver

A complexidade da função é dominada pelo cálculo das distâncias até o peão mais próximo em cada uma das K iterações. Em cada iteração, a rainha avalia todos os peões ainda não capturados, realizando até $O(K)$ chamadas à função `bfs`, cada uma com complexidade $O(N \cdot M)$. Assim, o tempo total é $O(K^2 \cdot N \cdot M)$. O espaço utilizado é $O(N \cdot M)$ para armazenar a matriz de distâncias usada em cada `bfs`, além de um vetor auxiliar de tamanho $O(K)$ para controlar os peões visitados. Portanto, a complexidade total de tempo é $O(K^2 \cdot N \cdot M)$ e a complexidade espacial é $O(N \cdot M + K)$.

6.5 main

A complexidade da `main`, desconsiderando a chamada aos algoritmos de resolução, é dominada pela leitura do tabuleiro e pela identificação das posições da rainha e dos peões. A leitura de N linhas com M caracteres ocorre em tempo $O(N \cdot M)$, e a verificação de cada caractere para identificar 'R' ou 'P' também é feita em $O(N \cdot M)$. O espaço utilizado é $O(N \cdot M)$ para armazenar o tabuleiro e $O(K)$ para armazenar as posições dos peões. Portanto, a complexidade total de tempo e espaço da função `main`, por si só, é $O(N \cdot M)$.

6.6 Análise geral

A complexidade total do programa é determinada pelas chamadas simultâneas às funções exactSolver e approximateSolver, após a leitura do tabuleiro. A função exactSolver realiza $O(K^2)$ chamadas à bfs, com tempo $O(N \cdot M)$ cada, para construir a matriz de distâncias, e aplica programação dinâmica com $O(2^K \cdot K^2)$ transições. Já a approximateSolver executa até $O(K)$ iterações, cada uma com até $O(K)$ chamadas à bfs, também com custo $O(N \cdot M)$ por chamada, resultando em tempo $O(K^2 \cdot N \cdot M)$. Como ambas utilizam internamente a função bfs, seus custos se acumulam. Assim, a complexidade total de tempo do programa é $O(K^2 \cdot N \cdot M + 2^K \cdot K^2)$ e o espaço total necessário é $O(2^K \cdot K + K^2 + N \cdot M)$, considerando o compartilhamento do tabuleiro e reutilização de estruturas auxiliares entre as chamadas.

7 Discussão dos Algorítmos

Após testar as 8 instâncias disponibilizadas pelo monitor e calcular os tempos de execução do algoritmo exato e aproximado, foi possível construir a seguinte tabela:

Tabela 1: Comparação entre os algoritmos exato e aproximado

Quantidade de Peões	Tempo Exato (ms)	Resposta Exata	Tempo Aproximado (ms)	Resposta Aproximada
3p	2	8	1	8
7p	79	22	25	23
10p	309	34	132	36
12p	674	42	320	43
15p	1026	43	488	50
16p	1214	57	550	68
17p	1581	53	635	53
18p	1917	51	667	58

Observa-se que, para instâncias pequenas (como 3 ou 7 peões), ambos os algoritmos produzem resultados idênticos ou bastante próximos, com tempos de execução baixos. No entanto, à medida que o número de peões aumenta, a diferença entre os tempos se torna mais acentuada. Por exemplo, com 18 peões, o algoritmo exato levou 1917 milissegundos para encontrar a melhor solução, enquanto o algoritmo aproximado produziu uma resposta em apenas 667 milissegundos (uma redução de mais de 65% no tempo).

Essa economia de tempo, entretanto, pode vir acompanhada de uma pequena perda de qualidade na solução. Em vários casos (como 15p e 16p), a solução aproximada apresenta um erro considerável em relação à solução ótima, chegando a ser 11 movimentos pior no pior caso observado. Isso ocorre porque a heurística gulosa não considera todas as permutações possíveis de captura dos peões, o que pode levar a decisões locais subótimas, dependendo da quantidade e localização dos obstáculos, por exemplo.

Apesar disso, o algoritmo aproximado ainda se mostra eficiente e aceitável para cenários em que o tempo de execução é crítico e uma solução próxima da ideal é suficiente. Já o algoritmo exato, embora mais lento, garante a resposta mínima possível e é adequado para instâncias pequenas ou quando a precisão é essencial.

Referências

[1] USP. Método Guloso. Disponível em: `https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/guloso.html`. Acesso em: 20 jun. 2025.

[2] COMP GEEK. Held-Karp Algorithm for TSP. Disponível em: `https://compgeek.co.in/held-karp-algorithm-for-tsp/`. Acesso em: 21 jun. 2025.

[3] CP-ALGORITHMS. Introduction to Dynamic Programming. Disponível em: `https://cp-algorithms.com/dynamic_programming/intro-to-dp.html`. Acesso em: 21 jun. 2025.