



Universidade Federal de São João del Rei
Campus Tancredo Neves - Ciência da Computação

TRABALHO PRÁTICO 2

Oscar Alves Jonson Neto
Geraldo Arthur Detomi

São João del-Rei
2025

Sumário

1. Introdução	3
1.1 Enunciado	3
1.2 Análise de entradas e saídas	3
2. Análise do Problema	4
2.1 Desafios e Componentes Fundamentais	4
2.2 Estratégia da Resolução	5
3. Desenvolvimento do Código	5
3.1 mundo_zambis.c	6
3.2 strategydp.c e dp_util.c	6
3.3 strategy_guloso.c	8
3.4 main.c	8
4. Testes	9
4.1 Análise de Complexidade Programação Dinâmica	9
4.1.1 Função get_max_habilidade_path	9
4.1.2 Função calc_maximo_habilidade_dp	9
4.2.1 Função get_caminho_max_habilidade	10
4.2.2 Função max_habilidade_guloso_por_povo	10
4.3 Análise de Resultados	11
5. Conclusão	12
6. Referências	12

1. Introdução

Este projeto tem como objetivo apresentar o desenvolvimento do Trabalho Prático 2 da disciplina de Projeto e Análise de Algoritmos, ofertada pelo docente Leonardo Chaves Dutra da Rocha.

1.1 Enunciado

O objetivo é auxiliar Zorc a definir um trajeto e uma estratégia de recrutamento que maximize a habilidade total do exército, respeitando as restrições da nave. O problema consiste em, dado um conjunto de P povos, as características dos soldados de cada povo (peso e habilidade), as distâncias d_{ij} entre os povos conectados, uma capacidade máxima de peso W e uma distância máxima D que pode ser percorrida, determinar o caminho e a quantidade de soldados a recrutar em cada povo visitado de forma a maximizar a soma total das habilidades dos soldados recrutados.

1.2 Análise de entradas e saídas

O programa recebe os dados de entrada a partir de um arquivo. A primeira linha do arquivo contém um inteiro K , indicando o número de entradas de teste. Para cada entrada temos:

- Uma linha contendo quatro inteiros: P (número de povos), D_{max} (distância máxima que Zorc pode andar), W_{max} (peso máximo da nave) e C (quantidade de caminhos diretos entre povos).
- P linhas seguintes, cada uma descrevendo um povo: p_i (identificador do povo), w_i (peso dos soldados do povo p_i) e h_i (habilidade dos soldados do povo p_i). A habilidade H varia de 1 a 10.
- C linhas seguintes, cada uma descrevendo um caminho: p_u , p_v (povos conectados) e d_{uv} (distância entre p_u e p_v). Se um par de povos não estiver listado, não há caminho direto entre eles.

Para cada entrada de teste, o programa deve imprimir a habilidade total máxima do exército recrutado, seguida pelas tuplas: povo visitado e a quantidade de soldados recrutados nesse povo.

Exemplo de entrada:

6 10 310 7 (Povos, Distância, Peso e Caminhos)

1 70 2 (Identificador do povo, peso do soldado e habilidade dos soldados)

2 100 3

3 20 7

4 90 4

5 20 3

6 10 1

1 2 3 (Povo i, Povo j e a distância que os liga)

1 5 2

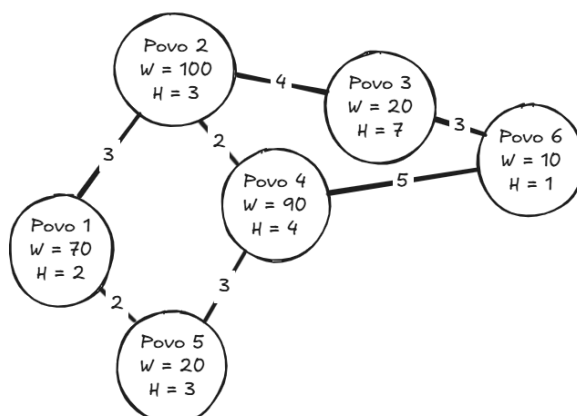
2 3 4

2 4 2

3 6 3

4 5 3

4 6 5



Grafo montado através da entrada fornecida acima

2. Análise do Problema

2.1 Desafios e Componentes Fundamentais

Zorc precisa montar o exército mais forte possível no Mundo de Zambis, e isso envolve uma série de decisões interligadas. Ele pode começar em qualquer um dos P povos e, a cada passo, decide para onde ir e quantos soldados recrutar.

As limitações são duas: o peso máximo que a nave pode carregar (W_{max}) e a distância total que Zorc pode percorrer (D_{max}). A cada novo grupo de soldados recrutados a

nave perde capacidade, e a cada deslocamento entre povos uma parte da distância permitida é consumida. Por isso, não é possível visitar todos os povos nem recrutar todos os soldados que quiser.

O objetivo é simples: recrutar o maior total de habilidade possível. Para isso, é preciso decidir bem quais povos visitar, em que ordem, e quantos soldados levar de cada um — tudo isso respeitando as restrições.

Problemas análogos ao de Zorc são frequentes no mundo real. Considere, por exemplo, a **logística de distribuição de mercadorias**: uma empresa precisa definir rotas para seus veículos (limitados por capacidade de carga e autonomia/tempo) para coletar ou entregar produtos em diferentes localidades, visando maximizar o lucro ou minimizar custos.

2.2 Estratégia da Resolução

Para resolver o problema de maximizar a habilidade total do exército de Zorc, foram implementadas duas estratégias distintas:

Programação Dinâmica (DP): Esta abordagem visa encontrar a solução ótima global. No contexto do problema de Zorc, a DP explora sistematicamente as escolhas de recrutar soldados no povo atual ou de se mover para um povo vizinho, mantendo o controle das restrições de peso e distância. Para evitar recálculos e garantir a eficiência, uma tabela de memorização é utilizada para armazenar os resultados de estados já visitados. O objetivo é encontrar a sequência de decisões que leva à maior habilidade acumulada ao final da jornada.

Heurística Gulosa: Esta estratégia busca uma solução boa de forma rápida, fazendo escolhas localmente ótimas em cada etapa. A decisão gulosa se baseia em uma métrica de "eficiência" calculada para cada possível próximo passo (visitar um vizinho e/ou recrutar). Esta métrica considera a habilidade e o peso dos soldados, bem como a distância para alcançar o próximo povo.

3. Desenvolvimento do Código

Diante da estratégia de resolução, usamos dois métodos para resolver o problema, sendo o primeiro a **Programação Dinâmica** que nos dá uma solução ótima porém utilizando mais memória e processamento, e também uma heurística de nossa escolha, que no caso, escolhemos uma **Heurística Gulosa** que busca uma solução boa, utilizando

menos recursos e processamento que a programação dinâmica, e que em alguns casos pode dar também, a solução ótima do problema.

Para facilitar o desenvolvimento e entendimento do código, dividimos o algoritmo em módulos:

- dp_util.c
- entrada.c
- main.c
- mundo_zambis.c
- resposta.c
- strategy_guloso.c
- strategydp.c
- tempo.c

Mas os módulos principais são mundo_zambis.c, strategydp.c, dp_util.c, strategy_guloso.c e main.c, que vamos analisar agora.

3.1 mundo_zambis.c

O módulo **mundo_zambis.c** é central para a representação do ambiente do problema. Ele é o responsável por gerenciar a struct MundoZambis e também a struct NaveZorc, que encapsulam todos os dados relativos ao mapa, aos povos e às e às restrições da nave de Zorc. A função **criar_mundo_zambis** é responsável por alocar dinamicamente a memória necessária para estas estruturas, incluindo uma matriz de adjacência para o grafo de povos, e ponteiros para as habilidades e pesos dos soldados de cada povo.

Funções como **get_habilidade_por_povo**, **get_peso_por_povo**, **sao_povos_vizinhos** e **get_distancia_entre_povos**, fornecem interfaces controladas para acessar esses dados. Já a liberação da memória alocada é feita por **destruir_mundo_zambis** que recebe a struct MundoZambis e muda todos os valores dos ponteiros para NULL e os desaloca para evitar vazamentos de memória.

3.2 strategydp.c e dp_util.c

O módulo **strategydp.c** é o responsável pela implementação da estratégia de Programação Dinâmica, cujo objetivo é determinar a solução ótima para o problema de recrutamento de Zorc.

A função central desta estratégia é **calc_maximo_habilidade_dp**, a função calcula recursivamente a máxima habilidade que pode ser obtida a partir de um estado, considerando duas ações principais:

(1) Recrutar soldados no povo atual, o que consome peso e adiciona habilidade, mantendo Zorc no mesmo povo para uma possível próxima ação de recrutamento;

(2) Mover-se para um povo vizinho, consumindo distância e começando um novo estado no povo vizinho.

Os resultados de cada estado (povo, peso, distância) são computados e armazenados em uma tabela de memoização *dp_data->memo*, fornecida pelo módulo **dp_util.c**, para evitar recálculos.

O módulo **dp_util.c** fornece as ferramentas e estruturas de dados para a programação dinâmica. A struct *DPData* encapsula as matrizes tridimensionais *memo* e *estados*, sendo *estados* uma struct *Estado* que está interligada com *DPData*. Este módulo contém funções para a alocação dinâmica através dos métodos **alocar_matriz_tridimensional** e **alocar_matriz_tridimensional_estado**, inicialização da struct *DPData* com o método **inicializar_DPData** e reinicialização através de **resetar_DPData**, que preenche *memo* com -1, e por fim, para a liberação dessas matrizes e structs, temos **liberar_matriz_tridimensional**, **liberar_matriz_tridimensional_estado**, e **liberar_DPData** que desalocam as memórias e as preenche com NULL.

Após a tabela *memo* ser completamente preenchida pela função recursiva para um dado ponto de partida, o método **reconstruir_caminho_dp** é chamado. Este método utiliza a tabela *dp_data->estados*, que armazena, para cada estado, o estado predecessor que levou à sua solução ótima para traçar o caminho de decisões tomadas, identificando os povos visitados e a quantidade de soldados recrutados em cada um.

A função de interface do módulo, **get_max_habilidade_path**, orquestra todo o processo da programação dinâmica. Ela itera por todos os *P* povos, tratando cada um como um potencial ponto de partida. Para cada partida, *DPData* é resetado, **calc_maximo_habilidade_dp** é chamada, e o caminho resultante é reconstruído. A função então compara a habilidade total de cada caminho gerado e retorna o *CaminhoSolucao* que representa a solução globalmente ótima.

3.3 strategy_guloso.c

O módulo **strategy_guloso.c** implementa uma estratégia heurística como alternativa à Programação Dinâmica. Seu objetivo é encontrar boas soluções em menos tempo, especialmente em instâncias maiores, embora sem garantir a otimalidade.

A heurística toma decisões com base em uma métrica local de eficiência, calculada pela função **eficiencia_visita**. Essa eficiência é dada por:

$$(\text{habilidade}[\text{povo}[\text{vizinho}]]/\text{peso}[\text{povo}[\text{vizinho}]])/\text{distancia}(\text{povo}[\text{origem}],\text{povo}[\text{vizinho}])$$

Ou seja, mede a habilidade pelo peso, penalizada pela distância, priorizando os vizinhos mais vantajosos.

A lógica principal está no método **max_habilidade_guloso_por_povo**, que, a partir de um povo inicial, o algoritmo recruta o máximo possível de soldados respeitando o peso disponível na nave. Em seguida, identifica os vizinhos acessíveis, calcula sua eficiência e os ordena com QuickSort. O vizinho mais eficiente e viável é escolhido como próximo destino. O processo se repete até não haver mais movimentos válidos, seja por distância, peso ou falta de vizinhos vantajosos. Ao final, é construído um *CaminhoSolucao* com o trajeto percorrido e os recrutamentos feitos.

Por fim, a função **get_caminho_max_habilidade** executa a heurística para cada um dos P povos como ponto de partida e retorna o caminho com maior habilidade total.

3.4 main.c

O módulo **main.c**, é o responsável por executar a função principal do programa, responsável por ler os dados de entrada, processar as instâncias do problema e medir o tempo de execução. Inicialmente, os argumentos da linha de comando são validados; em caso de erro, uma mensagem é exibida e a execução é encerrada.

Com os parâmetros válidos, o programa tenta abrir o arquivo de entrada. Se falhar, trata o erro. Um temporizador é iniciado para medir o tempo total de execução. Em seguida, o programa entra em um laço de repetição K vezes.

A cada laço é lido os parâmetros da entrada: número de povos P , distância máxima D_{max} , peso máximo W_{max} e quantidade de caminhos C , a struct *MundoZambis* é criada através da função **criar_mundo_zambis**, é iniciado um timer referente a entrada atual, e

por fim através de parâmetro, oferecido na linha de comando, é chamada a função **get_max_habilidade_path** ou **get_caminho_max_habilidade**, que retorna com o caminho solução, o timer da entrada então é finalizado, a resposta é escrita no terminal e em um arquivo de saída, as alocações feitas são liberadas e isso se repete K vezes.

4. Testes

Agora que entendemos como o programa funciona, precisamos rodá-lo e analisar seus resultados e também sua complexidade.

4.1 Análise de Complexidade Programação Dinâmica

A estratégia de Programação Dinâmica foi projetada para encontrar a solução **ótima**. Sua análise de complexidade considera as seguintes variáveis principais:

- P (número total de povos),
- W_{max} (capacidade máxima de peso da nave) e
- D_{max} (distância máxima que Zorc pode percorrer).

4.1.1 Função **get_max_habilidade_path**

A função **get_max_habilidade_path** é a rotina principal da estratégia de DP, iterando sobre cada um dos P povos como ponto de partida e invocando **calc_maximo_habilidade_dp** para cada um. A complexidade desta função é, portanto, P multiplicado pela complexidade de uma chamada completa a **calc_maximo_habilidade_dp**.

4.1.2 Função **calc_maximo_habilidade_dp**

Esta é a função central da DP. O número de estados distintos que podem ser alcançados e memorizados é dado pelo produto das dimensões:

$$P \cdot (W_{max} + 1) \cdot (D_{max} + 1)$$

Considerando que a tabela de memorização é preenchida uma vez, o custo total para uma execução da DP a partir de um ponto de partida (preenchendo toda a matriz *memo*) é

$$O(P \cdot W_{max} \cdot D_{max} \cdot P) = O(P^2 \cdot W_{max} \cdot D_{max})$$

Portanto, a **complexidade de tempo total** da função **get_max_habilidade_path** é dominada pelo preenchimento da tabela de DP para cada ponto de partida, resultando em

$$O(P \cdot (P^2 \cdot W_{max} \cdot D_{max})) = O(P^3 \cdot W_{max} \cdot D_{max})$$

A função **reconstruir_caminho_dp** percorre os estados armazenados para montar a solução final. Sua complexidade é aproximadamente

$$O(P + W_{max} + D_{max})$$

que é geralmente menor que o custo de preenchimento da tabela de DP.

Ou seja, o algoritmo de programação dinâmica é um algoritmo pseudo-polinomial, e pra piorar, ele também é extremamente custoso, com a complexidade de espaço das matrizes *memo* e *estados* estando em $O(P \cdot W_{max} \cdot D_{max})$.

4.2 Análise de Complexidade da Heurística Gulosa

A estratégia Gulosa opera iterativamente, fazendo escolhas locais que chegam em um resultado bom, e às vezes em um resultado ótimo. Sua complexidade é analisada em função de P

4.2.1 Função **get_caminho_max_habilidade**

Esta função chama **max_habilidade_guloso_por_povo** para cada um dos P povos, então sua complexidade é P vezes a da função interna.

4.2.2 Função **max_habilidade_guloso_por_povo**

A função **max_habilidade_guloso_por_povo** executa a partir de um *povo_inicio* específico, em cada etapa no *povo_atual*, o algoritmo recruta soldados e, em seguida, ordena os vizinhos alcançáveis (até P-1) pela métrica de *eficiencia_visita* usando quickSort (*custo* $O(P \log P)$) para escolher o próximo destino. O loop principal executa no máximo *Lmax* vezes, portanto a complexidade de tempo desta função é:

$$O(L_{max} \cdot P \log P)$$

No **pior caso**, com $L_{max} \approx P$, a complexidade é $O(P^2 \log P)$. Consequentemente, a função **get_caminho_max_habilidade**, que invoca **max_habilidade_guloso_por_povo** P vezes, opera em:

$$O(P^3 \log P)$$

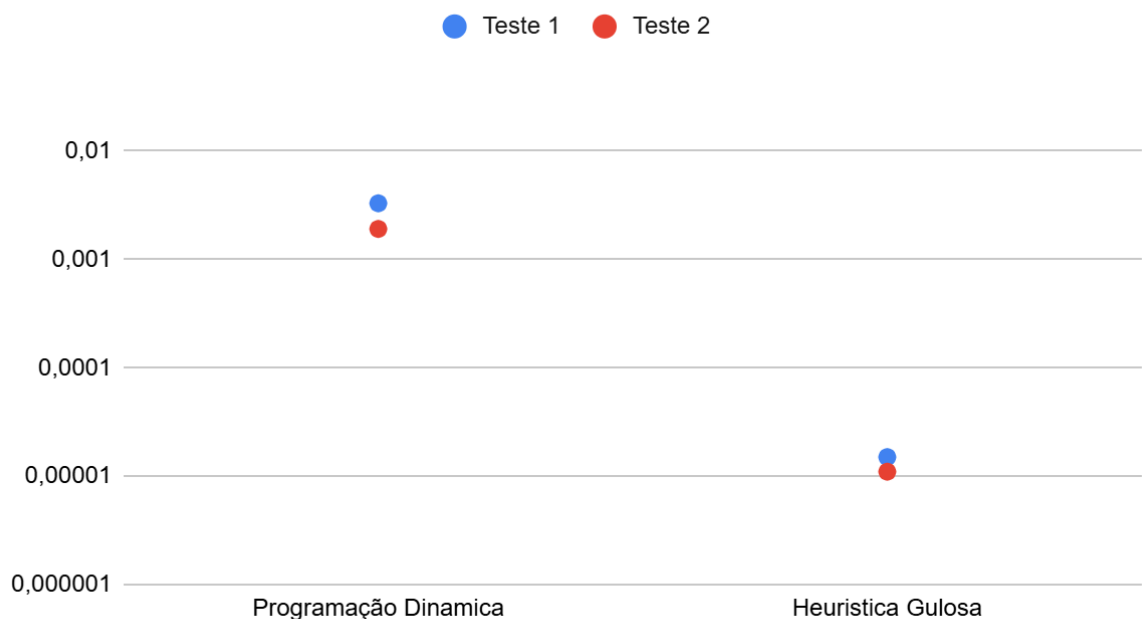
O **melhor caso** para **max_habilidade_guloso_por_povo** ocorreria com um caminho muito curto (L_{max} pequeno) e poucos vizinhos (k) a ordenar em cada passo, aproximando-se de $O(L_{max} \cdot k \log k)$, que consequentemente iria acabar operando em $O(P \cdot L_{max} \cdot k \log k)$ já que **get_caminho_max_habilidade** roda P vezes.

4.3 Análise de Resultados

Agora que entendemos a complexidade do código, vamos rodar o mesmo e analisar os resultados, primeiro vamos analisar o tempo de ambos os códigos:

Rodamos as entradas oferecidas no enunciado do TP e esses foram os resultados

Resultados TP2



Ou seja, a heurística gulosa foi aproximadamente 100 vezes mais rápida que a solução com programação dinâmica, mas lembrando que em geral a heurística gulosa **difícilmente** retorna uma solução ótima, diferente da programação dinâmica que garante uma solução ótima.

Outro ponto de grande diferença das duas é o uso de memória onde nos testes feitos, tivemos o seguinte retorno:

- Programação Dinâmica: Foram alocados 673,852 bytes ou 673KB.
- Heurística Gulosa: Foram alocados 11380 bytes ou 11KB.

Ou seja, o uso de memória da programação dinâmica é significativamente mais elevado, aproximadamente 59 vezes maior que o da heurística gulosa nos testes realizados.

5. Conclusão

Este trabalho abordou o problema de otimização do recrutamento de exércitos para o guerreiro Zorc, por meio de duas estratégias: Programação Dinâmica (DP) e uma Heurística Gulosa.

A abordagem por DP garantiu a otimalidade da solução, explorando exaustivamente os estados possíveis com auxílio de memorização. Apesar de seu alto custo computacional em tempo e em espaço, a estratégia entrega sempre a melhor resposta.

Já a Heurística Gulosa prioriza eficiência, com complexidade em tempo e em espaço. Embora não assegure otimalidade, é uma opção prática para instâncias grandes ou quando o tempo de resposta é mais importante.

A comparação entre as abordagens evidencia o dilema entre ótimos resultados e eficiência, reforçando a importância de escolher a estratégia mais adequada ao contexto do problema.

6. Referências

[CORMEN et al., 2012] CORMEN, T. H., Leiserson, C., Rivest, R., and Stein, C (2012). Algoritmos: teoria e prática. LTC.

BACKES, A. LINGUAGEM C: COMPLETA E DESCOMPLICADA. [s.l.] Elsevier, 2012.

