Universidade Federal
de São João del-Rei

Trabalho Prático 2

Disciplina: Projeto e Análise de Algoritmos
Integrantes: Gabriel da Silva Souza
José Vitor Santos Alves
Professor: Leonardo Rocha

São João del-Rei - Maio, 2025

Sumário

1. **O Problema**
2. **Algoritmos em Detalhe**
   - 2.1. Modelagem do Problema no Mundo Real
   - 2.2. Modelagem da Solução no Código
   - 2.3. Resolução com Heurística Gulosa
   - 2.4. Resolução com Programação Dinâmica
3. **Testes e Resultados**
4. **Compilação e Execução**
5. **Saída Gerada**
6. **Análise de Complexidade**
   - 6.1. Funções de Leitura e Estruturas de Dados
   - 6.2. Heurística Gulosa
   - 6.3. Programação Dinâmica
7. **Conclusão**

1. O Problema

O comandante Zorc precisa recrutar soldados de diferentes povos para uma missão. Cada povo possui uma habilidade específica e um peso associado aos seus soldados. Zorc possui uma nave com autonomia máxima de distância (**D**) que pode viajar e uma capacidade máxima de peso (**W**) que pode carregar. Existem caminhos definidos entre os povos, cada um com uma distância específica.
O objetivo é determinar a estratégia de recrutamento que maximize a habilidade total dos soldados recrutados, respeitando as restrições de autonomia da nave e a capacidade da nave. O problema considera que Zorc pode começar em qualquer povo e pode recrutar soldados de um mesmo povo várias vezes.
Usamos para este trabalho duas abordagens para soluções o problema:

- **Heurística Gulosa:** Uma abordagem rápida que busca uma boa solução, mas não necessariamente a solução ótima, dando preferência aos povos com melhor razão habilidade/peso e que sejam alcançáveis;
- **Programação Dinâmica:** Uma abordagem mais complexa que explora os povos visitados e os recursos restantes por meio de estados para encontrar a solução ótima.

2. Algoritmos em Detalhe

2.1. Modelagem do Problema no Mundo Real e no Código

Antes de detalhar os códigos, é importante entender como esse problema pode ser representado no mundo real e como as estruturas podem ser representadas.

Representação no Mundo Real:

Imagine uma ONG de ajuda humanitária que precisa coletar diferentes tipos de suprimentos de vários depósitos para entregar em uma zona de desastre.

- Cada tipo de suprimento tem sua importância (habilidade) e ocupa um certo espaço no caminhão (nave);
- Os depósitos estão conectados por ruas/estradas (caminhos), cada uma com seu custo de deslocamento;
- O caminhão tem uma autonomia máxima de deslocação antes de precisar reabastecer;

- O objetivo é maximizar a importância dos suprimentos para entregar na zona de desastre.

Modelagem da Solução:

Mapa de Caminhos e Povos:

- Os *Povos* e *Caminhos* são representados como um grafo não direcionado. A função *criarGrafo* constrói essa representação utilizando uma lista de adjacência. Cada *Povo* é um vértice, e cada *Caminho* é uma aresta com um peso correspondente à distância. A estrutura *Povos* armazena a lista de todos os povos com seus atributos: id, peso e habilidade.

Recursos e Restrições:

- A **habilidade** e o **peso** de cada soldado de um *Povo* são atributos da struct *Povo*;
- A distância de um caminho é um atributo da struct *Caminho* e é usada como peso da aresta;
- A autonomia máxima (**D**) e o peso máximo (**W**) são parâmetros globais para a solução, passados para as funções *resolverComHeuristica* e *resolverComPD*;

Acompanhamento da Solução e Progresso:

- Para a abordagem *resolverComPD*, a struct *Estado* é crucial:
    - **povo:** Onde Zorc está no momento;
    - **distancia_restante:** Quanto a nave ainda pode viajar;
    - **peso_usado:** Quanto da capacidade **W** já foi preenchida;
    - **habilidade:** A habilidade total acumulada até o momento;
    - **caminho** e **tamanho_caminho:** Guardam a sequência dos povos visitados;
    - **quantidades:** Armazena quantos soldados de cada id foram recrutados.
- A struct *Resultado* armazena a solução final encontrada por ambas as abordagens, contendo a *habilidadeTotal*, a lista de visitados e *quantidadeRecrutada*.

Objetivo da Otimização:

- O campo **habilidadeTotal** na struct *Resultado* (ou **habilidade** na struct *Estado*) é o que se busca maximizar. Os algoritmos tentam diferentes sequências de visitação e recrutamento para encontrar o maior valor possível para este campo, sempre respeitando **D** e **W**.

2.2. Resolução com Heurística Gulosa

Esta abordagem busca uma solução de forma rápida, tomando decisões que parecem melhores no momento.

Fluxo Geral Detalhado:

2.2.1. Construção do Grafo:

- Primeiramente, um grafo completo com todos os povos e caminhos é montado usando *criarGrafo*.
- Em seguida, para aplicar um filtro de alcance, a função *bfs_distancias* é chamada. Esta função calcula as menores distâncias de um povo inicial fixo a todos os outros povos. Ela opera de forma similar a um algoritmo de Dijkstra simplificado, atualizando as distâncias aos vizinhos iterativamente.
- Com estas distâncias, *filtrarPorDistancia* cria uma nova lista contendo apenas os povos cuja dist[id] é menor ou igual à autonomia D.
- O grafo completo é então liberado com a função *liberarGrafo*, pois a heurística prossegue apenas com a lista de povos filtrados.

2.2.2. Ordenação por Eficiência:

- A lista de Povos filtrados é ordenada. A função *comparePovosByRatio* é usada como critério de ordenação.
- **Justificativa da Razão Habilidade/Peso:** Os povos são ordenados de forma decrescente pela razão habilidade/peso. Esta é uma escolha gulosa clássica para o problema da mochila fracionária. A ideia é priorizar povos que oferecem a maior quantidade de habilidade por unidade de peso consumida. Povos com peso igual à 0 e habilidade maior que 0 são tratados como infinitamente eficientes e colocados no início.

2.2.3. Seleção Gulosa e Recrutamento:

- O algoritmo itera sobre os povos ordenados, do mais eficiente para o menos eficiente;
- Para cada povo na lista ordenada:
    - Verifica-se o **peso_restante** = **W** - **peso_ja_carregado**.
    - Se o peso do povo é maior que 0: Calcula **quantidade_a_recrutar** = peso_restante / povo->peso.
    - Se o peso do povo é igual à 0 e a habilidade do povo é maior que 0: Pode se recrutar quantos soldados forem, pois não consomem peso.

- Se **quantidade_a_recrutar** é maior que 0:
  - O id do povo e a quantidade recrutada são adicionadas ao resultado;
  - O **peso_ja_carregado** é atualizado, diminuindo a capacidade **W** restante para os próximos povos.

- O processo para quando todos os povos filtrados forem considerados ou quando o peso carregado atinge **W**.

2.2.4. Retorno e Limpeza:

- A estrutura *Resultado*, contendo a sequência de recrutamento e a habilidade total, é retornada.
- A memória alocada para a lista de povos filtrados é liberada usando **destruirPovos**.

2.3. Resolução com Programação Dinâmica

Esta abordagem é mais complexa e visa explorar o espaço de soluções de maneira mais sistemática para encontrar a solução ótima. Ela pode ser vista como uma forma de Branch and Bound.

Fluxo Geral Detalhado:

2.3.1. Construção do Grafo (criarGrafo):

- Assim como na heurística, um grafo representando todos os povos e caminhos é criado usando *criarGrafo* para permitir a navegação entre os povos.

2.3.2. Inicialização da Fila de Prioridade e Melhor Resultado:

- Uma FilaPrioridade é criada usando *criar_fila*. O critério de ordenação é dado por *comparador_habilidade*, que prioriza estados com maior habilidade acumulada. Isso significa que a fila sempre fornecerá o melhor estado para expansão.
- Uma estrutura *Resultado *melhor* é inicializada para guardar a melhor solução completa encontrada durante a busca.

2.3.3. Geração de Estados Iniciais:

- O diferencial desta abordagem é que Zorc pode começar de qualquer povo.
- Para cada *povo_inicial* na lista de todos os povos um estado é criado:
  - **povo** = **povo_inicial** -> **id;**
  - **distancia_restante** = **D;**
  - **peso_usado** = 0;
  - **habilidade** = 0;

    - **caminho** é alocado e contém apenas o id desse povo;
    - **quantidades** é alocado e zerado.
- **Recrutamento no Povo Inicial:** Recruta-se o máximo possível de soldados do **povo_inicial**, atualizando **estado_inicial.peso_usado**, **estado_inicial.habilidade** e **estado_inicial.quantidades [povo_inicial->id]**.
- Este **estado_inicial** é então inserido na fila de prioridade usando enfileirar. A função enfileirar insere o novo estado mantendo a ordem da fila.

2.3.4. Finalização:

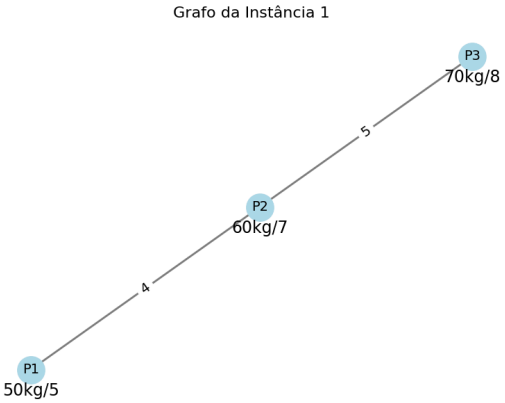
- Quando a fila se esvazia, todos os estados promissores foram visitados;
- A memória da fila e do grafo é liberada (*liberar_fila*, *liberarGrafo*);
- O melhor resultado é retornado.

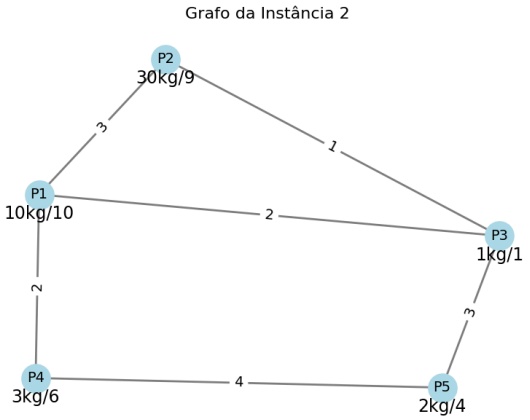
Esta abordagem é mais custosa porque pode explorar um número muito grande de estados. A fila de prioridade ajuda a focar nos caminhos que já acumularam mais habilidade, mas não impede a exploração de muitos ramos se várias rotas parecerem boas inicialmente.

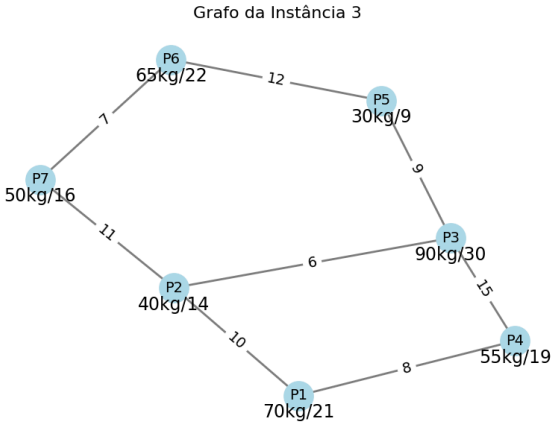
3. Testes e Resultados

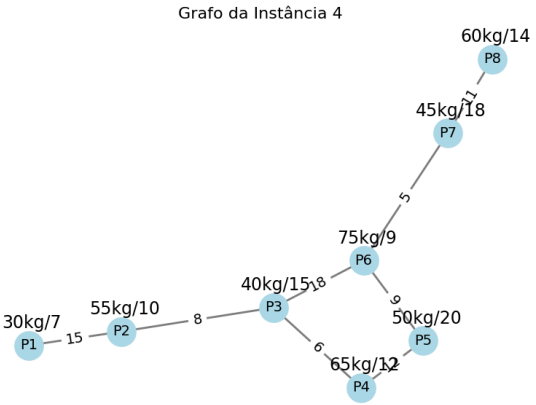
Número de povos:	Número de caminhos	Resultado Heurística	Resultado Programação dinâmica	Tempo Heurística / Programação dinâmica (ms)
3	2	14	16	0.008000/ 0.012000
5	6	634	634	0.009000/ 0.060000
7	8	98	102	0.013000/ 0.081000
8	8	135	144	0.011000/ 0.171000
9	9	180	180	0.014000/ 0.233000

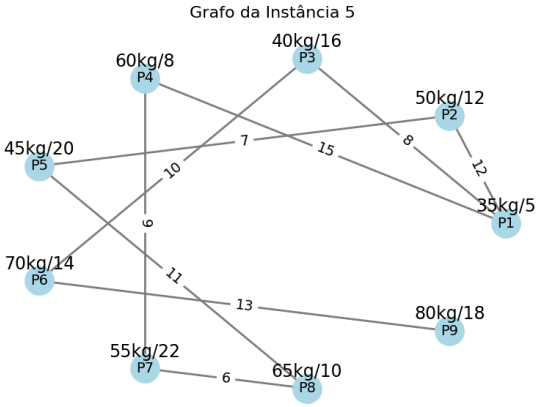
Grafos:











4. Compilação e Execução

Para compilação é usado a Makefile:

```
objetos = IO.o Caminhos.o Solucao.o
```

```
run: tp2
   ./tp2 entrada.txt

tp2: tp2.c $(objetos)
   gcc -o tp2 tp2.c $(objetos) -lm

IO.o: IO.c IO.h
   gcc -c IO.c

Caminhos.o: Caminhos.c Caminhos.h
   gcc -c Caminhos.c

Solucao.o: Solucao.c Solucao.h Caminhos.h
   gcc -c Solucao.c

clean:
   rm -f tp2 $(objetos)
   rm -f *.o
   # remove os txt menos a entrada
     find . -maxdepth 1 -type f -name '*.txt' ! -name 'entrada.txt'
-delete
```

5. Saída Gerada

Cada Instância é salva em diferentes arquivos, com o nome do arquivo indicando qual método foi usado, e qual o número da instância, dentro do arquivo tem o seguinte formato:
O primeiro valor é a habilidade total, seguido pelo número do povo visitado, e o número de habitantes recrutados.
ex: o Arquivo “SaidaPD1.txt” onde PD é o método, 1 significa que é a primeira instância
conteúdo:
106 3 15 6 1
habilidade total 106, passando pelo povo 3 e recrutando 15, depois pelo povo 6 e recrutando 1.

6. Análise de complexidade

Seja **P** o número de povos, **C** o número de caminhos, **D** a autonomia e **W** o peso máximo.

6.1. Funções de Leitura e Estruturas de Dados

- *lerEntrada*: O (L), onde **L** é o tamanho total do conteúdo do arquivo de entrada;
- *criarGrafo*: O (C), pois cria a lista de adjacência inicializando o grafo e adicionando cada aresta uma única vez;
- *bfs_distancias*: O (P + C), pois cada vértice e aresta é visitado no máximo uma vez durante a busca em largura.

6.2. Heurística Gulosa

- *resolverComHeuristica*: O (C + PlogP), pois realiza uma filtragem linear dos caminhos, depois uma ordenação dos povos pela razão e uma estratégia gulosa.

6.3. Programação Dinâmica

- *resolverComPD*: O (P*D*W), pois utiliza programação dinâmica para explorar todos os estados possíveis com base no povo atual, distância restante e peso acumulado, considerando as transições entre vizinhos.

7. Conclusão

Ambas as estratégias alcançaram o objetivo de maximizar a habilidade total dos soldados recrutados, sempre respeitando as restrições de peso e distância percorrida.

A abordagem da heurística Gulosa apresenta um resultado com tempos significativamente menores, principalmente em casos maiores, a solução foi boa, com pequenas diferenças nos testes realizados, em comparação com a solução ótima.

Por outro lado a solução da programação dinâmica, sempre garante a solução ótima mas com um custo computacional elevado, além de um maior consumo de memória, por guardar múltiplos estados.

Em problemas maiores as duas soluções enfrentam problemas, a Heurística por conta das possíveis diferenças entre os resultados, e a programação dinâmica pelo alto consumo de memória e tempo de execução.

Em aplicações práticas a escolha do método deve levar em consideração:

Criticidade da solução: onde a solução tem que ser a ótima, a programação dinâmica é mais aconselhada.

Escala do problema: para problemas grandes onde a solução não precisa ser ótima, a heurística gulosa oferece melhor equilíbrio de tempo e qualidade.