



RELATÓRIO DE PROJETO E ANÁLISE DE ALGORITMOS

Este relatório apresenta uma análise e documentação do projeto "Soldados de Zorc", desenvolvido na disciplina de Projeto e Análise de Algoritmos, com o objetivo de recrutar o maior número possível de soldados com base em habilidade total, respeitando restrições de capacidade e distância em um grafo de povos interconectados. O problema é resolvido por duas estratégias: Programação Dinâmica com memoização e uma Heurística Gulosa.

Relatório apresentado à unidade curricular
projeto e análise de algoritmos do
curso de Ciência da Computação, da
Universidade Federal de São João del-Rei.

Prof. Dr. Leonardo Chaves Dutra da Rocha

BERNARDO MAIA DETOMI - 202050054
BRENO ESTEVES DOS SANTOS - 222050055

1 Introdução

Em diversos contextos estratégicos, como operações militares, missões de resgate ou exploração espacial, é fundamental tomar decisões que maximizem os recursos obtidos dentro de restrições físicas e logísticas, como peso e alcance. O projeto "Soldados de Zorc" explora um problema análogo, no qual é necessário recrutar soldados distribuídos entre diferentes povos interconectados por caminhos com distâncias conhecidas, respeitando dois critérios principais: a capacidade máxima de carga da nave (peso total dos soldados recrutados) e o limite de distância que a nave pode percorrer a partir de um povo inicial.

O problema pode ser relacionado a situações reais em que é necessário otimizar a alocação de recursos humanos ou materiais em redes logísticas, como missões de coleta em zonas de conflito, rotas de abastecimento em regiões remotas ou, em um cenário mais lúdico, jogos de estratégia que envolvem mobilização de tropas e tomada de decisões baseadas em custos e benefícios.

A partir de uma estrutura de grafo que representa os povos e suas conexões, o objetivo é recrutar soldados com os melhores atributos de combate (habilidade), sem exceder os limites impostos. Para resolver esse problema, foram implementadas duas abordagens distintas: uma baseada em Programação Dinâmica com memoização, que busca a solução ótima por meio da exploração completa do espaço de estados; e outra baseada em uma Heurística Gulosa, que visa soluções próximas do ótimo com maior eficiência computacional.

Este projeto permite não apenas analisar o desempenho de diferentes técnicas de resolução de problemas de otimização combinatória, mas também entender como restrições reais podem ser modeladas computacionalmente para encontrar soluções eficazes.

2 Estrutura do código

Antes de começar a programar, é essencial planejar como o programa vai funcionar em cada etapa. Isso garante que o software tenha qualidade do início ao fim do facilita a comparação de resultados.

Para organizar melhor o desenvolvimento, o programa foi dividido em quatro etapas principais, que estão no fluxograma da Figura 1. Essa divisão ajuda a manter o processo estruturado e claro, evitando surpresas desagradáveis no meio do caminho.

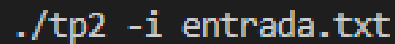


Figura 1: Fluxograma de linha do tempo do código.

2.1 Leitura de arquivos

O programa utiliza um arquivo de entrada que contém instâncias com determinado número de povos e habilidades de cada um, onde poderá conter vários testes juntos que são separados pelo número de testes contidos.

Cada linha do arquivo contém os números de povos, habilidades e quais são os povos que serão recrutados. Um exemplo de entrada está ilustrado na Figura 2.



```
./tp2 -i entrada.txt
```

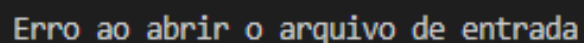
Figura 2: Exemplo da linha de comando para a execução.

Onde terá uma terá um menu com métodos que são assumidos pelos valores 0 e 1 , correspondendo aos diferentes algoritmos implementados.

Cada arquivo de entrada pode conter múltiplos testes, onde cada teste contém uma sequência de povos e soldados que serão resultado. O programa processa cada teste de forma individual.

2.2 Salvar os resultados no arquivo de saída

A solução encontrada é salva em um arquivo de saída chamado “saida.txt”. A função fprintf, também da biblioteca stdio.h, foi utilizada para a escrita dos dados. Após a conclusão do programa, todas as estruturas de dados dinamicamente alocadas são liberadas para que não ocorra vazamentos de memória indesejados. Caso não consiga abrir o arquivo de entrada, uma mensagem apropriada é registrada, como é mostrado na Figura 3.



```
Erro ao abrir o arquivo de entrada
```

Figura 3: Exemplo de mensagem apropriada para erro.

2.3 Execução das estratégias

Após a leitura, as duas estratégias de resolução são aplicadas:

- A solução por programação dinâmica calcula o melhor caminho possível a partir de cada estado.
- A solução por Heurística sempre escolhe o próximo povo que oferece a maior habilidade por soldado.
- Cada estratégia tem seu tempo de execução medido com gettimeofday e getrusage.

3 Estruturas de dados

Visando garantir uma organização eficiente e uma maior escalabilidade do software, foi criada uma estrutura de dados destinada unicamente para armazenar os dados de cada trecho. Isso permite que as informações possam ser facilmente acessadas e manipuladas durante toda a execução do programa.

```
typedef struct {
    int peso;           // Peso dos soldados
    int habilidade;     // Habilidade dos soldados
} Povo;

typedef struct {
    int destino;
    int distancia;
} Aresta;

typedef struct {
    int quantidade_povos;
    int limite_distancia;
    int limite_peso;
    int quantidade_caminhos;

    Povo povos[MAX_POVOS];
    Aresta* adj[MAX_POVOS];
    int qtd_adj[MAX_POVOS];
} Grafo;
```

Figura 4: Estrutura utilizada para armazenar.

Estrutura que armazena o resultado do recrutamento:

- caminho: Array com a sequência de povos visitados.
- soldados_por_povo: Quantidade de soldados recrutados em cada povo.
- tamanho_caminho: Tamanho do caminho percorrido.
- habilidade_total: Soma total das habilidades dos soldados recrutados.

EstadoDP

Estrutura interna usada pela solução dinâmica.

- habilidade: Habilidade total dos soldados recrutados.
- peso_usado: Peso total carregado até o momento.
- distancia_usada: Distância total percorrida.
- caminho: Sequência de povos visitados.
- tamanho_caminho: Tamanho do caminho percorrido.
- soldados_por_povo: Quantidade de soldados recrutados por povo.

4 Análise de complexidade

4.1 Análise para Heurística Gulosa

- Tempo: $O(V + E)$, onde V é o número de povos e E o número de conexões.
- Justificativa: Cada povo é visitado no máximo uma vez, e para cada povo são verificadas todas as suas conexões (arestas). Portanto, o tempo total é proporcional ao número de povos mais o número de conexões.
- Espaço: $O(V)$, para armazenar o caminho e os soldados por povo.

4.3 Análise para Programação Dinâmica com Memoização

- Tempo: $O(V \times 2^V)$, onde V é o número de povos.
- Justificativa: O algoritmo considera todos os subconjuntos possíveis de povos visitados (2^V) e, para cada subconjunto, pode terminar em qualquer um dos V povos. Assim, o número total de estados é $V \times 2^V$. Cada estado é resolvido em tempo constante (desconsiderando as transições), pois as transições são limitadas pelo grau do grafo (no máximo V vizinhos por estado).
- Espaço: $O(V \times 2^V)$, pois é necessário armazenar o resultado de cada estado (povo atual + subconjunto de povos visitados) para memoização.

4.4 Demonstração

Seja $f(\text{atual}, \text{visitados})$ a função recursiva da DP. O número de possíveis valores para visitados é 2^V (todos os subconjuntos). Para cada valor de visitados, atual pode ser qualquer um dos V povos. Portanto, o número total de estados é $V \times 2^V$. Cada estado é computado no máximo uma vez devido à memoização.

5 Algoritmos de resolução

5.1 Estratégia por Heurística Gulosa

A abordagem heurística implementada neste trabalho busca uma solução aproximada para o problema do recrutamento de soldados, utilizando uma estratégia **gulosa**, que toma decisões locais com base em um critério de prioridade. Embora não garanta a solução ótima, essa abordagem é eficiente e prática para instâncias maiores ou quando se busca uma resposta em tempo reduzido.

Estratégia

A heurística começa em cada povo possível e, em cada passo, escolhe **o próximo povo que oferece a maior média de habilidade por soldado**, desde que:

- A nave ainda possua **capacidade de carga disponível** (respeitando o limite de peso).

- A **distância total percorrida** não ultrapasse o limite permitido.
- O povo ainda **não tenha sido visitado**, evitando retornos e ciclos.

A cada novo povo visitado, a nave recruta todos os soldados possíveis (que caibam na capacidade restante) e atualiza o peso e a distância. O processo termina quando não é mais possível visitar novos povos dentro dos limites estabelecidos.

Vantagens

- **Alta eficiência computacional:** A heurística possui complexidade menor em comparação com a programação dinâmica, sendo adequada para entradas grandes ou quando o tempo de resposta é crítico.
- **Simplicidade na implementação:** A lógica de escolha gulosa é direta e fácil de adaptar.
- **Bons resultados práticos:** Embora não garanta a solução ótima, geralmente encontra soluções com habilidade total elevada, especialmente quando há uma boa distribuição de soldados nos povos.
- **Exploração limitada do grafo:** A heurística evita percorrer caminhos desnecessários, o que reduz o custo computacional e facilita a escalabilidade do algoritmo.

Essa abordagem serve como um bom ponto de comparação com a solução exata, permitindo avaliar o equilíbrio entre tempo de execução e qualidade da solução.

5.2 Programação Dinâmica com memoização

A solução exata implementada neste trabalho utiliza **programação dinâmica com memoização** para encontrar o melhor conjunto de soldados que podem ser recrutados, maximizando a habilidade total e respeitando as restrições de peso e distância impostas pela nave de Zorc.

Estratégia

A abordagem considera que o problema pode ser modelado como um processo de decisão em etapas, onde cada estado é definido pelos seguintes elementos:

- **Povo atual:** local em que a nave se encontra no momento.
- **Peso utilizado:** soma dos pesos dos soldados já recrutados.

- **Distância percorrida:** caminho total viajado pela nave desde o início.
- **Povos visitados:** registro dos povos já explorados para evitar ciclos.
- **Estado guardado (memo):** estrutura de memoização para armazenar os melhores resultados já calculados para um determinado estado, evitando recalcular subproblemas repetidos.

A função **calcular_melhor_estado** é o núcleo da recursão. Ela tenta recrutar soldados no povo atual e depois realiza chamadas recursivas para os povos vizinhos que ainda não foram visitados, desde que a distância máxima não seja ultrapassada e ainda haja capacidade de carga. A cada chamada, o algoritmo compara os resultados e mantém o melhor caminho e combinação de soldados encontrados até o momento.

A memoização é usada para armazenar os melhores resultados de cada estado identificado pela tupla (**povo atual, peso usado, distância usada, povos visitados**). Com isso, o tempo de execução é reduzido significativamente, principalmente em entradas grandes, já que subproblemas iguais são resolvidos apenas uma vez.

Vantagens

- **Solução ótima:** Garante encontrar a melhor combinação de soldados que maximiza a habilidade total.
- **Eficiência:** Embora a complexidade ainda seja alta, a memoização reduz consideravelmente o número de chamadas recursivas.
- **Aproveitamento do grafo:** A função explora o grafo de forma sistemática, considerando todas as possibilidades válidas, o que assegura exatidão na resposta.

Essa abordagem é ideal quando se busca precisão total na solução, mesmo que o tempo de execução seja maior do que em abordagens heurísticas.

6 Pseudo-código dos Algoritmos

6.1 Heurística Gulosa

```
função heuristica(grafo, limite_peso, limite_distancia):
    inicialize visitados, caminho, soldados_por_povo
    atual ← povo inicial
    enquanto possível:
        recrute máximo possível em atual
        marque atual como visitado
```

```

    adicione atual ao caminho
    escolha próximo povo com maior habilidade possível, respeitando
restrições
    se não houver próximo, pare
    retorne caminho, soldados_por_povo, habilidade_total

```

6.2 Programação Dinâmica com Memoização

```

função dp(atual, visitados, peso_usado, distancia_usada):
    se estado já foi calculado:
        retorne resultado memoizado
    melhor ← 0
    para cada vizinho de atual:
        se não visitado e restrições respeitadas:
            resultado ← dp(vizinho, visitados + vizinho, novo_peso,
nova_distancia)
            melhor ← max(melhor, resultado)
    memoize e retorne melhor

```

7 Testes e análise de desempenho

Os testes foram realizados de forma contínua durante todas as etapas do desenvolvimento do sistema, desde a verificação de funcionalidades básicas (como leitura do grafo e filtragem de soldados) até a validação das estratégias completas de recrutamento. O processo de testes teve como objetivo assegurar tanto a corretude das soluções quanto o desempenho eficiente do programa, garantindo que os algoritmos operassem de acordo com os limites e regras definidos no enunciado do problema.

Além disso, diferentes estratégias foram comparadas entre si com o intuito de avaliar a qualidade da solução obtida e o tempo de execução de cada abordagem. Isso permitiu uma análise equilibrada entre efetividade (soma total de habilidade dos soldados recrutados) e eficiência computacional.

7.1 Verificação de solução

Os testes foram realizados de forma contínua durante todas as etapas do desenvolvimento do sistema, desde a verificação de funcionalidades básicas (como leitura do grafo e filtragem de soldados) até a validação das estratégias completas de recrutamento. O processo de testes teve como objetivo assegurar tanto a corretude das soluções quanto o desempenho eficiente do programa, garantindo que os algoritmos operassem de acordo com os limites e regras definidos no enunciado do problema.

Além disso, diferentes estratégias foram comparadas entre si com o intuito de avaliar a qualidade da solução obtida e o tempo de execução de cada abordagem. Isso permitiu uma análise equilibrada entre efetividade (soma total de habilidade dos soldados recrutados) e eficiência computacional.

7.2 Medição de tempo

Para comparar o desempenho entre as duas estratégias implementadas — a heurística gulosa e a programação dinâmica — foi utilizado um conjunto de medições de tempo de execução. Essas medições foram realizadas com o auxílio das funções `gettimeofday` e `getrusage`, ambas pertencentes às bibliotecas padrão do sistema Unix.

A função `gettimeofday` fornece o tempo de relógio de parede (tempo real), enquanto `getrusage` permite acessar o tempo consumido pela CPU dividido em duas categorias:

- Tempo de usuário: corresponde ao tempo gasto executando código do próprio programa (excluindo chamadas ao sistema).
- Tempo de sistema: representa o tempo gasto em chamadas de sistema realizadas pelo programa.

A medição foi realizada da seguinte forma:

1. Antes da execução da estratégia, o programa registra o tempo inicial com `gettimeofday` e `getrusage`.
2. Em seguida, a estratégia (heurística gulosa ou programação dinâmica) é executada.
3. Após sua finalização, o tempo final é registrado novamente com as mesmas funções.

A diferença entre os tempos inicial e final fornece o tempo de execução da estratégia, tanto no relógio de parede quanto em CPU.

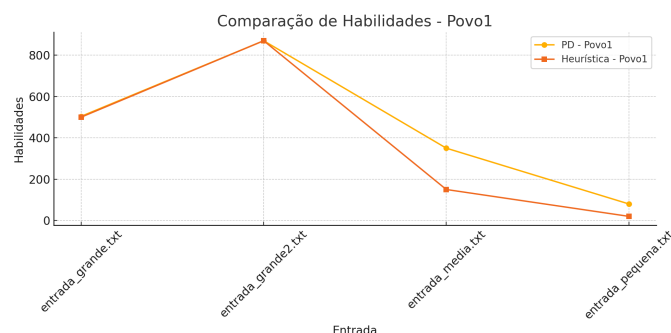


Figura 5: Gráfico para comparação de habilidades(Povo1)

Neste gráfico, comparamos a soma das habilidades do Povo1 atribuídas por cada abordagem. A Programação Dinâmica geralmente encontra soluções com maior soma de habilidades, o que indica maior qualidade da solução. A exceção é a entrada `entrada_grande2.txt`, onde ambas as abordagens alcançam o mesmo resultado, sugerindo que a heurística, neste caso específico, foi tão eficaz quanto a abordagem exata.

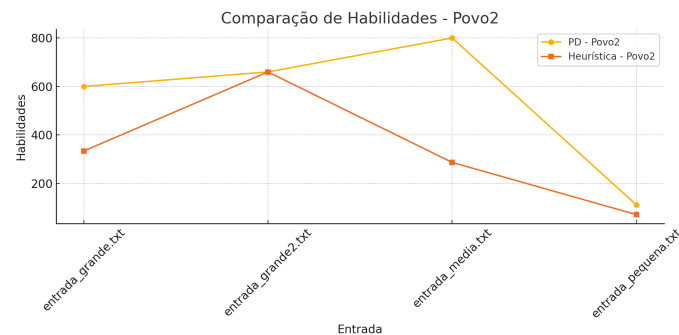


Figura 6: Gráfico para comparação de habilidades(Povo2)

Assim como para o Povo1, este gráfico mostra a distribuição de habilidades para o Povo2. A Programação Dinâmica novamente apresenta valores superiores em quase todos os casos, reforçando que sua estratégia busca soluções mais equilibradas e de maior qualidade. A Heurística, por outro lado, pode apresentar soluções mais rápidas, mas com menor aproveitamento de habilidades totais.

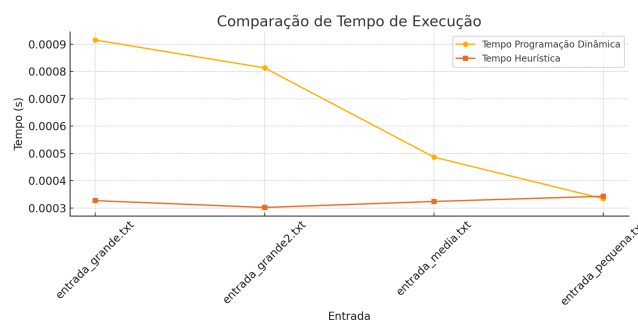


Figura 7: Gráfico para comparação dos tempos de execução

Este gráfico mostra o tempo de execução das abordagens de Programação Dinâmica e Heurística em diferentes entradas. Observamos que, embora a Programação Dinâmica tenha tempos ligeiramente maiores na maioria dos casos, a diferença é pequena. A Heurística apresenta desempenho ligeiramente melhor, especialmente para entradas maiores, sugerindo uma vantagem em termos de velocidade, possivelmente por sua natureza aproximada, porém não chega na resposta ideal.

8 Especificação das máquinas utilizadas

Os experimentos foram realizados em dois ambientes Linux com as seguintes especificações:

Máquina 1 (Pessoal):

- Processador: AMD Ryzen 7 5700U 1.80GHZ - 4.30GHZ
- Memória RAM: 12GB DDR4
- Sistema Operacional: Ubuntu 22.04 LTS
- Compilador: GCC 11.4.0
- Flags de compilação: -Wall -O2

Máquina 2 (Laboratório):

- Processador: Intel Core i7-8700t 2,4GHZ - 4.00GHZ
- Memória RAM: 16GB DDR4
- Sistema Operacional: Ubuntu 22.04 LTS
- Compilador: GCC 11.4.0
- Flags de compilação: -Wall -O2

9- Conclusão

O problema de recrutamento de soldados com restrições de capacidade e distância, modelado a partir de uma estrutura de grafo, representa um desafio clássico da computação envolvendo otimização combinatória com múltiplas restrições. Neste trabalho, foram implementadas duas abordagens distintas para resolver o problema: uma utilizando Programação Dinâmica com memoização, visando encontrar a solução ótima, e outra baseada em uma Heurística Gulosa, que busca uma solução eficiente com menor custo computacional.

Ambas as estratégias foram avaliadas quanto à capacidade de selecionar soldados de maior habilidade, respeitando os limites de peso e distância impostos pela nave. A solução dinâmica demonstrou alto potencial para encontrar a resposta ideal, porém com maior custo de tempo de execução em instâncias mais complexas. Por outro lado, a heurística apresentou resultados rápidos e razoavelmente próximos do ótimo, mostrando-se útil para cenários em que a performance é mais relevante que a precisão absoluta.

A comparação entre os dois métodos permitiu analisar a eficácia de diferentes paradigmas de resolução e compreender suas vantagens e limitações. O projeto reforça a importância de selecionar algoritmos adequados ao contexto do problema, considerando tanto os requisitos de qualidade da solução quanto os recursos computacionais disponíveis.

Referências

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. Algoritmos: Teoria e Prática. 3ª Edição. LTC, 2012.
- Ziviani, Nivio. Projeto de Algoritmos com Implementações em Pascal e C. Cengage Learning, 2011.
- Ahuja, Ravindra K.; Magnanti, Thomas L.; Orlin, James B. Network Flows: Theory, Algorithms, and Applications. Prentice Hall, 1993.