UNIVERSIDADE FEDERAL DE SÃO JOÃO DEL-REI

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

DOCUMENTAÇÃO TRABALHO PRÁTICO II

PROJETO E ANÁLISE DE ALGORITMOS

Discentes: Júlia Martins Silva, Ramon C.Paula

São João Del-Rei

Maio de 2025

Sumário

1. Introdução

Contextualização e Descrição do problema

Foi apresentado um mundo fictício onde um grande guerreiro viajara entre povos para fazer um recrutamento de suas forças. Existem caminhos entre algumas cidades que tem uma determinada distância. A espaçonave do grande guerreiro tem um limite do quanto pode voar entre os povos e um limite de peso que consegue carregar. Cada povo possui um tipo único de soldado disponível, mas para cara povo existirá soldados com peso P e habilidade H.

O problema consiste em recrutar o número máximo de soldados possível considerando o peso limite da espaçonave e a distância máxima que ela pode percorrer entre as cidades. Pode-se considerar que as cidades são vértices e os caminhos arestas com pesos.

Temos como entrada desse problema o número de instâncias(Quantas requisições serão feitas para o programa). Após isso é inserido respectivamente o número de povoados(P), distância máxima que a nave pode percorrer(D), peso máximo que a nave consegue carregar(W) e quantos caminhos existem(C). Nas próximas P linhas será descrito peso e habilidade de cada povo. Após isso terá mais C linhas onde o primeiro caractere dirá qual a cidade origem, o segundo a cidade destino e o terceiro a distância entre eles(Exemplo: 1 2 4).

Este documento detalha um programa em C desenvolvido para resolver o problema de otimização de recrutamento de soldados. Dado um conjunto de cidades, cada uma com soldados de um determinado peso (custo) e habilidade, e um mapa com as distâncias entre essas cidades, o objetivo do programa é determinar uma estratégia de recrutamento que maximize a habilidade total dos soldados recrutados. Essa estratégia está sujeita a duas restrições principais: a capacidade máxima de peso que uma nave de transporte pode carregar e a distância máxima total que a nave pode percorrer. O programa deve apresentar como saída a habilidade total obtida e a sequência de cidades visitadas, juntamente com o número de soldados recrutados em cada uma.

2 Estrutura do código

O código é projetado para processar múltiplas instâncias de problemas de recrutamento. Para cada instância, ele lê os dados de entrada, constrói uma representação do Mundo de Zorc, aplica um algoritmo de resolução para encontrar uma rota de recrutamento e, por fim, apresenta os resultados. O fluxo geral do programa, conforme executado pela função main, envolve:

1. Ler o número total de instâncias do problema.
2. Para cada instância:
    - Ler os parâmetros da instância: número de cidades, distância máxima, capacidade de peso da nave e número de caminhos.
    - Ler os dados das cidades (peso e habilidade dos soldados).
    - Construir o grafo que representa o Mundo de Zorc, incluindo as distâncias entre as cidades.
    - Aplicar o algoritmo de resolução (atualmente, uma heurística) para encontrar a melhor estratégia de recrutamento.
    - Imprimir a solução encontrada.
    - Liberar a memória alocada para a instância atual.
3. Repetir até que todas as instâncias sejam processadas.

Este fluxo permite a avaliação de diferentes cenários de recrutamento de forma sequencial.

2.1 Leitura de arquivos

O programa obtém todos os dados necessários de um arquivo de entrada padrão denominado entrada.txt. A estrutura deste arquivo é de extrema importância para o funcionamento do programa:

- **Número de Instâncias**: A primeira linha do arquivo deve conter um único inteiro, indicando quantas instâncias do problema de recrutamento serão processadas.
- **Parâmetros da Instância**: Para cada instância, a primeira linha subsequente contém quatro inteiros separados por espaços:

1. maxCidades: O número total de cidades existentes no Mundo de Zorc para esta instância.
2. maxDist: A distância máxima que a nave de recrutamento pode percorrer.
3. maxPeso: A capacidade máxima de peso que a nave pode transportar.
4. maxCaminhos: O número de conexões (caminhos diretos) entre pares de cidades.

- **Dados das Cidades**: Após a linha de parâmetros, seguem-se maxCidades linhas, cada uma descrevendo uma cidade. Cada linha contém três inteiros: idCidade (um identificador numérico, de 1 até maxCidades), Peso (o peso de um soldado recrutado nesta cidade) e Habilidade (a habilidade de um soldado desta cidade). A função lerCidades é responsável por processar esta parte do arquivo.
- **Dados dos Caminhos**: Após as descrições das cidades, seguem-se maxCaminhos linhas, cada uma detalhando um caminho bidirecional. Cada linha contém três inteiros: cidadeOrigem, cidadeDestino e pesoCaminho (a distância do caminho entre as duas cidades). A função lerMatAdj utiliza estas informações para popular a representação do mapa.

A saída do programa é direcionada para o console (saída padrão). Para cada instância, se uma solução viável for encontrada, o programa imprime a habilidadeTotal alcançada, seguida por uma sequência de pares idCidade e soldadosRecrutados, detalhando a rota e o recrutamento em cada cidade visitada. Se nenhum soldado puder ser recrutado, o programa imprime "0".

2.2 Criação do Mundo de Zorc através de um grafo

O Mundo de Zorc, com suas cidades e as distâncias entre elas, é representado internamente como um grafo. As cidades são os nós do grafo, e as distâncias são os pesos das arestas que conectam esses nós.

As principais estruturas de dados para esta representação são:

- **Cidade**: Define as propriedades de cada nó (cidade) no grafo:
    - int peso: Custo associado ao recrutamento de um soldado.
    - int habilidade: Valor de habilidade de um soldado.

    - int id: Identificador interno da cidade (geralmente o índice no array).
    - double razao: Um campo calculado (habilidade / peso), utilizado pela heurística de resolução.
- **MatAdj**: Representa o grafo (mapa do mundo) usando uma matriz de adjacência:

    - int **matAdj: Ponteiro para uma matriz alocada dinamicamente, onde matAdj[i][j] armazena a distância entre a cidade i e a cidade j. Um valor de 0 (após inicialização) ou um valor específico pode indicar ausência de caminho direto, dependendo da convenção usada após a leitura dos caminhos.
    - int rows_columns: A dimensão da matriz, correspondente ao número total de cidades (maxCidades).

As funções responsáveis pela criação e manipulação desta estrutura do grafo são:

- **criarMatAdj(int n)**: Aloca a memória necessária para a matriz de adjacência de tamanho n x n e a inicializa, geralmente com zeros, indicando que inicialmente não há caminhos conhecidos.
- **lerMatAdj(MatAdj *adjMatrix, FILE *arqEnt, int maxCaminhos)**: Popula a adjMatrix com as distâncias lidas do arquivo de entrada. Como os caminhos são bidirecionais, se existe um caminho de A para B com distância D, o caminho de B para A com a mesma distância D também é registrado na matriz.
- **freeMatAdj(MatAdj *freeMat)**: Libera a memória que foi dinamicamente alocada para a matriz de adjacência, evitando vazamentos de memória.

Esta representação em grafo permite que os algoritmos de resolução consultem eficientemente as distâncias entre as cidades para tomar decisões sobre as rotas de recrutamento.

2.3 Formas de resolução

Para fazer esse trabalho e solucionar o problema de otimização do recrutamento no Mundo de Zorc, utilizamos duas diferentes abordagens, um algoritmo utilizando programação dinâmica e um algoritmo utilizando uma heurística.

2.3.1 Programação Dinâmica

- Estruturas de dados utilizadas

O algoritmo faz uso de duas estruturas principais: uma matriz de adjacência que representa os caminhos e distâncias entre os povoados, e um vetor de estruturas do tipo City, onde cada entrada armazena o peso e a habilidade dos soldados disponíveis em um determinado povoado. Essas informações são fundamentais para determinar tanto o custo de se mover entre localidades quanto o ganho estratégico ao recrutar soldados.

- Parâmetros de entrada da função principal

A função recebe como entrada a matriz de adjacência já montada, o vetor de cidades com os dados dos soldados, o peso máximo que o guerreiro pode carregar e a distância máxima que a nave pode percorrer. Com esses dados, a ideia é explorar todas as rotas e combinações possíveis dentro desses limites, buscando o maior valor de habilidade recrutada.

- Criação da tabela de programação dinâmica (DP)

Uma matriz tridimensional é alocada para representar todos os estados possíveis: cada célula guarda a melhor habilidade alcançável ao se estar em determinada cidade, após percorrer certa distância e carregar um peso específico. Assim, a tabela funciona como um histórico otimizado de decisões parciais.

- Inicialização da tabela

A tabela é inicialmente considerada inacessível em todos os estados, exceto os que representam o início do percurso. Para cada cidade, o ponto de partida com distância e peso iguais a zero é configurado com habilidade igual a zero. Isso permite começar a simulação a partir de qualquer cidade, já considerando que o guerreiro pode iniciar em qualquer povoado.

Funcionamento do código usando Programação Dinâmica:

1. **Preenchimento da tabelaDinamica**

A tabela é preenchida de forma ordenada, sempre crescendo em termos de distância e peso, o que garante que todos os estados anteriores relevantes já tenham sido processados. Em cada ponto, o algoritmo avalia duas ações:

- Recrutar soldados, desde que ainda haja espaço de carga; para cada quantidade recrutada, calcula-se o novo peso e a nova habilidade. Se essa habilidade for melhor que a já existente para aquele estado, o valor é atualizado.

- Mover para uma cidade vizinha caso haja conexão direta e a distância adicional não ultrapasse o limite. Nessa transição, a habilidade é propagada, mas sem acréscimo — pois não houve recrutamento.

2. **Busca pela melhor habilidade**

Depois que toda a tabela foi preenchida, percorre-se todos os estados possíveis para encontrar aquele com o maior valor de habilidade. Esse valor será a solução final do problema, e os índices de cidade, distância e peso são salvos para possibilitar o rastreamento do caminho até esse estado ótimo.

3. **Reconstrução do caminho**

A reconstrução é feita a partir do melhor estado encontrado. Uma estrutura chamada caminhoReverso é usada para registrar os passos que levaram até ali. Primeiro verifica-se se houve recrutamento naquele ponto. Caso sim, ajusta-se o peso e registra-se o número de soldados recrutados. Caso contrário, tenta-se descobrir de qual cidade anterior veio o movimento, garantindo que o valor de habilidade permaneceu o mesmo. Isso indica que houve apenas movimentação, sem recrutamento. Esse processo continua até não ser mais possível voltar, formando o caminho completo até a solução.

4. **Resultado final**

Ao final, tem-se o valor máximo de habilidade possível respeitando os limites dados, além do caminho feito com todas as ações realizadas: em que cidades se recrutou, em quais se moveu e quanto peso foi carregado em cada passo. Esse trajeto completo é essencial para entender como a solução foi construída.

2.3.2 Heurística (Júlia)

O código implementado utiliza uma abordagem heurística gulosa, para encontrar uma solução para o problema de recrutamento de forma eficiente. Esta heurística não garante a solução ótima global, mas visa fornecer resultados de alta qualidade em tempo computacionalmente viável.

As estruturas de dados auxiliares específicas para esta heurística incluem:

- **infosRecrutamento**: Armazena o resultado do recrutamento em uma cidade específica:
    - int idCidade: O ID da cidade visitada.
    - int soldadosRecrutados: Quantos soldados foram recrutados nessa cidade.
- **Solucao**: Agrega todas as informações sobre uma rota de recrutamento candidata:
    - infosRecrutamento etapas[MAX_POVOS_DEFINIDOS]: Um array que registra cada etapa da rota de recrutamento.
    - int numEtapas: O número de cidades visitadas na rota.
    - int habilidadeTotal: A soma da habilidade de todos os soldados recrutados.
    - int pesoTotalAtual: O peso total acumulado dos soldados na nave.
    - int distanciaTotalPercorrida: A distância total percorrida pela nave.
- **Ordenacao**: Estrutura temporária usada para ordenar as cidades:
    - int idOriginal: O índice original (ID) da cidade.
    - double razaoCalculada: A razão habilidade / peso da cidade.

O algoritmo da heurística (recrutar) opera da seguinte maneira:

1. **Inicialização e Preparação**:
    - Uma estrutura Solucao é inicializada (usando inicializarSolucao) para armazenar a melhor rota encontrada.
    - Para cada cidade, calcula-se a razão habilidade / peso. Cidades com peso <= 0 são desconsideradas para este cálculo.
2. **Ordenação Gulosa**:
    - As cidades válidas são ordenadas em ordem decrescente com base na razão habilidade / peso calculada. A função compararRazao é usada para este propósito com qsort. Isso prioriza cidades que oferecem mais "habilidade por unidade de peso".
3. **Construção Iterativa da Rota**:
    - O algoritmo itera sobre a lista ordenada de cidades.
    - Para cada cidade candidata, ele verifica se a adição desta cidade à rota atual é viável, considerando:

        - **Capacidade da Nave**: O pesoTotalAtual não deve exceder capacidadeNave.
        - **Distância Máxima**: A distanciaTotalPercorrida mais a distância para alcançar a cidade candidata (obtida da MatAdj do Mundo de Zorc) não deve exceder maxDistancia. O caminho para a candidata deve existir (distância > 0, a menos que seja a primeira cidade).
        - **Visitas**: A cidade candidata não deve ter sido visitada anteriormente na rota atual.
    - **Recrutamento**: Se a cidade candidata puder ser visitada:
        - Calcula-se o número máximo de soldados que podem ser recrutados daquela cidade, limitado pelo pesoDisponivel na nave.
        - A solução (Solucao) é atualizada: distanciaTotalPercorrida, pesoTotalAtual, habilidadeTotal são incrementados, e os detalhes da etapa de recrutamento são registrados.
        - A cidade é marcada como visitada (para a rota atual), e ela se torna o ponto de partida para considerar a próxima cidade.
4. **Finalização**:
    - O processo continua até que nenhuma cidade adicional possa ser adicionada à rota (devido a restrições ou esgotamento de candidatos viáveis).
    - A Solucao final é retornada.

Esta abordagem tenta construir a melhor rota possível tomando decisões localmente ótimas (escolhendo a cidade com a melhor razão disponível) a cada passo.

3 Análise Matemática

3.1 Programação Dinâmica

Vamos agora analisar a programação dinâmica usada nessa função, seguindo a mesma lógica adotada na análise do algoritmo anterior.

1. **Inicialização da tabela:**

   O primeiro passo do algoritmo é criar uma estrutura tridimensional com tamanho n*(D+1)*(P+1) onde:

   - **n** é o número de cidades,
   - **D** representa a distância máxima que pode ser percorrida,
   - **P** é o limite de peso que pode ser transportado.

Essa estrutura é preenchida inicialmente com valores padrão usando três laços aninhados, o que leva a uma complexidade de: O(n*D*P)

2. **Preenchimento da tabela dinâmica:**

Esta é a parte mais trabalhosa do algoritmo. Para cada combinação possível de cidade, distância percorrida e peso carregado, o algoritmo realiza duas operações principais:

- **Recrutamento de soldados:** que pode envolver até **P** iterações no pior caso (por exemplo, se cada soldado pesar 1).
- **Movimentação entre cidades:** onde são verificadas as ligações com até **n** cidades vizinhas.

Como tudo isso acontece dentro de três laços externos (n, D e P), a complexidade dessa etapa é:
O(n*D*P*(P+n))

3. **Reconstrução do caminho percorrido:**

   Após preencher toda a tabela, o algoritmo reconstrói o melhor caminho a partir do estado final. Embora essa parte tenda a ser bem mais rápida na prática (porque só percorre os estados válidos encontrados), no pior caso ela pode também depender de todos os estados gerados: O(n*D*P)

Complexidade Final: O(n*D*P*(P+n)).

3.2 Heurística

A análise da heurística implementada (função recrutar e suas funções auxiliares) é a seguinte, considerando Nc como o número de cidades e Np como o número de caminhos:

1. **lerCidades(...):** Percorre as Nc cidades para ler seus dados. Complexidade: O(Nc).
2. **criarMatAdj(N_c):** Aloca e inicializa uma matriz Nc×Nc. Complexidade: O(Nc2).
3. **lerMatAdj(..., N_p):** Itera sobre os Np caminhos. Complexidade: O(Np).
4. Função **recrutar:**
    - Cálculo das razões: Um loop por Nc cidades. Complexidade: O(Nc).
    - Ordenação (qsort): Ordena Nc cidades. Complexidade média: O(NclogNc).
    - Loop principal de recrutamento: Itera no máximo Nc vezes (pelas cidades ordenadas). As operações internas (verificações, acesso à matriz de adjacência) são O(1). Complexidade: O(Nc).
    - Portanto, a complexidade dominante da função recrutar é O(NclogNc).

Considerando o processamento de uma única instância no main: A complexidade total é a soma das etapas: leitura, criação do grafo e execução da heurística. Tinstancia=O(Nc)+O(Nc2)+O(Np)+O(NclogNc). Simplificando, a complexidade por instância é O(Nc2+Np+NclogNc). O termo Nc2 (da criação da matriz de adjacência) é frequentemente significativo. Se o grafo for denso, Np pode ser da ordem de Nc2. Se for esparso, Np pode ser da ordem de Nc.

4 Conclusão

4.1 Programação Modular

Programação feita com foco na resolução do problema de forma eficiente e rápida, utilizando resultados já “conquistados”. Tendo uma análise matemática que demonstra alto nível de eficiência e que resolve o problema em um tempo relativamente baixo.

4.2 Heurística

Atualmente, uma abordagem heurística gulosa (desenvolvida por Júlia) está implementada. Esta heurística prioriza cidades com alta razão habilidade/peso e constrói iterativamente uma rota de recrutamento. Sua análise matemática indica uma complexidade polinomial (O(Nc2+Np+NclogNc) por instância), tornando-a viável para instâncias de tamanho considerável.