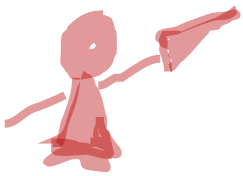

Editorial TP 01 (Trabalho Prático 01)

Algoritmos I

10 de junho de 2022

1 Introdução

Este documento é um compilado de informações associadas ao Trabalho Prático 1, e inclui informações sobre: O problema, a solução esperada e orientações para o próximo trabalho prático (TP02). Os casos de teste serão disponibilizados em um documento em separado.

1.1 Objetivos do trabalho

Este trabalho foi elaborado com o propósito de familiarizar os estudantes com as mecânicas da disciplina. Por esse motivo, foi proposto um problema com uma solução direta e extensa, de forma que a maioria dos erros (Em teoria) viria da documentação e, portanto, as penalidades sobre as notas não seriam tão pesadas. Pelo mesmo motivo, fatores como eficiência do código foram considerados secundários (Em particular, soluções de complexidade computacional mais alta que a necessária foram aceitas).

1.2 Pitfalls

Infelizmente, não sou muito familiar com a grade curricular da universidade, de forma que não esperava que tantos estudantes preferissem utilizar a abordagem orientada ao objeto, o que tornou muitas soluções demasiado extensas. De fato, não era a minha intenção passar um trabalho que demandasse 500+ linhas de código e, portanto, devo me desculpar pelo meu exagero.

2 Recapitulação do problema

O objetivo deste trabalho é alocar visitantes a bicicletas, satisfazendo alguns critérios. Considere um processo de alocação no qual n visitantes devem ser alocados a n bicicletas, distribuídas entre diversos pontos de controle. Ao acessar o aplicativo, visitantes fornecem uma lista de preferências que, nesse primeiro momento, contempla todas as bicicletas disponíveis. Preferências são dadas na forma de um número inteiro, de forma que, se uma bicicleta b_1 apresenta $pref(b_1) > pref(b_2)$ em relação a outra bicicleta b_2, dizemos que o visitante prefere b_1 a b_2. Um visitante não se importa em caminhar um pouco mais para obter uma bicicleta melhor (segundo suas preferências).

A condição essencial que deve ser satisfeita para uma alocação é que ela seja *justa*. Dizemos que uma alocação é justa se, quando alocamos uma pessoa p a uma bicicleta b_1, não houver uma bicicleta b_2 que, ao mesmo tempo, esteja mais próxima de p e que p considere preferível em relação a b_1. Em outras palavras, queremos respeitar as preferências dos usuários ao máximo possível, porém sem fazê-los andar em excesso para buscar sua bicicleta.

Para possibilitar os cálculos da distância de visitantes a bicicletas, o aplicativo dispõe de um mapa (desenvolvido por outro membro da equipe) que descreve a lagoa como uma grade NxM contendo pessoas, bicicletas e obstáculos que não podem ser atravessados. Suponha que sempre haja um caminho de qualquer pessoa a qualquer bicicleta. Uma vez que estas localizações estão em constante mudança, está também sob a sua responsabilidade calcular as distâncias entre os visitantes e cada bicicleta. Para simplificar os cálculos, suponha que uma pessoa pode mover-se apenas horizontalmente ou verticalmente.

3 Solução

O problema pode ser visto como um problema de pareamento em um grafo bipartido, onde nosso pareamento deve satisfazer as condições de uma alocação justa. A primeira coisa a se notar é que definição de uma alocação justa é bastante similar às condições de estabilidade no problema do Casamento Estável: Embora os critérios sejam diferentes, em ambos os casos, desejamos obter um pareamento que deve ser otimizado sob dois critérios diferentes, de forma que nenhuma mudança no pareamento gere melhorias em ambos os critérios ao mesmo tempo. Alternativamente, os estudantes podem chegar a este *insight* observando que a matéria não cobriu outros algoritmos de pareamento no momento da publicação do trabalho.

Para modelar o problema como uma instância de casamento estável, é necessário converter os critérios de otimização em listas de preferências. No caso dos visitantes, onde são dados "graus de preferências", basta utilizar um algoritmo de ordenação (Maior para menor) para converter as informações em uma lista de preferências ordenada.

Já no caso das bicicletas, a situação é um pouco mais complicada, pois não existe uma "lista de preferências"para bicicletas. Contudo, considerando que o segundo critério de otimização são as distâncias de cada bicicleta para cada visitante, podemos pensar de forma mais abstrata e dizer que a distância de uma bicicleta para um visitante é o seu "grau de preferência". Nesse caso, basta proceder da mesma forma que procedemos com os visitantes para obter uma "lista de preferências"ordenada para as bicicletas.

Para calcular a distância de um visitante para uma bicicleta, podemos visualizar o mapa da cidade como um grafo não-ponderado com NxM vértices e aproximadamente 4*(NxM) arestas, onde cada posição (i,j) é um vértice conectado aos vértices (i-1,j), (i+1,j), (i,j+1) e (i,j-1). Como bem sabemos, o caminho mais curto entre dois vértices em um grafo não-ponderado pode facilmente ser calculado a partir de uma BFS (Busca em largura).

Observação: Note que uma DFS (Busca em profundidade) *não* resolve o problema do caminho mais curto entre dois vértices. É interessante notar que qualquer outro algoritmo de caminho mínimo (Dijkstra, Bellman-Ford, etc.) também pode ser aplicado nesse passo, embora possuam um custo computacional pior (Por serem algoritmos de uso mais geral, aplicáveis também em grafos ponderados).

Com ambas as listas de preferências em mãos, basta utilizar o algoritmo de Gale-Shapley para encontrar um casamento estável entre os visitantes e as bicicletas. O casamento estável irá, por definição, gerar um pareamento livre de alocações injustas. Como descrito no enunciado do trabalho, o casamento deverá ser ótimo do ponto de vista dos visitantes, de forma que os visitantes deverão "se declarar"para as bicicletas.

4 Orientações para os próximos trabalhos

Tendo em vista as dificuldades observadas na correção do TP01, segue uma lista de orientações básicas para o Trabalho Prático 2.

- **Compilação**: O código deve incluir instruções de compilação, em substituição ao comando "Make". É recomendado que os estudantes testem seus códigos nas máquinas do DCC, "Jaguar"e "Tigre",

para evitar possíveis transtornos. Caso o código compile na sua máquina, mas não nas máquinas do laboratório, o trabalho poderá receber 0 no quesito de compilação.

- **Entrada**: Todos os trabalhos deverão ler os dados pela entrada padrão, e escrever na saída padrão. Isto é, devem ser utilizados os comandos "$scanf$", "$cin >>$", "$getline(cin)$", "$cout <<$, "$printf$", entre outros. Um arquivo pode ser transferido para a entrada padrão utilizando o operador '<', exemplo: "$./tp02.exe < teste1.txt$".

  - Trabalhos que lêem os dados a partir de qualquer fonte que não seja a entrada padrão receberão 0 no quesito de implementação. Isso inclui (mas não se limita a): Receber o arquivo de teste como argv[1]; Solicitar que o usuário digite o nome do arquivo de teste durante a execução do programa; Solicitar que o usuário *modifique o código-fonte para escrever o nome do arquivo de teste*. Em todos esses casos, o trabalho receberá nota 0 (Em implementação).

- **Qualidade de código**: Será avaliada de forma mais estrita durante os próximos trabalhos. O código deve ser estruturado de forma que os colegas de vocês (Ou seus futuros colegas de trabalho) possam abrir o código e, apenas lendo os comentários, possa entender o que o seu código está fazendo.

  - Comentários são importantes, mas muito texto nem sempre é um ponto positivo. Tente fazer comentários pontuais, destacando as principais funcionalidades do algoritmo (Onde está a BFS, onde está o Gale-Shapley) e detalhes de implementação mais importantes ("Este trecho transforma letras em IDs", por exemplo).
  - Tentem definir nomes significativos para suas variáveis e funções mais importantes. Evitem ao máximo utilizar variáveis como "aux", "a", "lwujrgh", etc., e funções como "solve()", "f()", etc. Nomes como "v-pref"ao invés de "lista-preferencias-visitantes"são aceitáveis.
  - Indentação é bastante importante para acompanhar o fluxo de execução do código de vocês. Não se esqueçam de tabular todos os "for"e "while"adequadamente.
  - Se forem utilizados muitos arquivos-fonte, cuidem para que eles sejam divididos em pastas diferentes para melhor navegação. Por exemplo, uma pasta "headers"para os arquivos .h e uma pasta "src"para os arquivos .cpp.
  - Se for utilizado um único arquivo-fonte, tentem dividir a função principal em partes significativas, mantendo o mínimo possível de código na função "main()".

    Note que um código do tipo "int main(){ solve(); return 0; }", onde "solve()"contém todo o restante do código, também não é aceitável. De forma geral, tentem manter todas as funções importantes com um número similar de linhas de código.
  - Recomendo realizar a leitura e impressão dos dados na função main() ou em uma função separada - nunca em uma das funções principais do código, como o Gale-Shapley ou a BFS.
  - Todos esses itens são apenas recomendações. A nota da qualidade de código será dada pela facilidade com a qual nós conseguimos navegar pelo código de vocês com base apenas nos comentários e organização do código.

- **Apresentação (Clareza)**: Um dos aspectos mais importantes da apresentação do trabalho.

  - Erros gramaticais não são muito importantes (mas tentem evitar excessos). Erros de sintaxe e semântica, por outro lado, são mais graves, pois podem prejudicar a clareza do texto.
  - Evitem frases avulsas. Se existe uma frase no seu texto que não se conecta aos argumentos principais do seu trabalho, você pode: a) Escrever um pouco mais de texto buscando conectá-la a esses argumentos; b) Removê-la do trabalho.

- **Apresentação (Organização)**: Embora não haja um modelo oficial para o trabalho, peço que os estudantes tenham um mínimo de senso de estética. Orientações comuns incluem:
  - Não misturar pseudocódigo com texto corrente;
  - Não misturar títulos com texto corrente;
  - Não exceder as margens do seu próprio documento;
  - Evitar pseudocódigos extensos demais;
  - Evitar fontes grandes (Arial 24) ou pequenas demais (Calibri 8) no texto corrente;
  - Evitar margens pequenas demais (Próximas demais da borda do documento);
- Em caso de confusão, vocês podem utilizar algum template como o modelo para artigos da SBC (https://www.overleaf.com/latex/templates/sbc-conferences-template/blbxwjwzdngr). Não é obrigatório, mas não tem como errar se você estiver usando um template oficial.
- Atentem para o limite de 5 páginas. O limite não inclui as instruções de compilação.
- **Modelagem**: A modelagem consiste em uma visão geral da sua solução em alto nível. Seu trabalho dizer *no que consiste* a sua solução, e *como* isso resolve o problema proposto. Utilize o capítulo 3 deste documento como referência para elaborar uma modelagem, se necessário.
- **Descrição da solução**: Após apresentar o overview da sua solução, é hora de entrar em detalhes. A BFS de vocês retorna a distância entre uma bicicleta e um visitante, ou entre uma bicicleta e todos os visitantes ao mesmo tempo (Flood-Fill)? O Gale-Shapley de vocês é ótimo para a bicicleta ou para o visitante? Quais estruturas de dados vocês estão utilizando? Se vocês usaram Dijkstra para obter o caminho mínimo, qual variante de Dijkstra vocês estão utilizando (Fila de prioridade, vetor, heap de Fibonacci...)? Se vocês usaram uma Tabela Hash, qual função de Hashing vocês utilizaram? Praticamente todo algoritmo possui detalhes de implementação. Este é o momento para vocês serem *bastante* específicos a respeito disso.
- **Análise de complexidade**: A análise de complexidade deve mostrar como o tamanho da entrada afeta o tempo de execução do código de vocês.
  - Peço clareza nas funções que vocês utilizam. Uma expressão $O(n^2)$ não significa nada até você definir quem é "n".
  - Em uma nota similar, $O(n * m)$ não é a mesma coisa que $O(n^2)$ (A menos que n = m). Tomem MUITO cuidado com as manipulações algébricas que vocês realizam.
  - *Justifiquem* as expressões apresentadas. Sua justificativa conta para a sua nota no quesito de análise de complexidade.
  - Para algoritmos já conhecidos, vocês podem utilizar a referência da literatura como justificativa. Porém, recomendo que verifiquem se o código de vocês não executa passos adicionais que possam modificar essa complexidade.
  - Sua análise deve apresentar a complexidade total do seu programa, não apenas de suas funções principais. Note que a complexidade total do programa é simplesmente a soma das complexidades de suas funções principais. No TP01, por exemplo, essa complexidade é dada pela soma dos custos de (pre-processamento() + calcular-preferencias-bicicletas() + calcular-preferencias-visitantes + gale-shapley()).
  - Lembrem, o custo de executar uma operação O(n) m vezes geralmente é O(m*n) (Exceções existem, mas vocês não precisam se preocupar com elas nesse semestre).

Algumas dicas adicionais para o TP02:

- Diferente do TP01, a relação entre o problema e o modelo não é tão trivial. Vocês podem escolher entre utilizar argumentos empíricos, provas por indução/contradição ou qualquer outro método que julgarem adequado para demonstrar que a solução de vocês de fato resolve o problema proposto. Irei pegar um pouco leve na hora da correção, pois não estamos fazendo Projeto e Análise de Algoritmos ou Matemática Discreta, mas vocês devem achar um jeito de convencer o leitor de que a solução de vocês *funciona*.

  Inclusive, a argumentação de vocês será uma excelente ferramenta de detecção de plágio na parte da documentação .

- Os seguintes erros resultarão em morte súbita (para o trabalho, não para o estudante), i.e. nota 0 nos quesitos relevantes:

  - Código e/ou documentação plagiados;
  - Documentação com mais de 5 páginas (Excluindo capa e instruções para compilação);
  - Leitura de entrada incorreta;
  - Ausência de instruções para compilação;