Laboratório: Placas de Circuito Impresso

MC458 - Projeto e Análise de Algoritmos I (1s2019)

Universidade Estadual de Campinas (Unicamp)

1 Introdução

As placas de circuito impresso (PCI) possuem grandes vantagens quando comparadas com os modos antigos de se desenvolver equipamentos eletrônicos. No passado, todos componentes dentro de um dispositivo eletrônico deveriam ser conectado por fios. Desse modo, o interior desses equipamentos não só ocupavam muito espaço como eram muito bagunçados! As PCI's utilizam uma abordagem diferente: os componentes são montados em placas não condutoras, geralmente de fibra de vidro, e conectados por finos "traços" de cobre (que são condutores, e assim, simulam os fios).

Durante a fabricação de PCI's é necessário não só "imprimir" os traços de cobre sobre a placa, como também perfurar a placa, para que os componentes possam ser encaixados posteriormente. Entretanto, além de fibra de vidro, as placas também utilizam resina de epóxi - material de dureza semelhante a do granito. Portanto, é necessário ferramentas especiais para fazer a perfuração das PCI's.

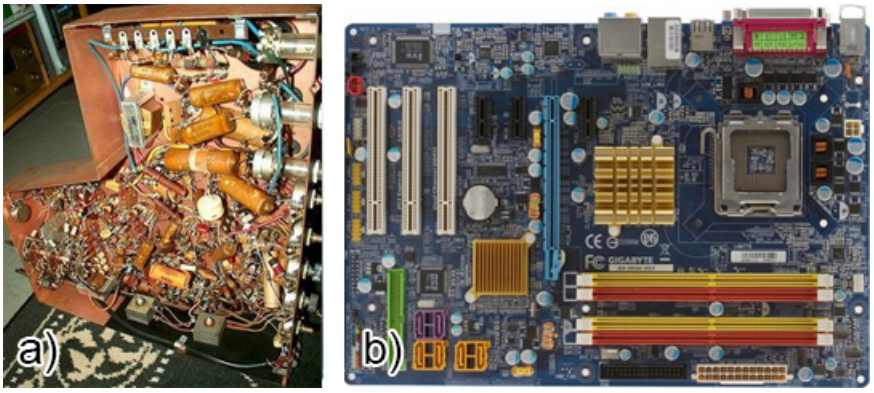

Figura 1: a): interior de uma televisão de 1948; b): uma placa mãe de 2007. Imagine como seria se tivéssemos que ter fios para cada "traço" da placa mãe!

Suponha que exista uma fábrica de PCI's que não consegue satisfazer a demanda, pois a máquina que cria os "buracos" na placa é um pouco antiga, e não considera a ordem em que os pontos devem ser perfurados, demorando muito mais tempo que o necessário para realizar a tarefa. Você foi contratado por essa

fábrica para otimizar esse processo, desenvolvendo um algoritmo que encontra a melhor ordem para perfurar os pontos.

Seja $p_1, \ldots, p_{n-2}$ os pontos que devem ser perfurados na placa. Além disso, por algum motivo, essa máquina antiga deve sempre começar sua broca em algum ponto p_0 e terminá-la em outro ponto p_{n-1}. Podemos assumir que cada ponto p_i é descrito por um par de coordenadas $(x_i, y_i) \in \mathbb{R}^2$.

A distância entre dois pontos $p_i = (x_i, y_i)$ e $p_j = (x_j, y_j)$ é dada pela distância euclidiana, isto é:

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}.$$

Dessa forma, seu objetivo é encontrar uma permutação $p'_1, \ldots, p'_{n-2}$ dos vértices $p_1, \ldots, p_{n-2}$ que minimize a distância total D percorrida pela broca da máquina:

$$D = d(p_0, p'_1) + d(p'_1, p'_2) + \ldots + d(p'_{n-3}, p'_{n-2}) + d(p'_{n-2}, p_{n-1}).$$

Também diremos que, se $p'_i = p_j$, então $\pi(i) = j$. Dessa forma, poderíamos reescrever a distância total como:

$$D = d(p_0, p_{\pi(1)}) + d(p_{\pi(1)}, p_{\pi(2)}) + \ldots + d(p_{\pi(n-3)}, p_{\pi(n-2)}) + d(p_{\pi(n-2)}, p_{n-1}).$$

2 Algoritmos

Nesse laboratório você deve implementar dois algoritmos baseados na técnica de programação dinâmica. O algoritmo **A** deverá usar a abordagem "bottom-up" para resolver o problema proposto, enquanto o algoritmo **B**, deverá usar a abordagem "top-down".

Lembre-se que, na abordagem "bottom-up" nós geralmente utilizamos uma **tabela** e construímos ela a partir do caso base. Por outro lado, a abordagem "top-down" é muito semelhante a um algoritmo recursivo de força-bruta, mas para evitar a computação de subproblemas repetidos, nós utilizamos a técnica de **memoização** (para saber mais: https://www.codesdope.com/course/algorithms-dynamic-programming/).

Note que, se por um lado a abordagem "top-down" pode consumir mais tempo e memória, devido a pilha das funções recursivas, ela também pode evitar calcular subproblemas que não são necessários para se construir a solução ótima. A abordagem "bottom-up", por sua vez, sempre computa todos os subproblemas menores, mas não tem a necessidade de utilizar funções recursivas.

Dica: na sua implementação procure usar uma representação compacta para representar um conjunto de pontos (por exemplo, um bitset https://www.geeksforgeeks.org/c-bitset-and-its-application/), pois os algoritmos desse laboratório podem utilizar muita memória.

3 Implementação

Esse laboratório será feito em C++. A fim de facilitar a implementação, fornecemos nesse laboratório 7 arquivos:

- *Makefile*: é o makefile do projeto.
- *point.h*: define a estrutura de um ponto (dois números em representação de ponto flutuante).
- *instance.h*: define a estrutura de uma instância do problema (um vetor de pontos, o valor de n e o nome da instância).
- *solver.cpp, solver.h*: contém os algoritmos que resolvem o problema.
- *main.cpp, main.h*: definem funções que leem os argumentos da linha de comando, realiza a leitura da instância, chama os algoritmos implementados em *solver.cpp*, computa o tempo de execução e, por fim, cria uma representação gráfica da solução encontrada.

Os únicos arquivos que você deverá alterar são os arquivos *solver.cpp* e *solver.h*. Nesses arquivos você deve implementar os algoritmos **A** e **B** nas funções *solveBottomUp* e *solveTopDown*, respectivamente. Essas funções recebem de entrada uma instância (definida em *instance.h*) e retorna um vetor de inteiros *sol*, onde $sol[i] = \pi(i + 1)$, isto é, o vetor *sol* indica a ordem em que os pontos $p_1, \ldots, p_{n-2}$ devem ser perfurados. Note que nesses arquivos já está presente uma função *getDistance* que computa a distância euclidiana entre dois pontos. Você **deve** utilizar essa função para computar a distância entre dois pontos nos seus algoritmos.

As funções *solveBottomUp* e *solveTopDown* também recebem de parâmetro o tempo em que o programa começou a executar e o tempo limite. Você deve utilizar esses parâmetros para que o seu programa não ultrapasse muito o tempo limite (ver o arquivo *solver.cpp* para ter um exemplo).

4 Compilação e Execução

Para o programa compilar basta digitar *make* na raiz do projeto. Para executar o programa você deve digitar *./solver* com as seguintes *flags*:

- *-i nome_do_arquivo*: onde *nome_do_arquivo* deve ser o arquivo de entrada que representa a instância a ser resolvida pelo programa
- *-a algoritmo*: onde *algoritmo* é uma string que pode ser **A** ou **B** e denota o algoritmo que deve ser utilizado para resolver o problema.
- *-g*: essa flag é opcional, caso você coloque ela, seu programa irá gerar um arquivo *out.pdf* que contém uma visualização da solução encontrada pelo seu algoritmo. Para que a visualização funcione você deve ter o graphviz (https://www.graphviz.org/download/) instalado em sua máquina.

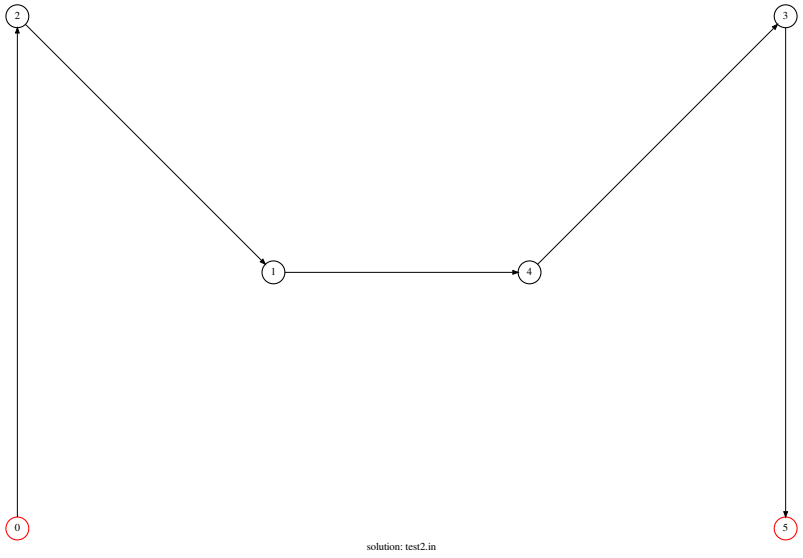

Figura 2: Exemplo de visualização de uma instância resolvida.

- *-t tempo*: essa flag é opcional, caso você coloque ela, seu programa deverá ter um tempo limite de *tempo* para executar. Caso você não coloque essa flag, o tempo limite será de 60 segundos.

Por exemplo, se digitarmos
./solver -i test.in -a A -g
iremos executar o algoritmo de programação dinâmica "Bottom-Up" para resolver a instância *test.in* com o tempo limite de 60 segundos, além disso, o programa também vai gerar uma visualização no arquivo *out.pdf*.

Obs.: caso precise utilizar um debugger, você também pode compilar o código com o comando *make debug*.

5 Relatório

Além dos códigos você também deve submeter um relatório que explique brevemente o seu algoritmo. O seu relatório também deve conter uma seção de **Resultados Computacionais** que inclui tabelas com informações de:

1. nome da instância,
2. qual algoritmo executou (A ou B),
3. valor da solução obtida dentro do tempo limite,
4. tempo limite,
5. tempo de execução.

Por fim, você também pode adicionar qualquer informação que acredite ser relevante para a correção do seu programa.

6 Avaliação

Os arquivos (*solver.cpp*, *solver.h* e *relatorio.pdf*), deverão ser submetidos na plataforma Susy. Desconsiderando o bônus, a nota máxima desse laboratório é 10. Assim, os algoritmos **A** e **B** valem 5 pontos cada. Para ganhar os 5 pontos o seu código deve executar corretamente para as instâncias da plataforma Susy.

Além desses pontos, também teremos um **bônus** para os melhores algoritmos. Esse bônus será considerado somente para os programas que passarem em todos os testes do Susy, tanto do algoritmo **A** quanto do algoritmo **B**. Nós iremos avaliar esses programas em algumas instâncias fechadas (e mais difíceis). Os 5 programas que apresentarem os melhores desempenho nessas instâncias ganharão um bônus no laboratório. O 1º, 2º, 3º, 4º e 5º programa ganharão, respectivamente, 2.5, 2.0, 1.5, 1.0, 0.5 de bônus.