Universidade Federal de Minas Gerais - Instituto de Ciências Exatas
Departamento de Ciência da Computação
AEDS III (DCC005) - 2017/2

Trabalho Prático 2 – Uaibucks

Nome: Marcel Henrique da Silva Mendes

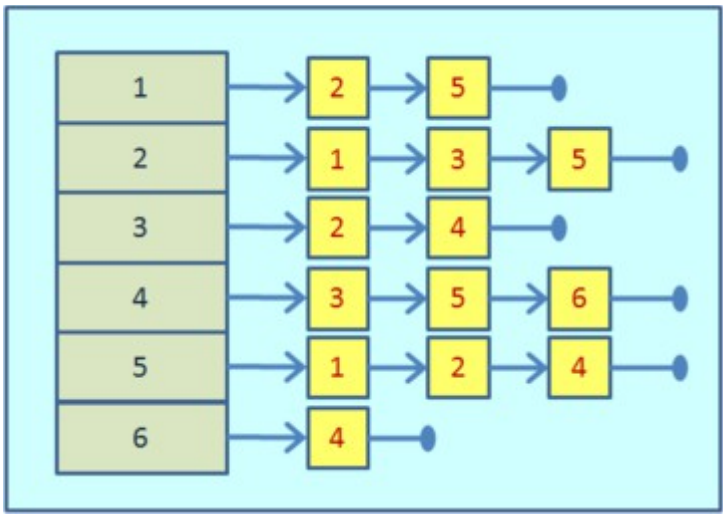
I. Introdução

O tema do trabalho proposto basicamente inclui achar localidades adequadas para a instalação de novas cafeterias uaibucks, de modo que não existam cafeterias nas duas esquinas de uma mesma rua e o número da demanda de clientes seja o maior possível. Foi percebido que esse problema pode ser modelado usando Grafos, e a resolução dele se resume a encontrar a solução de conjuntos independentes de vértices em um grafo não direcionado. De acordo com o enunciado o problema de encontrar as localidades adequadas para a instalação das filiais é NP–Difícil e o problema de decisão associado a ele pertence a classe dos NP-Completo, ao longo da documentação essa prova será feita. Por esse motivo foi pedido a criação de uma heurística para resolver o problema, uma heurística como já é conhecido nem sempre nos dá uma solução exata, mas na maioria das vezes retorna uma boa solução, uma solução aproximada que ajuda a resolver o problema. Para este trabalho foi adotado uma estratégia gulosa para a resolução da heurística, essa estratégia será explicada mais adiante. Para encontrar as respostas exatas foi feito um teste exaustivo de todas as possibilidades possíveis.

II. Decisões de projeto

Algumas etapas foram importantes para a implementação do trabalho, são elas:

Estrutura de dados utilizada: Para a implementação desse projeto a estrutura de dados escolhida foi lista de adjacência, um vetor de listas onde cada lista representa um vértice.



(a)Lista de adjacência

Heurística utilizada para encontrar uma solução: A heurística feita no trabalho prático usa uma estratégia gulosa, basicamente, escolhe o vértice de maior demanda e insere no conjunto solução depois escolhe o segundo vértice de maior demanda, verifica se o vértice é vizinho do vértice inserido, se sim ignora, se não ele insere no conjunto solução. Ele repete esse passo de inserção e verificação até todos os vértices serem analisados.

Foi criado um vetor auxiliar dAux que guarda os vértices e suas respectivas demandas, para evitar o acesso constante ao grafo e facilitar a ordenação das demandas. A ordenação das demandas é feita pelo algoritmo shellSort, foi escolhido por ser de fácil implementação e ter um tempo de execução razoavelmente bom.

Algoritmo para calculo exato: O algoritmo exato foi implementado usando força bruta, ou seja, é testado todas as maneiras de se formar um conjunto de vértices. A cada conjunto de vértice candidato a solução, é verificado se entre eles existem vértices vizinhos se não existir ele se confirma como um candidato a solução, logo depois é feito a soma das demandas e comparado com a variável max que guarda a maior soma de demandas, se for maior o candidato é atualizado como a melhor solução temporariamente, até que todas as possibilidades de soluções tenham sido verificadas. Para fazer essa força bruta foi criado uma função chamada de conv, que será explicada com mais detalhes a frente, onde converte números naturais em números binários, que são colocados em um vetor de inteiros chamado vSol de tamanho n, onde n é tamanho da entrada, é feita uma iteração do numero 1 até 2^n toda vez que o índice i do vetor vSol for igual a 1 o vértice i será um candidato a solução. Exemplo seja n igual a 8 o vetor vSol após 10 iterações se encontrará da seguinte forma [0,0,0,0,1,0,1,0] ou seja os vértices que serão avaliados ,verificados se possuem ou não vizinhos entre si, serão os vértices [5,6].A solução exata é guardada no vetor solFinal que contem os vértices e suas respectivas demandas.

Funções e procedimentos:

void FLvazia(TipoLista *lista): Esse procedimento cria uma lista vazia.
int Vazia(TipoLista *lista): Função que verifica se a lista está vazia ou não.
void Insere(TItem x,TipoLista *lista): Insere um elemento na lista, ou seja, um vizinho em um determinado vértice.
void insereGrafo(TipoLista *lista, int v): Insere uma aresta no grafo, usa a função Insere para inserir a demanda e o vértice em uma determinada lista.
void insereHead(TipoLista *lista,int p) :Insere a demanda na célula cabeça de cada lista(vértice).
int verifVizinho(TipoLista lista,TItem x): Verifica se existe vizinho entre x.vertice e a lista que representa determinado vértice.
void escolhe(TItem dAux,TItem *cSol,TipoLista lista): A função escolhe se o elemento que está no vetor dAux irá fazer parte da solução ou não, se sim ele é adicionado ao vetor cSol.
void conv(int k,int *buf,int n): Converte um número inteiro em binário e o coloca em um vetor, essa função é utilizada para ser possível considerar todas as combinações de vértices existentes. A cada iteração ela é chamada e o número inteiro é convertido, índices que tiverem 1 em seu conteúdo serão os vértices avaliados.
int escolheExato(TItem *cSol,TipoLista *lista,int n): Verifica se um determinado conjunto candidato a solução é um conjunto independente, se sim o valor da demanda é retornado e depois comparado para verificar se ele corresponde a demanda máxima.

void Ordena(TItem *t,int size): Procedimento que ordena as demandas em ordem crescente, é utilizado na implementação o método de ordenação shellsort.

III. Complexidade

Funções e procedimentos:

void FLvazia(TipoLista *lista): O(1),realiza somente uma operação de atribuição.
int Vazia(TipoLista *lista): O(1),Verifica se lista primeiro é igual a lista último
void Insere(TItem x,TipoLista *lista): O(1), sempre Insere o elemento no começo da lista.
void insereGrafo(TipoLista *lista, int v): O(1), usa a função Insere para inserir um vértice e uma demanda no inicio da lista.
void insereHead(TipoLista *lista,int p): O(1),Insere um elemento na célula cabeça da lista.
int verifVizinho(TipoLista lista,TItem x): O(m), onde m é a quantidade de vizinhos que um determinado vértice possui, no pior caso quando um grafo é completo,O(m) = O(n).
void escolhe(TItem dAux,TItem *cSol,TipoLista lista): $O(n^2)$,apesar da complexidade depender de quantos elementos temos na no conjunto solução ela tem limite superior quadrático.
void conv(int k,int *buf,int n): O(n), o procedimento conv, converte um número inteiro em um binário essa conversão depende do tamanho da entrada n.
int escolheExato(TItem *cSol,TipoLista *lista,int n): $O(n^2)$, Essa função verifica se existe vizinhos em um conjunto candidato a solução, possui duas iterações uma que depende da entrada e outra dependente dos elementos que estão no conjunto solução, mas seu limite superior é quadrático.

IV. Funçoes Principais:

heurística.c

- A função principal do algoritmo de heurística começa com a leitura das variáveis de número de vértices e quantidade de arestas, são feitas as alocações dinâmicas das variáveis dAux, que será utilizada para guardar demandas e os vértices inseridos, e cSol que é a variável que guarda o conjunto solução, ou seja, a variável que contêm o conjunto solução.

- É feita a inicialização do vetor de listas, ou seja é criado uma célula cabeça para cada um.

- É inserido as demandas e os vértices no grafo e no vetor auxiliar dAux.

- A inserção das arestas é feita usando a função insereGrafo.

- Logo depois o vetor dAux é ordenado em ordem crescente, pela função ordena .

- Então é feita a escolha de quais vértices serão inseridos no conjunto solução pela função escolhe, de acordo com os critérios já explicados.

- Finalmente é feita a impressão do resultado.

A complexidade da main é definida pela complexidade do loop que contém a função escolhe apesar de na prática o tempo execução ser bem menor, o limite superior para essa complexidade é $O(n^3)$.A complexidade de espaço O(|V| + |A|) que é exatamente a complexidade de se guardar o grafo (lista de adjacência) onde |V| é a quantidade de vértices e |A| as arestas desse grafo.

exato.c

- A função principal do algoritmo que calcula o resultado exato começa com a leitura das duas variáveis n e m, número de vértices e de arestas.

- Logo depois são alocadas as variáveis dAux, cSol, vSol e solFinal, note que a função da variável cSol não é a mesma do algoritmo de heurística agora ela guarda somente candidatos a solução e não mais a solução final.

- O grafo é alocado e inicializado, depois inserido as demandas no grafo e os vizinhos de cada vértice.

- É feito um loop que vai de 1 até 2^n que tem como função considerar todas as possibilidades de conjunto de vértices possível.

- Dentro do loop é chamada a função escolhe exato que calcula a soma das demandas dos candidatos a solução, logo após é feito a comparação onde se determina se o conjunto de solução que acabou de ser calculado é o melhor escolhido até o momento.

A complexidade da main é definida pelo loop que é feito, a partir disso podemos observar que a complexidade total do programa tem limite superior igual a $O(2^n)$.A complexidade de espaço é a mesma da heurística O(V| + |A|) definida pelo tamanho do grafo.

Prova que conjunto independente é NP-completo:

Considere π^1 o problema clique e π^2 o problema conjunto independente de vértices. A instância I de clique consiste de um grafo G = (V;A) e um inteiro k > 0. A instância f(I) de conjunto independente pode ser obtida considerando-se o grafo complementar ^G de G e o mesmo inteiro k. f(I) é uma transformação polinomial:

1. ^G pode ser obtido a partir de G em tempo polinomial.

2. G possui clique de tamanho ≥ k se e somente se ^G possui conjunto independente de vértices de tamanho ≥ k.

Se existe um algoritmo que resolve o conjunto independente em tempo polinomial, ele pode ser utilizado para resolver clique também em tempo polinomial.

Diz-se que clique α conjunto independente.

Como mostrado no artigo de Richard M. Karp de 1972 “Reducibility among combinatorial problems” o problema de achar uma clique em um grafo é NP-Completo. De acordo com o teorema de cook mostrando que podemos reduzir solucionar a clique em solucionar o problema de conjunto independente de um grafo, mostramos que conjunto independente também é NP-Completo.

v. TESTES

Execução: Foi feito dois makefiles e a execução é feita de maneira separada. ./heuristica <entrada.in> <saida.out>
./exato <entrada.in> <saida.out>
Entrada e saída padrão (stdin/stdout).

Resultados obtidos

(a) Tabela 1

Demandas obtidas exato	*Demandas obtidas heurística*	
118	100	Teste 1
168	121	Teste 2
170	170	Teste 3
10	10	Teste 4
304	304	Teste 5
139	82	Teste 6
30	25	Teste 7
265	261	Teste 8
312	312	Teste 9
198	87	Teste 10
80	80	Teste 11
10	10	Teste 12
127	127	Teste 13
7	7	Teste 14
80	62	Teste 15

Os testes acima realizado se encontram na pasta testes,que está no arquivo de entrega do trabalho.Foi adicionado mais alguns testes aos que foram passados pelos monitores da disciplina.

A taxa de acerto da heurística para estes testes em específico foi de aproximadamente 46%.A maior variação de resultado foi de 111 unidades de demanda.Foi percebido em relação a heurística que quando existe grafos que contém um vértice de grande demanda conectado com muitos outros vértices o calculo feito, muitas vezes fica longe do esperado.

VI. Conclusão

A implementação do trabalho foi realizado sem grandes problemas e os resultados ficaram dentro do esperado. O maior desafio encontrado foi a decisão de qual heurística escolher,a escolha desse algoritmo teve como objetivo aproveitar ao máximo os vértices de maior demanda,considerando um bairro ou cidade por exemplo sempre haverá avenidas ou ruas que possuem muita demanda,creio que esse algoritmo aplicado em situações reais teria boa qualidade de respostas.