Instituto Federal de Educação, Ciência e Tecnologia de São Paulo

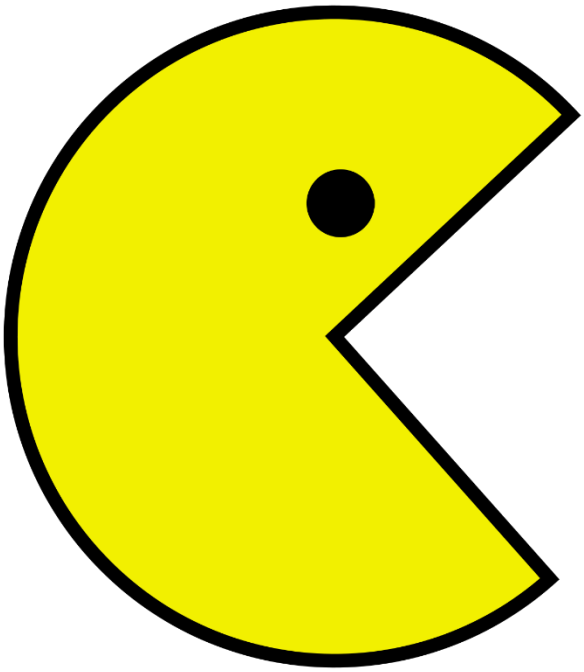
Trabalho Análise de Algoritmos

Docente: Bento Rafael Siqueira

Discentes:

Rodrigo Luis Tavano Bosso – PC3005623

Algoritmos Gulosos



0-Resumo:

O seguinte trabalho tem como objetivo explicar o funcionamento de Algoritmos Gulosos (Greedy Algorithm) passando por seus conceitos, análise de desempenho e apresentação de alguns exemplos.

Para que se possa realizar tal discussão, estaremos fundamentados em uma série de pesquisas em links e livros que explicam o funcionamento geral dos Algoritmos Gulosos. Consoante a tais fatores partiremos para o texto propriamente dito.

Palavras-chave: Algoritmos, Gulosos, Heurística.

1- Introdução:

1.1- O que são algoritmos gulosos?

Algoritmo ganancioso(gulosos) é qualquer algoritmo que segue uma solução de problemas de maneira heurista , em que ocorre a escolha de um local ideal em cada estágio.
Em muitos problemas, uma estratégia gananciosa não produza uma solução ideal, mas uma heurística gananciosa pode produzir soluções locais ideias que se aproximam de uma solução global ideal em um período de tempo razoável.

“Um algoritmo guloso nunca se arrepende, não desfaz escolhas já feitas” uma vez que ele não toma decisões com base em execuções passadas e nem futuras apenas com as execuções diretas e momentâneas, também por isso é chamado de algoritmo míope.

2- Conceitos Básicos:

2.1- Vantagens:

Algoritmos gananciosos(gulosos) tendem a ser de simples implementação. Pela diminuta complexidade de etapas.

Além disso algoritmos gulosos também tendem a ser bastante eficientes, sendo uma boa opção para se resolver problemas.

2.2 - Desvantagens:

Apesar de tenderem a uma ótima solução, nem sempre isso se prova uma verdade, assim não levando a uma solução ótima global.

Difícil realizar a prova de melhor desempenho com relação aos ótimos locais e desempenhos.

2.3 - Componentes:

Em geral o algoritmo guloso tem cinco componentes :

- Um conjunto candidato, a partir do qual é criada uma solução.
- Uma função de seleção, que seleciona o melhor candidato para ser adicionado à solução.
- Uma função de viabilidade, que é usada para determinar se um candidato pode ser utilizado para contribuir para uma solução.
- Uma função objetivo, que atribui um valor a uma solução, ou uma solução parcial.
- Uma função de solução, que irá indicar quando tivermos descoberto uma solução completa.

3- Desempenho:

"A avaliação de qualidade da solução obtida por um algoritmo é feita através da comparação com um limite inferior para a solução do problema, o que pode ser expresso pela seguinte fórmula: $\frac{x^2(I)}{x^*(I)}$, onde $x^h(I)$ é o valor da solução heurística (aproximada) e $x^*(I)$ é o valor da solução ótima."

Em outras palavras sabemos que o desempenho está relacionado com a ordem relacional conhecida como divisão e conquista, em que temos os seguintes desempenhos já predeterminados.

4- Exemplos:

4.1- Problema de seleção de atividade:

O problema de seleção de atividades é um problema de otimização combinatória referente à seleção de atividades não conflitantes a serem executadas dentro de um determinado período, dado um conjunto de atividades, cada uma marcada por um tempo de início (si) e tempo de término (fi).

Algoritmo:

```
Greedy-Iterative-Activity-Selector(A, s, f):
    Sort A by finish times stored in f
    S = {A[1]}
    k = 1
    n = A.length
    for i = 2 to n:
        if s[i] ≥ f[k]:
            S = S U {A[i]}
            k = i
    return S;
```

Explicação:

Linha 1 : Este algoritmo é chamado *Seletor de atividade iterativa gananciosa*, porque é antes de tudo um algoritmo ganancioso e depois é iterativo. Há também uma versão recursiva desse algoritmo ganancioso.

- ***A*** é uma matriz que contém as atividades.
- ***s*** é uma matriz que contém o *horários de início* das atividades .
- ***f*** é uma matriz que contém os tempos *de chegada* das atividades .

Linha 3 : Classifica *ordem crescente dos tempos de chegada* a variedade de atividades ***A*** usando os tempos de acabamento armazenados na matriz ***f***. Esta operação pode ser feita em $\boldsymbol{O(n * \log(n))}$ tempo, usando, por exemplo, tipos de mesclagem, tipos de pilha ou algoritmos de classificação rápida.

Linha 4 : Cria um conjunto ***S*** para guardar o *atividades selecionadas*, e inicializa com a atividade ***A[1]*** que tem o primeiro tempo de chegada.

Linha 5 : Cria uma variável ***k*** que acompanha o índice da última atividade selecionada.

Linha 9 : Começa a iterar a partir do segundo elemento dessa matriz ***A*** até o último elemento.

Linhas 10,11 : Se o *hora de começar* **s(i)** do **ith** atividade **A(i)** é maior ou igual ao *hora de terminar* **f[k]** do *última atividade selecionada* (**A[k]**) então **A[i]** é compatível com as atividades selecionadas no conjunto **S**, e assim pode ser adicionado a **S.**

4.2- Algoritmo de Prim:

O Algoritmo de Prin é um tipo de algoritmo de Árvore de extensão mínima.

Outros algoritmos conhecidos para encontrar árvores geradoras mínimas são o algoritmo de Kruskal e algoritmo de Boruvka, onde este último pode ser empregado em grafos desconexos, enquanto o algoritmo de Prim e o Algoritmo de Kruskal precisam de um grafo conexo.

O algoritmo de Prim neste caso fornecerá uma resposta ótima para este problema que não necessariamente é única. A etapa f) da figura 1 demonstra como estas cidades devem ser conectadas com as arestas em negrito.

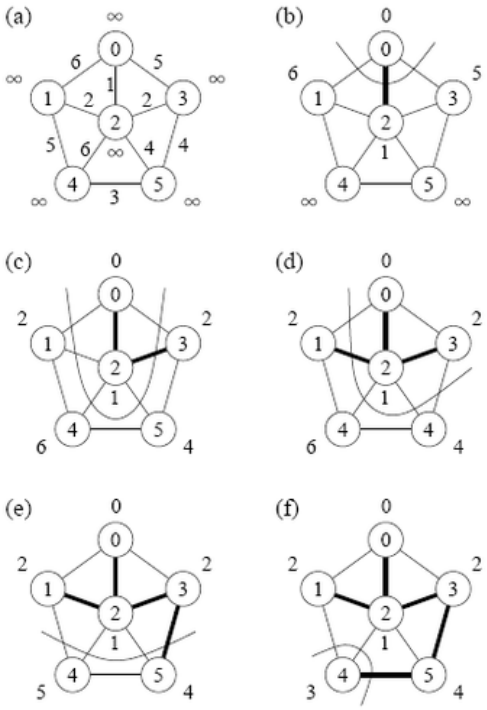


Figura 1: passo a passo da execução do algoritmo de Prim iniciado pelo vértice 0

Algoritmo:

```
int primMST(LAdj *g, int p[], int w[]) {
 int i, imin, v, r=0, cor[g->nvert];
 Nodo *aux;
 int fsize=0, fringe[g->nvert]; // ORLA (stack de vértices)
 for (i=0; i<g->nvert; i++) {
  p[i] = -1;
  cor[i] = BLACK;
 }
 cor[0] = GREY;
 w[0] = 0;
 fringe[fsize++] = 0; //f = addV(f, 0, 0);
  //ciclo principal...
 while (fsize>1) {
   imin = 0;
  for (i=1; i<fsize; i++)
    if (w[fringe[imin]] < w[fringe[i]]) imin = i;
   v = fringe[imin];
  fringe[imin] = fringe[++fsize];
   cor[v] = BLACK;
  r += w[v];
  for (aux=g->adj[v]; aux; aux=aux->prox)
    switch (cor[aux->dest])
          {
          case WHITE:
            cor[aux->dest] = GREY;
            fringe[fsize++] = aux->dest; //f = addV(f, aux->dest, aux->peso);
            w[aux->dest] = aux->peso;
```

```
            p[aux->dest] = v;
            break;
          case GREY:
            if (aux->peso > w[aux->dest]) {
              //f = updateV(f, aux->dest, aux->peso);
              p[aux->dest] = aux->peso;
              w[aux->dest] = v;
            }
          default:
            break;
          }
      }
      return r;
    }
```

Complexidade:

A complexidade do algoritmo de Prim muda conforme a estrutura de dados utilizada para representar o grafo.

As implementações mais comuns para um grafo são por listas de adjacência e por matrizes de adjacência e suas respectivas complexidades $\mathbf{O}([\mathbf{A}]\mathbf{log}[\boldsymbol{V}])$ e $\mathbf{O}(\boldsymbol{V^2})$ no pior caso.

5- Referências Bibliográficas:

1. https://github.com/pedropaiola/unesp-progcomp/blob/main/LPC%20I%20-%202021/Aula%205%20-%20Algoritmo%20Guloso%20-%20Divis%C3%A3o%20e%20Conquista/Algoritmo%20Guloso%20-%20Divis%C3%A3o%20e%20Conquista.pdf

2. https://xlinux.nist.gov/dads//HTML/greedyalgo.html

3. https://xlinux.nist.gov/dads//HTML/greedyHeuristic.html

4. https://www.youtube.com/watch?v=Yq6JlFP520o

5. Cormen, Thomas; Leiserson, Charles; Rivest, Ronald; Stein, Clifford (2001). «23». Introduction to Algorithms (em inglês) 2 ed. [S.l.]: MIT Press and McGraw-Hill. ISBN 0-262-03293-7

6. https://pt.wikipedia.org/wiki/Algoritmo_de_Prim#Descri%C3%A7%C3%A3o

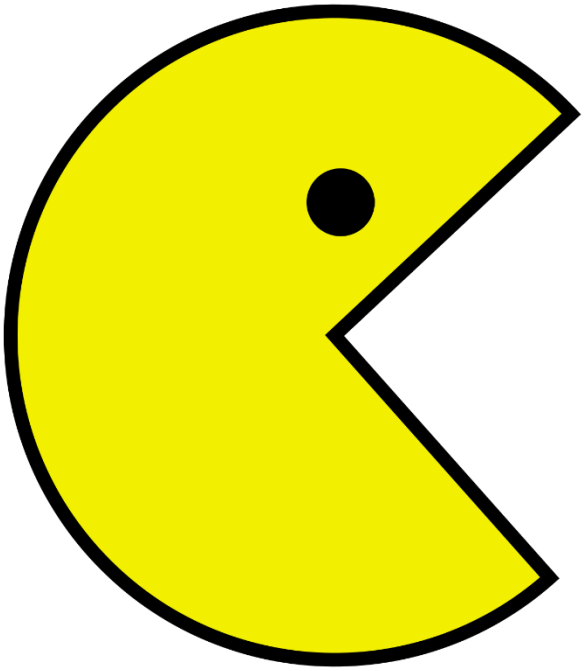
Trabalho Análise de Algoritmos

Docente: Bento Rafael Siqueira

Discentes:

Rodrigo Luis Tavano Bosso – PC3005623

Algoritmos Gulosos



FIM