Bacharelado em Ciência da Computação
Centro de Ciências Exatas e Tecnológicas - CCET

Disciplina: Projeto e Análise de Algoritmos
Série: 4º
Data: 04/06/2024
Descrição: Trabalho 04 - Análise de Grafos

unioeste
Universidade Estadual do Oeste do Paraná

Alunas: Eduarda Elger, Ellen Bonafin e Heloisa Alves

Algoritmo Prim

Algoritmo utilizado para a análise:

```
#define Graph Digraph // Custo: O(1) - Esta é uma operação de substi-tuição de texto e não tem impacto no tempo de execução do algoritmo.

void GRAPHmstP1( Graph G, Vertex parent[]){   // Custo: O(1) - Declaração de uma função, custo constante.

   Vertex v0, w, frj[maxV]; link a;  // Custo: O(1) - Declaração de variáveis, custo constante.
   double price[maxV], c; // Custo: O(1) - Declaração de variáveis, custo constante.

   for (w = 0; w < G->V; w++){  // Custo: O(V) - Este loop executa V vezes, onde V é o número de vértices no grafo.

      parent[w] = -1, price[w] = INFINITO; // Custo: O(V) - Para cada vértice no grafo, atribui um valor a dois arrays (parent e price).
      v0 = 0; // Custo: O(1) - Atribuição de valor fixo à variável.
      parent[v0] = v0; // Custo: O(1) - Atribuição de valor fixo à variável.

      for (a = G->adj[v0]; a != NULL; a = a->next) { // Custo: O(E_0) - Este loop executa uma vez para cada aresta adjacente ao vértice v0, onde E_0 é o número de arestas adjacentes ao vértice v0.

         price[a->w] = a->cost; // Custo: O(E_0) - Atribui um valor a um array para cada aresta adjacente ao vértice v0.
         frj[a->w] = v0; // Custo: O(E_0) - Atribui um valor a um array para cada aresta adjacente ao vértice v0.
      }

      while (1) { // Custo: Não pode ser determinado diretamente, pois depende do número de iterações. Pode ser tratado como O(1) se o número de iterações for limitado.
         double minprice = INFINITO; // Custo: O(1) - Atribuição de valor fixo à variável.
         for (w = 0; w < G->V; w++){  // Custo: O(V) - Este loop executa V vezes, onde V é o número de vértices no grafo.

            if (parent[w] == -1 && minprice > price[w]) // Custo: O(V) - Cada condição dentro do loop é verificada uma vez para cada vértice.
               minprice = price[v0=w]; // Custo: O(V) - Esta linha é executada uma vez para cada vértice no pior caso.
```

```
            if (minprice == INFINITO) break; // Custo: O(1) - Condição de saída do loop.
                parent[v0] = frj[v0]; // Custo: O(1) - Atribuição de valor fixo à variável.

            for (a = G->adj[v0]; a != NULL; a = a->next) { // Custo: O(E_v0) - Este loop executa uma vez para cada aresta adjacente ao vértice v0, onde E_v0 é o número de arestas adjacentes ao vértice v0.
                w = a->w, c = a->cost; // Custo: O(E_v0) - Atribui um valor a duas variáveis para cada aresta adjacente ao vértice v0.

            if (parent[w] == -1 && price[w] > c) { // Custo: O(E_v0) - Cada condição dentro do loop é verificada uma vez para cada aresta adjacente ao vértice v0.
                price[w] = c; // Custo: O(E_v0) - Esta linha é executada uma vez para cada aresta adjacente ao vértice v0 no pior caso.
                frj[w] = v0; // Custo: O(E_v0) - Esta linha é executada uma vez para cada aresta adjacente ao vértice v0 no pior caso.
            }
        }
    }
}
```

Disponível em:
https://gist.github.com/koziel101/18974c3e75ff0a6cb221955790fda876

O custo total do algoritmo é a soma dos custos das partes principais

O (V) + O(E0) + O(v2) + O(E)

simplificando fica:

O(v2+E)

Algoritmo Kruskral

Algoritmo utilizado para a análise:

```
#include<stdio.h>

int visited[10]={0}; // Declaração e inicialização de um array de inteiros com 10 elementos. Custo: O(1)

void kruskal(int w[10][10],int n) // Declaração da função kruskal(). Custo: O(1)
{
    int min,sum=0,ne=0,i,j,u,v,a,b; // Declaração de variáveis inteiras. Custo: O(1)

    while(ne<n-1) // Laço while que executa até que o número de arestas seja n - 1. Custo: O(n)
    {
        min=999; // Atribuição de um valor fixo a min. Custo: O(1)

        for(i=1;i<=n;i++) // Laço for que percorre os vértices. Custo: O(n)
            for(j=1;j<=n;j++) // Laço for interno que percorre os vértices novamente. Custo: O(n^2)
```

```
                    if(w[i][j]<min) // Verifica se o peso de uma aresta é menor que min. Custo: O(1)
                    {
                        min=w[i][j]; // Atualiza min se o peso da aresta for menor. Custo: O(1)
                        u=a=i; // Atualiza os vértices u e a. Custo: O(1)
                        v=b=j; // Atualiza os vértices v e b. Custo: O(1)
                    }

            while(visited[u]) // Laço while que percorre a árvore para encontrar o vértice u. Custo: O(n)
                u=visited[u]; // Atualiza u para o próximo vértice visitado. Custo: O(1)

            while(visited[v]) // Laço while que percorre a árvore para encontrar o vértice v. Custo: O(n)
                v=visited[v]; // Atualiza v para o próximo vértice visitado. Custo: O(1)

            if(u!=v) // Verifica se os vértices u e v são diferentes. Custo: O(1)
            {
                ne++; // Incrementa o número de arestas. Custo: O(1)
                sum+=min; // Soma o peso da aresta mínima. Custo: O(1)
                printf("\nEdge ( %d , %d ) --> %d",a,b,min); // Imprime a aresta mínima. Custo: O(1)
                visited[v]=u; // Atualiza a árvore de visitados. Custo: O(1)
            }

            w[a][b]=w[b][a]=999; // Atualiza o peso da aresta removida. Custo: O(1)
        }

        printf("\nCost of minimum spanning tree : %d\n",sum); // Imprime o custo total da árvore. Custo: O(1)
    }

    int main()
    {
        int w[10][10],n,i,j; // Declaração de variáveis inteiras e matriz. Custo: O(1)

        printf("\nProgram to implement Kruskal's Algorithm : \n"); // Imprime uma mensagem. Custo: O(1)
        printf("\nEnter no. of vertices : "); // Imprime uma mensagem. Custo: O(1)
        scanf("%d",&n); // Lê o número de vértices. Custo: O(1)

        printf("\nEnter the adjacency matrix : \n"); // Imprime uma mensagem. Custo: O(1)
        for(i=1;i<=n;i++) // Laço for que percorre os vértices. Custo: O(n)
            for(j=1;j<=n;j++) // Laço for interno que percorre os vértices novamente. Custo: O(n^2)
                scanf("%d",&w[i][j]); // Lê o peso das arestas. Custo: O(1)

        for(i=1;i<=n;i++) // Laço for que percorre os vértices. Custo: O(n)
            for(j=1;j<=n;j++) // Laço for interno que percorre os vértices novamente. Custo: O(n^2)
                if(w[i][j]==0) // Verifica se o peso é zero. Custo: O(1)
                    w[i][j]=999; // Atualiza o peso para um valor grande. Custo: O(1)
```

```
    kruskal(w,n); // Chama a função kruskal(). Custo: O(n^2)

    return 0; // Custo: O(1)
}
```

Disponível em:
https://github.com/mksjs/algo/blob/master/kruskal.c

O custo total do algoritmo é a soma dos custos das partes principais

O (V) + O(E log E) + O(E log V)

simplificando fica:

O(v2+E)

Para grafos densos, onde o número de arestas é próximo do número máximo possível de arestas (ou seja, $O(V^2)$), o custo do algoritmo de Prim é $O(V^2)$, enquanto o custo do algoritmo de Kruskal é O(E log V), que pode ser aproximado para $O(V^2 \log V)$, pois em um grafo denso E é próximo de V^2.

Para grafos esparsos, onde o número de arestas é proporcional ao número de vértices (ou seja, O(V)), o custo do algoritmo de Prim é $O(V^2)$, enquanto o custo do algoritmo de Kruskal é O(E log V), que, para um grafo esparso, também é O(V log V).

Portanto, no pior caso, o custo tanto para o algoritmo de Prim quanto para o algoritmo de Kruskal é de O(V?) para um grafo denso e O(V log V) para um grafo esparsos.

No caso de um grafo denso, onde o número de arestas é próximo do número máximo possível de arestas (ou seja, $O(V^2)$), o algoritmo de Prim tem um custo de tempo de $O(V^2)$, enquanto o algoritmo de Kruskal tem um custo de tempo de O(E log V), onde V é o número de vértices e E é o número de arestas.

No caso de um grafo esparsos, onde o número de arestas é proporcional ao número de vértices (ou seja, O(V)), o algoritmo de Prim tem um custo de tempo de $O(V^2)$, enquanto o algoritmo de Kruskal tem um custo de tempo de O(E log V) que, para um grafo esparso, também é O(V log V).

Portanto, no melhor caso, o custo tanto para o algoritmo de Prim quanto para o algoritmo de Kruskal é de $O(V^2)$ para um grafo denso e O(V log V) para um grafo esparsos.