Algoritmika grafů

Martin Klíma

2022-2023

Obsah

1 Depth First Search

Algoritmus pro procházení grafu do hloubky (anglicky Depth-First Search, DFS) je jedním ze základních algoritmů pro procházení grafů. Algoritmus DFS prochází graf postupně z jednoho vrcholu do dalších, tak dlouho, dokud to bude možné. Pokud narazí na vrchol, který již byl navštíven, algoritmus se vrátí zpět a pokračuje v procházení grafu z jiného vrcholu.

Při procházení grafu DFS používá zásobník pro ukládání vrcholů, které budou následně zpracovány. Na začátku se do zásobníku vloží počáteční vrchol grafu. Poté se postupně vybírají vrcholy ze zásobníku a zpracovávají se. Při zpracování vrcholu se navštíví všechny jeho následníky, kteří ještě nebyli navštíveni. Tyto následníky se poté vloží na vrchol zásobníku a proces se opakuje.

Algoritmus DFS je velmi jednoduchý a intuitivní, ale má několik nevýhod. Jednou z nevýhod je, že může být velmi pomalý pro grafy s mnoha vrcholy a hranami. Dále může vést k rekurzivnímu zanořování a potenciálně k překročení maximální hloubky rekurze.

Tento kód provede DFS pro procházení grafu do hloubky, kdy výstupem bude seznam navštívených vrcholů v pořadí, v jakém byly navštíveny:

```
using System;
using System.Collections.Generic;

public class Program
{
    public static void Main()
    {
        // create a new stack to use in DFS
        Stack<int> stack = new Stack<int>();

        // create a new graph with 4 vertices
        Graph g = new Graph(4);

        // add edges to the graph
        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);

        // perform DFS starting from vertex 2
        g.DFS(2);
    }
}

```

```
27 public class Graph
28 {
29     int V; // the number of vertices in the graph
30     List<int>[] adj; // an array of adjacency lists to
          represent the graph
31
32     // constructor to initialize the graph with the given
          number of vertices
33     public Graph(int v)
34     {
35         V = v;
36         adj = new List<int>[v];
37         for (int i = 0; i < v; i++)
38         {
39             adj[i] = new List<int>();
40         }
41     }
42
43     // method to add an edge between two vertices in the graph
44     public void addEdge(int v, int w)
45     {
46         adj[v].Add(w);
47     }
48
49     // method to perform depth-first search starting from the
          given source vertex
50     public void DFS(int s)
51     {
52         bool[] visited = new bool[V]; // array to keep track
              of visited vertices
53         for (int i = 0; i < V; i++)
54             visited[i] = false;
55         Stack<int> stack = new Stack<int>(); // create a new
              stack for DFS
56         stack.Push(s); // add the source vertex to the stack
57         visited[s] = true; // mark the source vertex as visited
58         while (stack.Count != 0) // loop until the stack is
              empty
59         {
60             int pop = stack.Pop(); // remove the next vertex
                  from the stack
61             Console.Write(pop + "␣"); // print the vertex
62             foreach (var vert in adj[pop]) // iterate over the
                  adjacent vertices
63             {
```

```
64                  if (!visited[vert]) // if the adjacent vertex
                        is not visited
65                  {
66                      visited[vert] = true; // mark it as visited
67                      stack.Push(vert); // add it to the stack
68                  }
69              }
70          }
71      }
72  }
```

2 Breadth First Search

Prohledávání grafu do šířky (anglicky Breadth-First Search, zkráceně BFS) je algoritmus pro prohledávání grafů, který postupuje systematicky a rozšiřuje průzkum postupně po vrstvách, tedy nejprve navštíví všechny sousedy zdrojového vrcholu, poté všechny sousedy sousedů a tak dále. Algoritmus je založen na principu prohledávání grafu pomocí fronty.

BFS začíná v zadaném vrcholu a postupně prochází všechny jeho sousedy. Tyto sousedy vloží do fronty, aby se prohledaly v následujícím kroku. Následně se vezme první vrchol z fronty a opakuje se proces pro všechny jeho sousedy, kteří nebyli navštíveni. Postup se opakuje, dokud fronta není prázdná.

```
1   using System;
2   using System.Collections.Generic;
3
4   public class Graph
5   {
6       private int V; // number of vertices in the graph
7       private List<int>[] adj; // adjacency list representing
            the graph
8
9       // Constructor to initialize the graph
10      public Graph(int v)
11      {
12          V = v;
13          adj = new List<int>[v];
14
15          // Initialize each element of the adjacency list to an
                empty list
16          for (int i = 0; i < v; i++)
17          {
18              adj[i] = new List<int>();
19          }
20      }
```

```
21
22      // Method to add an edge between two vertices
23      public void AddEdge(int v, int w)
24      {
25          // Add vertex w to the adjacency list of vertex v
26          adj[v].Add(w);
27      }
28
29      // Method to perform BFS traversal of the graph
30      public void BFS(int s)
31      {
32          // Create a boolean array to keep track of visited
                vertices
33          bool[] visited = new bool[V];
34
35          // Create a queue to store the vertices to be visited
36          Queue<int> queue = new Queue<int>();
37
38          // Mark the source vertex as visited and enqueue it
39          visited[s] = true;
40          queue.Enqueue(s);
41
42          while (queue.Count != 0)
43          {
44              // Dequeue a vertex from the queue and print it
45              s = queue.Dequeue();
46              Console.Write(s + "␣");
47
48              // Get all adjacent vertices of the dequeued
                    vertex s
49              // If an adjacent vertex has not been visited,
                    mark it as visited and enqueue it
50              foreach (int i in adj[s])
51              {
52                  if (!visited[i])
53                  {
54                      visited[i] = true;
55                      queue.Enqueue(i);
56                  }
57              }
58          }
59      }
60  }
61
62  // Driver code to test the implementation
63  public class BFS
```

```
{
    public static void Main()
    {
        Graph g = new Graph(4); // Create a graph with 4 vertices

        // Add edges to the graph
        g.AddEdge(0, 1);
        g.AddEdge(0, 2);
        g.AddEdge(1, 2);
        g.AddEdge(2, 0);
        g.AddEdge(2, 3);
        g.AddEdge(3, 3);

        Console.WriteLine("BFS traversal starting from vertex 2:");
        g.BFS(2); // Perform BFS traversal starting from vertex 2
    }
}
```

3 Dijkstrův algoritmus

Dijkstrův algoritmus na hledání nejkratších cest v grafu je algoritmus z oblasti teorie grafů. Jeho účelem je najít nejkratší cestu ze zvoleného počátečního uzlu do všech ostatních uzlů v grafu s ohledem na délku hran mezi nimi. Algoritmus je pojmenován po nizozemském vědci Edsgeru W. Dijkstrovi, který ho popsal v roce 1959.

Dijkstrův algoritmus je založen na principu postupného procházení grafu, přičemž pro každý uzlu uchovává nejkratší známou cestu z počátečního uzlu. Algoritmus se řídí principem tzv. "hladového algoritmu"(angl. greedy algorithm), kdy v každé iteraci vybírá z nezpracovaných uzlů ten s nejmenší dosud známou vzdáleností od počátečního uzlu.

Postup algoritmu je následující:

1. Nejprve se inicializuje graf a počáteční uzel. Pro každý uzel se uchovává jeho dosud nejkratší vzdálenost od počátečního uzlu, pro počáteční uzel se nastaví tato vzdálenost na 0, zatímco pro všechny ostatní uzly se nastaví na nekonečno.

2. Následně se vybírá nezpracovaný uzel s nejmenší dosud známou vzdáleností od počátečního uzlu a zpracuje se. V této fázi se do uzlu přidávají sousední uzly, kde sousední uzel je takový uzel, který je spojen hranou s aktuálním uzlem.

3. Pro každý sousední uzel se zkontroluje, zda je jeho nově vypočítaná vzdálenost menší než jeho dosud známá vzdálenost. Pokud ano, nová vzdálenost se uloží jako jeho nejkratší vzdálenost a uzel se přidá do seznamu nezpracovaných uzlů.

4. Postup se opakuje, dokud nejsou zpracovány všechny uzly, pro které existuje cesta z počátečního uzlu.

Na konci algoritmu jsou tedy pro každý uzel grafu známy jeho nejkratší vzdálenosti od počátečního uzlu, což umožňuje nalezení nejkratší cesty z počátečního uzlu do kteréhokoliv jiného uzlu v grafu.

```
1       using System;
2
3       class GFG {
4       static int V = 9; // Number of vertices in the graph
5
6   // This function finds the vertex with the minimum distance
        value,
7   // from the set of vertices not yet included in shortest path
        tree.
8   int minDistance(int[] dist, bool[] sptSet)
9   {
10      int min = int.MaxValue, min_index = -1;
11
12      // Loop through all vertices
13      for (int v = 0; v < V; v++)
14          // Check if vertex v is not in sptSet and if its
                distance
15          // from source is less than the current minimum
                distance
16          if (sptSet[v] == false && dist[v] <= min) {
17              // Update the minimum distance and the index of
                    the vertex
18              min = dist[v];
19              min_index = v;
20          }
21
22      // Return the index of the vertex with the minimum distance
23      return min_index;
24  }
25
26  // This function prints the distance of all vertices from the
        source
27  void printSolution(int[] dist)
28  {
29      Console.Write("Vertex␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Distance␣"
30                    + "from␣Source
31  ");
32      for (int i = 0; i < V; i++)
33          Console.Write(i + "␣␣␣␣␣␣␣␣␣␣␣␣␣␣" + dist[i] + "
34  ");
```

```
35 }
36
37 // This function implements Dijkstra's algorithm to find the
   shortest
38 // path from a source node to all other nodes in a weighted
   graph.
39 void dijkstra(int[,] graph, int src)
40 {
41     int[] dist= new int[V]; // Array to store the shortest
       distance from the source to each vertex
42
43     bool[] sptSet = new bool[V]; // Array to store whether a
       vertex is included in the shortest path tree or not
44
45     // Initialize all distances to infinity and sptSet to false
46     for (int i = 0; i < V; i++) {
47         dist[i] = int.MaxValue;
48         sptSet[i] = false;
49     }
50
51     dist[src] = 0; // Set the distance of the source node to 0
52
53     // Loop through all vertices except the source
54     for (int count = 0; count < V - 1; count++) {
55         int u = minDistance(dist, sptSet); // Find the vertex
           with the minimum distance from the source
56
57         sptSet[u] = true; // Add the vertex to the shortest
           path tree
58
59         // Update the distance of all adjacent vertices if the
           new path through
60         // the current vertex is shorter than the previously
           known path
61         for (int v = 0; v < V; v++){
62             if (!sptSet[v] && graph[u, v] != 0
63                 && dist[u] != int.MaxValue
64                 && dist[u] + graph[u, v] < dist[v])
65                 dist[v] = dist[u] + graph[u, v];
66         }
67     }
68     // Print the shortest distances of all vertices from the
       source
69     printSolution(dist);
70 }
71
```

```
72  public static void Main()
73  {
74      // Example graph as an adjacency matrix
75      int[,] graph
76              = new int[,] { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
77                             { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
78                             { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
79                             { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
80                             { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
81                             { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
82                             { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
83                             { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
84                             { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };
85          GFG t = new GFG();
86
87          // Function call
88          t.dijkstra(graph, 0);
89      }
90  }
```

4 Vizualizace

Pro vizualizaci a zadávání grafů jsem využil html element canvas. Oproti canvasu v jiných jazycích (frameworcích) má canvas v html značnou nevýhodu pro mé využití. Například v C# WPF se na canvas přidávají jednotlivé objekty a je možné k nim následně přistupovat a pracovat s nimi. Takovouto možnost html canvas nenabízí, jelikož html canvas nezaznamenává jednotlivé objekty, ale pouze vybarvené pixely.

Proto jsem musel přijít s řešením pro interakci s prvkami grafu bez využití interaktivních funkcí jednotlivých objektů jako v C# WPF.

Mé řešení se zakládá na zaznamenávání polohy kliknutí a následné kontrole, zda se v dané oblasti nachází interaktivní prvek, či ne. Samotný kód pro toto řešení je vcelku jednoduchý, jelikož se jedná pouzo o cyklus procházející vykrelené prvky a složenou podmínku kontrolující polohu vůči souřadnicím kliknutí.

```
elements.forEach((element) => {
    if (
        y > element.top - element.height*2 &&
        y < element.top + element.height*2 &&
        x > element.left - element.width*2 &&
        x < element.left + element.width*2
    )
```

Jedna z dalších překážek vycházela opět z toho, že si canvas zaznamenává pouze vybarvená místa a proto není možné jednotlivým prvkům určovat "výšku"
umístění oproti ostatním prvkům na plátně. Proto není možné umístit cesty mezi vrcholy grafu na "nižší úroveň"bez opětovného vykreslování plátna v požadovaném pořadí pokaždé, kdy je přidána cesta do grafu, aniž by zasahovala do vrcholu grafu. Neustálé vykreslování plátna by mohlo být zbytečně zátěžné, proto jsem se rozhodl pro využití alalitické geometrie pro výpočet průsečíků cesty s okrajem kružnice značící vrchol grafu.

```
let dx = lineEndX - lineStartX;
    let dy = lineEndY - lineStartY;
    let length = Math.sqrt(dx * dx + dy * dy);
    if (length > 0){
        dx /= length;
        dy /= length;
    }
    dx *= length - 25;
    dy *= length - 25;

    lineEndX = lineStartX + dx
    lineEndY = lineStartY + dy

    dx = lineStartX - lineEndX;
    dy = lineStartY - lineEndY;
    length = Math.sqrt(dx*dx+dy*dy)
    if (length > 0){
        dx /= length;
        dy /= length;
    }
    dx *= length - 25;
    dy *= length - 25;

    lineStartX = lineEndX + dx
    lineStartY = lineEndY + dy
```

Pro vizualizaci průchodu grafu je nutné znovu vykreslovat plátno pro kažný "snímek", což sice není nejhezčí řešení, ovšem html canvas jiné nenabízí. Jako časovač pro oddělaní kroků průchodu grafu jsem využil funkci setInterval(), která po zadaném intervalu volá přidělenou funkci, odkud není zastavena. Při každném kroku animace se smaže plátno, podle pořadí průchodu grafu se určí další navštívený vrchol, kterému se změní barva pro vykreslení a následně se plátno vykreslí. Tento proces se opakuje dokud neprojde veškeré procházené prvky a poté zastaví interval.