Virtual Lab Computer Science and Engineering and IT

IIIT Hyderabad: Data Structures

JUIT Lab: Data Structures and Computer Programming Lab (10B17CI271)

1. Graph Traversals
2. Spanning Trees in Graphs

Experiment No 1 (18)

Introduction

Graph Traversals

In this experiment, we will see a fundamental problem related to graphs, Graph Traversal. It is nothing more than visiting every vertex of the given graph. Based on the order in which we visit the vertices, we define two different types of graph traversals. The order of vertices thus obtained can be useful in understanding several properties of the graph.

Two main techniques : Breadth First Search (BFS), and Depth First Search (DFS).

Theory

Breadth First Search (BFS)

BFS is similar to level order traversal in a tree, where we visit vertices in a level by level order. We specify a starting vertex *S* and start visiting vertices of the graph *G* from the vertex *S*. As a general graph can have cycles, we may visit the same vertex more than once. To solve this, we also maintain the state of each vertex. A vertex can be in one of three states, VISITED, NOT_VISITED, IN_PROCESS. The basic idea of breadth first search is to find the least number of edges between *S* and any other vertex in *G*. Starting from *S*, we visit vertices of distance *k* before visiting any vertex of distance *k+1*. For that purpose, define *ds(v)* to be the least number of edges between *S* and *v* in *G*. So, for vertices *v* that are not reachable from *S* we can have *ds(v)* = ∞. We can use a queue to store vertices in progress.

We also a maintain *from[u]* for each of the vertex *u*, which denotes the vertex that discovered *u*. This information can be used to define the predecessor subgraph of *G*, which has all edges *(from[u],u)*. These edges are called as tree edges. For a tree edge *(u, v), ds(v) = ds(u) + 1*. All other edges are called non-tree edges and are further classified as back edges and cross edges. A non-tree edge (u, v) is called as a back edge if *ds(v) < ds(u)*, meaning *u* can be reached from *S* via *v*, but there is yet another shortest path through path *u* can be reached from *S*. An edge *(u, v)* is called a cross edge if $d(v) \leq d(u) + 1$.

```
Procedure BFS(G)
 for each v in V(G) do
   from[v] = NIL; state[v] = NOT_VISITED; d(v) = ∞
 end-for

 ds[S] = 0; state[S] = IN_PROCESS; from[s] = NIL;
 Q = EMPTY; Q.Enqueue(S);

 While Q is not empty do
   v = Q.Dequeue();
   for each neighbour w of v do
    if state[w] == NOT_VISITED then
     state[w] = IN_PROCESS; from[w] = v; ds[w] = ds[v] + 1; Q.Enqueue(w);
    end-if
   state[v] = VISITED
   end-for
 end-while
End-BFS
```

<u>*Runtime:*</u> Each vertex enters the queue at most once and each edge is considered only once. Therefore, runtime of BFS is *O(m+n)*, where *n* is the number of vertices and *m* is the number of edges in *G*.

Depth First Search (DFS)

The idea of DFS is to start from a specified start vertex *S* and explore from *S* as deep as possible. We go from *S*, to one of its neighbors *x*, to a neighbor of *x*, and so on. We stop when there are no new neighbors to explore from a given vertex. If all vertices are not visited, pick another start vertex from such vertices. We have to keep track of the state of a vertex and similar to BFS, a vertex can be in three states: VISITED, NOT_VISITED, IN_PROCESS. We also a maintain *from[u]*, *d[u]*, *f[u]* for each of the vertex *u*, which denotes the vertex that discovered *u*, the discovery time of *u* and the finish time *u*.

```
Procedure Explore(v)
  d[v] = cur_time;
  cur_time = cur_time + 1;
  for each neighbor w of v
   if state(w) == NOT_VISITED then
    state(w) = IN_PROCESS;
    Explore(w);
   end-if
  end-for
  state(v) = VISITED;
  f[v] = cur_time;
  cur_time = cur_time + 1;
End-Explore

Procedure DFS(G)
  cur_time = 0
  for v = 1 to n do
   if state(v) == NOT_VISITED then
    state(v) = IN_PROCESS;
    Explore(v);
   end-if
  end-for
End-DFS
```

Runtime: Explore() is called for each vertex exactly once and each edge is considered only once. Therefore, runtime of DFS is *O(m+n)*, where *n* is the number of vertices and *m* is the number of edges in *G*.

Classification of edges, based on Depth First traversal is as follows

- Tree Edges : All edges *(from[u],u)* are called as tree edges and define a dfs tree, a subgraph of *G*.
    - *An edge (u,v)* is a tree edge if *from[v] = u*
- Back Edges : A non-tree edge *(u, v)* is called as a back edge if *v* is an ancestor of *u* in the dfs tree. *u* can be reached from *v* using tree edges, but there an edge from *u* to *v* also.
    - *An edge (u,v)* is a back edge if [d(u), f(u)] is a subinterval of [d(v), f(v)]
- Forward Edges : A non-tree edge *(u, v)* is called as a forward edge if *v* is a descendant of *u* in the dfs tree. *v* can be reached from *u* using tree edges, but there an edge from *u* to *v* also.
    - *An edge (u,v)* is a forward edge if [d(v), f(v)] is a subinterval of [d(u), f(u)].
- Cross Edges : Non-tree edges *(u, v)* where *u* and *v* do not share any ancestor/descendant relationship are called cross edges.
    - *An edge (u,v)* is a cross edge if the two intervals [d(u), f(u)] and [d(v), f(v)] do not overlap.

Objectives:

At the end of this experiment, you will be able to:

- * Perform a Breadth First Traversal on a graph
- * Perform a Depth First Traversal on a graph
- * Classify edges of the graph based on the traversal

Quizzes:

- Suppose that G is a directed graph with a vertex u with both a nonzero indegree and a nonzero outdegree. Can u end up in a DFS tree containing only itself? Justify your answer.
- Take an example directed graph of about 12 vertices and 20 edges. Perform DFS on the graph and record the discovery and finish time of each vertex.
- Repeat the above question with respect to BFS.
- Suppose that G is a directed graph and (u,v) is an edge in the graph. Is it true that the finish time of u is always larger than the finish time of v?
- Consider a directed graph G with an edge (u,v). Suppose that there is no path from v to u in G. Is it true that the finish time of u is always larger than the finish time of v?
- Consider the setting of the above question. Suppose now that C and C' are two subsets of vertices so that there is a path between every vertex in C (C') to every other vertex in C (C'), and there is an edge (u,v) with u in C and v in C'. Suppose also that there is no path from any vertex in C' to any vertex in C. Is it true that any vertex in C has a finish time larger than any vertex in C'? Justify your answer.
- Let G be an undirected graph and we apply BFS on G. Classify the edges of G as tree edges, back edges, forward edges, and cross edges. Are any of these classes empty? If so, why?
- Let G be an undirected graph. Use DFS on G to partition the vertices of G into subsets V1, V2, ..., so that two vertices u and v are in the same partition if and only if there is a path between u to v. Such subsets are called as the connected components of G.
- Let G have k connected components. How would k change if one edge is added to G? How would k change if one edge is removed from G.
- Consider the problem of arranging the vertices of a directed acyclic graph so that if (u,v) is an edge in the graph then u appears before v in the arragenment. Such an arrangement is called as a topological sorting of the graph. Design an algorithm using DFS to obtain a topological sort of a given directed acyclic graph. What is the runtime of your algorithm.
- Repeat the above question with using BFS instead of DFS. Is there any difference in the runtime? Which approach do you prefer and why?

Experiment No 2 (19)

Introduction

Spanning Trees in Graphs

In this experiment, we will see a famous problem in graphs, finding the Minimum Spanning Tree. Let $G = (V, E, W)$ be a weighted graph. Find a subgraph G' of G that is connected and has the smallest cost, where cost is defined as the sum of the edge weights of all edges in G'. Observe that if G' has a cycle, we can remove one of the edges along a cycle and still the resultant graph will remain connected and has smaller cost than G'. Hence, G' cannot be have a cycle and as it has to be connected, G' must be a tree. We define G' as a spanning subgraph of G iff $V(G) = V(G')$ and a spanning subgraph that is also a tree is called a spanning tree of G. Our aim is to find a spanning tree of G that has the least cost and such a spanning tree is called as minimum spanning tree (MST) of G.

Theory

Lets look at a simple method to find the MST of a given graph G. We start with a spanning graph of G with no edges initially and keep adding edges one by one. As we want to minimize the total cost, we should prefer to add edges with smaller weights first, but should not add edges that create cycles. This method of greedily picking the edges to form a MST is called the Kruskal's algorithm, and is described below.

```
Algorithm Kruskals(G)
  sort the edges of G in increasing order of weight as e1, e2, ..., em
  i = 1; E(T) = Φ
  while |E(T)| < n-1 do
   if E(T) ∪ ei does not have a cycle then
     E(T) = E(T) ∪ ei
   end-if
   i = i + 1;
  end-while
End-Kruskals
```

We still need to implement the cycle checker. The simplest way to do this is as follows. Suppose we want to check if adding a edge (u,v) can create a cycle or not. Before adding the edge (u,v), perform a depth first search (DFS) starting from v to see if the vertex u can be reached from v. If it can be reached, then adding (u,v) will create a cycle. The time taken for DFS is $O(m+n)$ and for a tree $m = O(n)$, so the running time is $O(n)$. In the worst case, we need to try all m edges, so the running time of Kruskal's algorithms if we use a simple cycle checker is $O(mn)$. Using some advanced datastructures, we can bring down the running time to $O(m \log n)$. We will now look at a much simpler solution with smaller runtime, using basic data structures.

In this approach, we maintain a single tree T at any time. In each iteration, T is extended by adding one vertex v not in T and one edge from v to some vertex in T. Starting from a tree of one node, this process is repeated (n-1) times. For each vertex v in G, there must be at least one edge in any MST. Considering the edge of the smallest weight is useful as it can decrease the cost of the spanning tree. So, for any vertex v, the edge with the least weight among all edges having one of its end points as v, is always contanied in any MST of G. Let T is a subtree of some MST of an undirected weighted graph T. Consider edges (u,v) in G such that u is in T and v is not in T. Of all such edges, let $e = (x,y)$ be the edge with the smallest weight. Then $T \cap \{e\}$ is also a subtree of some MST of *G*. This suggests an incemental method of constructing a MST. This algorithm is called Prim's algorithm, and is described below.

```
Algorithm Prims(G,v)
  Add v to T;
  While T has less than n Â–  1 edges do
    w = vertex s.t. (v,w) has the smallest weight amongst edges with one endpoint in T and another not in T.
    Add (v,w) to T.
  end-while
End-Prims
```

The only thing left is to efficiently find the vertex *w* in the above algorithm. For this purpose, we associate a key to every vertex and *key[v]* is the smallest weight of edges with *v* as one endpoint and another in the current tree *T*. A *key[v]* changes only when some vertex is added to *T* and also vertex with the smallest *key[v]* is the one to be added to *T*. When a vertex is added to *T*, only its neighboring vertices may have to update their keys. Therefore, we can maintain a heap of vertices with their *key[]* values and perform the above algorithm as follows.

```
Algorithm Prims_heap(G, u)
  for each vertex v do key[v] = ∞
  key[u] = 0;
  Add all vertices to a heap H.
  While T has less than n-1 edges do
    v = deleteMin();
    Add v to T via uv s.t. u is in T
    For each neighbor w of v do
      if W(vw) > key[w] then DecreaseKey(w)
    end-for
  end-while
End-Prims_heap
```

Runtime: Each vertex is deleted once from the heap. Each DeleteMin() takes *O(log n)* time. So, this accounts for a time of *O(n log n)*. Each edge may result in one call to DecreaseKey(). Over *m* edges, this accounts for a time of *O(m log n)*. Total time = *O((n+m)log n)*.

Objectives:

At the end of this experiment, you will be able to:

* Know the concept of spanning trees and minimum spanning trees

* Understand algorithmic approaches to finding minimum spanning trees in graphs

* Understand the data structures required to efficeintly implement algorithms for minimum spanning trees.

Quizzes:

- Take an example graph of about 10 vertices and 20 edges. Assign weights to edges and use the algorithm of Kruskal to find a minimum spanning tree.
- In a graph G, let the edge uv have the least weight. Is it true that uv is always part of any minimum spanning tree of G? Is it true that uv is always part of some minimum spanning tree of G? Justify your answers.
- Let G be a graph and T be a minimum spanning tree of G. Suppose that the weight of an edge e is decreased. How can you find the minimum spanning tree of the modified graph. What is the runtime of your solution?
- Let G be a graph and T be a minimum spanning tree of G. Suppose that the weight of an edge e that also belongs to T is increased. How can you find the minimum spanning tree of the modified graph. What is the runtime of your solution?
- Let G be a graph and T be a minimum spanning tree of G. If an edge e is added to G with weight w(e) to get the graph G', how can T be modified to T' so that T' is a minimum spanning tree of G'. What is the runtime of your solution?
- Let G be a graph and e be a maximum weight edge on some cycle in G. Let G' be the graph obtained by removing the edge e from G. show that G' has a minimum spanning tree T' that is also a minimum spanning tree of G.
- Suppose that the edge weights in a graph are all either 1 or 2. Can you modify Dijkstra's algorithm to run faster? If so, describe your approach and analyze the runtime of your approach.
- Generalize the above question where the edge weights are between 1 and W.