Basic Graph Algorithms

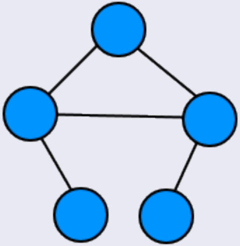
Oliver-Matis Lill

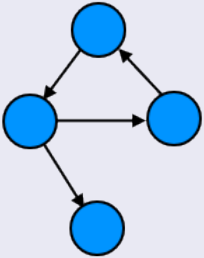
April 23, 2018

Graphs

- Graphs are basically a collection of nodes connected by edges
- They can be directed, or undirected
- Skill with OOP and knowledge of the Standard Library will be extremely useful

Example





Graph Code

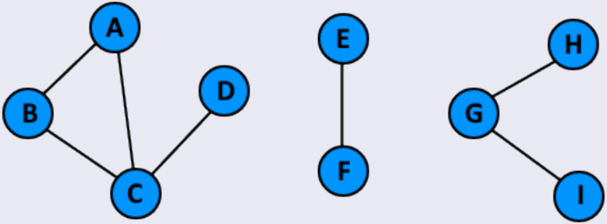
Example

```
struct Graph {
    struct Node {
        vector<Node*> arc;
    };
    deque<Node> nodes;
    int n = 0;

    Node* newNode() {
        nodes.push_back(Node());
        return &nodes.back();
    }
    void addArc(Node* a, Node* b) {
        a->arc.push_back(b);
    }
};
```

Graph Traversal

- Suppose you have an unconnected, undirected graph like this:

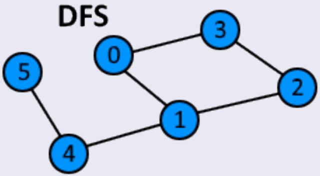


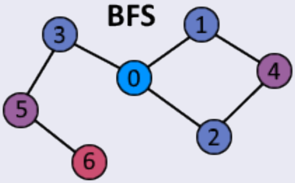
- How would you find what other nodes are connected to some chosen node (for instance D from above image)

Graph Traversal

- Intuitively you traverse the nodes while keeping track which ones are explored, performing traversal until all connected nodes are traversed
- Turns out it's simple to write a program to do that
- There are however different possible traversal approaches

- Two common and useful traversal techniques are:
  1. Depth-first search - greedily traverse from last node, move back when no new nodes to traverse to
  2. Breadth-first search - traverse in "waves", going from closer nodes to farther ones
- DFS is usually implemented with recursion, BFS with a queue
- The order in which nodes are traversed could look like:





Graph Traversal Code

Example

```
void DFS(Node* cur) {
    cur->explored = true;

    for(Node* next : cur->arc)
        if(!next->explored)
            DFS(next);
}
```

```
void BFS(Node* start) {
    deque<Node*> que(1, start);
    start->explored = true;

    while(que.size() > 0) {
        Node* cur = que.front();
        que.pop_front();

        for(Node* next : cur->arc)
            if(!next->explored) {
                next->explored = true;
                que.push_back(next);
            }
    }
}
```

- DFS is generally much simpler
- They have different properties that are useful in different scenarios
- If all edges have equal length, then BFS will be an effective technique for finding minimum path

Dijkstra Algorithm

- Suppose we have edges with differing non-negative lengths and we want to find a shortest path from A to some other vertex B
- Recall from the Dynamic Programming lecture, that if all edges are directed from smaller vertex index to larger, then there is a simple DP solution

Dijkstra Algorithm

Idea

- Let's number the nodes from 1 to N and the roads from 1 to M. Without loss of generality let's pick 1 to be the starting node and N to be the destination node
- Each road j is characterized by three integers: the starting node s_j, the ending node e_j and it's length l_j
- Let's denote that:

$$d_i = \text{the minimum path length from node 1 to node } i$$

Dijkstra Algorithm

Idea

- Notice that:

$$d_i = \min_{j \in \text{in}(i)} \{d_{s_j} + l_j\}$$

  where $\text{in}(i)$ is the set of edges entering node i
- That's because, suppose (s_j, e_j) is the last edge traversed to e_j in the minimum path. We obviously want the path to s_j we used to be minimum as well, therefore $d_{e_j} = d_{s_j} + l_j$
- Obviously $d_1 = 0$, but how to calculate the rest?

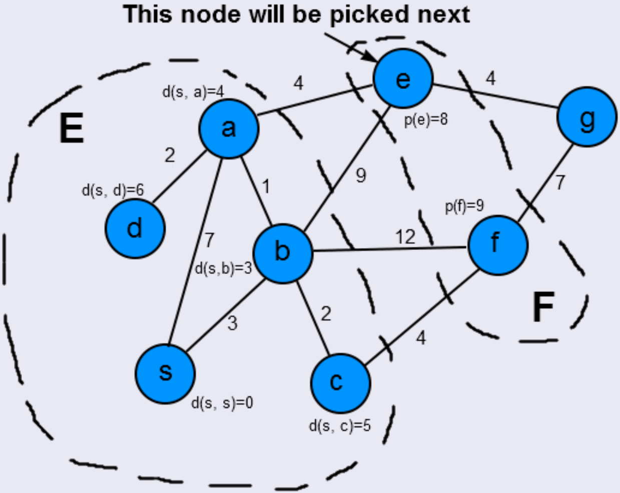
Idea

- Let's repeat the following algorithm until d_N is calculated:
  1. Let E be the set of every node i for which d_i is known
  2. Find j such that $s_j \in E$, $e_j \notin E$ and $d_{s_j} + l_j$ is minimal
  3. Set $d_{e_j} = d_{s_j} + l_j$ and add e_j to E
- Note that step 2 will set d_{e_j} to be the minimum possible, because thanks to step 2, every d_i calculated in the future will be greater (or equal)

Dijkstra Implementation

- Note that the algorithm is run $O(N)$ times
- An easy way to perform step 2 would be to iterate over all edges to find the minimal $d_{s_j} + l_j$. This would take $O(M)$ operations and give a total running time of $O(NM)$
- We can however do much better by keeping all candidates for the minima in a heap and then adding new candidates at step 3
- Each edge would add a single candidate to heap (when s_j is added to E at step 3) and heap operations will take $O(\log M)$ operations
- This would give us an $O(M \log M)$ algorithm.

Example


This node will be picked next
E
F
e
g
a
d
b
f
s
c
d(s, a)=4
d(s, d)=6
d(s,b)=3
d(s, s)=0
d(s, c)=5
p(e)=8
p(f)=9
4
4
2
1
9
7
7
12
3
2
4

Dijkstra Algorithm Code

Example

```
void Dijkstra(Node* start) {
    //Note: priority_queue is a max heap. Use the inverses of potentials
    priority_queue<pair<int, Node*> > front;
    front.push({0, start});
    while(front.size() > 0) {
        int dist; Node* cur;
        tie(dist, cur) = front.top();
        front.pop();
        if(cur->explored)
            continue;
        cur->explored = true;
        cur->distance = -dist;
        for(Arc* curArc : cur->arc)
            front.push({dist-curArc->length, curArc->destination});
    }
}
```