COMPSCI 330: Design and Analysis of Algorithms	09/11/14
Lecture 6	
Lecturer: Debmalya Panigrahi	*Scribe: Wenshun Liu*

1 Overview

In this lecture we introduced the concept of the shortest path between two vertices, as well as the related algorithms in finding such paths from a start vertex to all other reachable vertices in the graph. Content covered includes properties and applications of a shortest path, Breath First Search (BFS) algorithm, and Dijkstra's algorithm.

2 Definitions

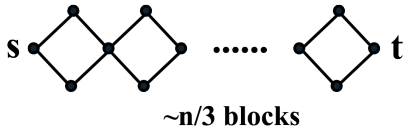
2.1 Length (Weight) of a Path

For an *unweighted* graph, the *length* of a path is simply the number of edges on the path.
For a *weighted* graph, every edge is associated with a length (weight). The *length* of a path is the sum of all edge lengths on the path.

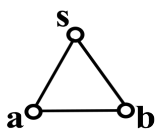
2.2 Distance between Two Vertices

The *distance* between two vertices u and v is defined as the length of *the shortest path* connecting them.

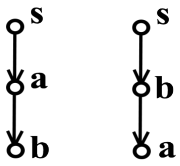
Remark 1. *The number of paths connecting two vertices could be exponential. As shown in the below illustration, there are around $\frac{n}{3}$ "blocks" to choose from given that there are n vertices in total and each block takes around 3 vertices. The number of paths available from s to t is therefore in $O(2^{\frac{n}{3}})$, which is exponential. Thus, it is unwise to find the shortest path by finding all possible paths and comparing their lengths.*



Remark 2. *DFS is not a good indicator of the shortest path between two vertices. As illustrated below, a DFS tree does not indicate the shortest path from a start vertex s to the target vertices. The rest of the lecture will focus on algorithms that are efficient in doing so.*



graph



two DFS tree representations starting at s

3 Breath First Search (BFS)

3.1 Description

BFS follows a layer-by-layer approach to traverse a graph. It first covers all vertices at distance 1 from the starting node. Then for each of such neighbour nodes, it inspects their unvisited neighbours, and so on.

BFS could be applied to both *directed* and *undirected* graphs.

Pseudocode:

```
1   BFS(s)
2           mark(s)
3           enqueue s in Q
4           while Q != empty
5                   dequeue v from Q
6                   for all neighbors u of v
7                           if (! marked)
8                                   mark(u)
9                                   enqueue u in Q
```

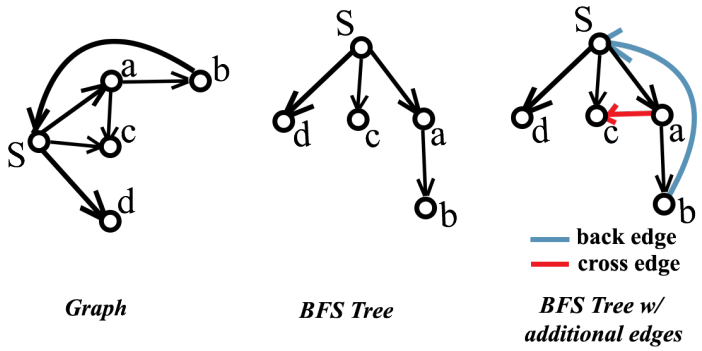
3.2 The BFS Tree and Finding the Shortest Path

The BFS tree marks the distance from the start node to any reachable vertex (this is proven in Theorem 3). We expand the above pseudocode so that such distances could be recorded in a *dist* array:

```
1   BFS(s)
2           mark(s);
3           dist(s) = 0
4           enqueue s in Q
5           while Q != empty
6                   dequeue v from Q
7                   for all neighbors u of v
8                           if (! marked)
9                                   mark(u);
10                                  dist(u) = dist(v) + 1
11                                  enqueue u in Q
```

Lemma 1. *The BFS tree does not contain any forward edges.*

Proof. We prove the lemma by contradiction. If such a *forward edge* (u,v) existed, v would have been put into the queue at an earlier time. The edge (u,v) would have become a *tree edge* when v was dequeued. □



Lemma 2. *If two vertices belong to the same connected component, they are at most $n-1$ steps away from each other following the shortest path.*

Proof. We prove the lemma by contradiction. Given a graph containing n vertices, a shortest path from vertex v to vertex u with more than $n-1$ steps would require traversing certain vertices more than once. Thus, the path must contain one or more cycles. This is a contradiction to the definition of a shortest path, as we could easily remove the cycle(s) and form a shorter path. □

Theorem 3. *For all vertices v, BFS finds the distance from the start vertex s to v.*

Proof. We prove the theorem by induction. We have the following **inductive hypothesis**: at time t_d,

(1) For all vertices at $distance \leq d$, there is a state with their correct distances labelled

(2) The queue contains exactly the vertices at distance d from s

(3) No vertex at $distance > d$ has been marked

Base Case: we have the base case when $d = 0$. At this time, we have only marked and enqueued s itself into the queue, and only the distance from s to itself has been recorded. The inductive hypothesis holds.

Inductive Step: consider any v at distance $d+1$ from s.

Let u be the vertex ahead of v in the path. The path from s to u is of the exact length d. Given the inductive hypothesis (2), we know that u is in the queue at time t_d.

Let t_{d+1} be the time when all vertices in the queue at time t_d have been dequeued.

Based on the inductive hypothesis (1), by time t_d, all vertices with *correct distance* $\leq d$ would have been labelled.

At time t_{d+1}, only vertices with $distance = d+1$ are put into the queue (this is because only vertices with $distance = d$ were present in the queue at time t_d given the inductive hypothesis (2), and only their unmarked neighbours are added into the queue at time t_{d+1}). The algorithm labels all vertices before putting them into the queue. Thus, at time t_{d+1}, all vertices with $distance \leq d+1$ must all have been labelled.

We hereby proved that the first assumption holds

We know v must be in the queue at time t_{d+1} as the inductive hypothesis (3) shows that v (with $distance > d$) was unmarked at time t_d. When u was dequeued, it would add the unmarked v into the queue.

A vertex with $distance < d+1$ must not be in the queue at t_{d+1}. Given the inductive hypothesis (3), such a vertex would have already been marked by time t_d. As the algorithm only adds unmarked vertex into the queue, it will not have the chance to be added back.

A vertex with $distance > d+1$ must not be in the queue at time t_{d+1}. Given the inductive hypothesis (2) and (3), it was neither marked nor presented in the queue at time t_d. Thus, the only chance for it to enter the queue would be at time t_{d+1}. However, this is not possible since it is not the neighbour of any vertex in the queue at time t_d (we've showed that only vertices with $distance = d$ were present in the queue at time t_d).

We hereby proved that the second assumption holds

Based on the inductive hypothesis (3), by time t_d, only vertices with $distance \leq d$ have been marked.

At time t_{d+1}, only vertices with $distance = d+1$ are put into the queue (we've proved this above). The algorithm marks all vertices before putting them into the queue. Thus, at time t_{d+1}, only vertices with $distance \leq d+1$ are marked.

We hereby proved that the third assumption holds

We conclude that at time t_{n-1}, all vertices at $distance \leq n-1$ have been marked and their correct distances from s have been labelled. We proved in Lemma 2 that if a vertex is reachable, its shortest path from s cannot take more than n-1 steps. Thus, the theorem holds for all vertices. □

3.3 Runtime Analysis

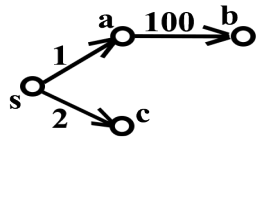
The runtime of BFS is $O(m)$, with m being the number of edges in the graph. This runtime comes from the fact that each vertex only enters the queue once, and each edge is only examined once.

4 Dijkstra's Algorithm

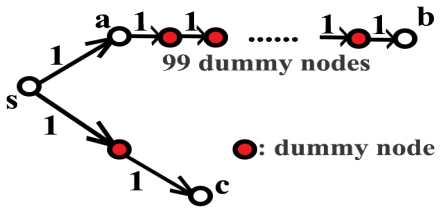
4.1 Limitation of BFS for Weighted Graph

BFS always returns the result with the fewest edges between the start node and the end node. *Thus, it is only optimal when all edges are weighted equally* (such as in the case of an unweighted graph).

When edges have weight, we could still convert the graph into an unweighted one to apply the BFS algorithm. We do so by breaking an edge of weight w into w sub-edges each of weight 1. An example can be seen with the below illustration.



Weighted graph



Transformed unweighted graph

The disadvantages of this approach are obvious. It takes extra memory and time to execute as the number of edges in this case is the sum of all the edge lengths. When this sum is large, we are looking at a large runtime. The time and memory are also wasted in looking at the "dummy nodes" that we don't care about.

4.2 Description of Dijkstra's Algorithm

Dijkstra's algorithm is able to find the shortest path of all vertices from a start vertex for any graph with *non-negative* edges.

Pseudocode:

```
1   DIJKSTRA(s)
2           dist[s] = 0;
3           dist[v] = infinity for all v != s
4           add all vertices to a HEAP H
5           while H != empty
6                   v = EXTRACT_MIN(H)
7                   for all edges(v, u)
8                           if dist[u] > dist[v] + l(u, v)
9                                   //commly refered to as DECREASE KEY[u, dist[u] + l(v, u)]
10                                  dist[u] = dist[v] + l(u, v)
```

We choose to implement Dijkstra's Algorithm using a heap because it has good runtime for both *DECREASE_KEY* and *EXTRACT_MIN* operations. Both operations take $O(\log n)$ time.

For weighted graphs, Dijkstra's Algorithm is more favourable than the described modification on BFS in that,

(1) it avoids creating and analysing all the dummy nodes we don't care about.

(2) it could work with any non-negative numbers including doubles. This is not the case for BFS as we could not create, for example, 99.9 dummy nodes.

4.3 Runtime Analysis

The runtime of Dijkstra's Algorithm is $O((m+n)\log n)$. This is because we have n *EXTRACT_MIN* operations (each vertex is extract only once as an extracted vertex never goes back into the heap) and m *DECREASE_KEY* operations (each *edge(v, u)* is examined only once to see if a *DECREASE_KEY* operation should be applied). The time to construct the heap is $O(n)$, which is ignorable in terms of the Big-O analysis.

Remark 3. *There exist other data structures that could further reduce the runtime. For example, with a Fibonacci heap, the runtime could be reduced to* $O(m+n\log n)$. *This is however beyond the scope of this course.*

4.4 Applications of the Shortest Path Problem

Road Networks: when we want to find the optimal path from an origin to a destination.
Facebook Outreach: when we want to take the shortest number of hops to reach to a famous person on Facebook.

5 Computational Complexity Theory

5.1 Combinatorial Parameters and Numerical Parameters

Definition 1. *Combinatorial parameters refer to the number of the elements given.*

For example, both the number of edges and the number of vertices are combinatorial parameters.

Definition 2. *Numerical parameters refer to the actual value of the elements.*

For example, the weight of each edge is considered as a numerical parameter.

5.2 Pseudo-, Weakly, and Strongly Polynomial

Definition 3. *Pseudo-polynomial time algorithms are polynomial in the numerical and combinatorial parameters.*

For example, the modification to BFS described above in finding the shortest path for weighted graphs is pseudo-polynomial, as the run-time is linear in both the number of edges created and the value of each edge.

Definition 4. *Weakly polynomial time algorithms are polynomial in the input representation.*

Definition 5. *Strongly polynomial time algorithms are polynomial in the combinatorial parameters*

For example, Dijkstra's algorithm is considered as strongly polynomial.

Remark 4. *There is a typically a strong desire to convert pseudo- and weakly-polynomial time algorithms into strongly-polynomial time algorithms (for instance, one of the biggest open problems in theoretical computer science is finding a strongly polynomial time algorithm for solving linear programs—all known approaches run in weakly polynomial time).*

6 Summary

In examining the shortest path problem, this lecture covered some widely-used graph algorithms (DFS, BFS, and Dijkstra's Algorithm), examined their approaches, runtimes, as well as their limitations in dealing with different types of graphs. The lecture also briefly discussed the definition of different polynomial time algorithms.