Search Algorithms

GraphSearch($V, E, s, searchtype$)

```
    // Input: Vertex set V, edge set E, starting vertex s, searchtype as "BFS" or "DFS"
    // Output: Path from s to every vertex in V
    // Running Time: O(|V| + |E|)
 1  for v in V
 2      color[v] ← White
 3      d[v] ← ∞
 4      π[v] ← [] // π[v] is parent of v
 5  color[s] ← Gray; d[s] ← 0; π[s] ← []
 6  if searchtype = "BFS"
 7      Q = queue()
 8  else
 9      Q = stack()
10  Q.push(s)
11  while not Q.empty()
12      u ← Q.pop()
13      for each v in Adj[u]
14          if color[v] = White // v not in Q
15              color[v] ← Gray
16              d[v] ← d[u] + 1; π[v] ← u
17              Q.push(v)
18      color[u] ← Black
```

The recursive version of DFS computes discovery times d and finishing times f for each vertex:

DFS(V, E)

```
    // Input: Vertex set V, edge set E
    // Output: Discovery and finish time for each vertex in V
    // Running Time: O(|V| + |E|)
1   for v in V
2       color[v] ← White
3       π[v] ← [] // π[v]isparentofv
4   time ← 0
5   for u ∈ V
6       if color[u] = White
7           DFS-Visit(u)
```

DFS-VISIT(u)

```
 1  color[u] ← Gray
 2  time ← time + 1
 3  d[u] ← time
 4  for v in Adj(u)
 5        if color[v] = White
 6              π[v] ← u
 7              DFS-VISIT(v)
 8  color[u] ← Black
 9  time ← time + 1
10  f[u] ← time
```

Contents

1 Implicit Representation **3**

2 Group Exercise: BFS and DFS Examples **3**

3 Depth First Search **6**

4 Breadth First Search **7**

1 Implicit Representation

Should we go over implicit representation again? It was squished at the end.

Possible to represent graph *implicitly*: Just implement Adj(u) that returns all neighbors of u.

Example: Infinite grid. Squares are adjacent to top, bottom, left, right. Impossible to write down adjacency matrix, easy to implement Adj.

2 Group Exercise: BFS and DFS Examples

We split up into two or four groups depending on recitation size and did examples on the board.

Breadth First Search

Show how BFS works on the graph in Figure 1 starting from vertex S. You should show the distance from start d that BFS computes for each node and the queue that BFS stores.

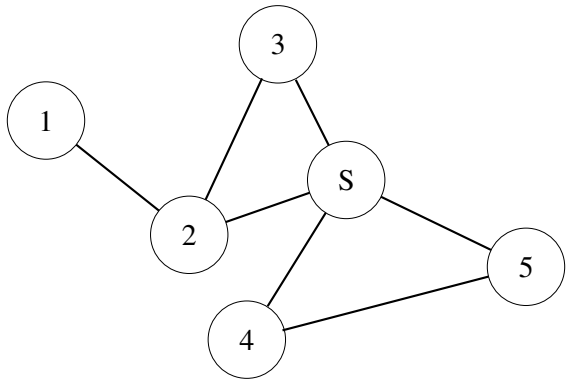

Figure 1: Show how BFS operates on this graph.

Depth First Search

Show how DFS works on the graph in Figure 2 assuming the first vertex you visit is S. You should show the starting time d and finishing time f that DFS computes for each node and the stack that DFS stores. Try to order your adjacency lists so that you show an example of where DFS fails to find the shortest path.

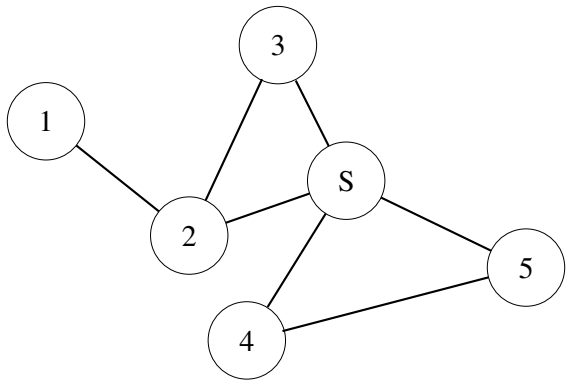

Figure 2: Show how DFS operates on this graph.

3 Depth First Search

Badness: If there is an infinite path and there *is* a solution, DFS may fail to find it

Data structure: stack, last in, first out.

But! The stack can be implemented for us in recursion.

Discovery and Finishing Times: DFS computes *discovery* and *finishing* times for each node. Times are between 1 and $2|V|$ because there is one discover event and one finish event per vertex. For $v \in V$, $d[v] < f[v]$.

The discovery time is the number of vertices visited when v is first noticed. The finishing time is after every path originating at v has been explored. Interesting things:

1. If and only if $[d[u], f[u]]$ disjoint from $[d[v], f[v]]$ then u is not a descendent of v and vice-versa
2. If and only if $[d[u], f[u]]$ is entirely contained in $[d[v], f[v]]$, u is a descendent of v
3. Since u and v *have* to either be descendents of each other or not, one of these conditions will always hold.

Forest of Trees: Depth first search divides the graph into connected *trees*. The whole graph is then a forest of trees. We are talking more about this in class on thursday.

Edge Classifications:

1. *Tree edges*: are edges actually traversed during the search. In other words, (u, v) is a tree edge if v was discovered exploring by (u, v).
2. *Back edges*: connect a descendent to one of its ancestors. For example, if (u, v) is a tree edge then (v, u) is a back edge.
3. *Forward edges*: are edges that connect an ancestor to a descendent but not the edge used to discover the descendent.
4. *Cross edges*: all other edges. Note that in a directed graph, cross edges can occur between vertexes in the same tree since one does not necessarily have to be a descendent of the other.

Undirected graphs have only tree and back edges: If one edge wasn't used for getting to a descendent, we can think of it being used to get to an ancestor since trees in an undirected graph are fully connected.

Correctness: We show that we visit every vertex v in the graph by induction on the shortest path length from the starting vertex s to v.

Base Case: We visit s on the first iteration.

Induction Step: Assume we will visit all vertexes with a shortest path from s of length $k - 1$. Consider a vertex v with shortest path length k. Then there is an edge from some vertex u with a shortest path

from s of length $k-1$ to v. When we visit u, we will examine this edge. If v is not White then we have already visited v. Otherwise, we will visit v when we traverse the (u, v) edge.

Note: This proof *does not* show that DFS finds the shortest path from s to v because it does not. We do not visit vertexes in order of their shortest path from s so it is possible that when we find the edge leading to v that is on the shortest path, v has already been visited. This proof simply shows that we will eventually visit all vertexes.

Running Time: Think of time at each vertex and time at each edge:

Vertexes: For each vertex, we call DFS-Visit once. In DFS-Visit, we change the color of the vertex and look at its descendents. Looking at its descendents involves traversing edges, though, so we consider that in edge work and not here. Therefore, time per vertex is constant.

Edges: We traverse each edge at most once for directed graphs and twice for undirected graphs. Traversing is constant time.

Total: We do constant work per edge on all edges and constant work per vertex on all vertexes for a total running time of $O(|V| + |E|)$.

4 Breadth First Search

Expand all nodes closer to start before any nodes farther from start.

Data structure: queue, first in, first out

Running time analysis: Split into what we do on each vertex and what we do on each edge:

Vertex work: We do a constant amount of work per vertex. We put it into the lists, expand it, etc. We only see each vertex once. Total work: $O(|V|)$

Edge work: For each vertex we expand, we look at all edges originating at it. For a directed graph, we only look at each edge at most once. For an undirected graph, we look at each edge at most twice $O(|V| + |E|)$

Total work: $O(|V| + |E|)$

Correctness proof: Some path length L between s and v, v added to queue at iteration $i = L$ or before. By induction.

Base Case: s is added on first iteration, length 0 from itself

Induction Step: Assume all vertices with some path length $L-1$ are in the queue. At iteration L we have added to the queue all vertices with some path length of L.

If there is a shorter path from s to v, then v is in the queue by induction.

If there is no shorter path from s to v, then there must be some vertex u such that there is a path from s to u of length $L-1$ and a path from u to v of length 1. At iteration L, we expand all vertices with shortest path length $L-1$ from s so we expand u. In expanding u, we add v to the queue.