Contents

1 Depth First Search: Characterizing Nodes and Edges

1.1 Discovery and Finishing Times

Discovery Time: The discovery time $d[v]$ is the number of nodes discovered or finished before first seeing v (call to DFS-Visit).

Finishing Time: The finishing time $f[v]$ is the number of nodes discovered or finished before finishing the expansion of v (return from DFS-Visit).

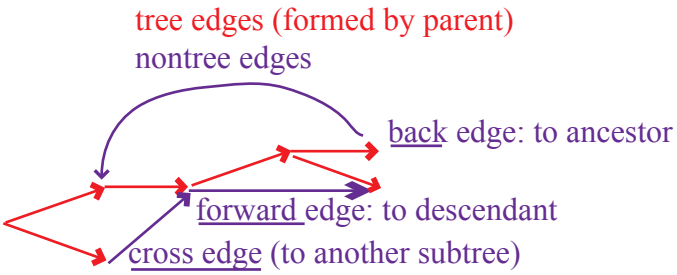

Figure 1: Edge classifications

For two nodes u and v, either $[d[u], f[u]] \subset [d[v], f[v]]$ (or vice versa). or the intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are disjoint.

Proof: If u is a descendent of v in the search then $d[v] < d[u]$. Moreover, we must return from DFS-Visit(u) before we return from DFS-Visit(v) so $f[u] > f[v]$. In this case $[d[u], f[u]] \subset [d[v], f[v]]$.

If u is not a descendent of v and v is not a descendent of u then we must either finish expanding v before we discover u or finish u before discovering v (since if we discover u while expanding v then u is a descendent of v). In this case $[d[u], f[u]]$ and $[d[v], f[v]]$ are disjoint.

1.2 Edge Classifications

When doing a DFS, we think about four types of edges:

- *Tree edges*: Edges traversed in the search. If the edge is (u, v) then, when we first saw edge (u, v), we expanded v. If there is a path of tree edges from w to s then w is an *ancestor* of s and $[d[s], f[s]] \subset [d[w], f[w]]$.
- *Back edges*: A non-tree edge leading from a node u to a node v where there is a path from v to u consisting of tree edges. If there is a back edge (u, v) then v is an ancestor of u so $[d[u], f[u]] \subset [d[v], f[v]]$.
- *Forward edges*: A non-tree or -back edge leading from a node u to a node v where there is a path of tree edges from u to v. Here u is an ancestor of v so $[d[v], f[v]] \subset [d[u], f[u]]$.
- *Cross edges*: Edges that are not tree, back, or forward edges. If (u, v) is a cross edge then $[d[u], f[u]]$ and $[d[v], f[v]]$ will be disjoint.

Examples of edge types are shown in Figure 1

1.3 Node Coloring

During depth first search, a node can be in three states:

- Never been seen (White)

- Currently on the stack (Gray)
- Already popped off the stack and fully expanded (Black)

The color of the node when we see it tells us a lot about the structure of the search to this point. Assume we are expanding node v and considering child u

- u is white:
  - We will expand u right now
  - u is a *descendent* of v
  - v is an *ancestor* of u
  - v can reach u (possibly u cannot reach v)
  - (v, u) is a *tree edge*
  - We discovered u after v and must finish expanding it before we finish expanding v so $[d[u], f[u]] \subset [d[v], f[v]]$.
- u is gray:
  - u is currently on the stack, therefore it is currently being expanded
  - u is an *ancestor* of v
  - v is a *descendent* of u
  - u can reach v **and** v can reach u
  - (v, u) is a *back edge*
  - There is a cycle in the graph involving v and u
  - We started expanding v during the expansion of u so v was discovered after u and must be finished before u: $[d[v], f[v]] \subset [d[u], f[u]]$.
- u is black
  - The graph must be directed
  - v is an ancestor of u **or** u is not an ancestor of v and v is not an ancestor of u
  - (v, u) is either a forward edge or a cross edge (which can be determined by starting and finishing times)
  - Either $[d[u], f[u]] \subset [d[v], f[v]]$ (forward edge) or $[d[u], f[u]]$ and $[d[v], f[v]]$ are disjoint.

Undirected Graphs have only tree and back edges: Let (u, v) be an edge not traversed during DFS. Then when we saw edge (u, v) we must have already pushed v onto the stack (since we do not expand v). Moreover, if we are currently visiting u then clearly we had not visited u when we pushed v onto the stack. Therefore, we cannot yet have finished v because there is a path from v to u (along edge (u, v) among others). Thus v is an ancestor of u and (u, v) is a back edge.

2 Topological Sort

Recall: We must sort vertices such that if u can reach v then u is sorted before v. We run DFS on a DAG and then sort by decreasing finish times. Given Section 1.1, it's clear why it works:

Assume u can reach v. While expanding u, we must see v. When we see v, it is either white, in which case v is a descendent of u and we have $f[v] < f[u]$ or it is black, in which case $f[v] < f[u]$ since $f[u]$ has yet to be assigned. Note that v cannot be gray since the graph is acyclic. Therefore if u can reach v, u will have a higher finishing time than v and be sorted first.

3 Graph Representations and Transformations

3.1 Implicit Representation

Sometimes we don't want to actually build the graph using an adjacency matrix or lists.

Example: An infinite grid. This cannot possibly be constructed... but that doesn't mean you can't view it as a graph! Given a point (x, y) on the grid, we can define its neighbors using an adjacency function

```
ADJ(x, y)
1  return [(x − 1, y), (x + 1, y), (x, y − 1), (x, y + 1)]
```

That's really all we need! Why is this useful? Because you may still care about things like the shortest path in the grid from one point to another. Even though you cannot possibly represent the graph, you can still do Dijkstra's early-termination algorithm on it!

3.2 Graph Transformations

Problem: Assume we add 1 to every weight in a graph. Can this change the shortest path from u to v? What if we multiply every weight by a positive constant a?

Solution: After adding 1 to every weight the path lengths change by:

$$w'(p) = w(p) + |p|$$

where $w'(p)$ is the cost of p after we add 1 to every weight, $w(p)$ is the cost before and $|p|$ is the length of the path. Therefore, adding 1 to every weight can change a path. Assume we have one path from s to v with three edges, each with weight 1 and another path with only one edge of weight 4. Then the shortest path from s to v with unmodified weights is along the 3-edge path (length 3) while the shortest path from s to v with modified weights is along the one edge (length 4).

If we multiply each weight by 1 then

$$w'(p) = aw(p)$$

Therefore if $w(p_1) < w(p_2)$, $w'(p_1) < w'(p_2)$ and the shortest path remains the same.

Fall 2009 Quiz 2 Problem 5: Given an undirected, weighted graph G, we have some subset of edges $R \subset E$ that are considered "rough". Give an algorithm to find the shortest point from a vertex s to all other vertices that uses *at most* one rough edge.

Solution: Create a new *directed* graph $G' = (V', E')$. For every vertex $v \in V$, add two vertices v_r and v_s to V' (so $|V| = 2|V'|$). For each smooth edge (u, v), add the edges (u_s, v_s), (v_s, u_s), (u_r, v_r) and (v_r, u_r) to E' (remember (u, v) was undirected). For each rough edge (u, v), add the edges (u_s, v_r) and (v_s, u_r) to E'. Run Dijkstra on G' from s_s. Then $\delta(s, v) = \min(d[v_r], d[v_s])$.

In this graph, both the smooth (subscript s) and rough (subscript r) clusters have only smooth edges. However, rough edges are only in the graph as a path from the smooth to the rough cluster. Once you have traversed a rough edge to the rough cluster, there is no path back. Therefore, if we start at s_s and finish at v_r, we have traversed exactly one rough edge. If we start at s_s and end at v_s, we traversed no rough edges.

4 Shortest Path Theorems

You can cite any of these during the quiz so you should definitely know them! Knowing their proofs will help you understand why all the shortest path algorithms work. All are proved in CLRS or in the notes from Recitation 15.

- **Subpath Theorem** Let $\{v_1, v_2, ..., v_n\}$ be a shortest path from v_1 to v_n. Then any subsequence of this path from v_i to v_j is a shortest path from v_i to v_j.
- **Triangle Inequality** $\forall u, v, x \in V$, we have $\delta(u, v) \leq \delta(u, x) + \delta(x, v)$.
- **Upper Bound Property** We always have $d[v] \geq \delta(s, v)$ and if we ever find $d[v] = \delta(s, v)$, $d[v]$ never changes.
- **Path Relaxation Property** Assume we have a graph G with no negative cycles. Let $p = \langle v_0, v_1, ..., v_j \rangle$ be a shortest path from v_0 to v_j. Any sequence of calls to RELAX that includes, in order, the relaxations of $(v_0, v_1), (v_1, v_2), ..., (v_{j-1}, v_j)$ produces $d[v_j] = \delta(v_0, v_j)$ after all of these relaxations and at all times afterwards. Note that this property holds regardless of what other relaxation calls are made before, during, or after these relaxations.

5 Bellman-Ford

5.1 Things to Know

- You cannot be sure that $d[v] = \delta(s, v)$ until the algorithm has finished.
- Bellman-Ford returns FALSE if it finds a negative cycle.
- The running time is $O(|V||E|)$. This is (much) worse than Dijkstra's algorithm.
- The running time of Bellman-Ford on a DAG is only $O(|E| + |V|)$. See below.
- The proof for why this works is in the book and also in the notes for Recitation 15. Knowing this proof is an excellent way of understanding how the algorithm works.

5.2 On a DAG

Bellman-Ford takes forever because we must relax all edges for every possible path in order. However, on a DAG, we can figure out the order of relaxation easily! Here's what you do:

1. Topologically sort the graph $O(|E| + |V|)$
2. Run *one* iteration of Bellman-Ford taking the vertices in topological order $O(|E| + |V|)$
3. Note: We know there are no cycles, so we don't need to do the negative cycle check at the end!

Assume $p = \langle (s = v_0, v_1), (v_2, v_3), ..., (v_{n-1}, v_n) \rangle$ is a shortest path. Now consider edge (v_i, v_{i+1}). Then v_i is sorted after all $v_{j<i}$ and before all $v_{j>i}$ so we relax all edges $(v_0, v_1), ..., (v_{i-1}, v_i)$ *before* (v_i, v_{i+1}) and all edges $(v_{i+1}, v_{i+2}, ..., (v_{n-1}, v_n)$ after (v_i, v_{i+1}). So by the path relaxation property (look it up in CLRS or Recitation 15), we will report the shortest path!

6 Dijkstra's Algorithm

6.1 Things to Know

- When a vertex v pops off the priority queue, $d[v] = \delta(s, v)$, the shortest path from s to v
- When a vertex v pops off the priority queue, no vertex u will pop off the priority queue at any point later in the algorithm with $d[u] < d[v]$.
- Once a vertex v has popped off the queue, we will never change $d[v]$.
- The running time of Dijkstra depends on your priority queue implementation. If you use a Fibonnacci heap, the running time is $O(|E| + |V| \log |V|)$. If you use a binary heap, the running time is $O((|E| + |V|) \log |V|)$.
- See the paper or Recitation 17 for speedups.

Fall 2008 Final Problem 9: Assume we have a directed graph $G = (V, E)$ with non-negative edge weights. We wish to find the shortest path from a vertex $s \in V$ to a vertex $t \in V$ with one caveat: While traversing a path from s to t you may set one edge weight of your choosing to zero. Given an algorithm for finding the shortest path with this caveat.

Solution: First run Dijkstra's algorithm to find the shortest path from s to every other vertex in the graph. Then run Dijkstra's on the transpose graph to find the shortest path from every vertex in the graph to t. Now iterate through the edges (u, v) calculating the path cost from s to t if we set that edge to zero weight:

$$w(s \to t) = \min(\delta(s, t), \delta(s, u) + \delta(u, v))$$

Choose to set the edge to zero that minimizes $w(s \to t)$. Note that if the path cost from s to t is non-zero then $w(s \to t)$ should be less than $\delta(s, t)$.

7 Example Problems

Fall 09 Quiz 2 Problem 3: Consider a graph $G = (V, E)$ that has both directed and undirected edges. There are no cycles in G that use only directed edges. Give an algorithm to assign each undirected edge a direction so that the completely directed graph has no cycles.

Solution: First topologically sort the graph using only the directed edges. Create an array so that for each vertex you store its number in the short. Then, for each undirected edge, draw the edge in the direction that goes from the vertex with the lower sort number to the higher sort number.

Fall 2008 Problem 3a: Given a directed graph G, you would like to get from s to t stopping at u if not too inconvient where "too inconvenient" means the shortest path that stops at u is more than 10% longer than the shortest path from s to t. Give an algorithm for returning the shortest path from s to t that stops at u if convenient.

Solution: Run Dijkstra's algorithm once from s and once from u. The shortest path from s to t is found in doing Dijkstra's algorithm from s. The shortest path from s to t through u is the shortest path from s to u plus the shortest path from u to t.

8 More Example Problems

Problem: Give an algorithm to find the shortest path containing an *even* number of edges in the directed graph with non-negative weights. Your algorithm should have the same running time as Dijkstra's.

Solution: Create a new graph G' as follows: for each $v \in V$, add two vertices v_{red} and v_{blue} to V' (so $|V'| = 2|V|$). Then for every edge (u, v) in E, add the edges $(u_{\text{red}}, v_{\text{blue}})$ and $(u_{\text{blue}}, v_{\text{red}})$ to E' (so $|E'| = 2|E|$). Run Dijkstra on G' starting at s_{red} and report the distance from s to v as the distance from s_{red} to v_{red}.

Every time you traverse an edge in G' you change the color of your cluster. Therefore, if you begin at a red vertex and end at a red vertex you must have traversed an even number of edges. Prove the correctness rigorously yourself as an exercise in how Dijkstra works!

Critical Edges: You are given a graph $G = (V, E)$ a weight function $w : E \to \Re$, and a source vertex s. Assume $w(e) \geq 0$ for all $e \in E$.

We say that an edge e is $u$pwards critical if by increasing $w(e)$ by any $\epsilon > 0$ we increase the shortest path distance from s to some vertex $v \in V$.

We say that an edge e is downwards critical if by decreasing $w(e)$ by any $\epsilon > 0$ we decrease the shortest path distance from s to some vertex $v \in V$ (however, by definition, if $w(e) = 0$ then e is not downwards critical, because we can't decrease its weight below 0).

1. Claim: an edge (u, v) is downwards critical if and only if there is a shortest path from s to v that ends at (u, v), and $w(u, v) > 0$. Prove the claim above.

Solution: First, note that if (u, v) is on any shortest path, then because subpaths of shortest paths are shortest paths, (u, v) will also be on a shortest path to v.

Second, we prove that (u, v) is downwards critical implies (u, v) is on the shortest path from s to v.

Proof by contradiction: Assume (u, v) is downwards critical, but it is not on the shortest path from s to v. Then $\delta[s, v] < \delta[s, u] + w(u, v)$, so let $\epsilon = (\delta[s, u] + w(u, v) - \delta[s, v])/2$ is positive. If we decrease $w(u, v)$ by ϵ, we'll only be changing the cost of the paths to v going through (u, v), so the cost of the minimum path will stay the same. By the choice of ϵ, the best path going through (u, v) will still cost more than the minimum path. So the minimum path cost doesn't change when $w(u, v)$ is decreased by ϵ. Contradiction.

Third, we prove that (u, v) is on the shortest path from s to v implies (u, v) is downwards critical.

If (u, v) is on a shortest path to v, then decreasing its weight by any $\epsilon > 0$ decreases the cost of that path. We know that no other path through v had a lower cost than $w(u, v)$, so the path containing (u, v) is still the shortest path to v. So by decreasing the weight of (u, v), the weight of the shortest path to v is decreased, which means (u, v) is downwards critical.

2. Make a claim similar to the one above, but for upwards critical edges, and prove it.

Solution: Claim: (u, v) is upwards critical if and only if all the shortest paths from s to v end at (u, v).

First, we again start by noting that if (u, v) is on all shortest paths to any particular node, then it must also be on all shortest paths to v.

Second, we prove that if (u, v) is upwards critical then all the shortest paths from s to v end at (u, v).

If (u, v) is upwards critical, then increasing $w(u, v)$ by $\epsilon > 0$ must increase the cost of all shortest paths from s to v, otherwise the minimum cost to get from s to v would stay the same. Increasing $w(u, v)$ only impacts the paths containing (u, v), therefore (u, v) must be contained on all shortest paths to v. Since all the edges have positive weights, (u, v) must be the last edge on any the shortest path from s to v.

Third, we prove that if all the shortest paths from s to v end at (u, v) then (u, v) is upwards critical.

If all the shortest paths from s to v include (u, v), then increasing $w(u, v)$ by any $\epsilon > 0$ increases the cost of all these paths. Therefore, the minimum cost to get from s to v is increased, so (u, v) is upwards critical.

3. Using the claims from the previous two parts, give an $O(E \log V)$ time algorithm that finds all downwards critical edges and all upwards critical edges in G.

Solution: Run Dijkstra using binary heaps as a priority queue (binary trees or Fibonacci heaps are also acceptable data structures here). Save the results in $d[v]$ and $\pi[v]$.

Iterate through all edges, and report an edge (u, v) as downwards critical if $d[u] + w(u, v) = d[v]$. This is correct because the edges satisfying the condition must belong to the shortest paths from s to v. While doing this, compute $dc[v]$ = the number of downwards critical edges coming into v.

Iterate through all the vertices, and report $(\pi[v], v)$ as upwards critical if $dc[v] = 1$. This is correct because the check implies that $(\pi[v], v)$ is the only edge coming into v that is on a shortest path from s to v.

Running time analysis: all vertices are reachable from v, so it must be that $V = O(E)$. Then the running time of Dijkstra is $O((V + E)logV) = O(ElogV)$. Reporting downwards critical edges takes $O(E)$, because we do $O(1)$ work per iteration over all the edges. Reporting upwards critical edges takes $O(V)$, because we do $O(1)$ work per iteration over all the vertices. So the total running time is $O(ElogV + E + V) = O(ElogV)$

9 Radix and Counting Sorts

9.1 Counting Sort

Counting sort can beat comparison sort bound *because* it doesn't use comparisons. In order not to use comparisons, we must have a little bit of "extra" knowledge. Namely: given an array A to sort, we need to be able to map every element that might appear in A uniquely to integers in $[1, k]$ where k is a small integer.

Input: A array to be sorted

Output: B sorted array of elements of A

Pass 1: Create array C of length k. $C[i]$ stores the number of times i appears in A. For each i, $C[A[i]] = C[A[i]] + 1$

Pass 2: For each entry i in C, put $C[i]$ i's in B.

This takes $O(n + k)$.

Problem: B is not *stably* sorted. Two equal keys may swap their relative orders. We would like to avoid that.

New algorithm: Create C', which stores in $C'[i]$ number of numbers in A less than or equal to i. Fill in C' after filling in C just by keeping running total.

Now, for $j = length[A]$ downto 1, place $A[j]$ at $C'[A[j]]$ in B and decrement $C'[A[j]]$. Now the sorting is stable.

Example: We know we are sorting numbers from 1 to 8

$$\begin{aligned} A &= [2, 7, 5, 3, 5, 4] \\ C &= [0, 1, 1, 1, 2, 0, 1, 0] \\ C' &= [0, 1, 2, 3, 5, 5, 6, 6] \end{aligned}$$

Stepping through the second pass (0 in B indicates nothing there yet):

1.
$$\begin{aligned} B &= [0, 0, 0, 0, 0, 0] \\ C' &= [0, 1, 2, 3, 5, 5, 6, 6] \end{aligned}$$

2.
$$\begin{aligned} B &= [0, 0, 4, 0, 0, 0] \\ C' &= [0, 1, 2, 2, 5, 5, 6, 6] \end{aligned}$$

3.

$$\begin{aligned} B &= [0,0,4,0,5,0] \\ C' &= [0,1,2,2,4,5,6,6] \end{aligned}$$

4.

$$\begin{aligned} B &= [0,3,4,0,5,0] \\ C' &= [0,1,1,2,4,5,6,6] \end{aligned}$$

5.

$$\begin{aligned} B &= [0,3,4,5,5,0] \\ C' &= [0,1,1,2,3,5,6,6] \end{aligned}$$

6.

$$\begin{aligned} B &= [0,3,4,5,5,7] \\ C' &= [0,1,1,2,3,5,5,6] \end{aligned}$$

7.

$$\begin{aligned} B &= [2,3,4,5,5,7] \\ C' &= [0,0,1,2,3,5,5,6] \end{aligned}$$

9.2 Radix Sort

Digit-by-digit sort of list of numbers. Sort on least-significant digit first using a *stable* sort.

Why least significant? Example: Most significant

33		33		52
55	$\rightarrow$	55	$\rightarrow$	33
52		52		55

Oops!

Why do we need a stable sort? Because we don't want to mess up orderings caused by earlier digits in later digits!

Example: Sort 33, 55, 52

33		52		33
55	$\rightarrow$	33	$\rightarrow$	52
52		55		55

We can only guarantee that 52 comes before 55 if we can guarantee the sort on the second digit is stable!

Running Time: Sorting n words of b bits each: Each word has b/r 2^r-base digits. Example: 32-bit word is has 4 8-bit digits. Each counting sort takes $O(n + 2^r)$, we do b/r sorts. Choose $r = \log n$, gives running time $O(nb/\log n)$.

Correctness: By induction on number of digits. Do it for practice!