Graph Theory Review

Samuel Hsiang

October 16, 2015

Enjoy the algorithm dump. This text is adapted from my text, which can be found at the link below. I like it better because it has color :D

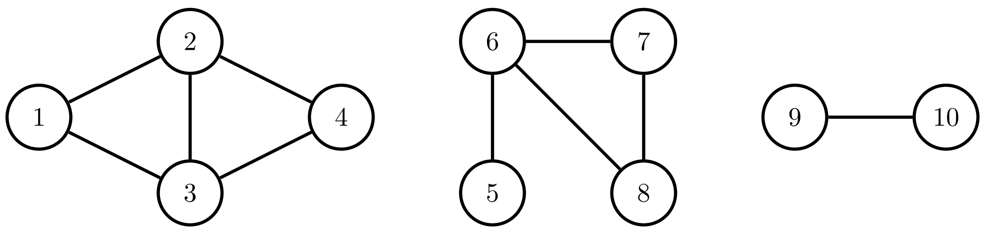
`https://www.dropbox.com/s/z2lur71042pjaet/guide.pdf?dl=0`

The majority of this *should* be review for you, though we may go more in depth into specific topics in this handout should I feel we would benefit from it.

Take a minute to skim this handout and refresh your memory. Ask me if you have any questions. **When you're ready, begin practicing problems on Codeforces.**

1 Connected Components

A *connected component* of an undirected graph is a subgraph such that, for any two vertices in the component, there exists a path from one to the other. The diagram illustrates three connected components of a graph.

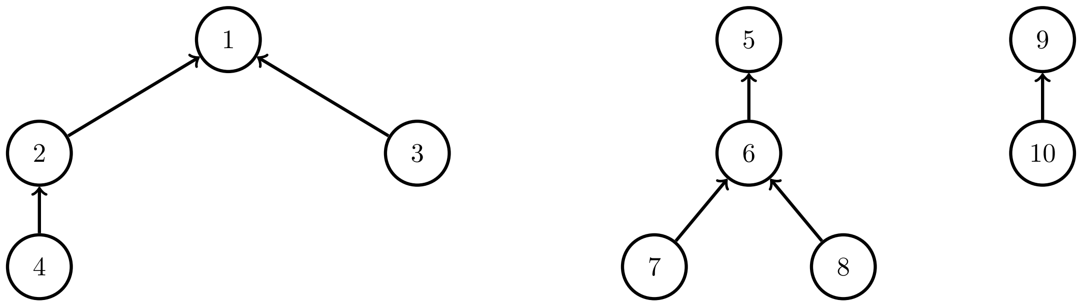


1.1 Flood Fill

Really any kind of search method solves the undirected graph connected components problem. We could use recursion with a depth-first search. To avoid using the run-time stack, we could use a queue to perform a breadth-first search. Both of these run in $O(E + V)$ time. I would recommend in general to use the BFS.

1.2 Union-Find (Disjoint Set Union)

We maintain a pointer to another vertex it's connected to, forming a *forest*, or collection of trees. To check whether two elements are in the same component, simply trace the tree up to the root by jumping up each pointer.



The idea of a pointer can easily be stored within an array.

-1	1	1	2	-1	5	6	6	-1	9
1	2	3	4	5	6	7	8	9	10

We want to support two operations: $find(v)$, which returns the root of the tree containing v, and $union(u, v)$, which merges the components containing u and v. This second operation is easy given the first; simply set the pointer of $find(u)$ to be $find(v)$.

But a problem quickly arises – the $find$ operation threatens to become linear. There are two simple things we can do to optimize this.

The first is to always add the shorter tree to the taller tree, as we want to minimize the maximum height. An easy heuristic for the height of the tree is simply the number of elements in that tree. We can keep track of the size of the tree with a second array. This heuristic is obviously not perfect, as a larger tree can be shorter than a smaller tree, but it turns out with our second optimization that this problem doesn't matter.

The second fix is to simply assign the pointer associated with v to be $find(v)$ at the end of the $find$ operation. We can design $find(v)$ to recursively call $find$ on the pointer associated with v, so this fix sets pointers associated with nodes along the entire chain from v to $find(v)$ to be $find(v)$. These two optimizations combined make the $union$ and $find$ operations $O(\alpha(V))$, where $\alpha(n)$ is the inverse Ackermann function, and for all practical values of n, $\alpha(n) < 5$.

Algorithm 1 Union-Find

```
function FIND($v$)
    if $v$ is the root then
        return $v$
    $parent(v) \leftarrow$ FIND($parent(v)$)
    return $parent(v)$
function UNION($u$, $v$)
    $uRoot \leftarrow$ FIND($u$)
    $vRoot \leftarrow$ FIND($v$)
    if $uRoot = vRoot$ then
        return
    if $size(uRoot) < size(vRoot)$ then
        $parent(uRoot) \leftarrow vRoot$
        $size(vRoot) \leftarrow size(uRoot) + size(vRoot)$
    else
        $parent(vRoot) \leftarrow uRoot$
        $size(uRoot) \leftarrow size(uRoot) + size(vRoot)$
```

2 Shortest Path

Assign nonnegative weights to each of the edges, where the weight of the edge (u, v) represents the distance from u to v. This graph can be either directed or undirected.

2.1 Dijkstra

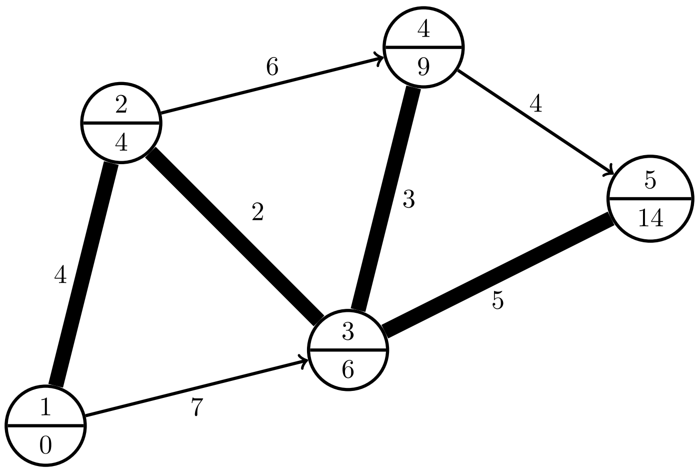
Dijkstra's algorithm solves the single-source shortest path problem. From any vertex, we can compute the shortest path to each of the remaining vertices in the graph. The two formulations of Dijkstra's algorithm run in $O(V^2)$ or $O(E \log V)$ time, whichever one suits us better. Note that it is possible to do better than $O(E \log V)$ using a Fibonacci heap. The former works nicely on dense graphs, as $E \approx V^2$, while the latter works better on sparse graphs, as $E \approx V$.

For every vertex v in the graph, we keep track of the shortest known distance $dist(v)$ from the source to v, a boolean $visited(v)$ to keep track of which nodes we "visited," and a pointer to the previous node in the shortest known path $prev(v)$ so that we can trace the shortest path once the algorithm finishes.

Dijkstra iteratively "visits" the next nearest vertex, updating the distances to that vertex's neighbors if necessary.

Algorithm 2 Dijkstra

```
for all vertices $v$ do
    $dist(v) \leftarrow \infty$
    $visited(v) \leftarrow 0$
    $prev(v) \leftarrow -1$
$dist(src) \leftarrow 0$
while $\exists v$ s.t. $visited(v) = 0$ do
    $v \equiv v$ s.t. $visited(v) = 0$ with min $dist(v)$
    $visited(v) \leftarrow 1$
    for all neighbors $u$ of $v$ do
        if $visited(u) = 0$ then
            $alt \leftarrow dist(v) + weight(v, u)$
            if $alt < dist(u)$ then
                $dist(u) \leftarrow alt$
                $prev(u) \leftarrow v$
```



The slow part of the $O(V^2)$ formulation is the linear search for the vertex v with the minimum $dist(v)$. We happen to have a data structure that resolves this problem – a binary heap. The main problem with using the standard library heap is having repeated vertices in the heap. We could just ignore this problem and discard visited vertices as they come out of the heap. Alternatively, we could choose never to have repeated vertices in the heap. To do this, we need to be able to change the value of the distances once they are already in the heap, or *decrease-key*. This is a pretty simple function to add, however, if you have a heap already coded. Either way, we achieve $O(E \log V)$, as we do $E + V$ updates to our heap, each costing $O(V)$.

2.2 Floyd-Warshall

Dijkstra is nice when we are dealing with edges with nonnegative weights and are looking for the distances from one vertex to all the others. Floyd-Warshall solves the shortest path problem for all pairs of vertices in $O(V^3)$ time, which is faster than V single-source Dijkstra runs on a dense graph. Floyd-Warshall works even if some edge weights are negative but not if the graph has a negative cycle.

Algorithm 3 Floyd-Warshall

```
for all vertices $v$ do
    $dist(v, v) = 0$
for all edges $(u, v)$ do
    $dist(u, v) = weight(u, v)$
for all vertices $k$ do
    for all vertices $i$ do
        for all vertices $j$ do
            if $dist(i, j) > dist(i, k) + dist(k, j)$ then
                $dist(i, j) \leftarrow dist(i, k) + dist(k, j)$
```

2.3 Bellman-Ford

Bellman-Ford is a single-source $O(VE)$ shortest path algorithm that works when edge weights can be negative. It is preferable to Floyd-Warshall when the graph is sparse and we only need the answer for one source. Like Floyd-Warshall, the algorithm fails if the graph contains a negative cycle, but the algorithm is still useful for detecting negative cycles.

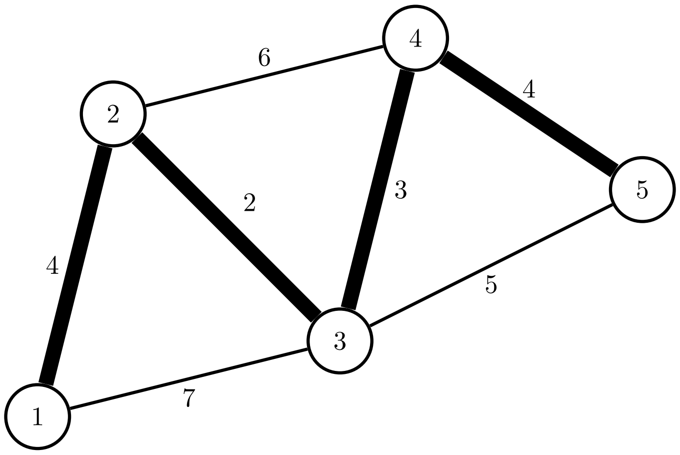
The idea here is the shortest path, assuming no negative cycles, has length at most $V - 1$.

Algorithm 4 Bellman-Ford

```
for all vertices $v$ do
    $dist(v) \leftarrow \infty$
    $prev(v) = \leftarrow -1$
$dist(src) \leftarrow 0$
for $i \equiv 1, V - 1$ do
    for all edges $(u, v)$ do
        if $dist(u) + weight(u, v) < dist(v)$ then
            $dist(v) \leftarrow dist(u) + weight(u, v)$
            $prev(v) \leftarrow u$
for all edges $(u, v)$ do                                    ▷ check for negative cycles
    if $dist(u) + weight(u, v) < dist(v)$ then
        negative cycle detected
```

3 Minimum Spanning Tree

Consider a connected, undirected graph. A *spanning tree* is a subgraph that is a tree and contains every vertex in the original graph. A *minimum spanning tree* is a spanning tree such that the sum of the edge weights of the tree is minimized. Finding the minimum spanning tree uses many of the same ideas discussed earlier.



3.1 Prim

Prim's algorithm for finding the minimum spanning tree is very similar to Dijkstra's algorithm for finding the shortest path. Like Dijkstra, it iteratively adds a new vertex at a time to build a tree. The only difference is $dist(v)$ stores the shortest distance from *any* visited node instead of the source.

Algorithm 5 Prim

```
for all vertices $v$ do
    $dist(v) \leftarrow \infty$
    $visited(v) \leftarrow 0$
    $prev(v) \leftarrow -1$
$dist(src) \leftarrow 0$
while $\exists v$ s.t. $visited(v) = 0$ do
    $v \equiv v$ s.t. $visited(v) = 0$ with min $dist(v)$
    $visited(v) \leftarrow 1$
    for all neighbors $u$ of $v$ do
        if $visited(u) = 0$ then
            if $weight(v, u) < dist(u)$ then
                $dist(u) \leftarrow weight(v, u)$
                $prev(u) \leftarrow v$
```

The proof of correctness is left as an exercise. The complexity of this algorithm depends on how the minimum unvisited vertex is calculated. Using the same approaches as Dijkstra, we can achieve $O(V^2)$ or $O(E \log V)$.

3.2 Kruskal

While Prim greedily adds vertices to the tree, Kruskal's algorithm greedily adds edges. It iterates over all the edges, sorted by weight. We need to watch out for adding a cycle, breaking the tree structure, which means we need to keep track of each vertex's connected component. If an edge connects two vertices from the same connected component, we don't want to add it to our tree. However, we have a union-find algorithm that works perfectly for this.

Algorithm 6 Kruskal

```
for all edges $(u, v)$ in sorted order do
    if FIND($u$) $\neq$ FIND($v$) then
        add $(u, v)$ to spanning tree
        UNION($u, v$)
```

This algorithm requires a sort of the edges and thus has complexity $O(E \log E) = O(E \log V)$.

4 Eulerian Tour

An *Eulerian tour* of a graph is a path that traverses every edge exactly once. If the tour ends exactly where it started, it is called an *Eulerian circuit*. A graph has an Eulerian circuit if it is connected and every vertex has even degree. A graph has an Eulerian path if it is connected and all vertices but exactly two have even degrees. The mathematical proofs for these graph properties hinge on the idea that removing a cycle from the graph maintains the Eulerian property. We construct an Eulerian tour by appealing to this idea.

Algorithm 7 Eulerian Tour

```
function FINDTOUR($v$)
    while $v$ has a neighbor $u$ do
        delete edge $(v, u)$
        FINDTOUR($u$)                ▷ FINDTOUR($u$) must trace a circuit back to $v$
    add $v$ to tour
```

It is not preferable to use the run-time stack; we can use our own stack if necessary.

If the graph contains an Eulerian circuit, we call this function on any vertex we like. If it contains an Eulerian path, we call this function on one of the vertices with odd degree.