Lecture 12

- Algorithms on graphs
- Breadth first, depth first searches
- Shortest path in unweighted graphs
- Greedy algorithms
- Djikstra’s algorithm for shortest path in weighted graphs

Reading: Weiss, Chapter 9, 10
Shortest path problems

- Suppose graph vertices represent computers, and graph edges represent network links between computers, and edge weights represent communications times...
  - ... then a shortest-path algorithm can find the fastest route to send email between one computer and another

- Suppose graph vertices represent cities, and graph edges represent airline routes between cities, and edge weights represent travel costs ...
  - ... then a shortest-path algorithm can find the cheapest route to travel by air between one city and another

- Many, many other examples...

- We will look at shortest-path algorithms in unweighted and weighted graphs

- These algorithms will find the shortest path from a “source” or “start” vertex to every other vertex in the graph

- (Often you may want only the shortest path from a source vertex to one particular destination vertex... but there is no known algorithm to do that with better worst-case time cost than a good algorithm to find shortest path to *every* other vertex!)
The unweighted shortest path problem

• Input: an unweighted directed graph \( G = (V,E) \) and a “source vertex” \( s \) in \( V \)

• Output: for each vertex \( v \) in \( V \), a representation of the shortest path in \( G \) that starts at \( s \) and ends at \( v \)

• What is the best approach to this problem?
Search in graphs

• The shortest-path problem is a search problem in a graph: starting at a vertex $s$, you are searching for the shortest path to another vertex $v$

• When searching in a graph, three important approaches are:
  • depth-first search
  • breadth-first search
  • best-first search

• These approaches can be applied to many different search problems

• We’ll consider applying depth-first and breadth-first to the unweighted shortest-path problem. (Best-first will arise in the weighted shortest-path case)
Breadth-first search

- Breadth-first search in a graph visits a node; and then all the nodes adjacent to that node; then all the nodes adjacent to those nodes; etc.

- A level-order traversal of a tree is a breadth-first search:
Depth first search

- Depth first search in a graph visits a node; and then recursively does depth-first search from each of the nodes adjacent to that node.

- A pre-order traversal of a tree is a depth-first search:
Depth-first search for shortest paths

- Consider searching for shortest paths with start vertex V0 in this unweighted graph, using depth-first search:
Depth-first search for shortest paths: frame 1

- Immediately we have the shortest path from V0 to V0, following 0 edges.

- Continue the depth-first search, following the edge to V1
Depth-first search for shortest paths: frame 2

- We have found a path to V1, of length 1.

- Now continuing the depth-first search, there are two choices. Suppose we follow the edge to V3...
Depth-first search for shortest paths: frame 3

- We have found a path to V3, of length 2.

- Continuing the depth-first search, we follow edges to V2, and then to V4...
Depth-first search for shortest paths: frame 4

• We have found a path to V2, of length 3, and a path to V4 of length 4.

• There are no untravelled edges to follow from V4. So we backtrack to the last node visited with an untravelled edge: V1, and follow the edge to V4
Depth-first search for shortest paths: frame 5

• Following the other edge from V1 to V4, we find a path to V4 of length 2, which is shorter than the one found previously (going through V3 and V2)

• Now all edges have been traversed, and we have visited all the nodes.

• However, note that if there were nodes reachable from V4, we would have to search them again, now having found a shorter path to V4!

• We can eventually find all shortest paths this way, but any simple implementation of this idea could be very inefficient
Breadth-first search for shortest paths

- Consider searching for shortest paths with start vertex $V_0$ in this unweighted graph, using breadth-first search:
Breadth-first search for shortest paths: frame 1

- First visit V0, giving the shortest path from V0 to V0, following 0 edges.

- Continue the breadth-first search, visiting all the unvisited nodes adjacent to V0. (There is only one: V1)
Breadth-first search for shortest paths: frame 2

- We have found a path to V1, of length 1. We can be sure this is the shortest path to V1.

- Now visit continuing the breadth-first search, there are two unvisited vertices adjacent to V1. They can be visited in any order, as long as they are visited before any other vertices.
Breadth-first search for shortest paths: frame 3

- We have found paths to V3 and V4, of length 2. We can be sure these are the shortest paths to these vertices.

- Continuing the breadth-first search, we follow edges to vertices adjacent to V3 and to V4. There is only one, V2.
Breadth-first search for shortest paths: frame 4

- We have found a path to V2, of length 3

- Now we have visited all the vertices in the graph, and have found the shortest path from V0 to each of them
Comparing depth- and breadth-first search for shortest path

• Either depth-first or breadth-first search will traverse the graph, visiting all nodes. But breadth-first search is better for the shortest-path problem in graphs

• In depth-first search:
  • you need to keep track of which edges have been traversed
  • you need to know which node to backtrack to when you reach a “dead end” (this can be done with a stack)
  • when you visit a node, you may not be sure if you have found the shortest path or not; you may find a shorter path later. If you do, you will have to repeat the depth-first search from that node! These repetitions can lead to exponentially many steps

• In breadth-first search:
  • you need to keep track of which nodes have been visited
  • you need to make sure you visit all nodes adjacent to a node before visiting any others (this can be done with a queue)
  • the first time you visit a node, you can be sure you have found the shortest path to it, so that node does not need to be visited again
Unweighted shortest path

- Input: an unweighted directed graph $G = (V, E)$; and a source vertex $s$ in $V$

- Output: for each vertex $v$ in $V$, a representation of the shortest path in $G$ that starts at $s$ and ends at $v$

- Ordinary breadth-first search solves this problem efficiently: worst-case time cost $O(|E| + |V|)$
Breadth-first search for unweighted shortest path: basic idea

• By distance between two nodes $u, v$ we mean the number of edges on the shortest path between $u$ and $v$. Now:

• Start at the start vertex $s$. It is at distance 0 from itself, and there are no other nodes at distance 0

• Consider all the nodes adjacent to $s$. These all are at distance at most 1 from $s$ (maybe less than 1, if $s$ has an edge to itself; but then we would have found a shorter path already) and there are no other nodes at distance 1

• Consider all the nodes adjacent to the nodes adjacent to $s$. These are all at distance at most 2 from $s$ (maybe less than 2; but then we would have found a shorter path already) and there are no other nodes at distance 2

• ... and so on. In this breadth-first search, as soon as we visit a node in the graph, we know the shortest path from $s$ to it; and so by the time we have visited all the nodes in the graph, we know the shortest path from $s$ to each of them
Unweighted shortest path: sketch of algorithm

• The basic idea is a breadth-first search of the graph, starting at source vertex $s$
• Initially, give all vertices in the graph a distance of INFINITY
• Start at $s$; give $s$ distance $= 0$
• Consider the vertices in the adjacency list of $s$
  • these all have a distance 1 from $s$, so give each of them distance $= 1$ unless a shorter distance (i.e. 0) has already been found
  • there is no shorter path from $s$ to any of these vertices!
• Now consider the vertices in the adjacency lists of vertices at distance 1 from $s$
  • these all have a distance 2 from $s$, so give each of them distance $= 2$ unless a shorter distance (i.e. 1 or 0) has already been found
  • there is no shorter path from $s$ to any of these vertices!
• Continue in this fashion until all vertices have been visited
• This finds the lengths of the shortest paths. Representing the actual sequence of vertices on the paths is easy to do, as we will see momentarily
Unweighted shortest path: auxiliary data structures

- Maintain a sequence (e.g. an array) of vertex objects, indexed by vertex number

  - Vertex objects contain these 2 fields (and others):
    - “dist”: the cost of the best (least-cost) path discovered so far from the start vertex to this vertex (initially INFINITY)
    - “prev”: the vertex number (index) of the previous node on that best path

- Maintain a queue, containing vertices, or vertex numbers (similar to use of queue in level-order traversal of a tree)

  - Initially the queue contains only the start vertex $s$, with dist field 0
  - A vertex is “visited” when it is enqueued: then shortest path to it is known
  - When a vertex $v$ is dequeued, vertices on $v$’s adjacency list that have not yet been visited are enqueued, and given a distance $= 1 + v$’s distance
  - When the queue is empty, the algorithm terminates
Unweighted shortest path: more algorithm details

- How do you tell if a vertex has been visited before or not?
  - Check to see if its distance is INFINITY. If it is, it has not been visited yet

- How do you figure out the actual paths, not just their costs?
  - When you enqueue a vertex $w$, it is because it is on the adjacency list of a vertex $v$, which is the previous vertex on the shortest path from the source vertex to $w$
  - You give $w$ a distance equal to the distance to $v$, plus 1; and you also set the previous-node field of $w$ to be $v$
  - So, the entire shortest path from source to $w$ can be found by tracing back these previous node indices to the source vertex

- What is the time complexity of this algorithm?
  - Each vertex is visited exactly once: so there are $O(|V|)$ enqueue and dequeue operations
  - When you dequeue a vertex, you traverse the adjacency list for that vertex; these traversals total $O(|E|)$ steps
  - So, the algorithm has total worst-case time cost $O(|V| + |E|)$
Unweighted shortest path: an example

- We will find shortest paths in this graph, with source vertex V0
Do it!

- The array of vertices, which include dist and prev fields (initialize dist to ‘INFINITY’):

  \[
  \begin{align*}
  V_0: & \quad \text{dist} = \quad \text{prev} = \quad \text{adj: } V_1 \\
  V_1: & \quad \text{dist} = \quad \text{prev} = \quad \text{adj: } V_3, V_4 \\
  V_2: & \quad \text{dist} = \quad \text{prev} = \quad \text{adj: } V_0, V_5 \\
  V_3: & \quad \text{dist} = \quad \text{prev} = \quad \text{adj: } V_2, V_5, V_6 \\
  V_4: & \quad \text{dist} = \quad \text{prev} = \quad \text{adj: } V_1, V_6 \\
  V_5: & \quad \text{dist} = \quad \text{prev} = \quad \text{adj: } \\
  V_6: & \quad \text{dist} = \quad \text{prev} = \quad \text{adj: } V_5
  \end{align*}
  \]

- The queue (give source vertex dist=0 and prev=-1 and enqueue to start):

  HEAD \quad TAIL
void unweightedShortestPath( int startNode ){
    queue<Vertex*> q;
    initData( ); // sets all Vertex dists to INFINITY, prevs to -1

    Vertex* s = vertexVec[startNode];
    s->dist = 0;
    q.push( s ) ;

    while( !q.empty() ) {
        Vertex* v = q.front(); // get a Vertex* from the queue
        q.pop(); // and remove it from the queue
        std::list<Edge*>::iterator it = v->adj.begin();
        for( ; it != v->adj.end() ; ++it ) { // go thru v’s adj list
            Vertex* w = vertexVec[ it->dest ];
            if( w->dist == INFINITY ) { // not yet visited
                w->dist = v->dist + 1;
                w->prev = v->indx;
                q.push( w );
            }
        }
    }
}
Weighted shortest path

• Input: a weighted directed graph $G = (V, E)$ with no negative edge weights; and a source vertex $s$ in $V$

• Output: for each vertex $v$ in $V$, a representation of the shortest weighted path in $G$ that starts at $s$ and ends at $v$

• The best general algorithm for this problem is Djikstra’s algorithm [Dijkstra, 1959]

• Djikstra’s algorithm uses ‘greedy’, ‘best-first’ search and runs in worst-case time $O(|E| \log|V|)$
Greedy algorithms

- A greedy algorithm is one that, at each stage of the algorithm, commits to a move that seems to be the best, considering just the state of computation at that stage.
- Greedy algorithms tend to be fast: they don’t require any backtracking or second-guessing.
  - However not every problem can be solved with a greedy algorithm!
  - And not every greedy algorithm is fast (may still have exponentially many moves).
- Example: Huffman’s algorithm for constructing a coding tree is a greedy algorithm.
  - At each step you choose two trees to join together, and you never second-guess the choice; but if symbols are not independent the result will not be optimal.
- Example: the usual algorithm for making change (in the U.S.) is a greedy algorithm:
  - Give back as many quarters as possible, then as many dimes as possible, then as many nickels as possible, then the remaining in pennies.
  - This algorithm is optimal: it always makes change with the fewest coins possible.
  - But suppose there were also a 12-cent coin; then the algorithm is not optimal (consider making 20 cents change...).
- Djikstra’s algorithm is fast, but it requires the precondition: no edge weights are negative.
**Weighted vs. unweighted shortest path algorithms**

- The basic idea is similar to the unweighted case

- A major difference is this:
  - In an unweighted graph, breadth-first search guarantees that when we first make it to a node $v$, we can be sure we have found the shortest path to it; more searching will never find a path to $v$ with fewer edges
  - In a weighted graph, when we first make it to a node $v$, we can’t be sure we have found the best path to $v$: there could be a path with more edges, but less overall cost, that we would find later

- Still, Dijkstra’s is a greedy algorithm:
  - At each stage of the algorithm, we will extend the best path we have found so far: this guarantees we will know when we have found the shortest (least-cost) path from the source vertex, if there are no negative-cost edges
  - Keeping track of paths in terms of which is best will require a priority queue -- a very common data structure in greedy algorithms

- (Note that Djikstra’s algorithm will work for an unweighted graph: treat it as a weighted graph with all edge weights the same, e.g. 1)
Weighted shortest path: auxiliary data structures

- Maintain a sequence (e.g. an array) of vertex objects, indexed by vertex number
  - Vertex objects contain these 3 fields (and others):
    - “dist”: the cost of the best (least-cost) path discovered so far from the start vertex to this vertex
    - “prev”: the vertex number (index) of the previous node on that best path
    - “done”: a boolean indicating whether the “dist” and “prev” fields contain the final best values for this vertex, or not

- Maintain a priority queue
  - The priority queue will contain (pointer-to-vertex, path cost) pairs
  - Path cost is priority, in the sense that low cost means high priority
  - Note: multiple pairs with the same “pointer-to-vertex” part can exist in the priority queue at the same time. These will usually differ in the “path cost” part
Weighted shortest path: Dijkstra’s algorithm

0. Initialize the vertex vector for the graph. Set all “dist” fields to INFINITY, and all “done” fields to false. Locate the start vertex \( s \). Set its “dist” field to 0, and its “prev” field to -1. Put the pair \((s,0)\) on the priority queue.

1. Is the priority queue empty? Done!

2. Remove from the priority queue the pair \((v, \text{cost})\) with the smallest cost.

3. Is the “done” field of the vertex \( v \) marked true? Go to 1.

4. Mark the “done” field of vertex \( v \) true. The shortest path from \( s \) to \( v \) is now known, and the “cost” and “prev” fields in \( v \) are correct.

5. Traverse \( v \)’s adjacency list. For each vertex \( w \) adjacent to \( v \):
   
   Compute the path cost \( c \) from \( s \) to \( w \) going through \( v \) (this is just the sum of the true cost from \( s \) to \( v \), which is now known, plus the cost of the edge from \( v \) to \( w \)).
   
   If \( c \) is less than the best cost from \( s \) to \( w \) known so far (this value is stored in the “dist” field of \( w \)), we have found a better path to \( w \). Update the “dist” field in \( w \) with this value, and insert the pair \((w,c)\) into the priority queue.

6. Go to 1.
Weighted shortest path: more algorithm details

• How do you figure out the actual paths, not just their costs?
  • Same as for the unweighted path algorithm

• How do you know that when you delete-min vertex $v$ from the priority queue, that we have searched enough to have found the shortest weighted path to $v$?
  • All the other vertices in the priority queue have at least as great a path cost to them, so we can’t do better by extending those paths to $v$
  • (If there were negative weights, continuing to add edges to a path can make its cost less, and this “greedy” approach does not work)

• What is the time complexity of Djikstra’s algorithm?
  • Each element of each adjacency list can be inserted and deleted from the priority queue; there are $|E|$ such elements
  • An insertion or delete-min in a binary heap implementation of a priority queue is $O(\log N)$; here $N = |E|$ worst-case
  • So, the algorithm has total worst-case time cost $O(|E| \log |E|)$
  • Since $|E| \leq |V|^2$, this is $O(|V|^2 \log |V|)$ and also $O(|E| \log |V|)$
Weighted shortest path: an example

- We will find shortest weighted paths in this graph, with start vertex $V_0$
The array of vertices, which include dist, prev, and done fields (initialize dist to ‘INFINITY’ and done to ‘false’):

\[
\begin{align*}
V0: & \quad \text{dist}= \quad \text{prev}= \quad \text{done}= \quad \text{adj}: (V1, 1), (V2, 6), (V3, 3) \\
V1: & \quad \text{dist}= \quad \text{prev}= \quad \text{done}= \quad \text{adj}: (V2, 4) \\
V2: & \quad \text{dist}= \quad \text{prev}= \quad \text{done}= \quad \text{adj}: \\
V3: & \quad \text{dist}= \quad \text{prev}= \quad \text{done}= \quad \text{adj}: (V2, 1)
\end{align*}
\]

The priority queue (set start vertex dist=0, prev=-1, and insert it with priority 0 to start)
Next time

- Connectedness in graphs
- Spanning trees in graphs
- Finding a minimal spanning tree
- Time costs of graph problems and NP-completeness
- Finding a minimal spanning tree: Prim’s and Kruskal’s algorithms
- Intro to disjoint subsets and union/find

Reading: Weiss, Ch. 9, Ch 8