CSE 332 Autumn 2023
Lecture 21: Dijkstra’s

Nathan Brunelle

http://www.cs.uw.edu/332
Breadth-First Search

• Input: a node $s$

• Behavior: Start with node $s$, visit all neighbors of $s$, then all neighbors of neighbors of $s$, ...

• Output:
  • How long is the shortest path?
  • Is the graph connected?
void bfs(graph, s) {
    found = new Queue();
    found.enqueue(s);
    mark s as "visited";
    While (!found.isEmpty()){
        current = found.dequeue();
        for (v : neighbors(current)){
            if (!v marked "visited"){
                mark v as "visited";
                found.enqueue(v);
            }
        }
    }
}

Running time: $\Theta(|V| + |E|)$
def shortestPath(graph, s, t):
    found = new Queue()
    layer = 0
    found.enqueue(s)
    mark s as “visited”
    While (!found.isEmpty()){
        current = found.dequeue()
        layer = depth of current
        for (v : neighbors(current)){
            if (!v marked “visited”){
                mark v as “visited”
                depth of v = layer + 1
                found.enqueue(v)
            }
        }
    }
    return depth of t

Idea: when it’s seen, remember its “layer” depth!
Find the quickest way to get from UVA to each of these other places

Given a graph $G = (V, E)$ and a start node $s \in V$, for each $v \in V$ find the least-weight path from $s \rightarrow v$ (call this weight $\delta(s, v)$)

(assumption: all edge weights are positive)
Some “Tricky” Observations

• Shortest path by sum of edge weights does not necessarily use the fewest edges.

• Negative Edges:
  • Today’s algorithm assumes that a path from A to B cannot be longer than a path from A to B to C.
    • Assumption is guaranteed to be true if no edges have negative weights
  • If there are negative weight cycles, problem is ill-defined
Dealing with Negative Edges (Incorrectly)

• Why doesn’t this work?
  • Take the most negative edge and add it’s absolute value to every other edge
Dijkstra’s Algorithm

• Input: graph with **no negative edge weights**, start node $s$, end node $t$
• Behavior: Start with node $s$, repeatedly go to the incomplete node “nearest” to $s$, stop when
• Output:
  • Distance from start to end
  • Distance from start to every node
Dijkstra’s Algorithm

Start: 0
End: 8

<table>
<thead>
<tr>
<th>Node</th>
<th>Done?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>∞</td>
</tr>
<tr>
<td>2</td>
<td>∞</td>
</tr>
<tr>
<td>3</td>
<td>∞</td>
</tr>
<tr>
<td>4</td>
<td>∞</td>
</tr>
<tr>
<td>5</td>
<td>∞</td>
</tr>
<tr>
<td>6</td>
<td>∞</td>
</tr>
<tr>
<td>7</td>
<td>∞</td>
</tr>
<tr>
<td>8</td>
<td>∞</td>
</tr>
</tbody>
</table>

Idea: When a node is the closest “unknown” node to the start, we have found its shortest path.
Dijkstra’s Algorithm

Start: 0
End: 8

<table>
<thead>
<tr>
<th>Node</th>
<th>Done?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>∞</td>
</tr>
<tr>
<td>4</td>
<td>∞</td>
</tr>
<tr>
<td>5</td>
<td>∞</td>
</tr>
<tr>
<td>6</td>
<td>∞</td>
</tr>
<tr>
<td>7</td>
<td>∞</td>
</tr>
<tr>
<td>8</td>
<td>∞</td>
</tr>
</tbody>
</table>

Idea: When a node is the closest “unknown” node to the start, we have found its shortest path.
Dijkstra’s Algorithm

Start: 0
End: 8

<table>
<thead>
<tr>
<th>Node</th>
<th>Done?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>∞</td>
</tr>
<tr>
<td>4</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>∞</td>
</tr>
<tr>
<td>6</td>
<td>∞</td>
</tr>
<tr>
<td>7</td>
<td>∞</td>
</tr>
<tr>
<td>8</td>
<td>∞</td>
</tr>
</tbody>
</table>

Idea: When a node is the closest “unknown” node to the start, we have found its shortest path.
Dijkstra’s Algorithm
Start: 0
End: 8

<table>
<thead>
<tr>
<th>Node</th>
<th>Done?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>6</td>
<td>∞</td>
</tr>
<tr>
<td>7</td>
<td>∞</td>
</tr>
<tr>
<td>8</td>
<td>∞</td>
</tr>
</tbody>
</table>

Idea: When a node is the closest “unknown” node to the start, we have found its shortest path.
Dijkstra’s Algorithm

Start: 0  
End: 8

<table>
<thead>
<tr>
<th>Node</th>
<th>Done?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>6</td>
<td>∞</td>
</tr>
<tr>
<td>7</td>
<td>20</td>
</tr>
<tr>
<td>8</td>
<td>∞</td>
</tr>
</tbody>
</table>

Idea: When a node is the closest “unknown” node to the start, we have found its shortest path.
Dijkstra’s Algorithm

```java
int dijkstras(graph, start, end){
    PQ = new minheap();
    PQ.insert(0, start);  // priority=0, value=start
    start.distance = 0;
    while (!PQ.isEmpty){
        current = PQ.extractmin();
        if (current.known){ continue;}
        current.known = true;
        for (neighbor : current.neighbors){
            if (!neighbor.known){
                new_dist = current.distance + weight(current,neighbor);
                if(neighbor.dist != ∞){ PQ.insert(new_dist, neighbor);}
                else if (new_dist < neighbor.distance){
                    neighbor.distance = new_dist;
                    PQ.decreaseKey(new_dist,neighbor); }
            }
        }
    }
    return end.distance;
}
```
Dijkstra’s Algorithm: Running Time

• How many total priority queue operations are necessary?
  • How many times is each node added to the priority queue?
  • How many times might a node’s priority be changed?

• What’s the running time of each priority queue operation?

• Overall running time:
  • \( \Theta(|E| \log |V|) \)
Dijkstra’s Algorithm: Correctness

• Claim: when a node is removed from the priority queue, we have found its shortest path
• Induction over number of completed nodes
• Base Case:
• Inductive Step:
Dijkstra’s Algorithm: Correctness

• Claim: when a node is removed from the priority queue, its distance is that of the shortest path

• Induction over number of completed nodes

• Base Case: Only the start node removed
  • It is indeed 0 away from itself

• Inductive Step:
  • If we have correctly found shortest paths for the first \( k \) nodes, then when we remove node \( k + 1 \) we have found its shortest path
Dijkstra’s Algorithm: Correctness

• Suppose $a$ is the next node removed from the queue. What do we know bout $a$?
Dijkstra’s Algorithm: Correctness

• Suppose $a$ is the next node removed from the queue.
  • No other node incomplete node has a shorter path discovered so far

• Claim: no undiscovered path to $a$ could be shorter
  • Consider any other incomplete node $b$ that is 1 edge away from a complete node
  • $a$ is the closest node that is one away from a complete node
  • Thus no path that includes $b$ can be a shorter path to $a$
  • Therefore the shortest path to $a$ must use only complete nodes, and therefore we have found it already!
Dijkstra’s Algorithm: Correctness

• Suppose \( a \) is the next node removed from the queue.
  • No other node incomplete node has a shorter path discovered so far

• Claim: no undiscovered path to \( a \) could be shorter
  • Consider any other incomplete node \( b \) that is 1 edge away from a complete node
  • \( a \) is the closest node that is one away from a complete node
  • No path from \( b \) to \( a \) can have negative weight
  • Thus no path that includes \( b \) can be a shorter path to \( a \)
  • Therefore the shortest path to \( a \) must use only complete nodes, and therefore we have found it already!
Depth-First Search
Depth-First Search

• Input: a node $s$

• Behavior: Start with node $s$, visit one neighbor of $s$, then all nodes reachable from that neighbor of $s$, then another neighbor of $s$, ...

• Output:
  • Does the graph have a cycle?
  • A **topological sort** of the graph.
void dfs(graph, s){
    found = new Stack();
    found.pop(s);
    mark s as “visited”;
    While (!found.isEmpty()){
        current = found.pop();
        for (v : neighbors(current)){
            if (!v marked “visited”){
                mark v as “visited”;
                found.push(v);
            }
        }
    }
}

Running time: \( \Theta(|V| + |E|) \)
DFS Recursively (more common)

```java
void dfs(graph, curr){
    mark curr as “visited”;
    for (v : neighbors(current)){
        if (! v marked “visited”){
            dfs(graph, v);
        }
    }
    mark curr as “done”;  
}
```
Using DFS

• Consider the “visited times” and “done times”

• Edges can be categorized:
  • Tree Edge
    • \((a, b)\) was followed when pushing
    • \((a, b)\) when \(b\) was unvisited when we were at \(a\)
  • Back Edge
    • \((a, b)\) goes to an “ancestor”
    • \(a\) and \(b\) visited but not done when we saw \((a, b)\)
    • \(t_{visited}(b) < t_{visited}(a) < t_{done}(a) < t_{done}(b)\)
  • Forward Edge
    • \((a, b)\) goes to a “descendent”
    • \(b\) was visited and done between when \(a\) was visited and done
    • \(t_{visited}(a) < t_{visited}(b) < t_{done}(b) < t_{done}(a)\)
  • Cross Edge
    • \((a, b)\) goes to a node that doesn’t connect to \(a\)
    • \(b\) was seen and done before \(a\) was ever visited
    • \(t_{done}(b) < t_{visited}(a)\)
boolean hasCycle(graph, curr){
    mark curr as “visited”;
    cycleFound = false;
    for (v : neighbors(current)){
        if (v marked “visited” && ! v marked “done”){
            cycleFound = true;
        }
        if (! v marked “visited” && ! cycleFound){
            cycleFound = hasCycle(graph, v);
        }
    }
    mark curr as “done”;
    return cycleFound;
}
Topological Sort

• A Topological Sort of a directed acyclic graph $G = (V, E)$ is a permutation of $V$ such that if $(u, v) \in E$ then $u$ is before $v$ in the permutation.
DFS Recursively

```c
void dfs(graph, curr){
    mark curr as “visited”;
    for (v : neighbors(current)){
        if (! v marked “visited”){
            dfs(graph, v);
        }
    }
    mark curr as “done”;
}
```
DFS: Topological sort

```python
def dfs(graph, s):
    seen = [False, False, False, ...]  # length matches |V|
    done = [False, False, False, ...]  # length matches |V|
    dfs_rec(graph, s, seen, done)

def dfs_rec(graph, curr, seen, done):
    mark curr as seen
    for v in neighbors(curr):
        if v not seen:
            dfs_rec(graph, v, seen, done)
    mark curr as done
```

Idea: List in reverse order by finish time
DFS Recursively

void dfs(graph, curr){
    mark curr as “visited”;
    for (v : neighbors(current)){
        if (! v marked “visited”){
            dfs(graph, v);
        }
    }
    mark curr as “done”;
}

Idea: List in reverse order by finish time
DFS: Topological sort

List topSort(graph){
    List<Nodes> finished = new List<>();
    for (Node v : graph.vertices){
        if (!v.visited){
            finishTime(graph, v, finished);
        }
    }
    finished.reverse();
    return finished;
}

void finishTime(graph, curr, finished){
    curr.visited = true;
    for (Node v : curr.neighbors){
        if (!v.visited){
            finishTime(graph, v, finished);
        }
    }
    finished.add(curr)
}

Idea: List in reverse order by finish time

finished: