Greedy algorithm

- Prim’s algorithm for constructing a Minimal Spanning Tree is a ***greedy algorithm***: it just adds the shortest edge without worrying about the overall structure, without looking ahead. It makes a locally optimal choice at each step.

Greedy Algorithms

- Dijkstra's algorithm: pick the vertex to which there is the shortest path currently known at the moment.
- For Dijkstra's algorithm, this also turns out to be globally optimal: can show that a shorter path to the vertex can never be discovered.
- There are also greedy strategies which are not globally optimal.

Example: non-optimal greedy algorithm

- Problem: given a number of coins, count the change in as few coins as possible.
- Greedy strategy: start with the largest coin which is available; for the remaining change, again pick the largest coin; and so on.

Shortest path

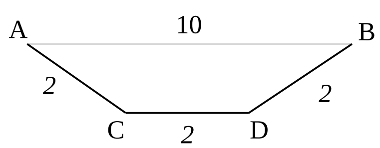
- Find the shortest route between two vertices u and v.
- It turns out that we can just as well compute shortest routes to ALL vertices reachable from u (including v). This is called *single-source shortest path problem* for weighted graphs, and u is the source.

Dijkstra’s Algorithm

- An algorithm for solving the single-source shortest path problem. Greedy algorithm.
- The first version of the Dijkstra's algorithm (traditionally given in textbooks) returns not the actual path, but a number - the shortest distance between u and v.
- (Assume that weights are distances, and the length of the path is the sum of the lengths of edges.)

Example

- Dijkstra’s algorithm should return 6 for the shortest path between A and B:



Dijkstra’s algorithm

To find the shortest paths (distances) from the start vertex s:

- keep a priority queue PQ of vertices to be processed
- keep an array with current known shortest distances from s to every vertex (initially set to be infinity for all but s, and 0 for s)
- order the queue so that the vertex with the shortest distance is at the front.

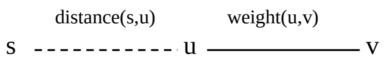
Dijkstra’s algorithm

Loop until there are vertices in the queue PQ:

- dequeue a vertex u
- recompute shortest distances for all vertices in the queue as follows: if there is an edge from u to a vertex v in PQ and the current shortest distance to v is greater than distance(s,u) + weight(u,v) then replace distance(s,v) with distance (s,u) + weight(u,v).

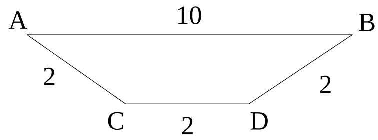
Computing the shortest distance

If the shortest distance from s to u is distance(s,u) and the weight of the edge between u and v is weight(u,v), then the current shortest distance from s to v is distance(s,u) + weight(u,v).



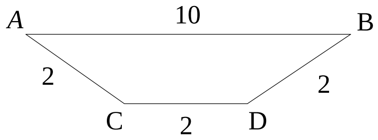
Example

- Distances: (A,0), (B,INF), (C,INF), (D,INF)
- PQ = {A,B,C,D}



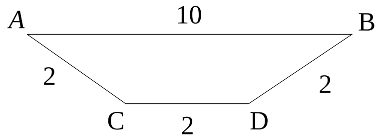
Example (dequeue A)

- Distances: (A,0), (B,INF), (C,INF), (D,INF)
- PQ = {B,C,D}



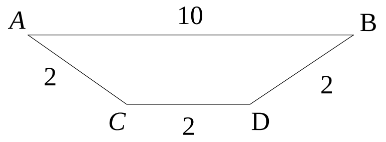
Example (recompute distances)

- Distances: (A,0), (B,10), (C,2), (D,INF)
- PQ= {C,B,D}



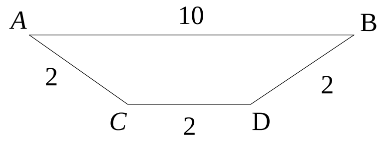
Example (dequeue C)

- Distances: (A,0), (B,10), (C,2), (D,INF)
- PQ = {B,D}



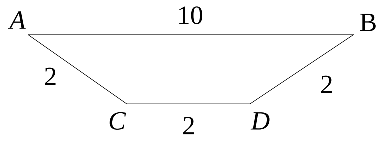
Example (recompute distances)

- Distances: (A,0), (B,10), (C,2), (D,4)
- PQ = {D,B}



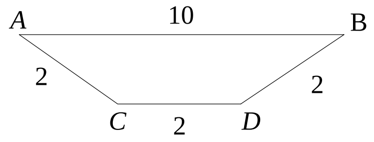
Example (dequeue D)

- Distances: (A,0), (B,10), (C,2), (D,4)
- PQ = {B}



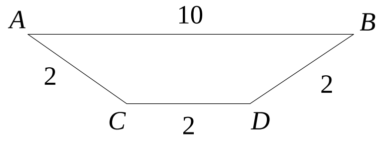
Example (recompute distances)

- Distances: (A,0), (B,6), (C,2), (D,4)
- PQ = {B}



Example (dequeue B)

- Distances: (A,0), (B,6), (C,2), (D,4)
- PQ = {}



Pseudocode for D's Algorithm

- INF is supposed to be greater than any number
- *dist* : array holding shortest distances from source s
- *PQ* : priority queue of unvisited vertices prioritised by shortest recorded distance from source
- *PQ.reorder()* reorders PQ if the values in *dist* change.

Pseudocode for Dijkstra's Algorithm

```
for(each v in V){
  dist[v] = INF;
  dist[s] = 0;
}
PriorityQueue PQ = new PriorityQueue();
// insert all vertices in PQ,
// in reverse order of dist[]
// values
```

Pseudocode for D's Algorithm

```
while (! PQ.isempty()){
  u = PQ.dequeue();
  for(each v in PQ adjacent to u){
    if(dist[v] > (dist[u]+weight(u,v)){
       dist[v] = (dist[u]+weight(u,v));
    }
  }
  PQ.reorder();
}
return dist;
```

Modified algorithm

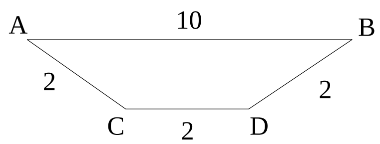
To make Dijkstra's algorithm to return the path itself, not just the distance:

- In addition to distances, maintain a path (list of vertices) for every vertex
- In the beginning paths are empty
- When assigning dist(s,v)=dist(s,u)+weight(u,v) also assign path(v)=path(u).
- When dequeuing a vertex, add it to its path.

Example

- Distances and paths:

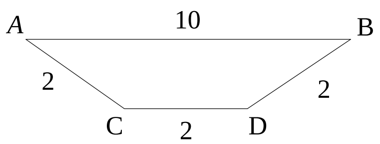
(A,0,{}), (B,INF,{}), (C,INF,{}), (D,INF,{})



Dequeue A, recompute paths

- Distances and paths:

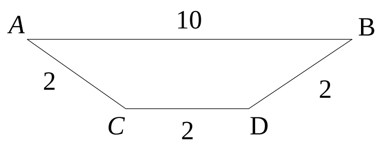
(A,0,{A}), (B,10,{A}), (C,2,{A}), (D,INF,{})



Dequeue C, recompute paths

- Distances and paths:

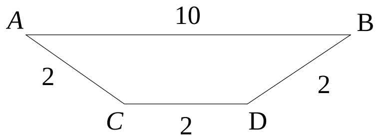
(A,0,{A}), (B,10,{A}), (C,2,{A,C}), (D,INF,{})



Dequeue C, recompute paths

- Distances and paths:

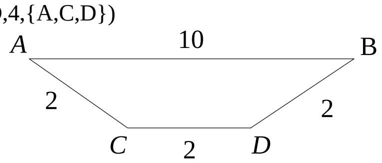
(A,0,{A}), (B,10,{A}), (C,2,{A,C}), (D,4,{A,C})



Dequeue D, recompute paths

- Distances and paths:

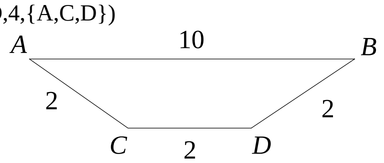
(A,0,{A}), (B,6,{A,C,D}), (C,2,{A,C}), (D,4,{A,C,D})



Dequeue B, recompute paths

- Distances and paths:

(A,0,{A}), (B,6,{A,C,D,B}), (C,2,{A,C}), (D,4,{A,C,D})



Optimality of Dijkstra's algorithm

So, why is Dijkstra’s algorithm optimal (gives the shortest path)?

Let us first see where it *could* go wrong.

What the algorithm does

- For every vertex in the priority queue, we keep updating the current distance downwards, until we remove the vertex from the queue.
- After that the shortest distance for the vertex is set.
- What if a shorter path can be discovered later?

Optimality proof

- Base case: the shortest distance to the start node is set correctly (0)
- Inductive step: assume that the shortest distances are set correctly for the first n vertices removed from the queue. Show that it will also be set correctly for the n+1st vertex.

Optimality proof

- Assume that the n+1st vertex is u. It is at the front of the priority queue and it's current known shortest distance is dist(s,u). We need to show that there is no path in the graph from s to u with the length smaller than dist(s,u).

Optimality proof

- Proof by contradiction: assume there is such a (shorter) path
- That path contains a vertex v1 to which the shortest distance is set (it may be that v1=s) which has an edge to a vertex v2 to which the distance is not set (maybe v2=u)

v1 v2
s ----------- ——— -------- u

Optimality proof

- So the vertices from s to v1 have correct shortest distances (inductive hypothesis) and v2 is still in the priority queue.

v1 v2
s ----------- ——— -------- u

Optimality proof

- So dist(s,v1) is indeed the shortest path from s to v1. Current distance to v2 is dist(s,v2)=dist(s,v1)+weight(v1,v2)

v1 v2
s ----------- ——— -------- u

Optimality proof

- If v2 is still in the priority queue, then dist(s,v1)+weight(v1,v2) >= dist(s,u)

v1 v2
s ----------- ——— -------- u

Optimality proof

- But then the path going through v1 and v2 cannot be shorter than dist(s,u). QED

v1 v2
s ----------- ——— -------- u

Complexity

- Assume that the priority queue is implemented as a heap;
- At each step (dequeueing a vertex u and recomputing distances) we do $O(|E_u|*\log(|V|))$ work, where E_u is the set of edges with source u.
- We do this for every vertex, so total complexity is $O((|V|+|E|)*\log(|V|))$.
- Really similar to BFS and DFS, but instead of choosing some successor, we re-order a priority queue at each step, hence the $|*\log(|V|)$ factor.

Implementation

- A Java implementation of Dijkstra's algorithm is given in Goodrich and Tamassia, Chapter 13.6.