Lecturer: Andrew Parkes
http://www.cs.nott.ac.uk/~ajp/

G52ADS 2014-15
Shortest Path & Dijkstra

Shortest path

- Given a graph with weights/distances on the edges
- 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.
- Assume that weights are non-negative (though possibly zero)
- Think of the weights as distances, and the length of the path is the sum of the lengths of edges.

Remarks

- Will develop the algorithm by repeated refinement – will see the same example 3 times, each time, with more detail.
- To understand this algorithm (and others), after each stage, ask yourself
  - “what is now known to be true and that was not known in previous stages?”
  - such understanding forms the basis of “appropriate assertions and loop invariants” and is assessable

Example in “code perspective”

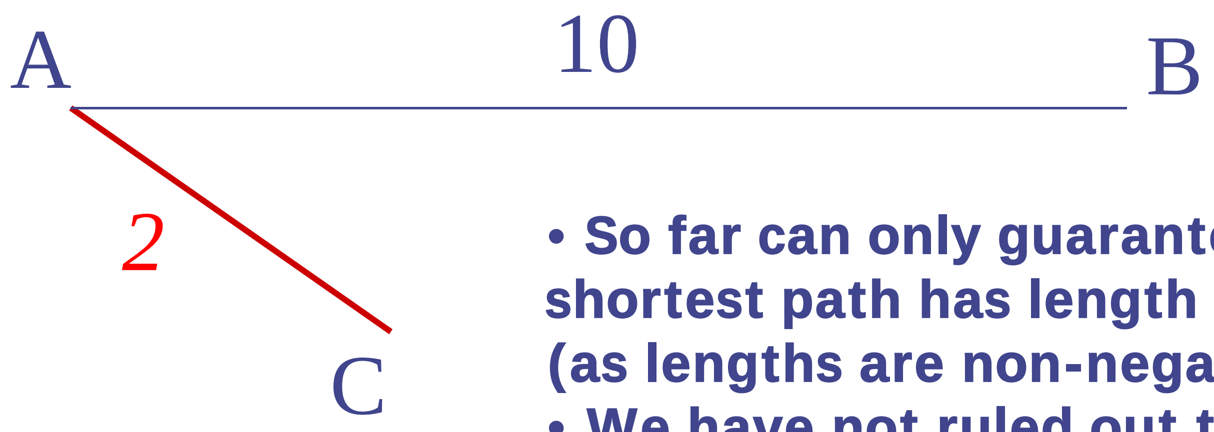
- Looking for shortest path from A to B
- Start from node A & find neighbours

A

Example

- Looking for shortest path from A to B
- So shortest path is 10

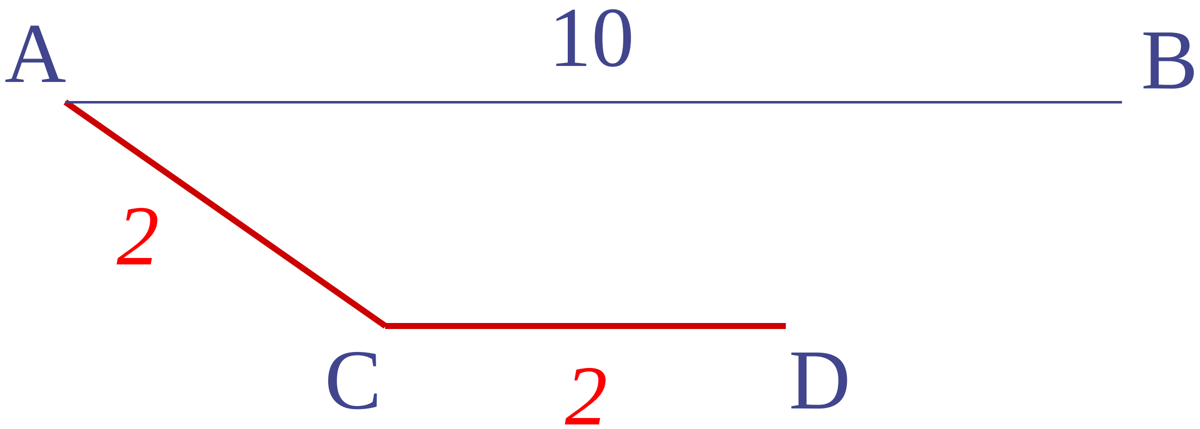
NO !!



• So far can only guarantee that the shortest path has length at least 2 (as lengths are non-negative)
• We have not ruled out the possibility of a path length L(A,B) with 2 < = L(A,B) < 10

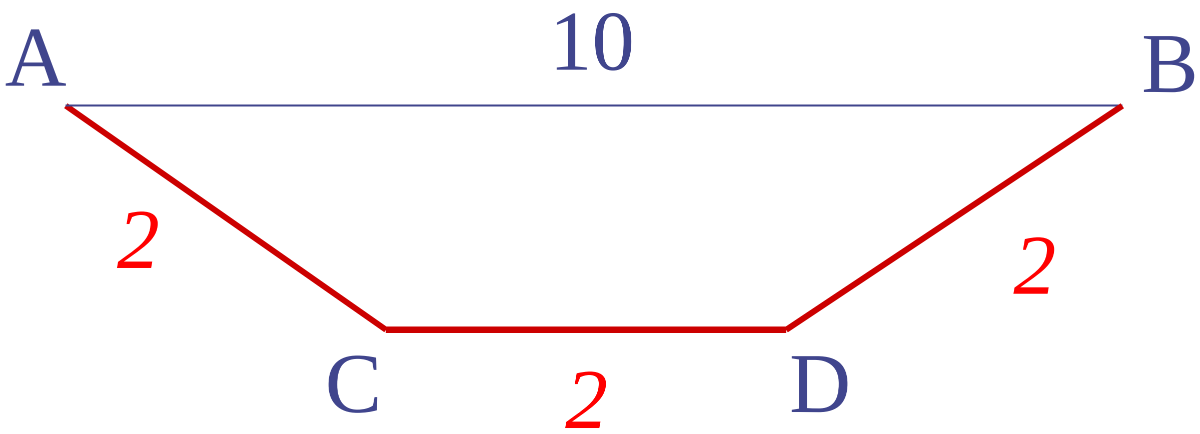
Example

- Which node should we expand next?
- Expand C as trying to rule out shortest paths
- Now know: shortest path A-C is 2



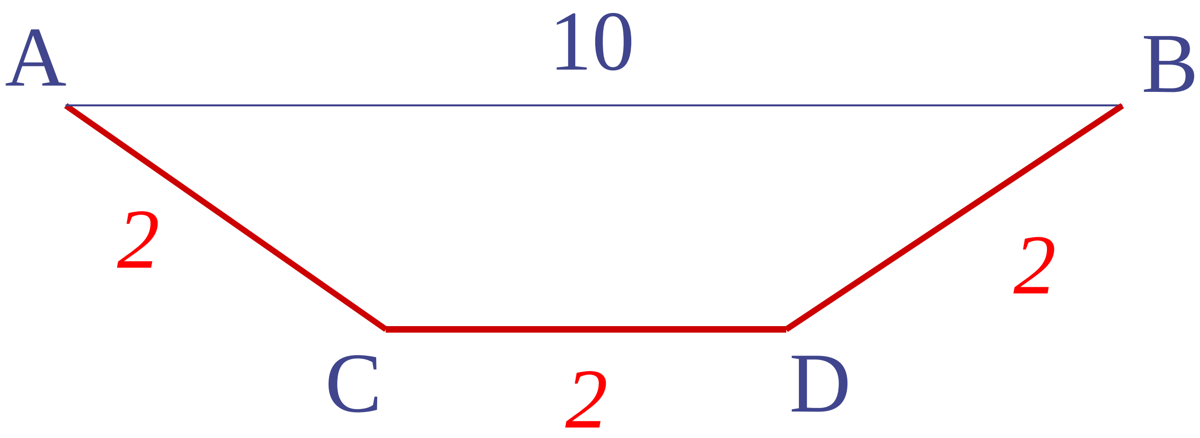
Example

- Next: expand D as it is
  - not yet expanded
  - the one with the shortest path and we are trying to rule out that L(A,B) is in the range [4 : 10]



Example

- Now we have reached B with L(A,B) = 6
- Are we finished?
- Yes, in this case, as all nodes are expanded (except B itself)



Core Ideas

- The previous simple example contains the core ideas of Dijkstra
  - “expand” means “add neighbours to a working list”
  - expand nodes with the shortest known current path as this is the only node for which we know the distance is really the shortest possible
  - **do not prematurely assume that have found the shortest path to a node**

Dijkstra’s algorithm

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

- keep a priority queue PQ of vertices to be processed
- for each u in the PQ maintain dist(s,u) as the shortest current known path length from s to u
  - e.g. 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)
- always order the queue so that the vertex with the shortest distance is at the front.
  - Exercise: ensure that you understand, and can explain, **why** this must be done.

Dijkstra's algorithm

Loop while there are vertices in the queue PQ:

- dequeue a vertex u – from the front, "popMin", hence with the least dist(s,u)
- expand node u:
  - recompute shortest distances for all vertices in the queue (i.e. not 'closed') as follows:
    - if there is an edge from u to a vertex v in PQ

      dist(s,v) ← min(dist(s,v) , dist(s,u) + w(u,v))
- close u, i.e. move to a "closed" list

Computing the shortest distance

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

distance(s,u) w(u,v)

s - - - - - - - - - - - - - - u ———————— v

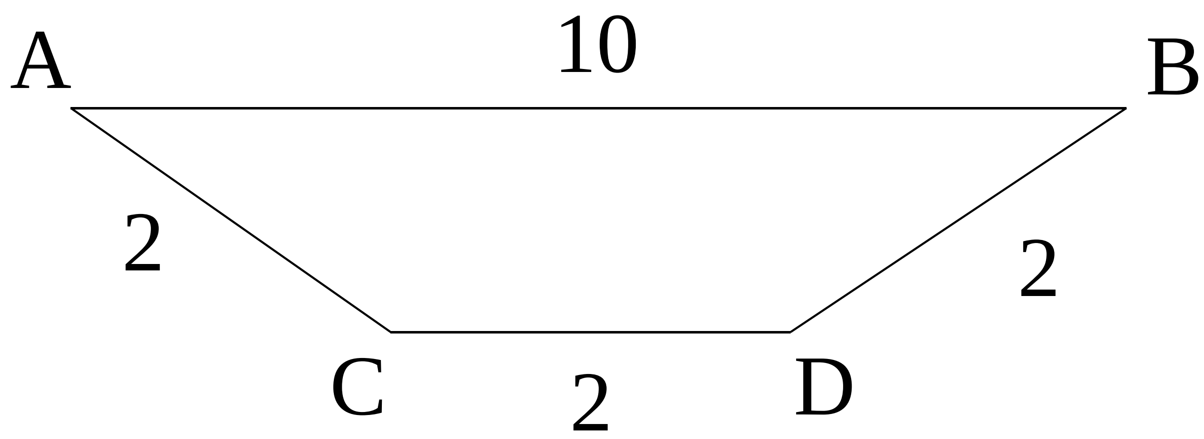
Important

- Do **NOT** conclude have the shortest path to a node until it has moved to front of the PQ and been dequeued and moved to the closed list

- Now do the same example again but this time with the PQ done explicitly:

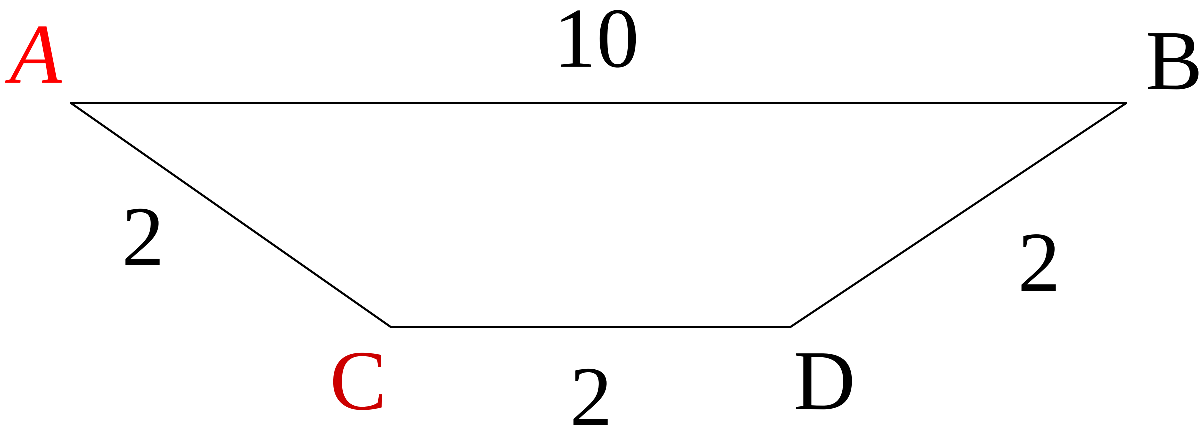
Example

- PQ = {A(0)} Closed= {}
- Dequeue and expand A



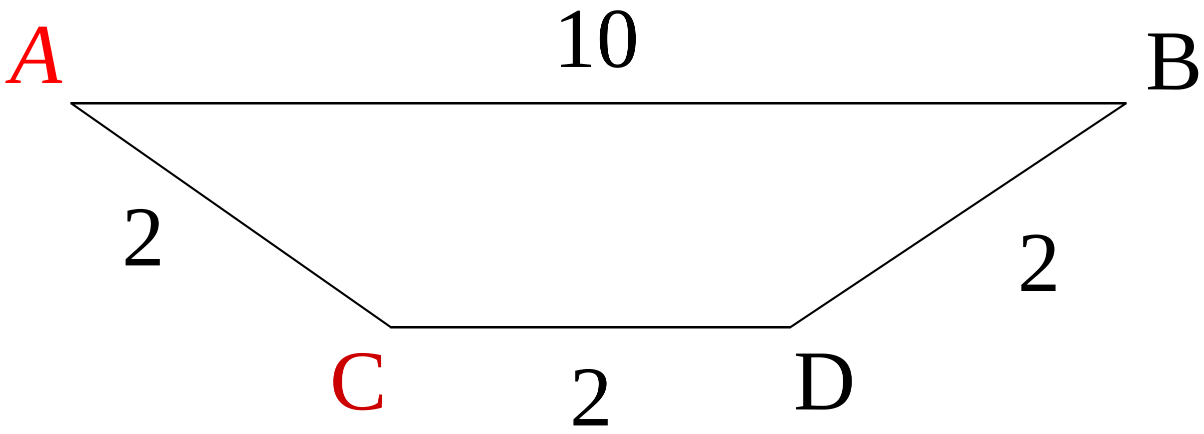
Example

- PQ = { C(2) , B(10)}   Closed = { A(0) }
- Dequeue and expand C



Example

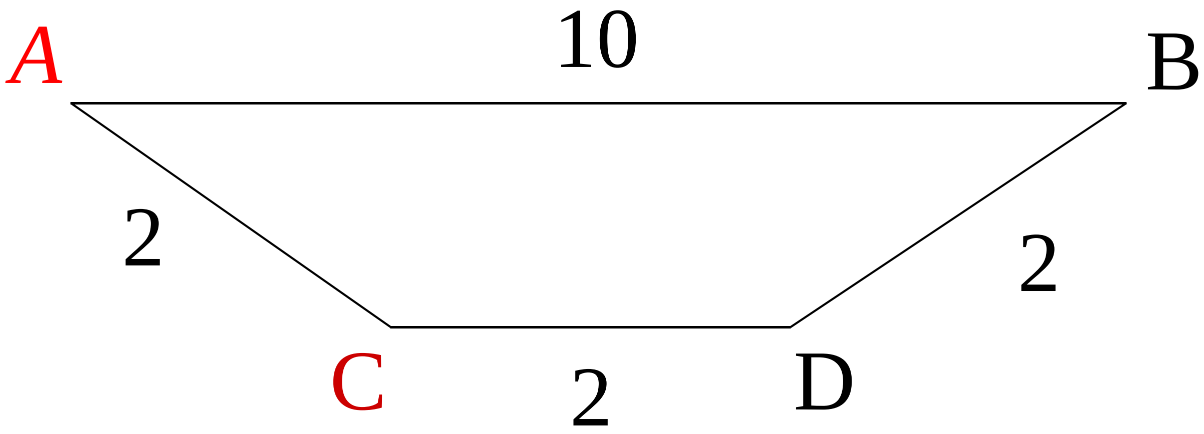
- PQ = { D(4) , B(10)} Closed = { A(0), C(2) }
- Dequeue and expand D & recompute B



= min (10, 4+2)

Example

- PQ = { B(6)}   Closed = { A(0), C(2), D(4) }
- Dequeue and close B and conclude L(A,B)=6



Pseudocode for D's Algorithm

- *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 D’s Algorithm

```
PriorityQueue PQ = new PriorityQueue();
while (! PQ.isempty()){
  u = PQ.dequeue();
  if ( u == target ) return dist[u];
  for(each v adjacent to u){
    add v to the PQ if not present and not
   already closed, else update the distance using
    if(dist[v] > (dist[u]+weight(u,v)){
       dist[v] = (dist[u]+weight(u,v));
    }
  }
  add u to list of closed nodes
  PQ.reorder(); // because some distances changed
}
return INFINITY; // no path to target
```

Implementing the PQ

- Many choices:
- It is not quite a heap – as might need to access nodes other than the minimum in order to change the distance
- Might just live with duplicates – and check when remove nodes that they are not already closed
- See GoTa textbook, etc, for advanced options

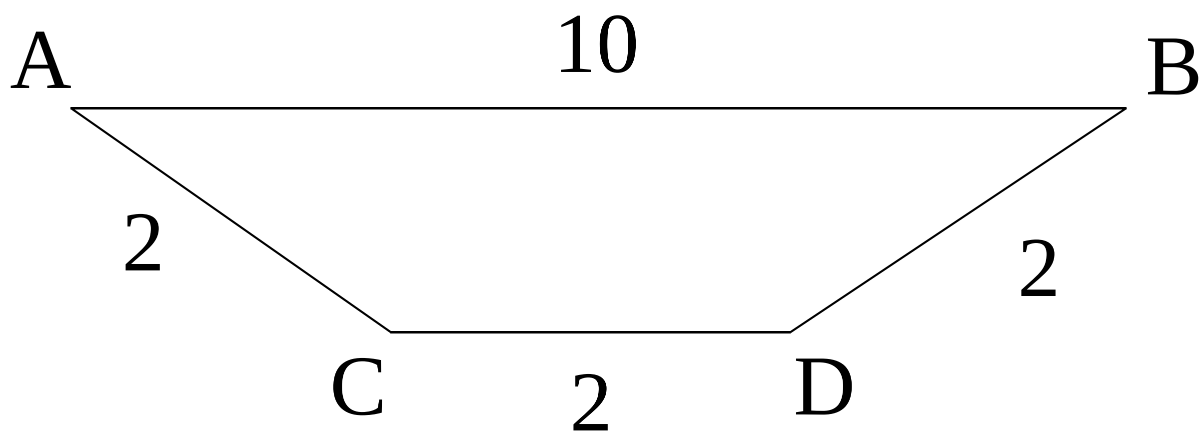
Finding the Path

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

- In addition to distances, maintain a “back pointer” back(u) a pointer to the previous node in the best path to u
- By following the back pointers can rebuild the path
- In the beginning paths are empty
- When adding a expanding u gives a new node v then back[v]=u
- When re-assigning dist(s,v)=dist(s,u)+weight(u,v) also re-assign back(v)=back(u).

Example

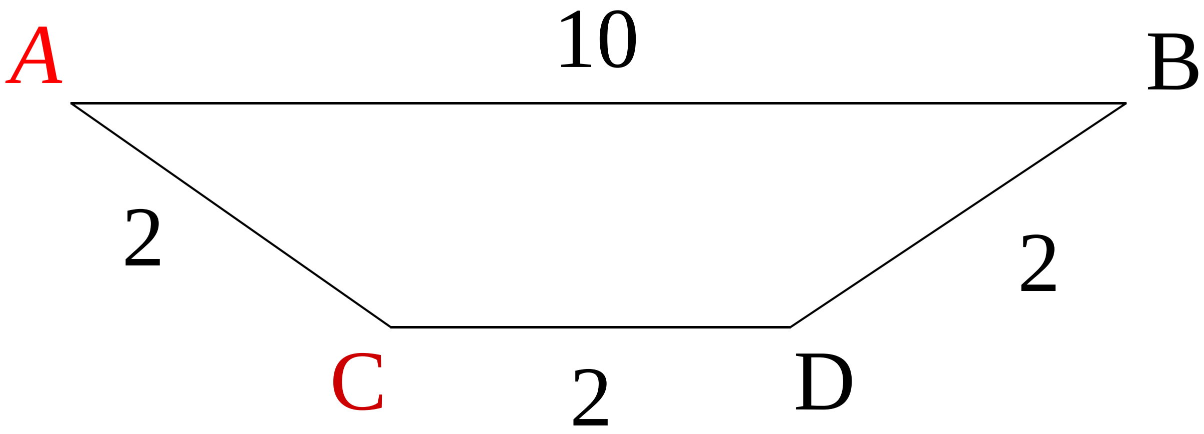
- PQ = {A(0,-)}   Closed= {}
- Dequeue and expand A



the back pointer from C to A

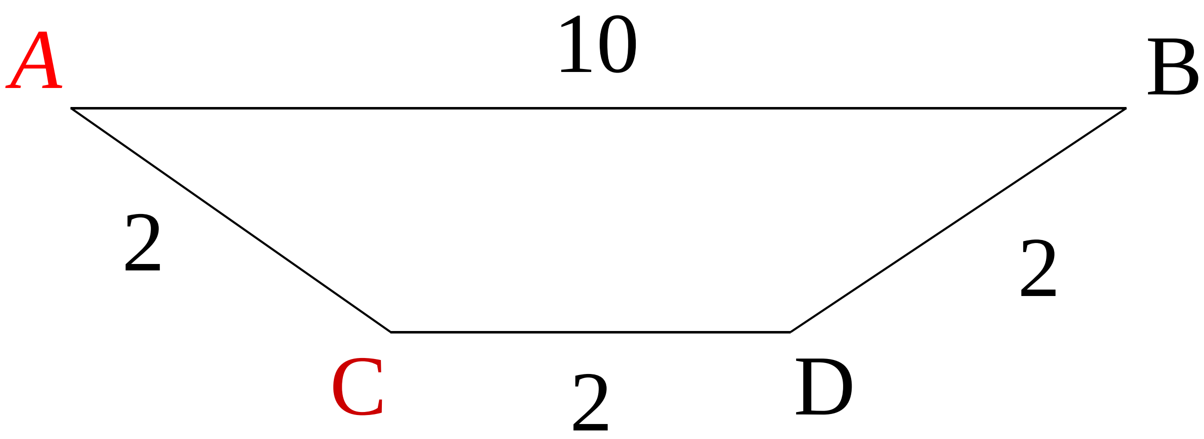
Example

- PQ = { C(2,A) , B(10,A)}   Closed = { A(0,-) }
- Dequeue and expand C



Example

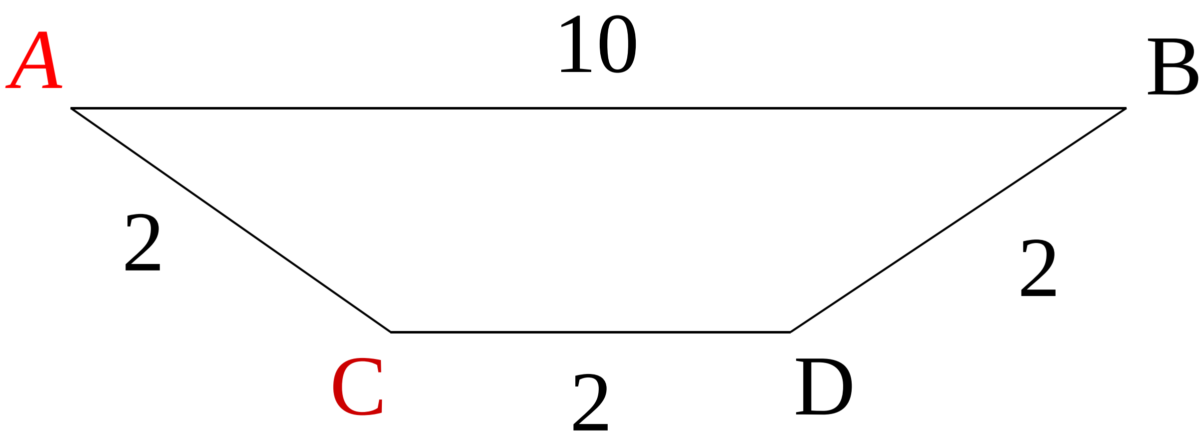
- PQ = { D(4,C) , B(10,A)} Closed = { A(0,-), C(2,A) }
- Dequeue and expand D & recompute B & back(B)



new back pointer

Example

- PQ = { B(6,D)}
  Closed = { A(0,-), C(2,A), D(4,C) }
- Close B and optimal back path is D,C,A



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).

- Proof: by contradiction – but non-essential so moved to "appendix" for self-study if desired

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 extra log(|V|) factor.

Exercise

- You are **highly** recommended to
  - create some small to medium graphs and work through this algorithm
  - repeat working examples until you understand it fully and can do it ‘by hand’ quickly and easily
    - Dijkstra is a classic algorithm, and the same ideas appear in many other algorithms

Minimum Expectations

- Know and understand definition of shortest path and Dijkstra's algorithm
- Be able to apply it, by hand, to small graphs
- Be able to argue or explain why it does give the shortest path
  - Formal proof not needed

Module Wrap-up

- General theme: Describing, reasoning about, and reducing the run-time of algorithms:
  - “Describing” - Big oh family
  - “Reasoning about” – counting operations, using heights of trees, etc
  - “Reducing” – using binary search, divide and conquer, binary trees, avoiding repeated work, etc
- Specifics:
  - “Linear structures”: Arrays, vectors, linked lists, stacks, queues, hashmaps
  - “Tree structures”: BST, heaps, pre- post- in-order traversals
  - “Graphs”: breadth & depth-first search, MST, Dijkstra

(All of these are core CS.)

END

“Appendix”

- Material that is
  - slightly more difficult
  - not strictly required
  - but that might still illuminate other aspects, or act as a test of understanding

Context: Greedy Algorithms

- Dijkstra's algorithm: has a stage “pick the vertex to which there is the shortest path currently known at the moment.”
- This is a “greedy” strategy
  - For Dijkstra's algorithm, this turned out to be globally optimal: a shorter path to the vertex can never be discovered.
  - However there are (many) problems for which 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.
- Example: Coins {5,2,2,2,2}. "Change": 8
  - the largest coin '5' is a 'fatal mistake' – not part of any desired solution
- "Exercise": find a fast greedy algorithm for the general version of this problem, or show one does not exist.
  - NOTE: $1m prize for solving this "exercise" ☺ – see G53COM
    - Note: not a real exercise for G52ADS! – but see "millennium prize problems"

Diameter of a graph

- http://en.wikipedia.org/wiki/Distance_%28graph_theory%29

- Diameter of a connected undirected graph is the length of the "longest shortest path"
  - the maximum over all pairs of nodes a, b
    - of the length of the shortest path between a and b

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)
  - Exercise: why must there be such a v2 if the path is to be shorter?

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