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

G52ACE 2017-18
Shortest Paths

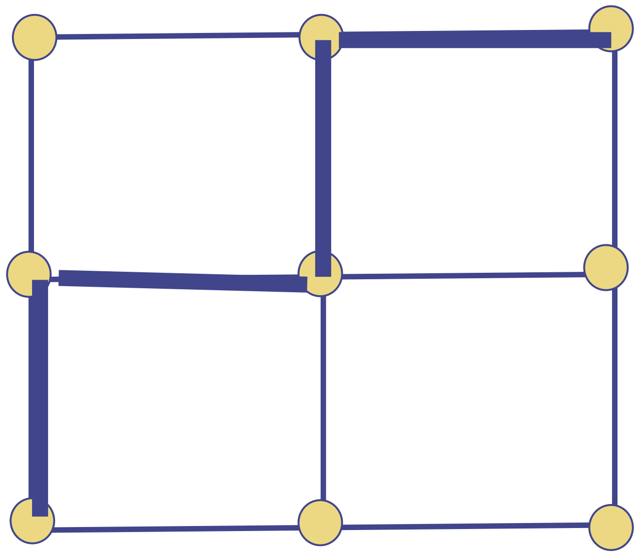
Shortest path

- 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.

Counting paths

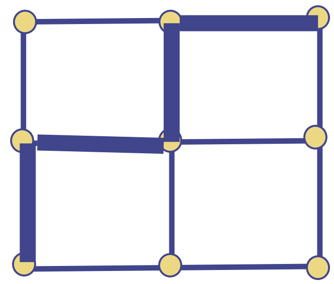
- Firstly we show that the number of paths is too large to simply generate them all and take the shortest
- Consider a simple example of an N x N set of city blocks with streets in between
  - Suppose need to travel from the bottom-left corner to the top-right corner:

Grid Example at N=2



- All shortest paths, from (0,0) to (2,2), are length 4
- An example is shown: “[u,r,u,r]” if we use labels
  - u for up
  - r for right
- Observe: All shortest paths have:
  - exactly 2 u and 2 r, but in any order.

Grid Example at N=2



- Shortest paths have exactly 2 u and 2 r, but in any order.
- There are 6 shortest paths

[u u r r], [u r u r] , [r u u r], [u r r u], [r u r u], [r r u u]

- We have to place the 2 x r within the list of 4, and the others are then u.
    - First r has 4 choices
    - Second r has 3 choices
    - But then swapping the different r, gives the same path so have "/ 2"
    - Giving 4 * 3 / 2 = 6 different optimal paths

Grid: Counting optimal paths

- Consider general N
- Shortest paths have exactly N u's and N r's, but in any order.
- We have to place the N x 'r' within the list of 2N moves, and the others are then u.
  - First 'r' has 2N choices
  - Second 'r' has 2N-1 choices, ….
  - N'th 'r' has 2N – (N-1) = N+1 choices
  - Hence total choices is 2N*(2N-1)*…*(N+1) = (2N)! / N!
  - But then swapping the N different r's, without moving them, gives the same path.
    - There are N! ways to order them, so have to divide by N!
  - Giving (2N)! / (N! * N!) different choices for the optimal paths
- This is called "2N choose N" or a "Binomial Coefficient"
- See: https://en.wikipedia.org/wiki/Binomial_coefficient

Grid: Counting optimal paths

- This is called "2N choose N" or a "Binomial Coefficient"
- See: https://en.wikipedia.org/wiki/Binomial_coefficient
- Compute this for various N
  - e.g. from online calculator such as http://www.ohrt.com/odds/binomial.php
- N=25 (50 25) = 126,410,606,437,752
- N=50 (100 50) = 100,891,344,545,564,193,334,812,497,256
- We see that the number of paths is very large and grows rapidly

From https://en.wikipedia.org/wiki/Binomial_coefficient#Bounds_and_asymptotic_formulas

- (2N choose N) ~ 4^N / sqrt(pi N) as N → \infty
  - So the number of paths grows exponentially.
  - This is standard (not just in the example). Hence

"Generating all paths and taking the shortest" is totally impractical

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, w(i,j), as distances, and the length of the path is the sum of the lengths of edges.

Example in “code perspective”

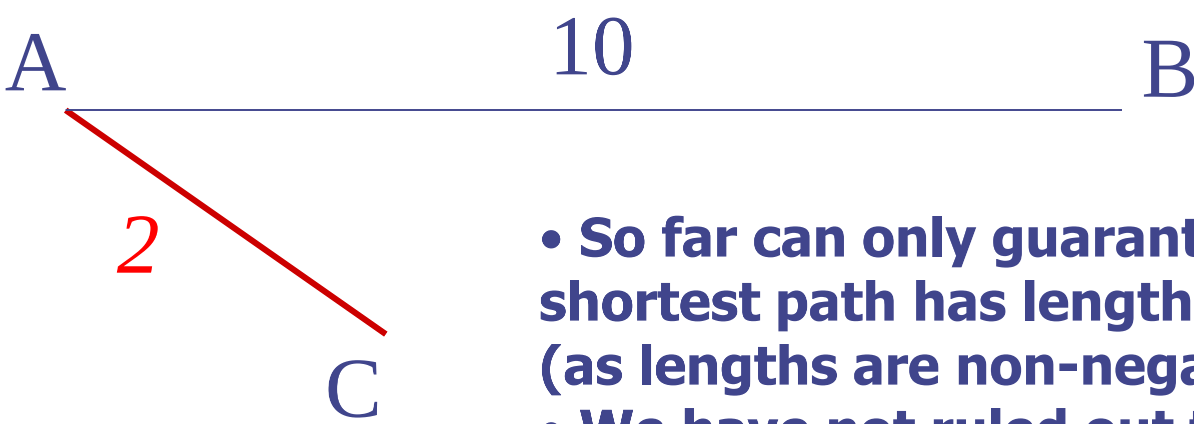
- Looking for shortest path from A to B
- Start from node A & find neighbours

A

Example

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

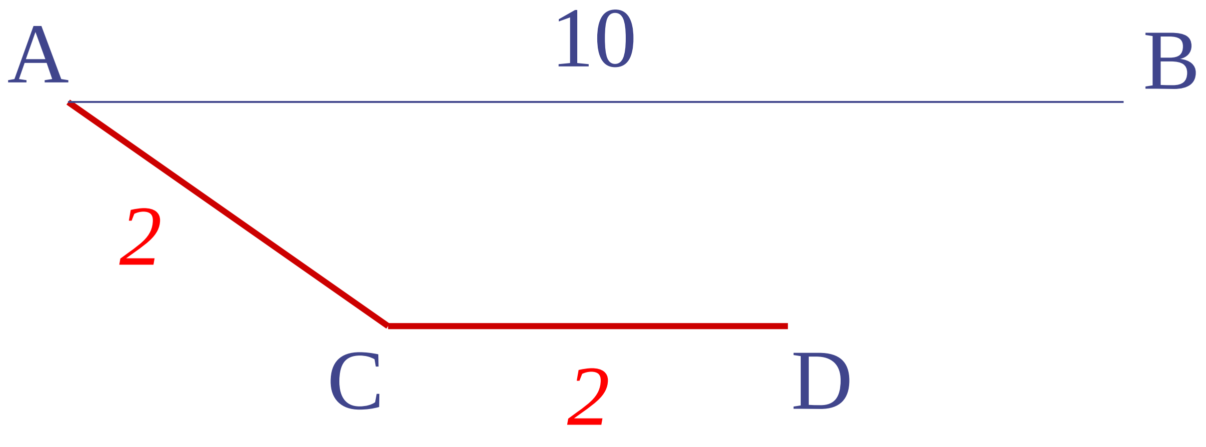
NO !!



- 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

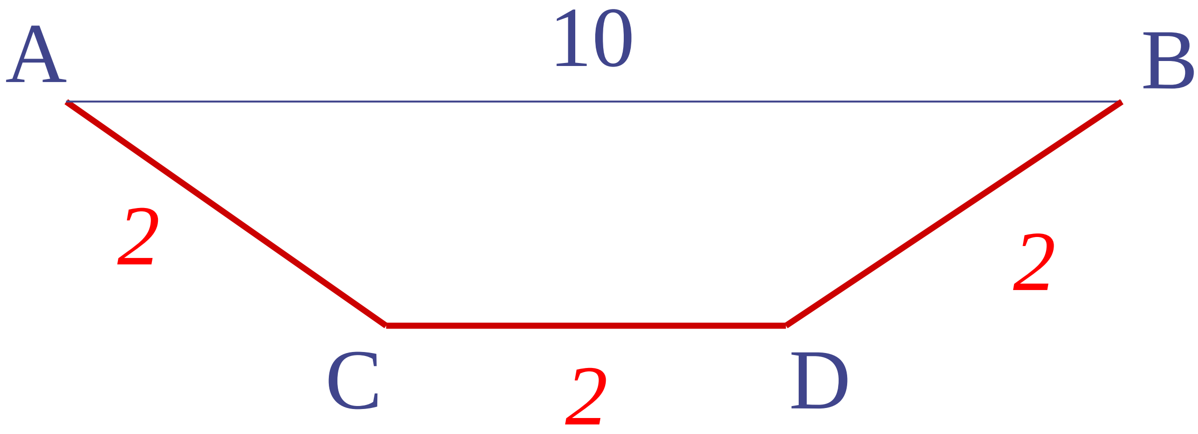
Example

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



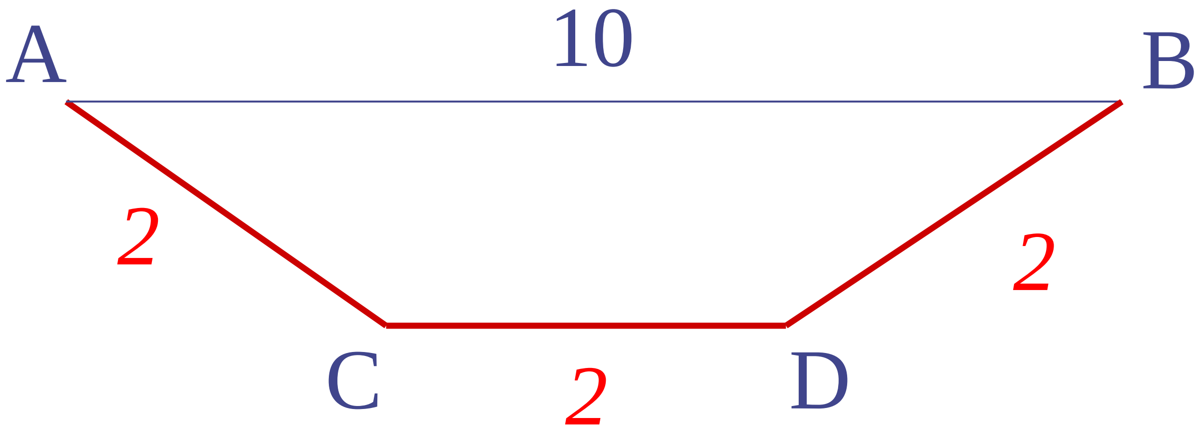
Example

- 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)



[VITAL] 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.
  - Note: 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

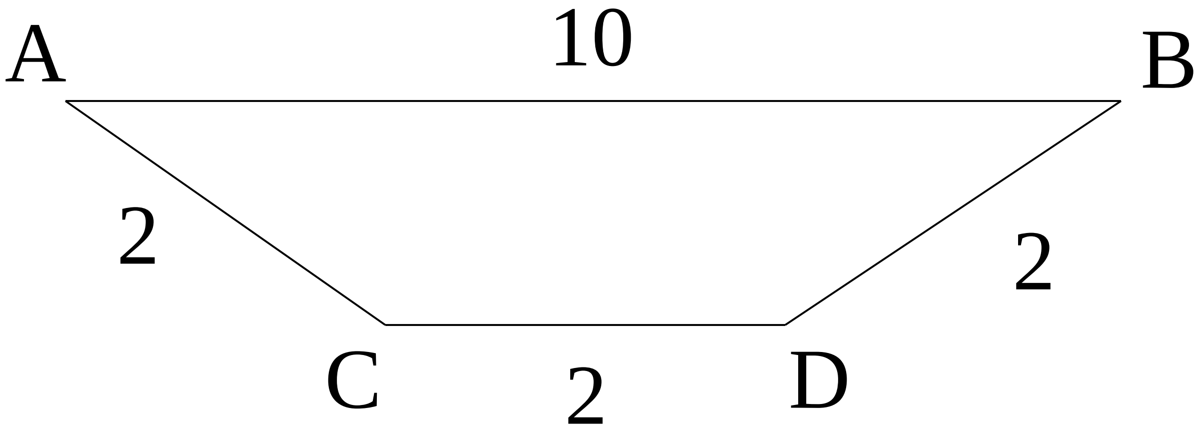
Important

- 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:

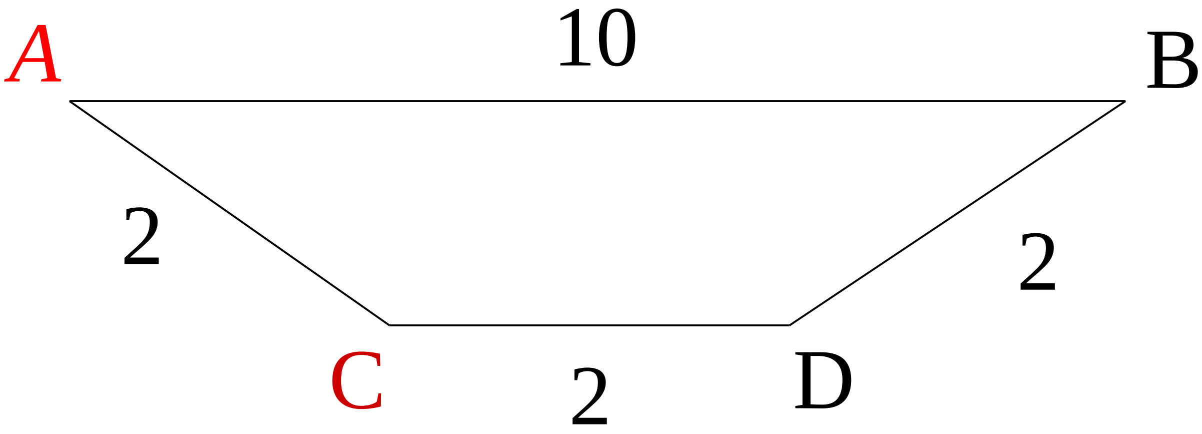
Example

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



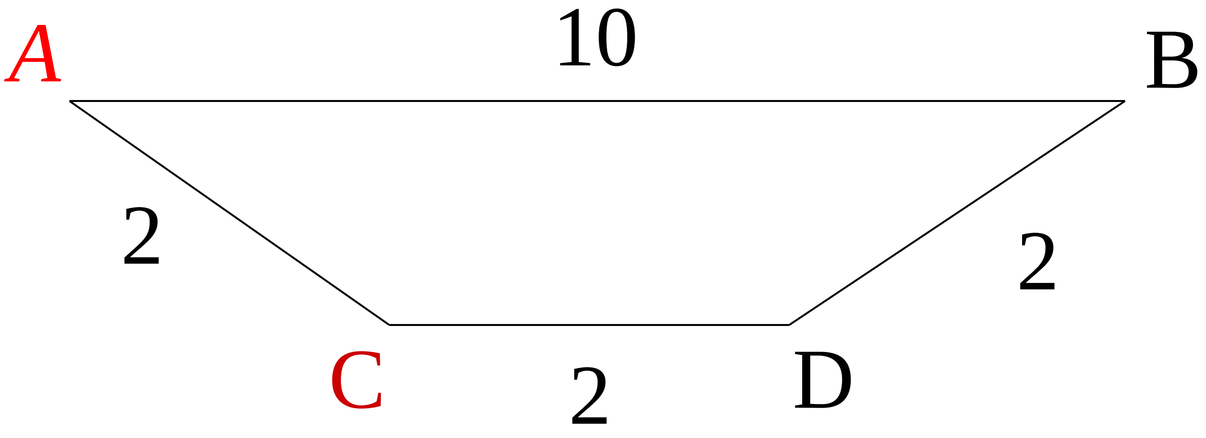
Example

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



Example

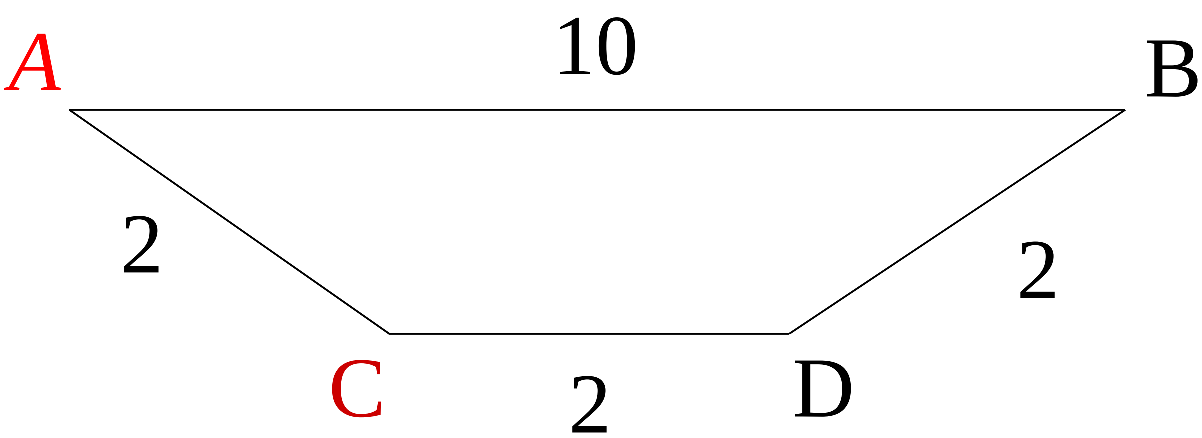
- PQ = { D(4) , B(10)} Closed = { A(0), C(2) }
- Dequeue and expand D & recompute B



= 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 textbook, etc, for advanced options

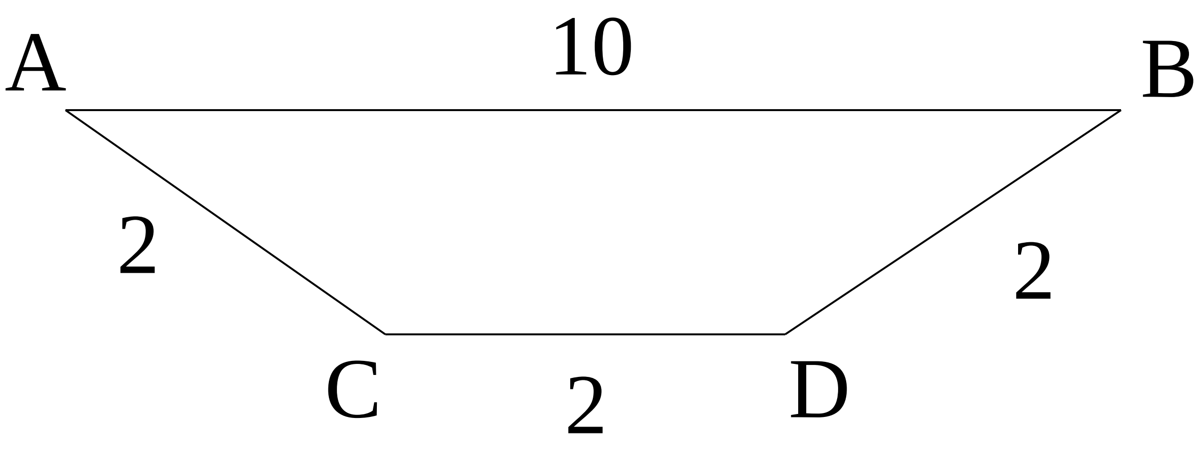
Finding the Path

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

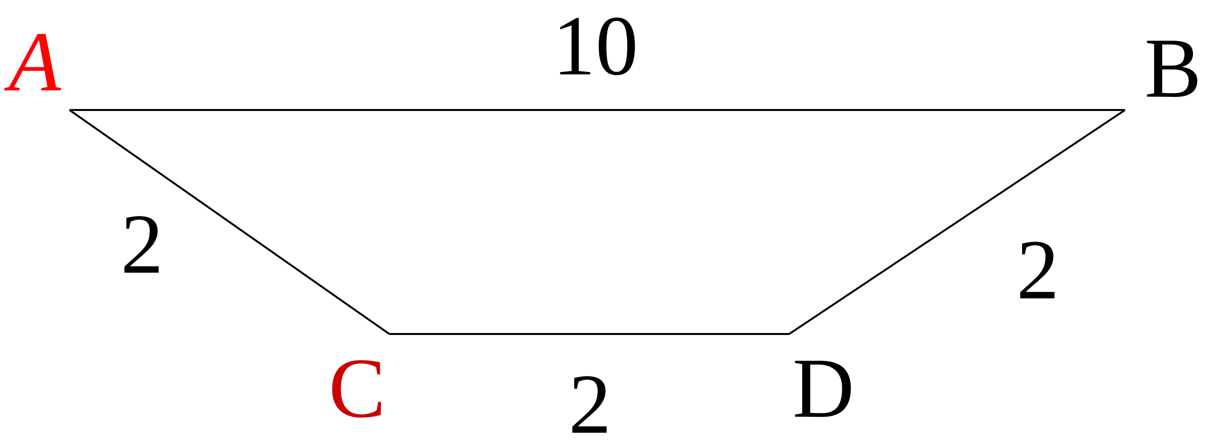
- PQ = {A(0,-)}   Closed= {}
- Dequeue and expand A



the back pointer from C to A

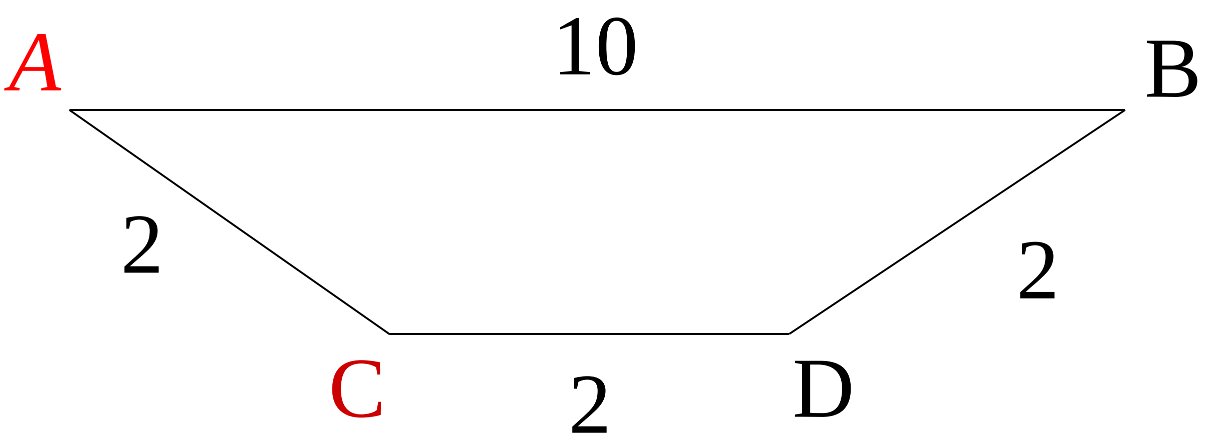
Example

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



Example

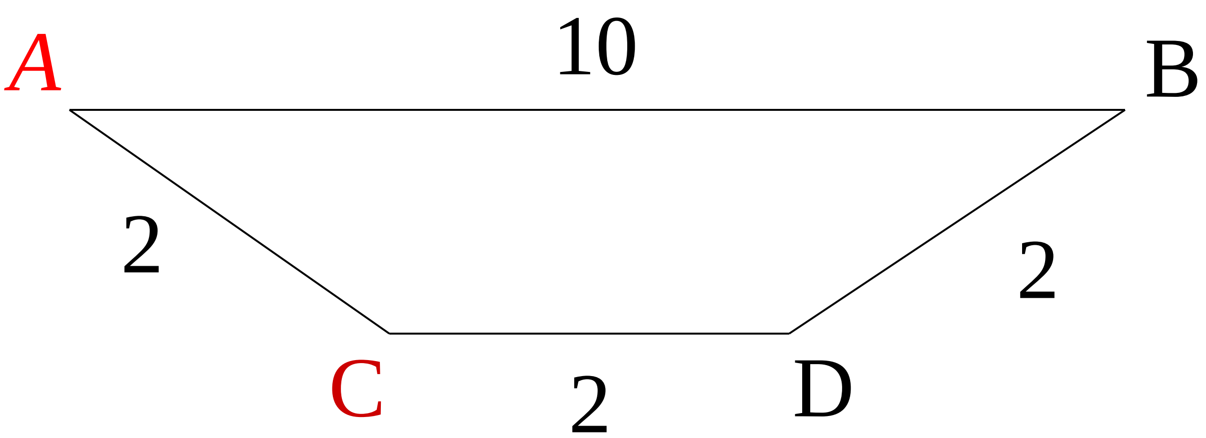
- PQ = { D(4,C) , B(10,A)} Closed = { A(0,-), C(2,A) }
- Dequeue and expand D & recompute B & back(B)



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



Complexity

- Assume that the priority queue is implemented as a heap;
  - Specifically, it should be a heap that allows efficient (log time) changes of the value of a key, such as a "Fibonacci Heap" (but these are outside the scope of this module)
- At each step (dequeueing a vertex u and recomputing distances) we expect no worse than $O(|E_u|*\log(|V|))$ work, where E_u is the set of edges with source u.
- We do this for every vertex, so total complexity is no worse than $O((|V|+|E|)* \log(|V|))$
  - Based on similarity to BFS and DFS, but instead of choosing some successor, and if we re-order a priority queue at each step, hence the extra $\log(|V|)$ factor.

  - With a good PQ implementation (e.g. using Fibonacci heap), we can get a (slightly better) complexity

$$O(|V| * \log(|V|) + |E|)$$

Exercise

- You are **highly** recommended to
  - create some small to medium graphs (directed and undirected) and work through the algorithm

Minimum Expectations

- Know and understand definition of shortest path, Dijkstra's algorithm
- Be able to apply it, by hand, to small graphs
  - Understand the complexity, etc/