Dijkstra's Shortest Paths

By: Ahmed Abdelmeged
and: Karl Lieberherr

Single Source Shortest Paths Problem

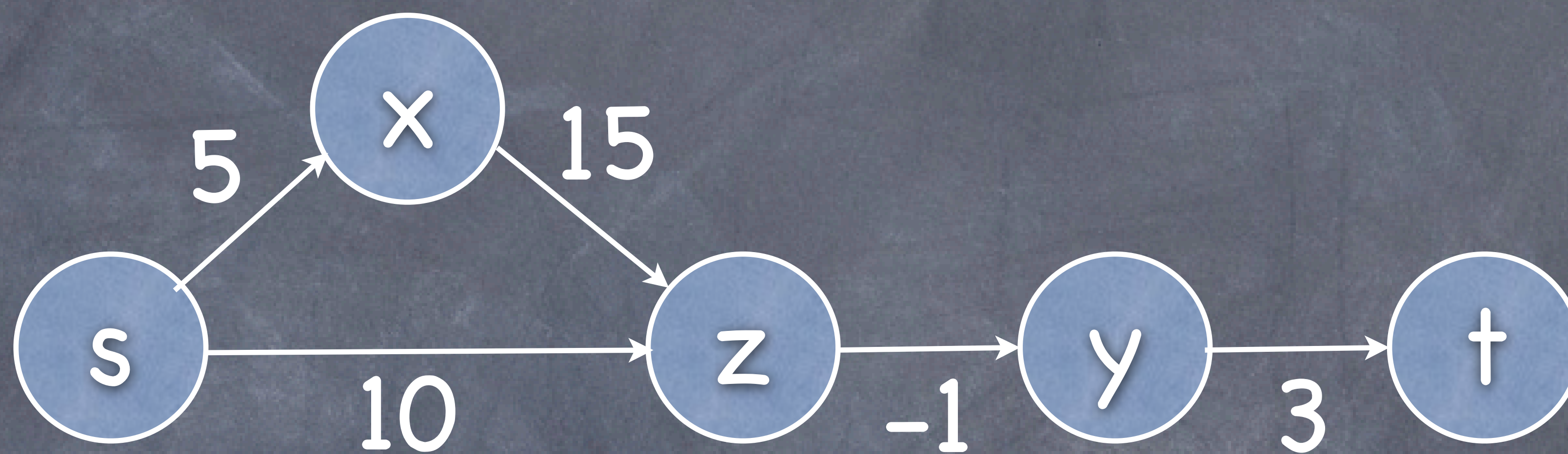
- Given:
  - a directed graph G(V,E),
  - edge weights w:E->R denoting costs/distances,
  - a designated source node $s \in V$.
- Find:
  - for each node $v \in V$ find the best (minimum) cost to reach v from s.

Brute Force Algorithm

- "**Try**" all paths.
- for v∈ V
  - let paths_v be the set of all paths from s to v
  - output v, min(cost(paths_v))

Q: Can you upper bound |paths_v|?

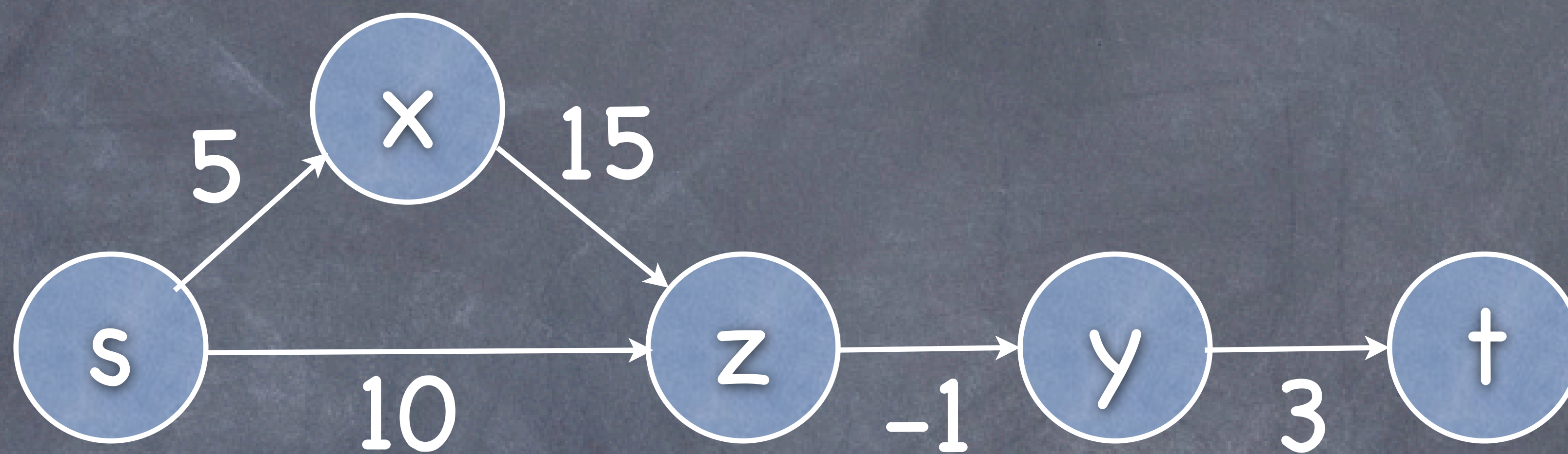
Can we asymptotically improve the brute force algorithm?



- s-x paths:{[s,x:5]}
- s-z paths {[s,z:10], [s,x,z:20]}
- s-y paths {[s,z,y:9],[s,x,z,y:19]}
- s-t paths{[s,z,y,t:12],[s,x,z,y,t:22]}

Can we say something about the best paths or other paths?

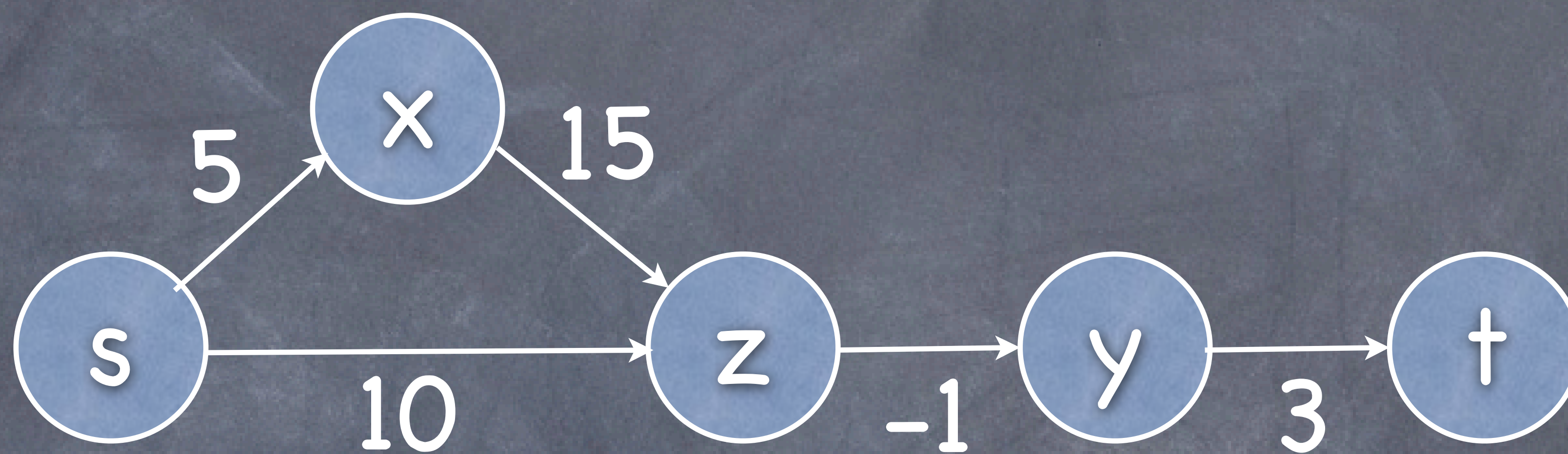
Yes, because the solution exhibits the optimal substructure property!



- s-x paths:{[s,x:5]}
- s-z paths {[s,z:10], [s,x,z:20]}
- s-y paths {[s,z,y:9],[s,x,z,y:19]}
- s-t paths{[s,z,y,t:12],[s,x,z,y,t:22]}

[s,x,z] is not the best path to z, so is every other path starting with [s,x,z,..]

Yes, because the solution exhibits the optimal substructure property!



- s-x paths:{[s,x:5]}
- s-z paths {[s,z:10], [s,x,z:20]}
- s-y paths {[s,z,y:9],[s,x,z,y:19]}
- s-t paths{[s,z,y,t:12],[s,x,z,y,t:22]}

[s,z,y,t] is the best path to t, so [s,z,y] **must be** the best path to y, and [s,z] **must be** the best path to z,...

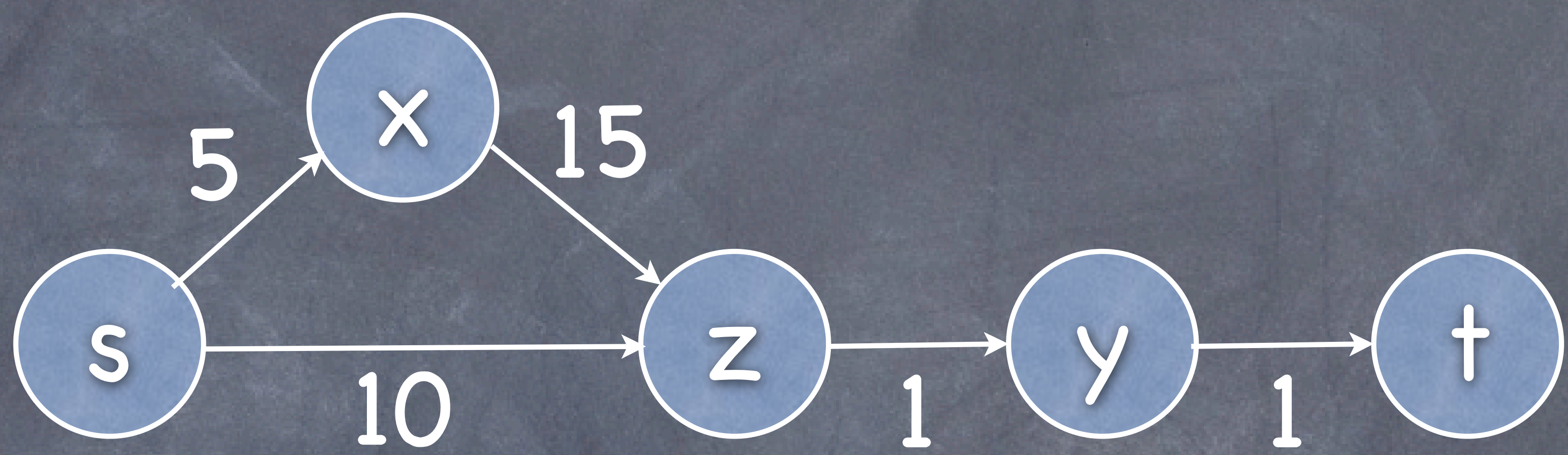
Dynamic Programming

- To find the best path to v:
  - "**Try**" to extend the best path to some node x (where $(x,v) \in E$) with v. Output the extension minimizing the overall path cost.
- BestPath(s,v) = min { BestPath(s,x) + w([x,v])}
- Q (we won't answer): when does the above recursion terminate? What is its complexity?

Can we do it without "Try"ing?

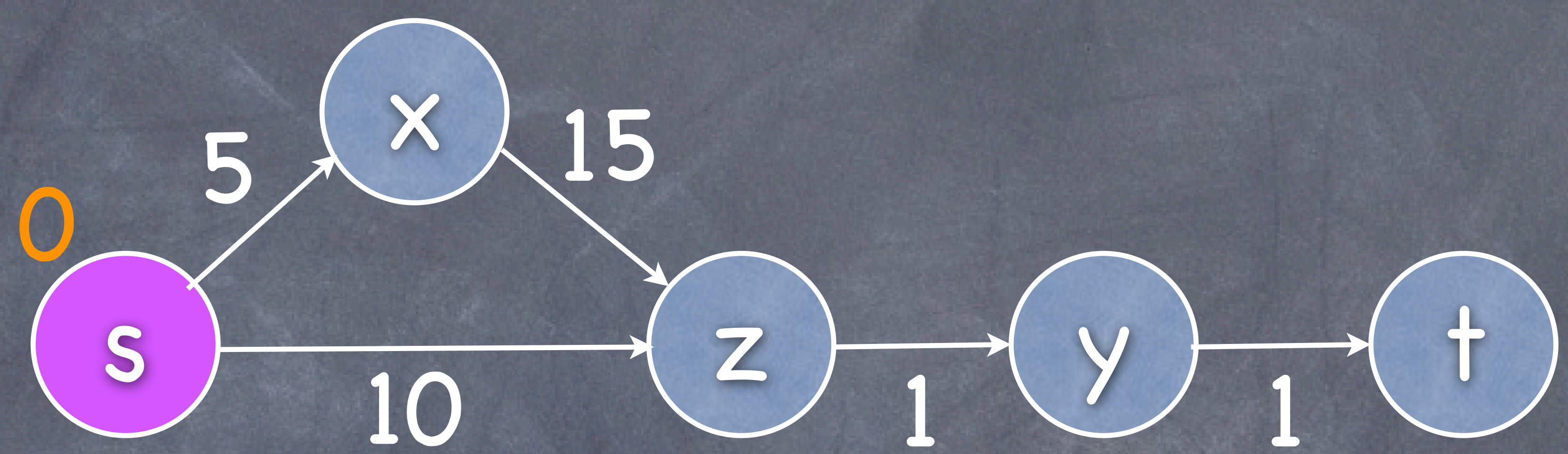
- We are after either a greedy or a divide and conquer algorithm.
- Dijkstra's algorithm (Greedy)
  - Edge weights must be positive ($w: E \to R^+$). As we go through the algorithm, try to figure out why this is critical! Now we can use the distance metaphor.

Dijkstra's Single Source Shortest Paths Algorithm (Data Structures)



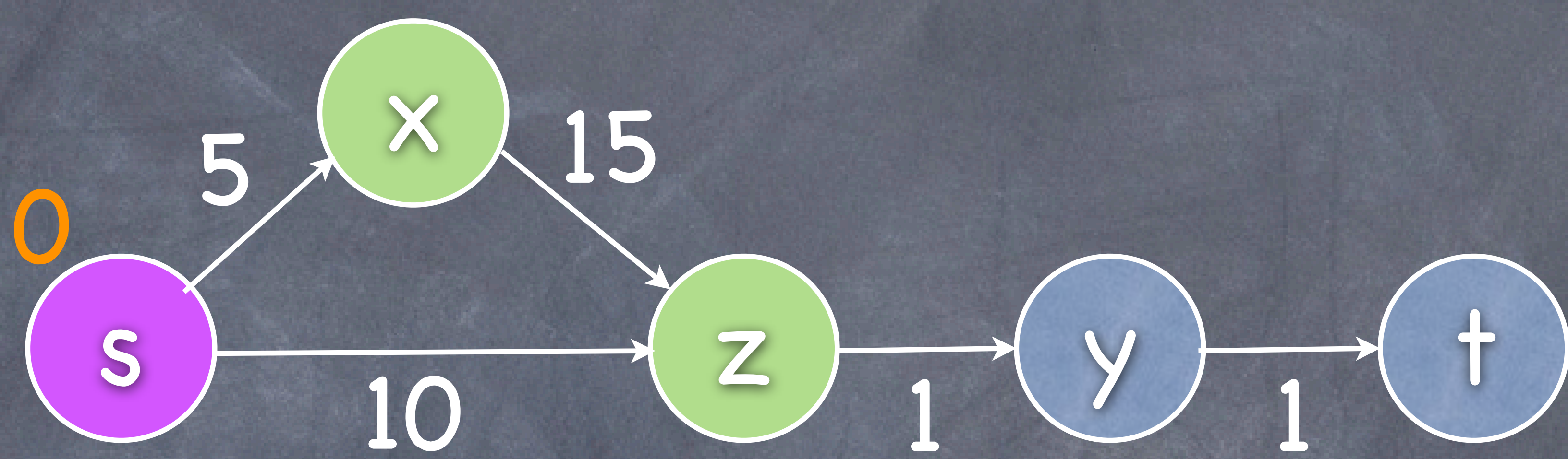
- For each node v in V we keep it's best known distance.
- We also keep a set S of nodes that we know their shortest distance.

Dijkstra's Single Source Shortest Paths Algorithm



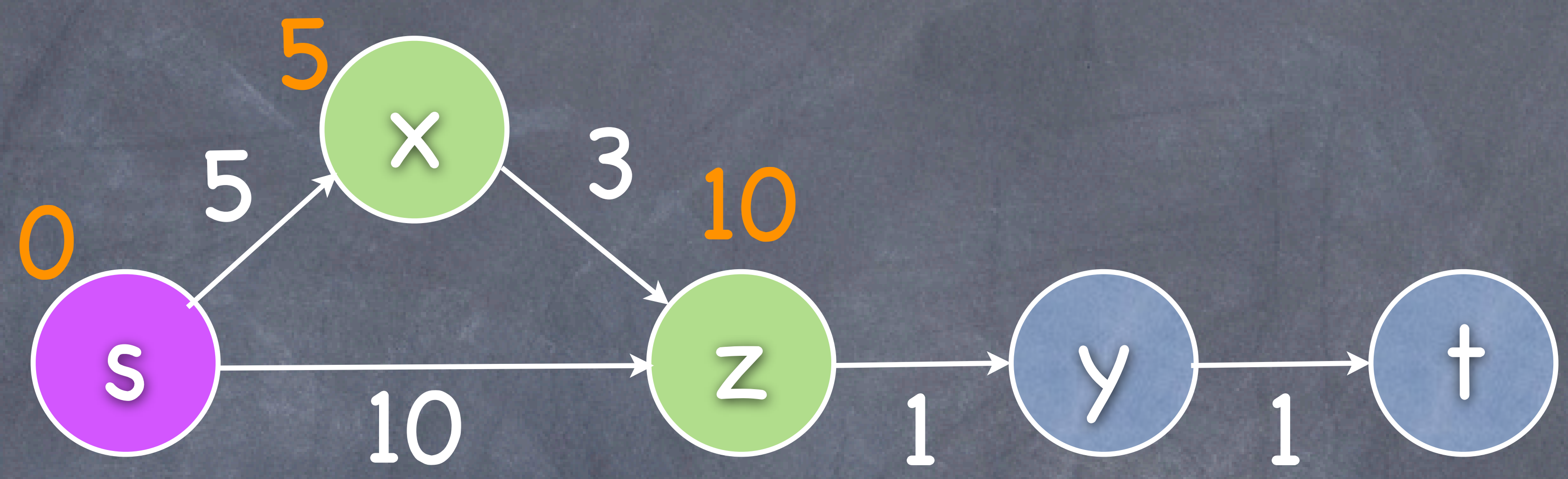
- Initialize S to {s} and bestKnownDistance(s) to 0;

Dijkstra's Single Source Shortest Paths Algorithm



- A node f not in S is called a frontier node iff it is the target of some edge coming out of some node in S.

Dijkstra's Single Source Shortest Paths Algorithm



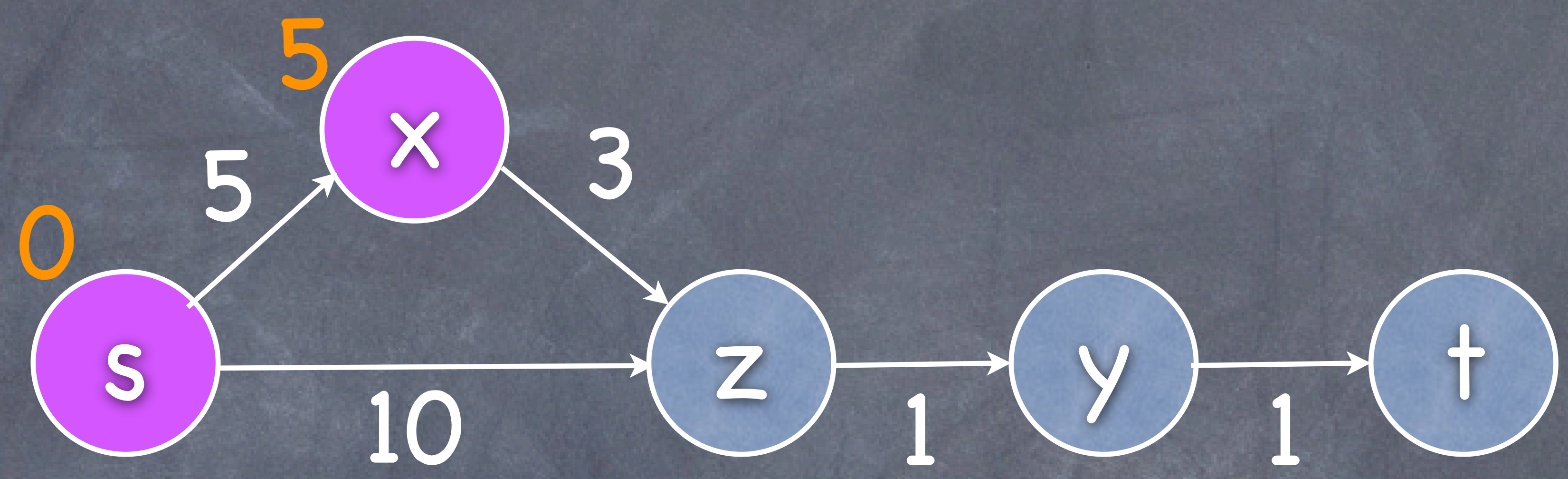
- while S != V{

    Compute the best known distance for frontier nodes using the distances of nodes in S and edges from nodes in S to nodes in F.

    Add the frontier node with the shortest best known distance to S.

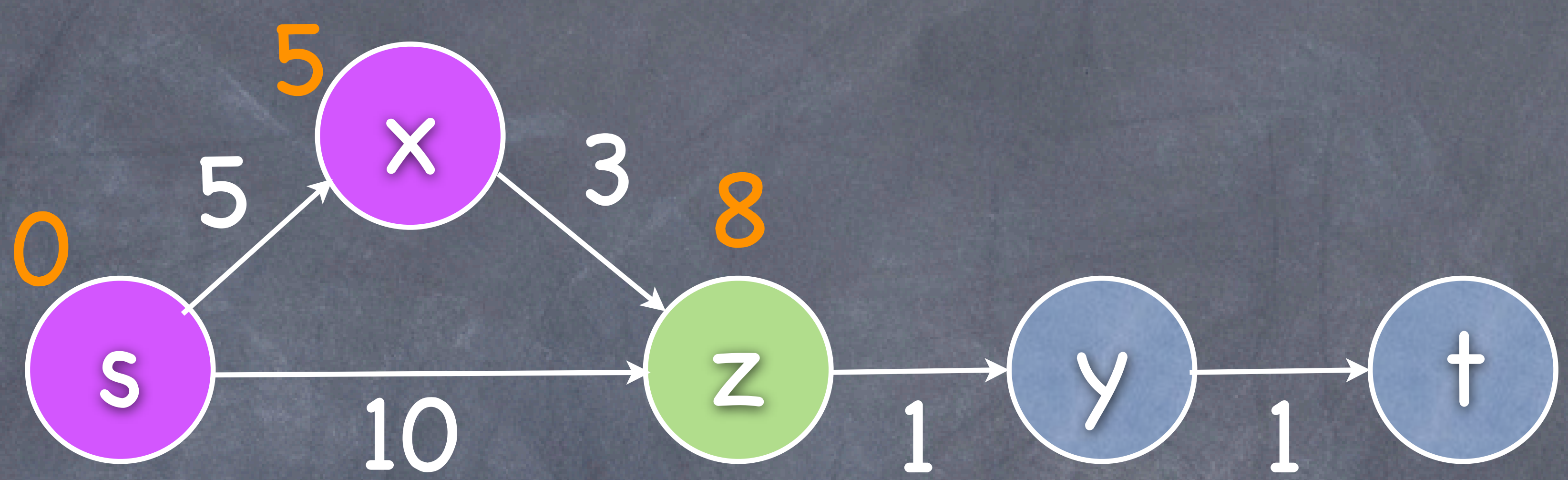
  }

Dijkstra's Single Source Shortest Paths Algorithm



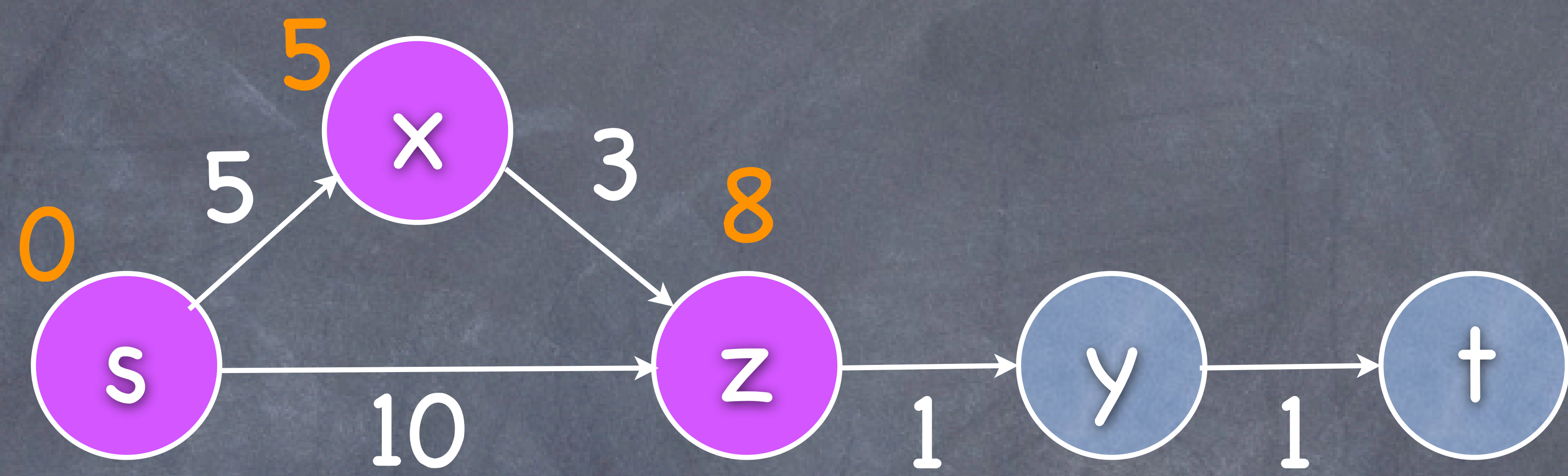
- while S != V{
    Compute the best known distance for frontier nodes using the distances of nodes in S and edges from nodes in S to nodes in F.
    Add the frontier node with the shortest best known distance to S.
  }

Dijkstra's Single Source Shortest Paths Algorithm



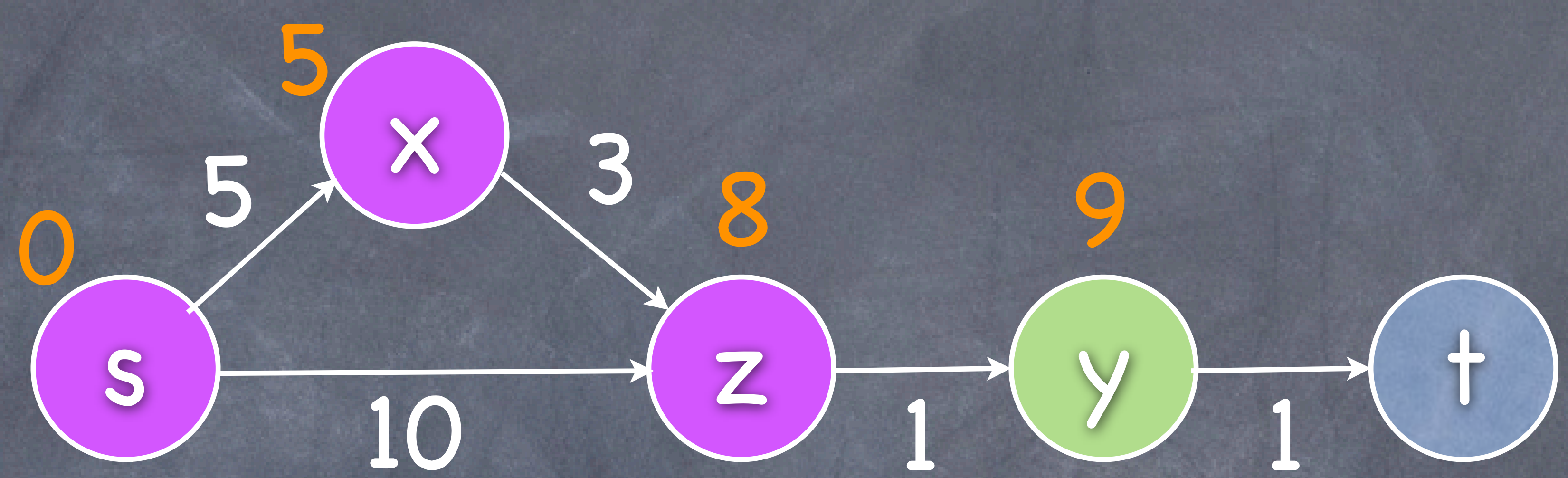
- while S != V{
    Compute the best known distance for frontier nodes using the distances of nodes in S and edges from nodes in S to nodes in F.
    Add the frontier node with the shortest best known distance to S.
  }

Dijkstra's Single Source Shortest Paths Algorithm



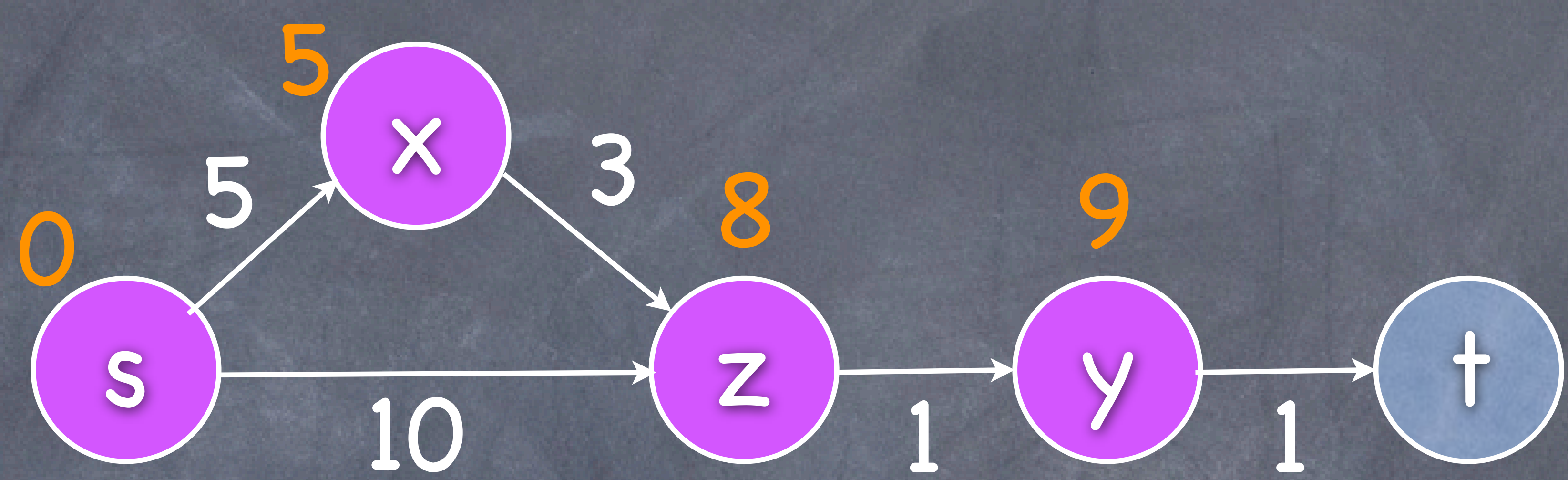
- while S != V{
  Compute the best known distance for frontier nodes using the distances of nodes in S and edges from nodes in S to nodes in F.
  Add the frontier node with the shortest best known distance to S.
  }

Dijkstra's Single Source Shortest Paths Algorithm



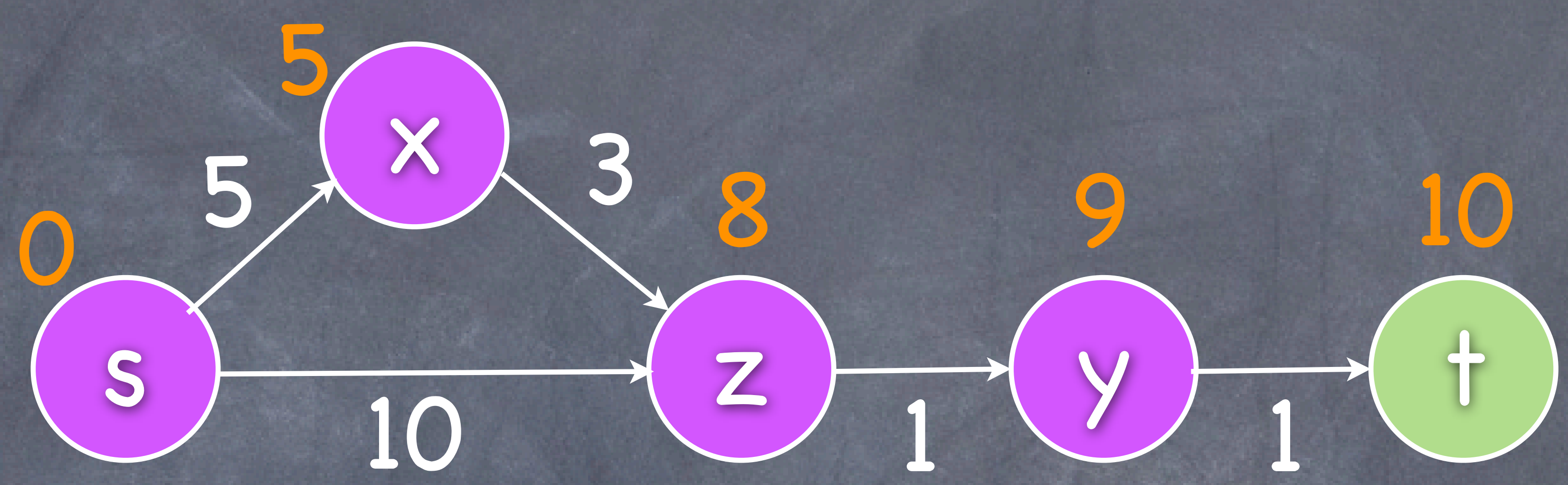
- while S != V{
    Compute the best known distance for frontier nodes using the distances of nodes in S and edges from nodes in S to nodes in F.
    Add the frontier node with the shortest best known distance to S.
  }

Dijkstra's Single Source Shortest Paths Algorithm



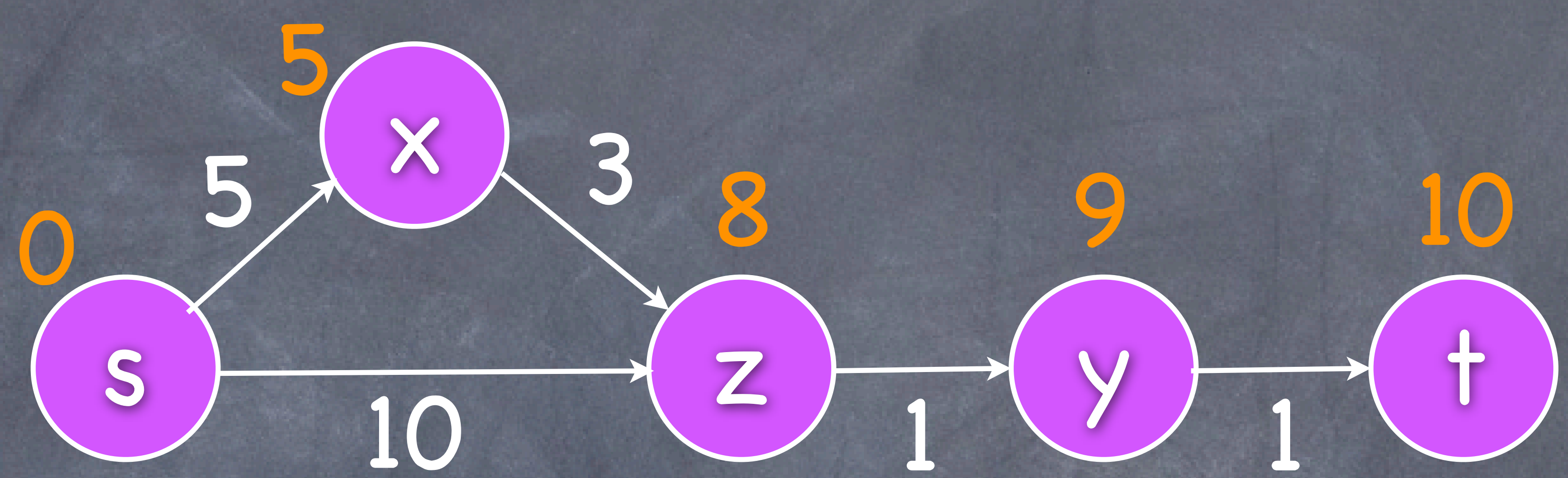
- while S != V{
    Compute the best known distance for frontier nodes using the distances of nodes in S and edges from nodes in S to nodes in F.
    <u>Add the frontier node with the shortest best known distance to S.</u>
  }

Dijkstra's Single Source Shortest Paths Algorithm



- while S != V{
  <u>Compute the best known distance for frontier nodes using the distances of nodes in S and edges from nodes in S to nodes in F.</u>
  Add the frontier node with the shortest best known distance to S.
  }

Dijkstra's Single Source Shortest Paths Algorithm



- while S != V{

    Compute the best known distance for frontier nodes using the distances of nodes in S and edges from nodes in S to nodes in F.

    <u>Add the frontier node with the shortest best known distance to S.</u>

  }

Correctness

- What does it mean that Dijkstra's algorithm is correct?

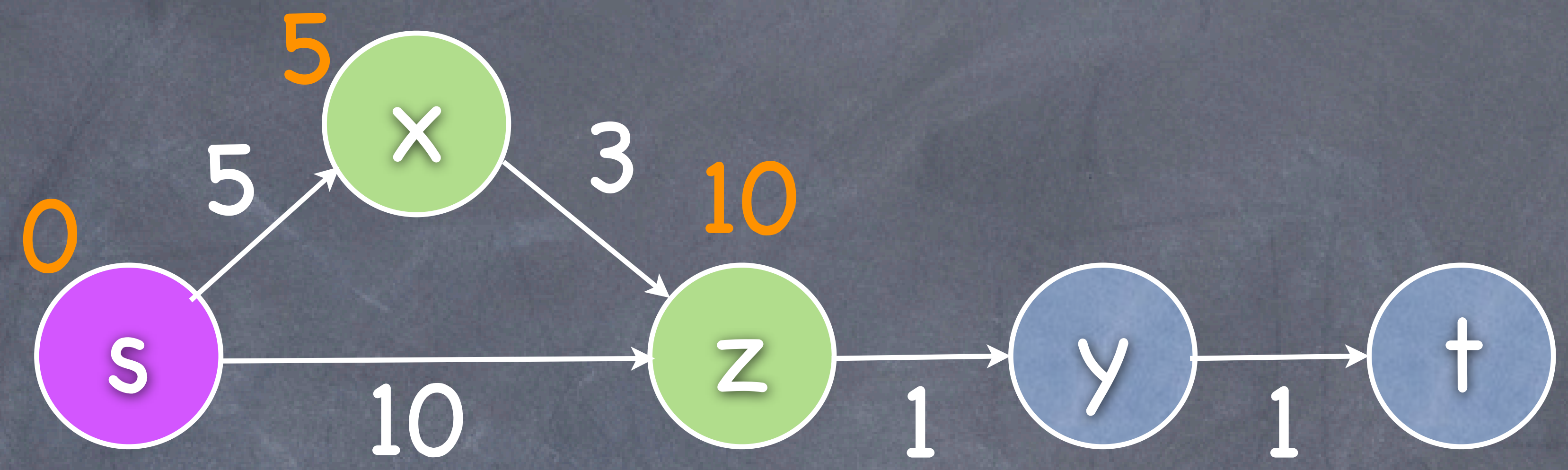
Correctness

- (When the algorithm finishes) The best known distances of nodes in S are the shortest distances.
- Noting that S is constructed by successive extension to some initial value, you are advised to generalize the above statement to:
  - At all times, the best known distances of nodes in S are the shortest distances. (An invariant on S)

Correctness

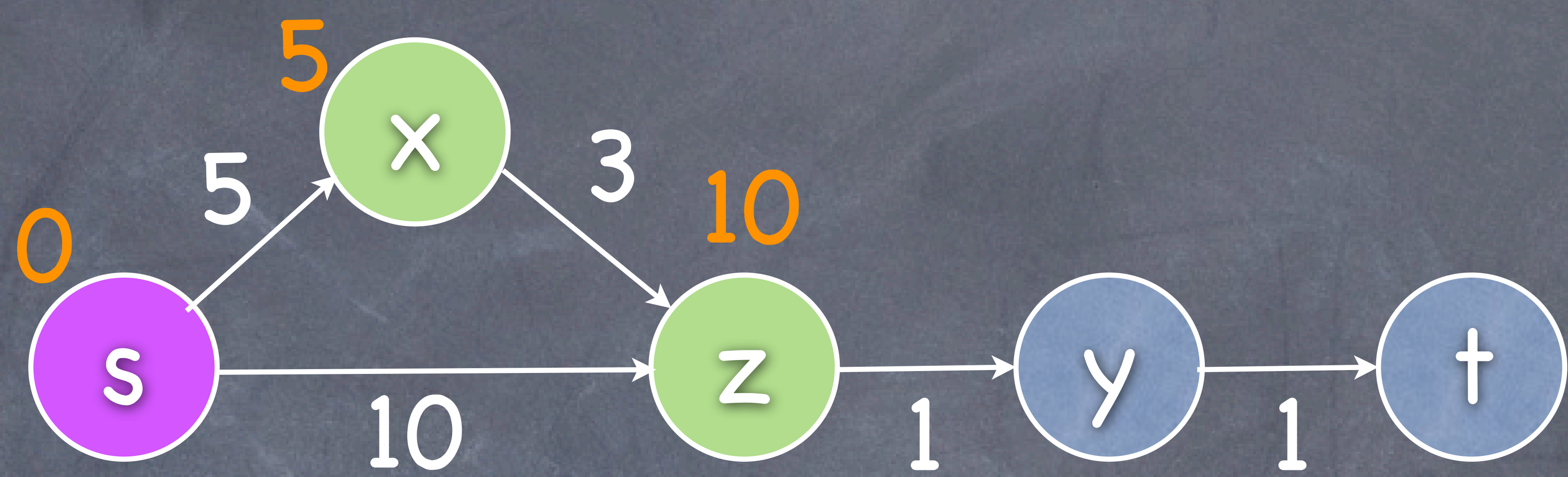
- Also, by noting that S is constructed by successive extension to some initial value, you are advised to use induction to prove the invariant.
- Base case: [S={s}, bkd(s) = 0] satisfies the invariant.
- Induction hypothesis: [S={v1..vk}, bkd(v) is shortest] imply that the bkd(x), where x is the node added by Dijkstra's algorithm to S, is the shortest.

Correctness



- The path from s to any node not in S must go through some frontier node (by definition of frontier nodes)
- The bkd for a frontier node is the shortest using only nodes in S (by definition of frontier nodes and IH)
- let x be the frontier node with the minimum bkd, then all other paths to x that go through nodes not in S are not less expensive than bkd(x). Why?

Correctness



- The path from s to any node not in S must go through some frontier node (by definition of frontier nodes)
- The bkd for a frontier node is the shortest using only nodes in S (by definition of frontier nodes and IH)
- let x be the frontier node with the minimum bkd, then all other paths to x that go through nodes not in S are not less expensive than bkd(x). Why? <u>Because these paths must go through a frontier node whose distance is not less than bkd(x) and that edge costs are positive.</u>

Complexity

- while S != V{
  Compute the best known distance for frontier nodes using the distances of nodes in S and edges from nodes in S to nodes in F.
  Add the frontier node with the shortest best known distance to S.
  }

Complexity

- while S != V{
    Compute the best known distance for frontier nodes using the distances of nodes in S and edges from nodes in S to nodes in F.
    Add the frontier node with the shortest best known distance to S.
  }
- The outer loop iterates n times for a graph with n nodes. Each iteration considers all edges coming out of some node in S. These edges are comparable to all edges in the graph especially in later iterations and thus are bounded by the total number of nodes in the graph m.
- Overall complexity = O(m.n)

Further Optimizations

- Incrementally maintain the best known distances of frontier nodes under element addition to S rather than recomputing it from scratch every time.
- Keep frontier nodes in a priority queue ordered by their best known distance this allows us to maintain the closest node under removal of nodes from the priority queue as well as under changes in best known distances.

```java
public Map<Node, Integer> simpleDijkstra(Node source) {
  Map<Node, Integer> distances = new HashMap<Node, Integer>();
  distances.put(source, 0);
  while(true){
    MinAcc acc = new MinAcc(null, Integer.MAX_VALUE);
    for (Entry<Node, Integer> entry : distances.entrySet()) {
      for (Edge edge : entry.getKey().getOutgoingEdges()) {
        Node trgt = edge.getTarget();
        if(!distances.containsKey(trgt)) {
          int srcDistance = entry.getValue();
          int trgtDistance = srcDistance + edge.getLength();
          acc.acc(trgt, trgtDistance);
        }
      }
    }
    if(acc.getClosest() == null) break;
    distances.put(acc.getClosest(), acc.getClosestDistance());
  }
  return distances;
}
```

```java
class MinAcc{
    Node closest;
    int closestDistance;
    //Constructor, getters elided
    public void acc(Node node, int distance){
        if(distance < closestDistance){
            closestDistance = distance;
            closest = node;
        }
    }
}
```

- S : keySet (distances)
- BKD is computed on the fly.
- Not terminating the while loop when S contains all of the graph nodes. Benefits?

```java
public Map<Node, Integer> simpleDijkstra(Node source) {
  Map<Node, Integer> distances = new HashMap<Node, Integer>();
  distances.put(source, 0);
  while(true){
    MinAcc acc = new MinAcc(null, Integer.MAX_VALUE);
    for (Entry<Node, Integer> entry : distances.entrySet()) {
      for (Edge edge : entry.getKey().getOutgoingEdges()) {
        Node trgt = edge.getTarget();
        if(!distances.containsKey(trgt)) {
          int srcDistance = entry.getValue();
          int trgtDistance = srcDistance + edge.getLength();
          acc.acc(trgt, trgtDistance);
        }
      }
    }
    if(acc.getClosest() == null) break;
    distances.put(acc.getClosest(), acc.getClosestDistance());
  }
  return distances;
}
```

```java
class MinAcc{
    Node closest;
    int closestDistance;
    //Constructor, getters elided
    public void acc(Node node, int distance){
        if(distance < closestDistance){
            closestDistance = distance;
            closest = node;
        }
    }
}
```

- What if the graph had a node that is not reachable from s?
- The algorithm is designed with an "ideal" world in mind.
- To have a "good" real world implementation, we need to consider corner cases and ask "what if ..." question.

```java
public Map<Node, Integer> simpleDijkstra(Node source) {
  Map<Node, Integer> distances = new HashMap<Node, Integer>();
  distances.put(source, 0);
  while(true){
    MinAcc acc = new MinAcc(null, Integer.MAX_VALUE);
    for (Entry<Node, Integer> entry : distances.entrySet()) {
      for (Edge edge : entry.getKey().getOutgoingEdges()) {
        Node trgt = edge.getTarget();
        if(!distances.containsKey(trgt)) {
          int srcDistance = entry.getValue();
          int trgtDistance = srcDistance + edge.getLength();
          acc.acc(trgt, trgtDistance);
        }
      }
    }
    if(acc.getClosest() == null) break;
    distances.put(acc.getClosest(), acc.getClosestDistance());
  }
  return distances;
}
```

```java
class MinAcc{
    Node closest;
    int closestDistance;
    //Constructor, getters elided
    public void acc(Node node, int distance){
        if(distance < closestDistance){
            closestDistance = distance;
            closest = node;
        }
    }
}
```

- Consider the underlined code block. What does it read and what does it produce?
- Note that it is executed several times with slightly modified distance object.
- Can the loops there be executed in any order?

```java
public Map<Node, Integer> simpleDijkstra(Node source) {
    Map<Node, Integer> distances = new HashMap<Node, Integer>();
    distances.put(source, 0);
    Execute the code that computes minAcc;
    if(!acc.getClosest() == null){
        distances.put(acc.getClosest(), acc.getClosestDistance());
        while(true){
            Maintain acc under adding the closest node to distances;
            if(acc.getClosest() == null) break;
            distances.put(acc.getClosest(), acc.getClosestDistance());
        }
    }
    return distances;
}
```

- We can incrementalize the computation of minAcc.

```java
class MinAcc{
    Node closest;
    int closestDistance;
     //Constructor, getters elided
    public void acc(Node node, int distance){
        if(distance < closestDistance){
            closestDistance = distance;
            closest = node;
        }
    }
}
```

Implementation

```
public Map<Node, Integer> simpleDijkstra(Node source) {
  Map<Node, Integer> distances = new HashMap<Node, Integer>();
  distances.put(source, 0);
  MinAcc acc = new MinAcc(null, Integer.MAX_VALUE);
  for (Entry<Node, Integer> entry : distances.entrySet()) {
    for (Edge edge : entry.getKey().getOutgoingEdges()) {
      Node trgt = edge.getTarget();
      if(!distances.containsKey(trgt)) {
        int srcDistance = entry.getValue();
        int trgtDistance = srcDistance + edge.getLength();
        acc.acc(trgt, trgtDistance);
      }
    }
  }
  if(!acc.getClosest() == null){
    distances.put(acc.getClosest(), acc.getClosestDistance());
    while(true){
      Maintain acc under adding the closest node to distances;
      if(acc.getClosest() == null) break;
        distances.put(acc.getClosest(), acc.getClosestDistance());
      }
    }
    return distances;
}
```

- Given that distances contain a single entry (source, 0) we can simplify the underlined block.

Implementation

```
public Map<Node, Integer> simpleDijkstra(Node source) {
    Map<Node, Integer> distances = new HashMap<Node, Integer>();
    distances.put(source, 0);
    MinAcc acc = new MinAcc(null, Integer.MAX_VALUE);
    for (Edge edge : source.getOutgoingEdges()) {
        Node trgt = edge.getTarget();
        int srcDistance = 0;
        int trgtDistance = srcDistance + edge.getLength();
        acc.acc(trgt, trgtDistance);
    }
    if(!acc.getClosest() == null){
        distances.put(acc.getClosest(), acc.getClosestDistance());
        while(true){
            Maintain acc under adding the closest node to distances;
            if(acc.getClosest() == null) break;
            distances.put(acc.getClosest(), acc.getClosestDistance());
        }
    }
    return distances;
}
```

```
class MinAcc{
    Node closest;
    int closestDistance;
    //Constructor, getters elided
    public void acc(Node node, int distance){
        if(distance < closestDistance){
            closestDistance = distance;
            closest = node;
        }
    }
}
```

- Given that distances contain a single entry (source, 0) we can simplify the underlined block.
- No need for the outer loop. or the if statement.
- entry.getKey() is source
- entry.getValue() is 0

Implementation

```
public Map<Node, Integer> simpleDijkstra(Node source) {
    Map<Node, Integer> distances = new HashMap<Node, Integer>();
    distances.put(source, 0);
    MinAcc acc = new MinAcc(null, Integer.MAX_VALUE);
    for (Edge edge : source.getOutgoingEdges()) {
        Node trgt = edge.getTarget();
        int trgtDistance = edge.getLength();
        acc.acc(trgt, trgtDistance);
    }
    if(!acc.getClosest() == null){
        distances.put(acc.getClosest(), acc.getClosestDistance());
        while(true){
            Maintain acc under adding the closest node to distances;
            if(acc.getClosest() == null) break;
            distances.put(acc.getClosest(), acc.getClosestDistance());
        }
    }
    return distances;
}
```

```
class MinAcc{
    Node closest;
    int closestDistance;
    //Constructor, getters elided
    public void acc(Node node, int distance){
        if(distance < closestDistance){
            closestDistance = distance;
            closest = node;
        }
    }
}
```

- We can do constant propagation of srcDistance and simplification.

Implementation

```
public Map<Node, Integer> simpleDijkstra(Node source) {
  Map<Node, Integer> distances = new HashMap<Node, Integer>();
  distances.put(source, 0);
  ...;
  if(!acc.getClosest() == null){
    distances.put(acc.getClosest(), acc.getClosestDistance());
    while(true){
      Maintain acc under adding the closest node to distances;
      if(acc.getClosest() == null) break;
        distances.put(acc.getClosest(), acc.getClosestDistance());
      }
    }
  return distances;
}
```

```
class MinAcc{
    Node closest;
    int closestDistance;
    //Constructor, getters elided
    public void acc(Node node, int distance){
        if(distance < closestDistance){
            closestDistance = distance;
            closest = node;
        }
    }
}
```

- How to maintain acc under distances.put (acc.getClosest(), acc.getClosestDistance())?

Implementation

```
for (Entry<Node, Integer> entry : distances.entrySet()) {
    for (Edge edge : entry.getKey().getOutgoingEdges()) {
        Node trgt = edge.getTarget();
        if(!distances.containsKey(trgt)) {
            int srcDistance = entry.getValue();
            int trgtDistance = srcDistance + edge.getLength();
            acc.acc(trgt, trgtDistance);
        }
    }
}
```

- Adding an entry to distances affects the control flow
  - The outer loop shall have one additional iteration.
  - The body of the if-statement should be “undone” for the cases where trgt is the same as the key of the newly added entry.

```
Node closest = acc.getClosest();
int closestDistance= acc.getClosestDistance();
for (Edge edge : closest.getOutgoingEdges()) {
    Node trgt = edge.getTarget();
    if(!distances.containsKey(trgt)) {
        int srcDistance = closestDistance;
        int trgtDistance = srcDistance + edge.getLength();
        acc.acc(trgt, trgtDistance);
    }
}
acc.remove(closest); <== How to implement? can be replaced by
acc.removeClosest()
```

- Adding an entry to distances affects the control flow
    - The outer loop shall have one additional iteration.
    - The body of the if-statement should be "undone" for the cases where trgt is the same as the key of the newly added entry.

Implementation

```
class MinAcc{
    PriorityQueue<Node,Integer> pq = ...;
   public MinAcc(Node closest, int closestDistance) {
       pq.put(closest,closestDistance);
   }
   public Node getClosest() {
       return pq.min().getValue();
   }
   public int getClosestDistance() {
       return pq.min().getKey();
   }
   public void acc(Node node, int distance){
         if(pq.containsValue(node){
            int bkd = pq.getKey(node);
            if(distance < bkd){
                  pq.decreseKey(node, distance);
              }
         }else{
          pq.put(closest,closestDistance);
       }
   }
   public void removeClosest(){
       pq.poll();
   }
}
```

Complexity

```
public Map<Node, Integer> simpleDijkstra(Node source) {
    Map<Node, Integer> distances = new HashMap<Node, Integer>();
    distances.put(source, 0);
    MinAcc acc = new MinAcc(null, Integer.MAX_VALUE);
    for (Edge edge : source.getOutgoingEdges()) {
        Node trgt = edge.getTarget();
        int trgtDistance = edge.getLength();
        acc.acc(trgt, trgtDistance);
    }
    if(!acc.getClosest() == null){
        distances.put(acc.getClosest(), acc.getClosestDistance());
        while(true){
            Node closest = acc.getClosest();
            int closestDistance= acc.getClosestDistance();
            for (Edge edge : closest.getOutgoingEdges()) {
                Node trgt = edge.getTarget();
                if(!distances.containsKey(trgt)) {
                    int srcDistance = closestDistance;
                    int trgtDistance = srcDistance + edge.getLength();
                    acc.acc(trgt, trgtDistance);
                }
            }
            acc.removeClosest();
            if(acc.getClosest() == null) break;
            distances.put(acc.getClosest(), acc.getClosestDistance());
        }
    }
    return distances;
}
```

- acc.removeClosest is invoked n times
- acc.acc is invoked m times

Complexity

```
class MinAcc{
    PriorityQueue<Node,Integer> pq = ...;
   public MinAcc(Node closest, int
closestDistance) {
      pq.put(closest,closestDistance);
   }
   public Node getClosest() {
      return pq.min().getValue(); O(1)
   }
   public int getClosestDistance() {
      return pq.min().getKey(); O(1)
   }
   public void acc(Node node, int distance){
         if(pq.containsValue(node){
            int bkd = pq.getKey(node);
            if(distance < bkd){
                  pq.decreaseKey(node, distance); O
(lg n)/O(1)
             }
         }else{
          pq.put(closest,closestDistance); O(lg n)/
      O(1)
      }
   }
   public void removeClosest(){
      pq.poll(); O(lg n)
   }
}
```

- acc.removeClosest is invoked n times
- acc.acc is invoked m times
- The priority queue is bounded by n (in reality would we have that large of a frontier?)
- decreaseKey and put are O(lg n) in binary heaps and O(1) in Fibonacci heaps.
- Using binary heaps, Dijkstra is O((m +n)*lg n) = O(m * lg n).
- Using Fibonacci heaps, Dijkstra is O(m + n * lg n)

Exercise - Tricking Dijkstra

- Give a weighted directed graph G, where weights are not necessarily positive, such that Dijkstra produces wrong results.

Questions?

Thank You