Dijkstra’s Shortest Paths

By: Ahmed Abdelmeged
and: Karl Lieberherr
Single Source Shortest Paths Problem

Given:
- a directed graph $G(V,E)$,
- edge weights $w:E\rightarrow \mathbb{R}$ 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 $s$. 

Wednesday, February 29, 2012
Brute Force Algorithm

- **Try** all paths.
- **for** $v \in V$
  - let $\text{paths}_v$ be the set of all paths from $s$ to $v$
  - output $v$, $\min(\text{cost}(\text{paths}_v))$

**Q:** Can you upper bound $|\text{paths}_v|$?
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) ∈ 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 \((w:E\rightarrow\mathbb{R}^+)\). 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 $\text{bestKnownDistance}(s)$ 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{
    \text{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.}
    \text{Add the frontier node with the shortest best known distance to S.}
}

Wednesday, February 29, 2012
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.
}

Wednesday, February 29, 2012
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.
}

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.
}

Wednesday, February 29, 2012
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

\[ \text{while } S \neq V \{
\text{Compute the best known distance for frontier nodes using the distances of nodes in } S \text{ and edges from nodes in } S \text{ to nodes in } F. \\
\text{Add the frontier node with the shortest best known distance to } S.
\} \]
Dijkstra's Single Source Shortest Paths Algorithm

while $S \neq V$

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 $S$.

}
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: $[S=\{s\}, \text{bkd}(s) = 0]$ satisfies the invariant.

Induction hypothesis: $[S=\{v_1..v_k\}, \text{bkd}(v) \text{ is shortest}]$ imply that the $\text{bkd}(x)$, where $x$ is the node added by Dijkstra's algorithm to $S$, is the shortest.
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 $\text{bkd}(x)$. Why?
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? Because these paths must go through a frontier node whose distance is not less than $bkd(x)$ and that edge costs are positive.
while $S \neq V$

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 $S \neq V$

    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.
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;
}

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.
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;
}

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

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

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

```java
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 trg = edge.getTarget();
            if (!distances.containsKey(trg)) {
                int srcDistance = entry.getValue();
                int trgDistance = srcDistance + edge.getLength();
                acc.acc(trg, trgDistance);
            }
        }
    }
    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;
}
```
Implementation

```java
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 trg = edge.getTarget();
        int srcDistance = 0;
        int trgDistance = srcDistance + edge.getLength();
        acc.acc(trg, trgDistance);
    }
    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
We can do constant propagation of srcDistance and simplification.

```java
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 trg = edge.getTarget();
        int trgDistance = edge.getLength();
        acc.acc(trg, trgDistance);
    }
    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;
        }
    }
}
```
Implementation

```java
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;
}
```

```java
class MinAcc{
    Node closest;
    int closestDistance;
    // Constructor, getters elided
    public void acc(Node node, int distance){
        if (distance < closestDistance){
            closestDistance = distance;
            closest = node;
        }
    }
}
```
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.
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.
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.containsKey(node)){
            int bkd = pq.getKey(node);
            if(distance < bkd){
                pq.decreaseKey(node, distance);
            }
        }else{
            pq.put(closest,closestDistance);
        }
    }
    public void removeClosest(){
        pq.poll();
    }
}
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 trg = edge.getTarget();
        int trgDistance = edge.getLength();
        acc.acc(trg, trgDistance);
    }
    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 trg = edge.getTarget();
                if (!distances.containsKey(trg)) {
                    int srcDistance = closestDistance;
                    int trgDistance = srcDistance + edge.getLength();
                    acc.acc(trg, trgDistance);
                }
            }
            acc.removeClosest();
            if (acc.getClosest() == null) break;
        }
        distances.put(acc.getClosest(), acc.getClosestDistance());
    }
    return distances;
}
Complexity

- 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 \( O((m + n)\star \lg n) = O(m \star \lg n) \).
- Using Fibonacci heaps, Dijkstra is \( O(m + n \star \lg n) \).
Exercise – Tricking Dijkstra

Give a weighted directed graph \( G \), where weights are not necessarily positive, such that Dijkstra produces wrong results.
Questions?
Thank You