Single Source Shortest Paths

- Given an edge weighted (non-negative weights) directed graph $G$ and a vertex $s$ in $G$, find shortest (least distance or cost) paths from vertex $s$ to all the other vertices in the graph.
- Every edge has a non-negative weight.
- The cost or distance of a path is the sum of the weight of the edges in the path.

Our algorithms finds shortests paths in increasing order of their cost (or distance).
Example

s is 1

Vertex | Cost of SP | SP
---|---|---
1 | 0 | 1
3 | 2 | 1 → 3
4 | 3 | 1 → 3 → 4
2 | 4 | 1 → 2
5 | 6 | 1 → 3 → 4 → 5
Ideas Behind Algorithm

Our algorithm computes the cost or distance of a shortest path and leaves a pointer to the parent of each node in a shortest path.

Greedy Criterion: (At each iteration) From all the vertices to which a shortest path from \( s \) has not yet been found, select one that results in a least cost (distance) path.
Iteration Invariant

- At each iteration the algorithm has computed the distance (or cost) of a shortest path from vertex \( s \) to every vertex in a set of vertices called \( S \) (initially \( S = \{s\} \)).

- A shortest path from \( s \) to a vertex that is not in \( S \) has distance (cost) that is at least as large as the distance of any shortest path from \( s \) to a vertex in \( S \).

- Every vertex has a value called \text{dist}. For a vertex \( v \) in \( S \), \text{dist} is the distance of a shortest path from \( s \) to \( v \).

- For every vertex \( v \) that is not in \( S \), \text{dist} is the distance of a shortest path from vertex \( s \) to vertex \( v \) that can only visit vertices in \( S \) as intermediate vertices.
• Let $v$ be a vertex that is not in $S$ with smallest $\text{dist}$ value. We claim that a shortest path from $s$ to $v$ that may visit any vertex in $G$ as an intermediate vertex has distance equal to $v->\text{dist}$.

• This claim follows from the fact that any path from $s$ to $v$ that visits other vertices that are not in $S$ has distance that is not smaller than $v->\text{dist}$ as the weight of each edge cannot be negative.

For example ...

Path $s - x - d - v$ is shorter than path $s - a - b - c - d - v$
class Vertex{

private:
    int dist;  // Distance of the currently best path from s to v.
    bool known;  // True if a path with distance dist is a shortest path from s to v.
    int id;  // Vertex id.
    Vertex *path;  // Pointer to the predecessor of v in the current best path from s to v.
Class Vertex (Cont‘)

```cpp
public:
    Vertex( int i) {
        dist = 10000; known = false;
        id = i; path = Null;}

    int get_dist() {return dist;}
    void set_dist(int val) {dist=val;}
    bool get_known() {return known;}
    void set_known(bool val) {known=val;}
    vertex get_path() {return path;}
    void set_path(Vertex *v) {path=v;}
    printPath();
};
```
v is set to 1
Set "known" for vertex 1 to True
### v is set to 4
Set known for vertex 4 to True

### v is set to 2
Set "known" for vertex 2 to True
v is set to 3
Set "known" for vertex 3 to True

v is set to 5
Set "known" for vertex 5 to True

<table>
<thead>
<tr>
<th>vertex</th>
<th>known</th>
<th>dist</th>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>0</td>
<td>Null</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>10000</td>
<td>Null</td>
</tr>
</tbody>
</table>
v is set to 6

Set "known" for vertex 6 to True
Recovering Shortest Paths

Reversed tree of Shortest paths (v→path)

void Vertex::printPath()
{
    if (path != Null)
    {
        path->printPath();
        cout << " to " ;
    }
    cout << this.id; }

v->printPath                  Prints
-------                   ------
  1                  1
  2                1 to 2
  3              1 to 4 to 3
  4              1 to 4
  5            1 to 2 to 5
  6          1 to 2 to 5 to 6
Algorithm

void Graph::Dijkstra( Vertex *s)
{
    Vertex *v, *w;
    Edge *e;
    // We use e->COST() to return the cost of the
    // edge e from vertex v->id to vertex w->id
    s->set_dist(0);
for( ; ; )
{
    if (all vertices have the "known" field set to true) exit;
    // let v to an "unmarked" vertex with smallest v->dist, i.e.
    // from all the vertices with the "known" field false
    // vertex v has least "dist" value.
    d = MIN { y->dist() | !(y->get_known()) and y is a Vertex}
    v = Element of { y | !(y->get_known()) && y->get_dist() is equal
    v->set_known(true);
    for each edge e incident from v do
    {
        Assume that edge e is directed from v->id to w->id
        if (!w->get_known)
            if (v->get_dist() + e->COST() < w->get_dist())
                { // update w //
                    w->set_dist(v->get_dist() + e->COST())
                    w->set_path(v);
                } } } } }
Graph $G$ is represented by adjacency lists ($n$ vertices and $m$ edges)

- Implementation using the code provided
  - Find min takes $O(n)$ time. Repeated $n$ times $\rightarrow O(n^2)$.
  - Update dist takes $O(1)$. Repeated $m$ times takes $O(m)$.
  - Therefore, total time is $O(n^2 + m)$.

- Implementation using a heap for dist of vertices with “known” value true
  - Find min takes $O(\log n)$ time. Repeated $n$ times $\rightarrow O(n \log n)$.
  - Update dist takes $O(\log n)$. Repeated $m$ times takes $O(m \log n)$.
  - Therefore, total time is $O(m \log n)$.

- Can be implemented to take $O(m + n \log n)$ time using Fibonacci Heaps (CMPSC 230).