Single Source Shortest Paths

Madhavan Mukund

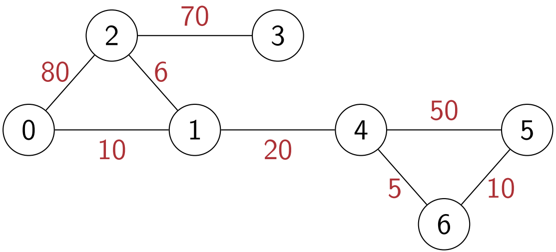
https://www.cmi.ac.in/~madhavan

Programming, Data Structures and Algorithms using Python

Week 5

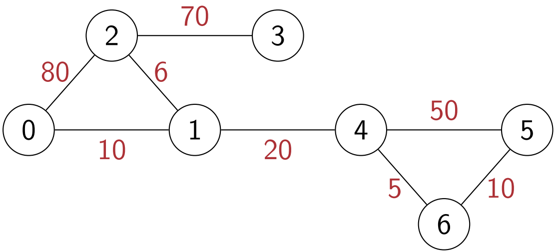
Single source shortest paths

- Weighted graph:
  - $G = (V, E)$
  - $W : E \to \mathbb{R}$
- Single source shortest paths
  - Find shortest paths from a fixed vertex to every other vertex



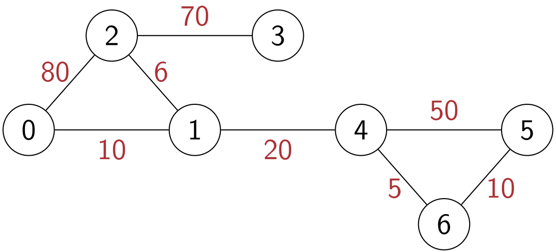
Single source shortest paths

- Weighted graph:
  - $G = (V, E)$
  - $W : E \to \mathbb{R}$
- Single source shortest paths
  - Find shortest paths from a fixed vertex to every other vertex
- Assume, for now, that edge weights are all non-negative



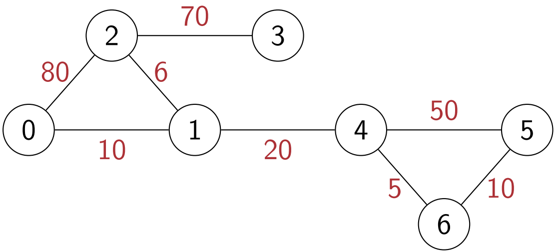
Single source shortest paths

- Compute shortest paths from 0 to all other vertices



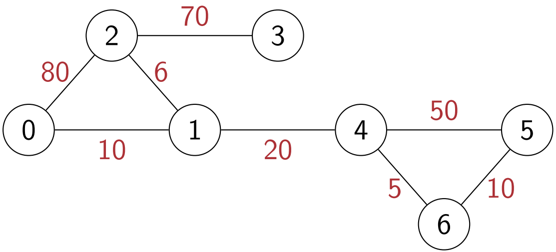
Single source shortest paths

- Compute shortest paths from 0 to all other vertices
- Imagine vertices are oil depots, edges are pipelines



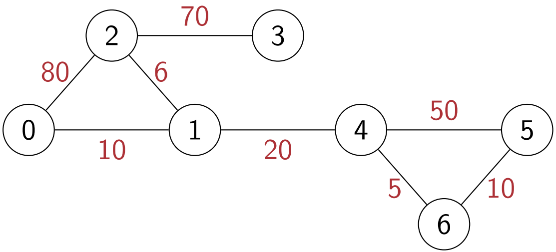
Single source shortest paths

- Compute shortest paths from 0 to all other vertices
- Imagine vertices are oil depots, edges are pipelines
- Set fire to oil depot at vertex 0



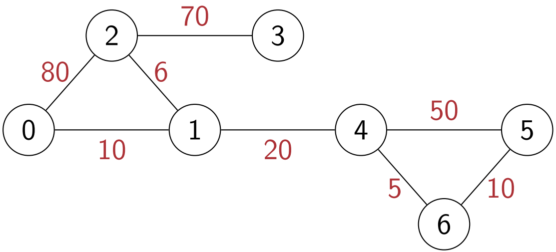
Single source shortest paths

- Compute shortest paths from 0 to all other vertices
- Imagine vertices are oil depots, edges are pipelines
- Set fire to oil depot at vertex 0
- Fire travels at uniform speed along each pipeline



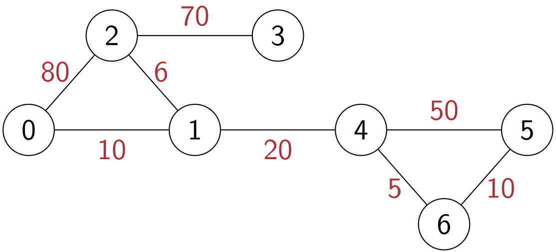
Single source shortest paths

- Compute shortest paths from 0 to all other vertices
- Imagine vertices are oil depots, edges are pipelines
- Set fire to oil depot at vertex 0
- Fire travels at uniform speed along each pipeline
- First oil depot to catch fire after 0 is nearest vertex



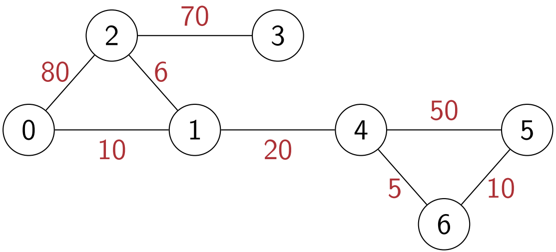
Single source shortest paths

- Compute shortest paths from 0 to all other vertices
- Imagine vertices are oil depots, edges are pipelines
- Set fire to oil depot at vertex 0
- Fire travels at uniform speed along each pipeline
- First oil depot to catch fire after 0 is nearest vertex
- Next oil depot is second nearest vertex



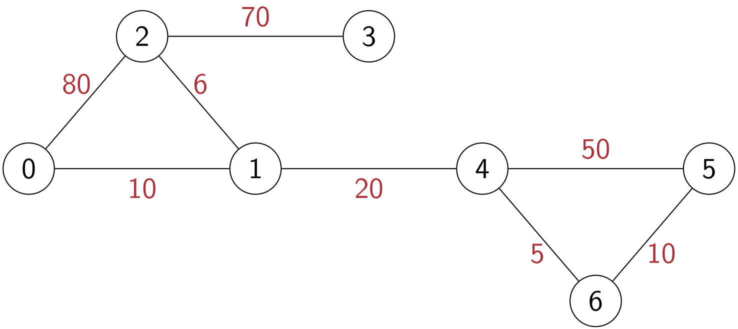
Single source shortest paths

- Compute shortest paths from 0 to all other vertices
- Imagine vertices are oil depots, edges are pipelines
- Set fire to oil depot at vertex 0
- Fire travels at uniform speed along each pipeline
- First oil depot to catch fire after 0 is nearest vertex
- Next oil depot is second nearest vertex
- ...



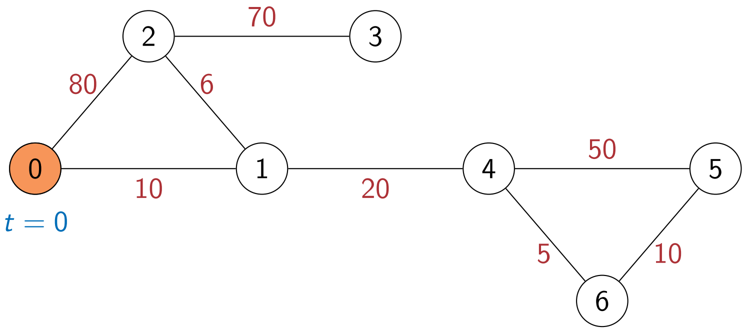
Single source shortest paths

- Set fire to oil depot at vertex 0
- Fire travels at uniform speed along each pipeline
- First oil depot to catch fire after 0 is nearest vertex
- Next oil depot is second nearest vertex
- . . .



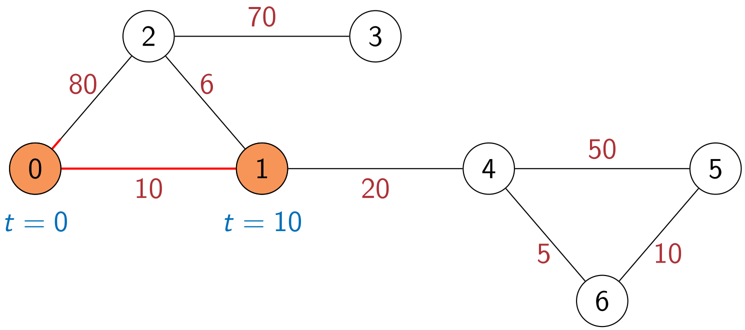
Single source shortest paths

- Set fire to oil depot at vertex 0
- Fire travels at uniform speed along each pipeline
- First oil depot to catch fire after 0 is nearest vertex
- Next oil depot is second nearest vertex
- ...



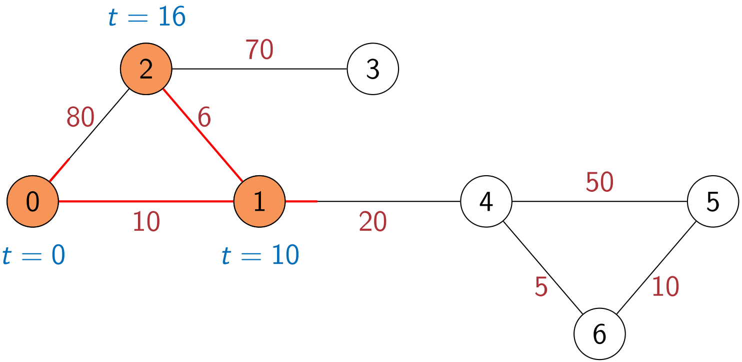
Single source shortest paths

- Set fire to oil depot at vertex 0
- Fire travels at uniform speed along each pipeline
- First oil depot to catch fire after 0 is nearest vertex
- Next oil depot is second nearest vertex
- ...



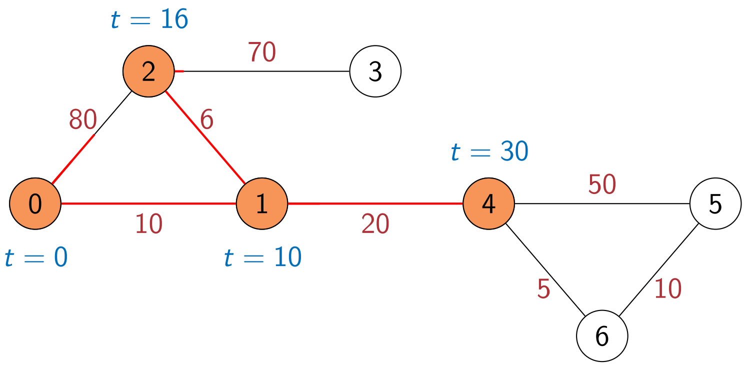
Single source shortest paths

- Set fire to oil depot at vertex 0
- Fire travels at uniform speed along each pipeline
- First oil depot to catch fire after 0 is nearest vertex
- Next oil depot is second nearest vertex
- ...



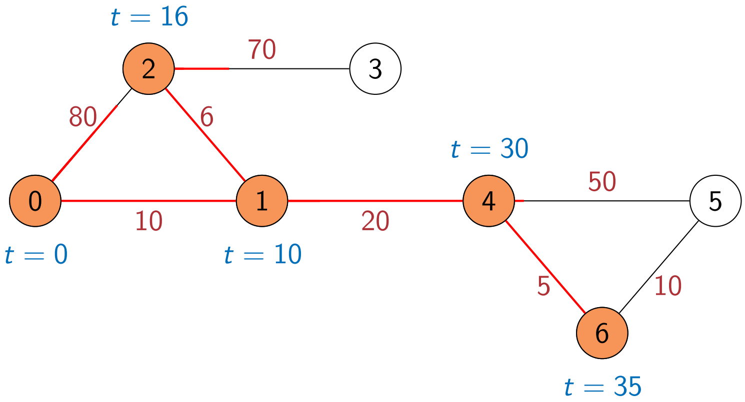
Single source shortest paths

- Set fire to oil depot at vertex 0
- Fire travels at uniform speed along each pipeline
- First oil depot to catch fire after 0 is nearest vertex
- Next oil depot is second nearest vertex
- ...



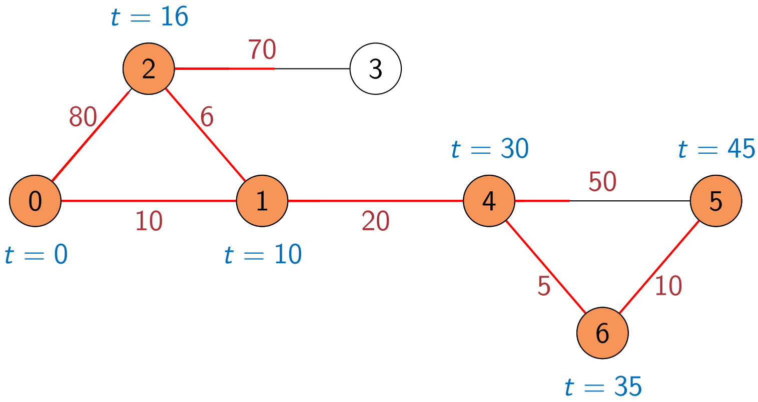
Single source shortest paths

- Set fire to oil depot at vertex 0
- Fire travels at uniform speed along each pipeline
- First oil depot to catch fire after 0 is nearest vertex
- Next oil depot is second nearest vertex
- ...



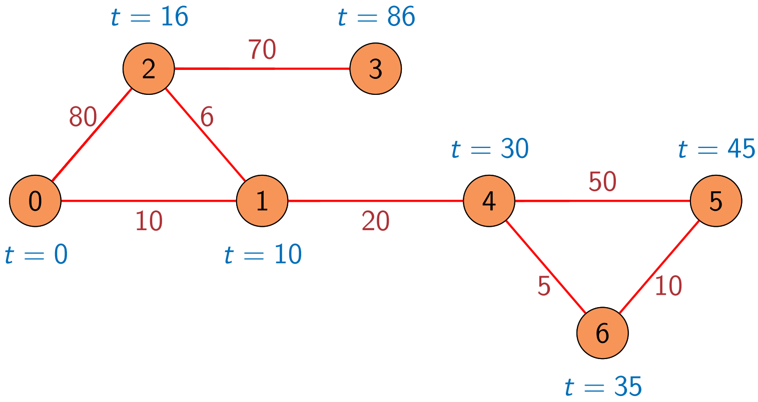
Single source shortest paths

- Set fire to oil depot at vertex 0
- Fire travels at uniform speed along each pipeline
- First oil depot to catch fire after 0 is nearest vertex
- Next oil depot is second nearest vertex
- ...



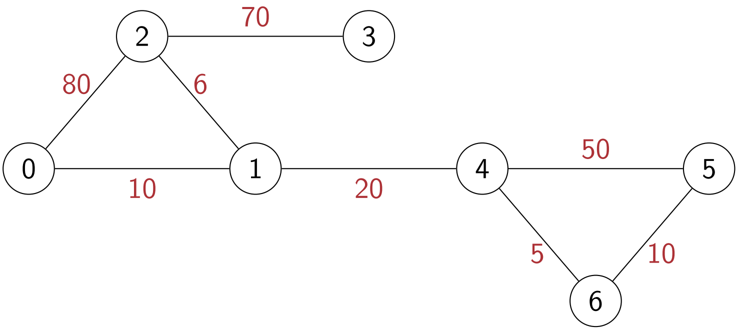
Single source shortest paths

- Set fire to oil depot at vertex 0
- Fire travels at uniform speed along each pipeline
- First oil depot to catch fire after 0 is nearest vertex
- Next oil depot is second nearest vertex
- ...



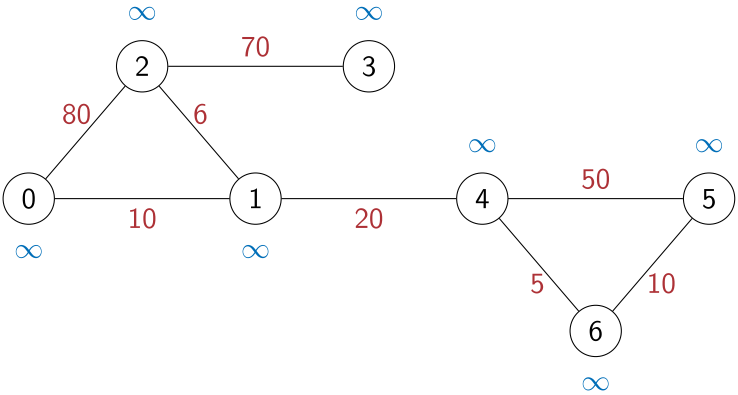
Single source shortest paths

- Compute expected burn time for each vertex
- Each time a new vertex burns, update the expected burn times of its neighbours



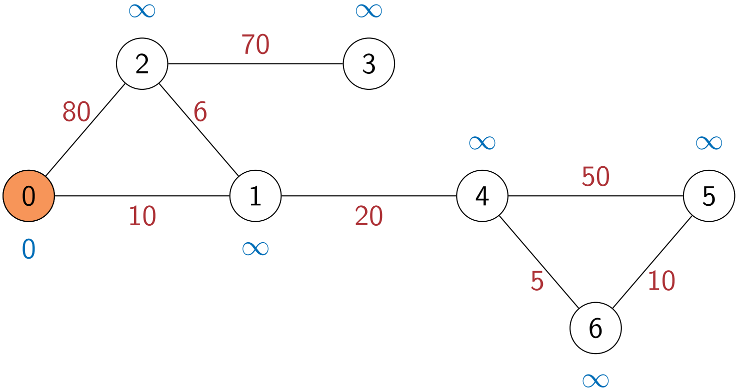
Single source shortest paths

- Compute expected burn time for each vertex
- Each time a new vertex burns, update the expected burn times of its neighbours



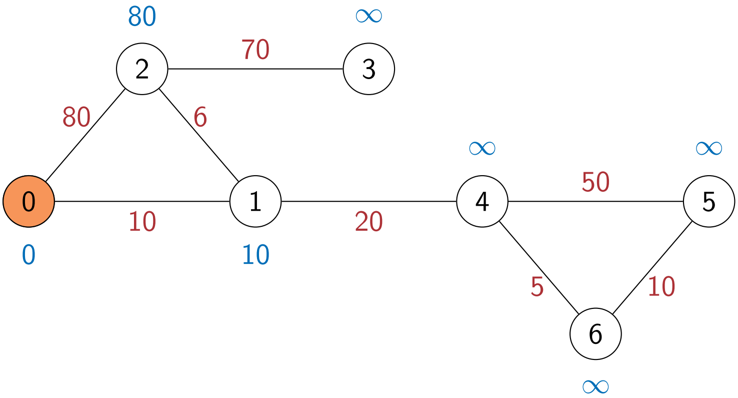
Single source shortest paths

- Compute expected burn time for each vertex
- Each time a new vertex burns, update the expected burn times of its neighbours



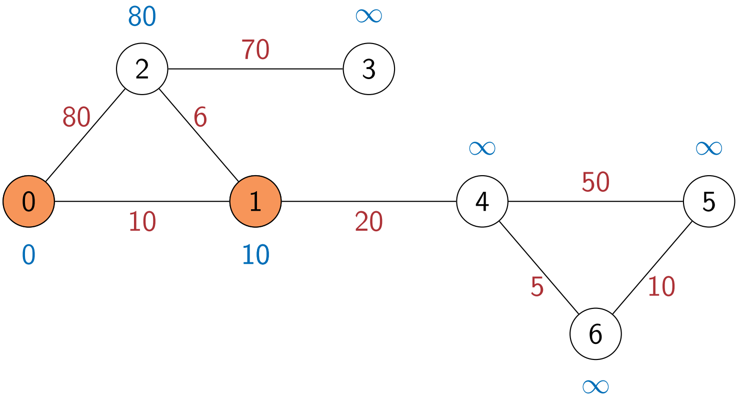
Single source shortest paths

- Compute expected burn time for each vertex
- Each time a new vertex burns, update the expected burn times of its neighbours



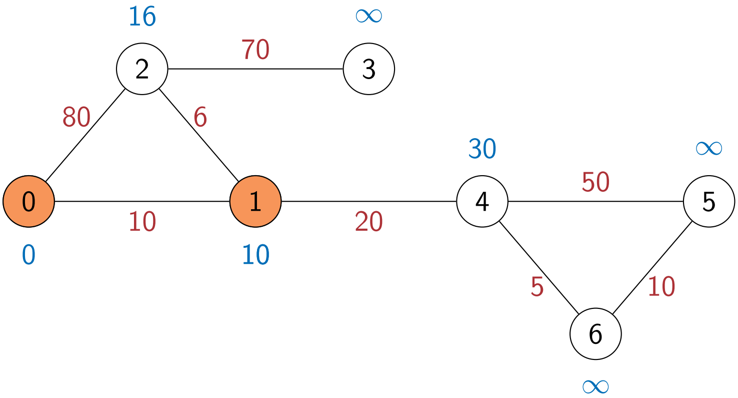
Single source shortest paths

- Compute expected burn time for each vertex
- Each time a new vertex burns, update the expected burn times of its neighbours



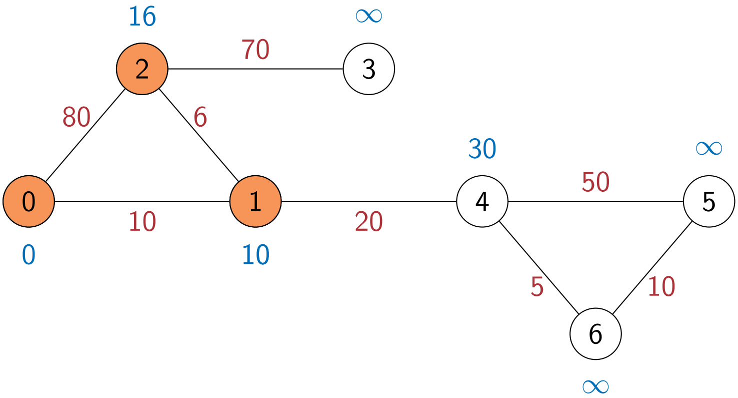
Single source shortest paths

- Compute expected burn time for each vertex
- Each time a new vertex burns, update the expected burn times of its neighbours



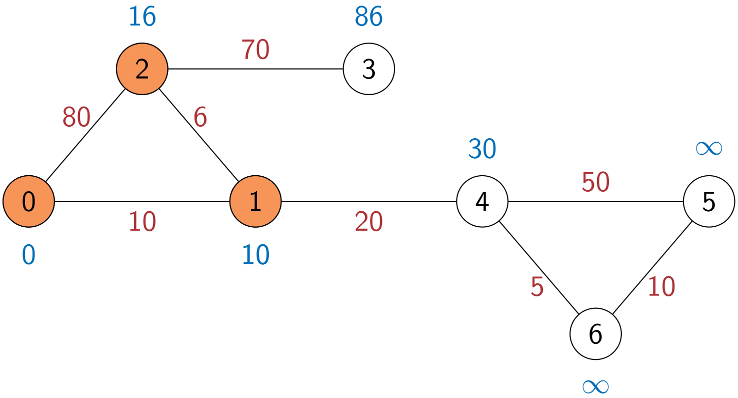
Single source shortest paths

- Compute expected burn time for each vertex
- Each time a new vertex burns, update the expected burn times of its neighbours



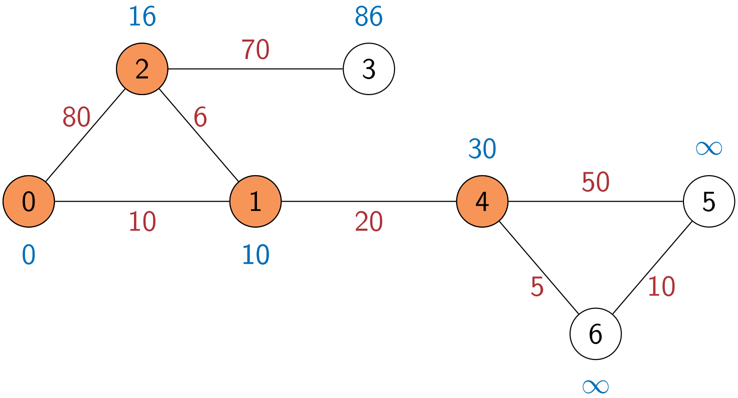
Single source shortest paths

- Compute expected burn time for each vertex
- Each time a new vertex burns, update the expected burn times of its neighbours



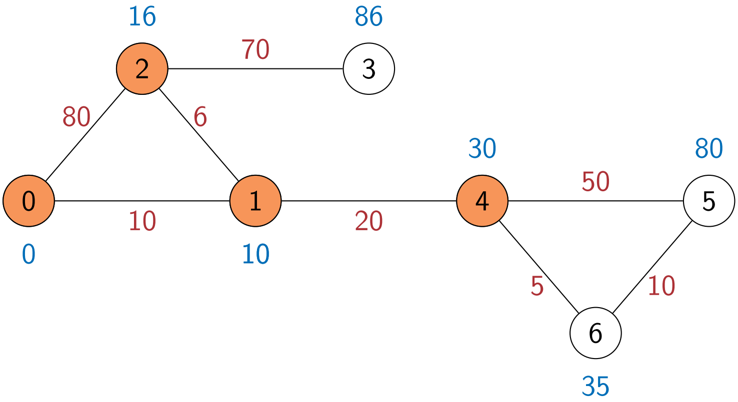
Single source shortest paths

- Compute expected burn time for each vertex
- Each time a new vertex burns, update the expected burn times of its neighbours



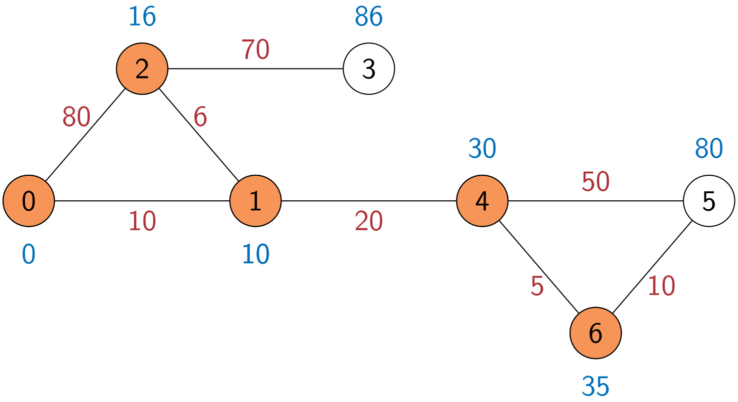
Single source shortest paths

- Compute expected burn time for each vertex
- Each time a new vertex burns, update the expected burn times of its neighbours



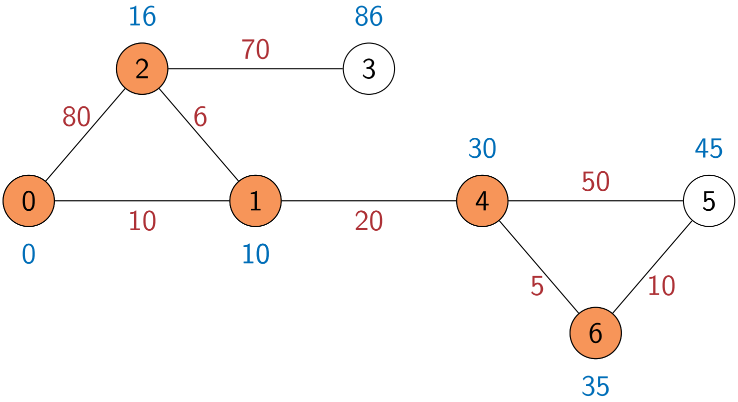
Single source shortest paths

- Compute expected burn time for each vertex
- Each time a new vertex burns, update the expected burn times of its neighbours



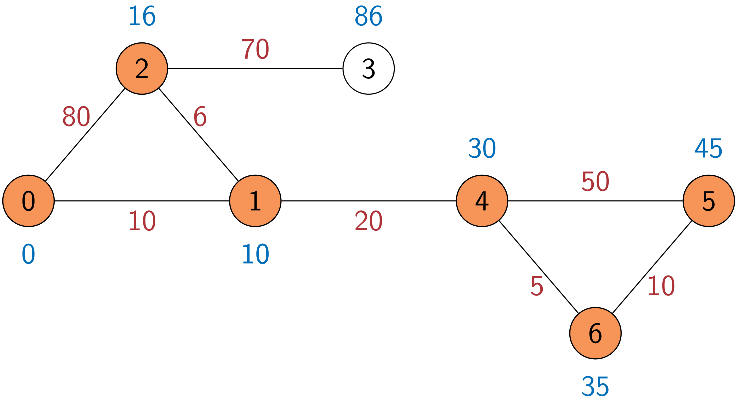
Single source shortest paths

- Compute expected burn time for each vertex
- Each time a new vertex burns, update the expected burn times of its neighbours



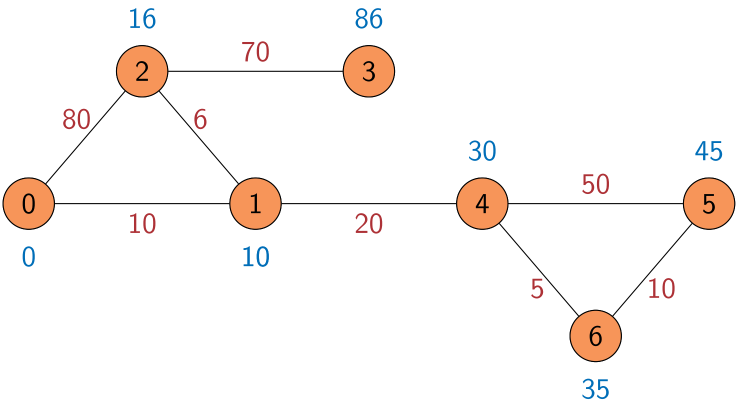
Single source shortest paths

- Compute expected burn time for each vertex
- Each time a new vertex burns, update the expected burn times of its neighbours



Single source shortest paths

- Compute expected burn time for each vertex
- Each time a new vertex burns, update the expected burn times of its neighbours
- Algorithm due to Edsger W Dikjstra



Dijkstra's algorithm: Proof of correctness

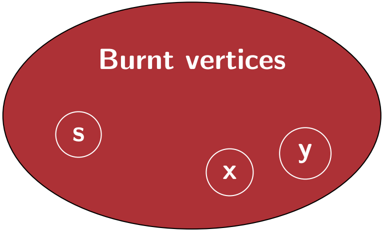
- Each new shortest path we discover extends an earlier one

Dijkstra's algorithm: Proof of correctness

- Each new shortest path we discover extends an earlier one
- By induction, assume we have found shortest paths to all vertices already burnt

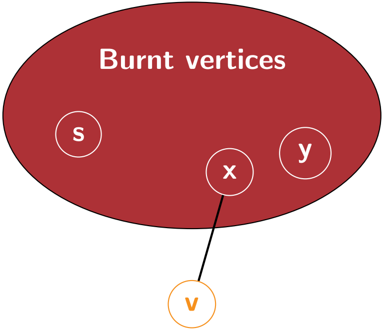
Dijkstra's algorithm: Proof of correctness

- Each new shortest path we discover extends an earlier one
- By induction, assume we have found shortest paths to all vertices already burnt



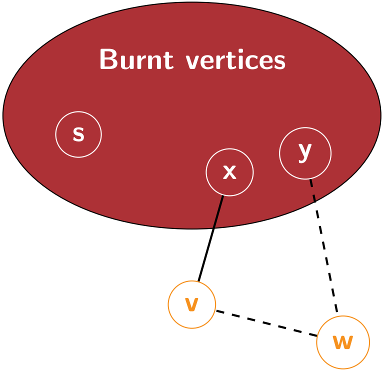
Dijkstra's algorithm: Proof of correctness

- Each new shortest path we discover extends an earlier one
- By induction, assume we have found shortest paths to all vertices already burnt
- Next vertex to burn is **v**, via **x**



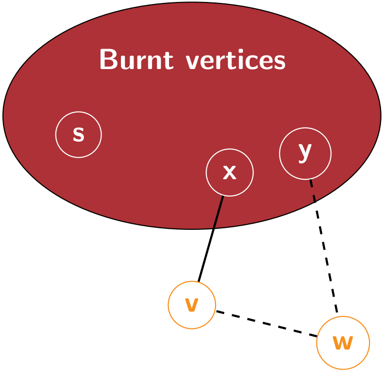
Dijkstra's algorithm: Proof of correctness

- Each new shortest path we discover extends an earlier one
- By induction, assume we have found shortest paths to all vertices already burnt
- Next vertex to burn is **v**, via **x**
- Cannot find a shorter path later from **y** to **v** via **w**
  - Burn time of **w** $\geq$ burn time of **v**
  - Edge from **w** to **v** has weight ≥ 0



Dijkstra's algorithm: Proof of correctness

- Each new shortest path we discover extends an earlier one
- By induction, assume we have found shortest paths to all vertices already burnt
- Next vertex to burn is **v**, via **x**
- Cannot find a shorter path later from **y** to **v** via **w**
  - Burn time of **w** $\geq$ burn time of **v**
  - Edge from **w** to **v** has weight ≥ 0
- This argument breaks down if edge (**w**,**v**) can have negative weight
  - Can't use Dijkstra's algorithm with negative edge weights



Implementation

- Maintain two dictionaries with vertices as keys
  - `visited`, initially `False` for all `v` (burnt vertices)
  - `distance`, initially `infinity` for all `v` (expected burn time)

```
def dijkstra(WMat,s):
  (rows,cols,x) = WMat.shape
  infinity = np.max(WMat)*rows+1
  (visited,distance) = ({},{})
  for v in range(rows):
    (visited[v],distance[v]) = (False,infinity)
  distance[s] = 0
  for u in range(rows):
    nextd = min([distance[v] for v in range(rows)
                if not visited[v]])
    nextvlist = [v for v in range(rows)
                if (not visited[v]) and
                   distance[v] == nextd]
    if nextvlist == []:
      break
    nextv = min(nextvlist)
    visited[nextv] = True
    for v in range(cols):
      if WMat[nextv,v,0] == 1 and (not visited[v]):
        distance[v] = min(distance[v],distance[nextv]
                                      +WMat[nextv,v,1])
  return(distance)
```

Implementation

- Maintain two dictionaries with vertices as keys
  - `visited`, initially `False` for all `v` (burnt vertices)
  - `distance`, initially `infinity` for all `v` (expected burn time)
- Set `distance[s]` to `0`

```
def dijkstra(WMat,s):
  (rows,cols,x) = WMat.shape
  infinity = np.max(WMat)*rows+1
  (visited,distance) = ({},{})
  for v in range(rows):
    (visited[v],distance[v]) = (False,infinity)
  distance[s] = 0
  for u in range(rows):
    nextd = min([distance[v] for v in range(rows)
                    if not visited[v]])
    nextvlist = [v for v in range(rows)
                    if (not visited[v]) and
                       distance[v] == nextd]
    if nextvlist == []:
      break
    nextv = min(nextvlist)
    visited[nextv] = True
    for v in range(cols):
      if WMat[nextv,v,0] == 1 and (not visited[v]):
        distance[v] = min(distance[v],distance[nextv]
                                     +WMat[nextv,v,1])
  return(distance)
```

Implementation

- Maintain two dictionaries with vertices as keys
  - `visited`, initially `False` for all `v` (burnt vertices)
  - `distance`, initially `infinity` for all `v` (expected burn time)
- Set `distance[s]` to `0`
- Repeat, until all reachable vertices are visited
  - Find unvisited vertex `nextv` with minimum distance
  - Set `visited[nextv]` to `True`
  - Recompute `distance[v]` for every neighbour `v` of `nextv`

```
def dijkstra(WMat,s):
  (rows,cols,x) = WMat.shape
  infinity = np.max(WMat)*rows+1
  (visited,distance) = ({},{})
  for v in range(rows):
    (visited[v],distance[v]) = (False,infinity)
  distance[s] = 0
  for u in range(rows):
    nextd = min([distance[v] for v in range(rows)
                    if not visited[v]])
    nextvlist = [v for v in range(rows)
                    if (not visited[v]) and
                       distance[v] == nextd]
    if nextvlist == []:
      break
    nextv = min(nextvlist)
    visited[nextv] = True
    for v in range(cols):
      if WMat[nextv,v,0] == 1 and (not visited[v]):
        distance[v] = min(distance[v],distance[nextv]
                                        +WMat[nextv,v,1])
  return(distance)
```

Complexity

- Setting `infinity` takes $O(n^2)$ time

```
def dijkstra(WMat,s):
  (rows,cols,x) = WMat.shape
  infinity = np.max(WMat)*rows+1
  (visited,distance) = ({},{})
  for v in range(rows):
    (visited[v],distance[v]) = (False,infinity)
  distance[s] = 0
  for u in range(rows):
    nextd = min([distance[v] for v in range(rows)
                    if not visited[v]])
    nextvlist = [v for v in range(rows)
                    if (not visited[v]) and
                       distance[v] == nextd]
    if nextvlist == []:
      break
    nextv = min(nextvlist)
    visited[nextv] = True
    for v in range(cols):
      if WMat[nextv,v,0] == 1 and (not visited[v]):
        distance[v] = min(distance[v],distance[nextv]
                                      +WMat[nextv,v,1])
  return(distance)
```

Complexity

- Setting `infinity` takes $O(n^2)$ time
- Main loop runs `n` times
  - Each iteration visits one more vertex
  - $O(n)$ to find next vertex to visit
  - $O(n)$ to update `distance[v]` for neighbours

```
def dijkstra(WMat,s):
  (rows,cols,x) = WMat.shape
  infinity = np.max(WMat)*rows+1
  (visited,distance) = ({},{})
  for v in range(rows):
    (visited[v],distance[v]) = (False,infinity)
  distance[s] = 0
  for u in range(rows):
    nextd = min([distance[v] for v in range(rows)
                if not visited[v]])
    nextvlist = [v for v in range(rows)
                if (not visited[v]) and
                   distance[v] == nextd]
    if nextvlist == []:
      break
    nextv = min(nextvlist)
    visited[nextv] = True
    for v in range(cols):
      if WMat[nextv,v,0] == 1 and (not visited[v]):
        distance[v] = min(distance[v],distance[nextv]
                                      +WMat[nextv,v,1])
  return(distance)
```

Complexity

- Setting `infinity` takes $O(n^2)$ time
- Main loop runs `n` times
  - Each iteration visits one more vertex
  - $O(n)$ to find next vertex to visit
  - $O(n)$ to update `distance[v]` for neighbours
- Overall $O(n^2)$

```
def dijkstra(WMat,s):
  (rows,cols,x) = WMat.shape
  infinity = np.max(WMat)*rows+1
  (visited,distance) = ({},{})
  for v in range(rows):
    (visited[v],distance[v]) = (False,infinity)
  distance[s] = 0
  for u in range(rows):
    nextd = min([distance[v] for v in range(rows)
                if not visited[v]])
    nextvlist = [v for v in range(rows)
                if (not visited[v]) and
                    distance[v] == nextd]
    if nextvlist == []:
      break
    nextv = min(nextvlist)
    visited[nextv] = True
    for v in range(cols):
      if WMat[nextv,v,0] == 1 and (not visited[v]):
        distance[v] = min(distance[v],distance[nextv]
                                      +WMat[nextv,v,1])
  return(distance)
```

Complexity

- Setting `infinity` takes $O(n^2)$ time
- Main loop runs `n` times
  - Each iteration visits one more vertex
  - $O(n)$ to find next vertex to visit
  - $O(n)$ to update `distance[v]` for neighbours
- Overall $O(n^2)$
- If we use an adjacency list
  - Setting `infinity` and updating distances both $O(m)$, amortised
  - $O(n)$ bottleneck remains to find next vertex to visit
  - Better data structure? Later ...

```
def dijkstralist(WList,s):
  infinity = 1 + len(WList.keys())*
                    max([d for u in WList.keys()
                            for (v,d) in WList[u]])
  (visited,distance) = ({},{})
  for v in WList.keys():
    (visited[v],distance[v]) = (False,infinity)
  distance[s] = 0
  for u in WList.keys():
    nextd = min([distance[v] for v in WList.keys()
                      if not visited[v]])
    nextvlist = [v for v in WList.keys()
                      if (not visited[v]) and
                          distance[v] == nextd]
    if nextvlist == []:
      break
    nextv = min(nextvlist)
    visited[nextv] = True
    for (v,d) in WList[nextv]:
      if not visited[v]:
        distance[v] = min(distance[v],distance[nextv]+d)
  return(distance)
```

Summary

- Dijkstra's algorithm computes single source shortest paths
- Use a greedy strategy to identify vertices to visit
  - Next vertex to visit is based on shortest distance computed so far
  - Need to prove that such a strategy is correct
  - Correctness requires edge weights to be non-negative
- Complexity is $O(n^2)$
  - Even with adjacency lists
  - Bottleneck is identifying unvisited vertex with minimum distance
  - Need a better data structure to identify and remove minimum (or maximum) from a collection