Lecture 6:
Shortest Path Algorithms: (Part I)

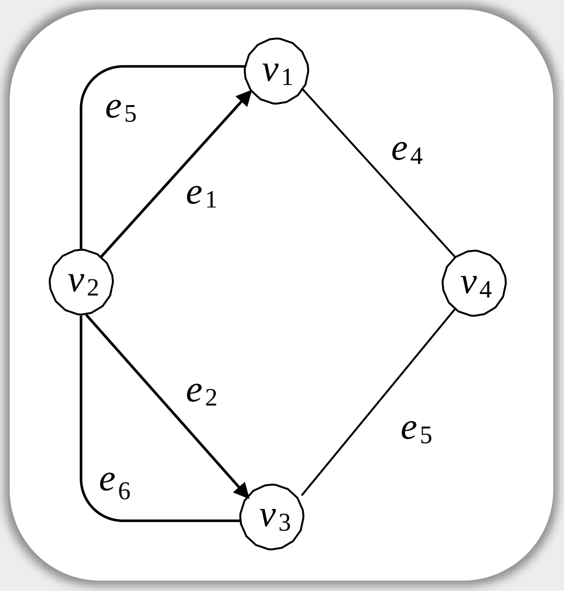
Prof. Krishna R. Pattipati
Dept. of Electrical and Computer Engineering
University of Connecticut
Contact: krishna@engr.uconn.edu; (860) 486-2890

Outline

- Graph terminology
- Computer representation of graphs
    - Weight matrix or adjacency matrix
    - List of edges
    - Linked adjacency list
    - Forward star
- Applications of shortest path problem
- A generic shortest path algorithm for single origin-multiple destinations problem
    - Dijkstra's algorithm . . . label setting methods
        - Heap implementation
        - Dial's bucket method
    - Label correcting methods
        - Bellman-Moore-D'Esopo-Pape algorithm
        - Threshold algorithm

Graph terminology

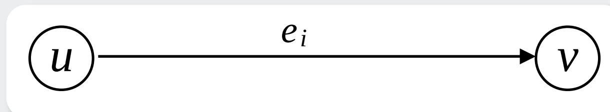
- Graph $G = (V, E)$
  - $V = \{v_1, v_2, \ldots, v_n\}$ a finite set of vertices, nodes, junctions, points, 0-cells, 0-simplices
  - $E = \{e_1, e_2, \ldots, e_m\}$ a finite set of edges, arcs, links, branches, elements, 1-cells, 1-simplices
  - To each edge e, there corresponds two distinct vertices u and $v \Rightarrow e$ is incident on u, v



Graph terminology

- **Directed graph** (or digraph) and **undirected graph**
  - If vertex pairs are ordered, i.e., *e* is directed from vertex *u* to vertex *v*, then the graph is called a **diagraph**

  ⇒ Direct edge <*u*, *v*>:



  ⇒ *u* is an immediate **predecessor** of *v* and *v* is an immediate **successor** of *u*

  - If the edges have no direction, then the graph is said to be an undirected graph

  ⇒ Vertices are unordered

  ⇒ Undirected edge: (*u*, *v*)

  - An undirected graph can be converted into a directed graph by adding bi-directional edges
  - We assume that there exists only one edge between two nodes in one direction

Graph terminology

- **Network**
  - A graph (directed or undirected) in which a real number is associated with each edge $\Rightarrow$ network = attributed graph
    - If have multiple attributes, it is a multi-attributed graph or network
  - This number is called the **weight** of the edge
  - No loss in generality
    - If a node has a weight, we can define a dummy node such that edge from dummy node to node has a weight
- **Degree of a vertex**
  - For an undirected graph G:
    - $d(v)$ = # of adjacent vertices or # of times v is an end point of edges
    - **Fact**: # of nodes of odd degree in a finite undirected graph is even
    - **Proof**:

$$\sum_{i=1}^{n} d(v_i) = 2m$$

Graph terminology

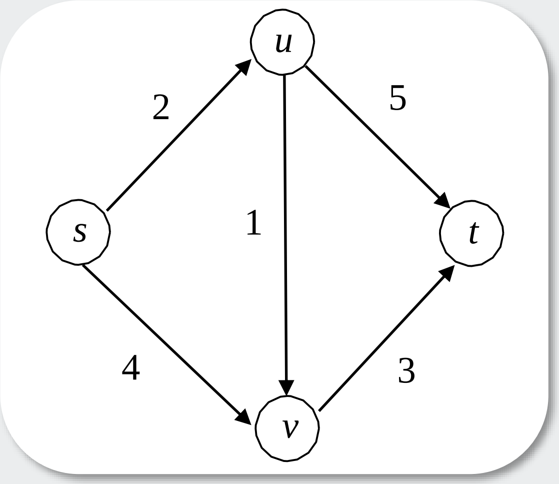
- **Walk or a path**
  - For an undirected graph G:
    - $(v_1, v_2, \ldots, v_k)$ is a walk in an undirected graph G if $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$ are edges on the walk
  - The walk is directed if each edge is directed (<>)
  - Note that **vertices may be repeated in a walk**
- **Simple path**
  - $(v_1, v_2, \ldots, v_k)$ is a simple path if all vertices are distinct
  - Directed simple path if all vertices are distinct and each edge is directed
- **Cycle**
  - A path in an undirected graph is a cycle if $k > 1$ and $v_1 = v_k$ and no edge is repeated
  - A path in a directed graph is a cycle if $k > 1$ and $v_1 = v_k$ … simple cycle if vertices $v_1, v_2, \ldots, v_{k-1}$ are distinct
  - A graph without cycles is ***acyclic***

Graph terminology

- **Connected graphs**
  - If there is a path from a vertex v_i to a vertex v_k, then v_k is reachable from v_i
  - A graph G is connected if every vertex v_k is reachable from every other vertex v_i, and disconnected otherwise
- **Weight (length) of a path**
  - Given a path $p = \langle v_1, v_2, \ldots, v_k \rangle$, we can speak of the length of the path or the weight of the path

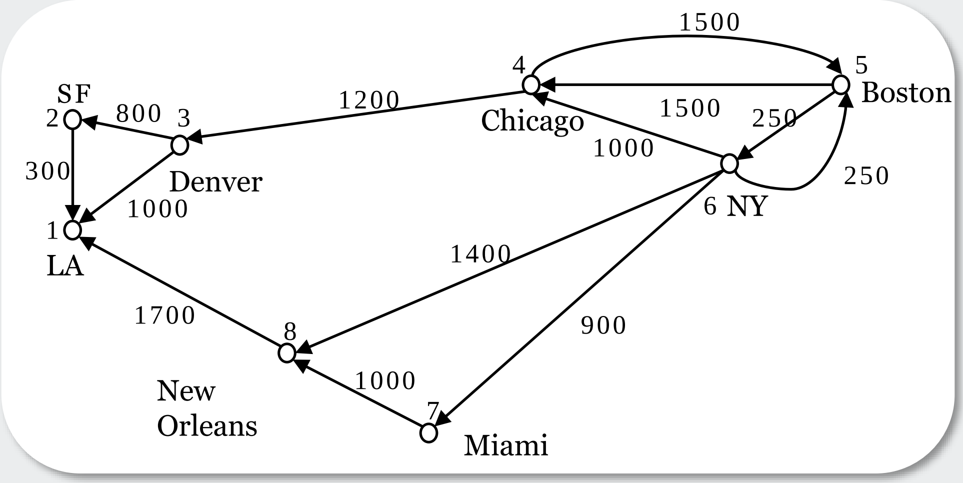
$$= c_{v_1 v_2} + c_{v_2 v_3} + \cdots + c_{v_{k-1} v_k}$$

  - Example: weight of path $s \rightarrow u \rightarrow t$: $c_{su} + c_{ut} = 7$



Computer representation of graphs

- Four methods
  - Weight matrix or adjacency matrix
  - List of edges
  - Linked adjacency list
  - Forward star



- Weight matrix
  - n nodes $\Rightarrow n \times n$ matrix $C = [c_{ij}]$
  - $c_{ij} \sim$ weight of edge <i, j>
  - No edge $\Rightarrow c_{ij} = \infty$ (e.g., 10^{20})
  - $c_{ii} = 0$
  - Undirected network $\Rightarrow C = C^T$ symmetric $\Rightarrow (\frac{n(n-1)}{2})$ elements/words
  - Directed network $\Rightarrow n(n-1)$ elements/words

List of edges representation of graphs

- List of edges
  - Useful when the graph is sparse
    $\Rightarrow$ # of edges $m \ll n(n-1)$
  - Needs three m vectors or a matrix $A(m, 3)$

Start node list (beginning node)	End node list (destination node)	Weight list
$b(1)$	$d(1)$	$c(1)$
$b(2)$	$d(2)$	$c(2)$
$\vdots$	$\vdots$	$\vdots$
$b(m)$	$d(m)$	$c(m)$

$b = [8, 5, 4, 6, 4, 3, 7, 6, 6, 3, 2, 5, 6]'$

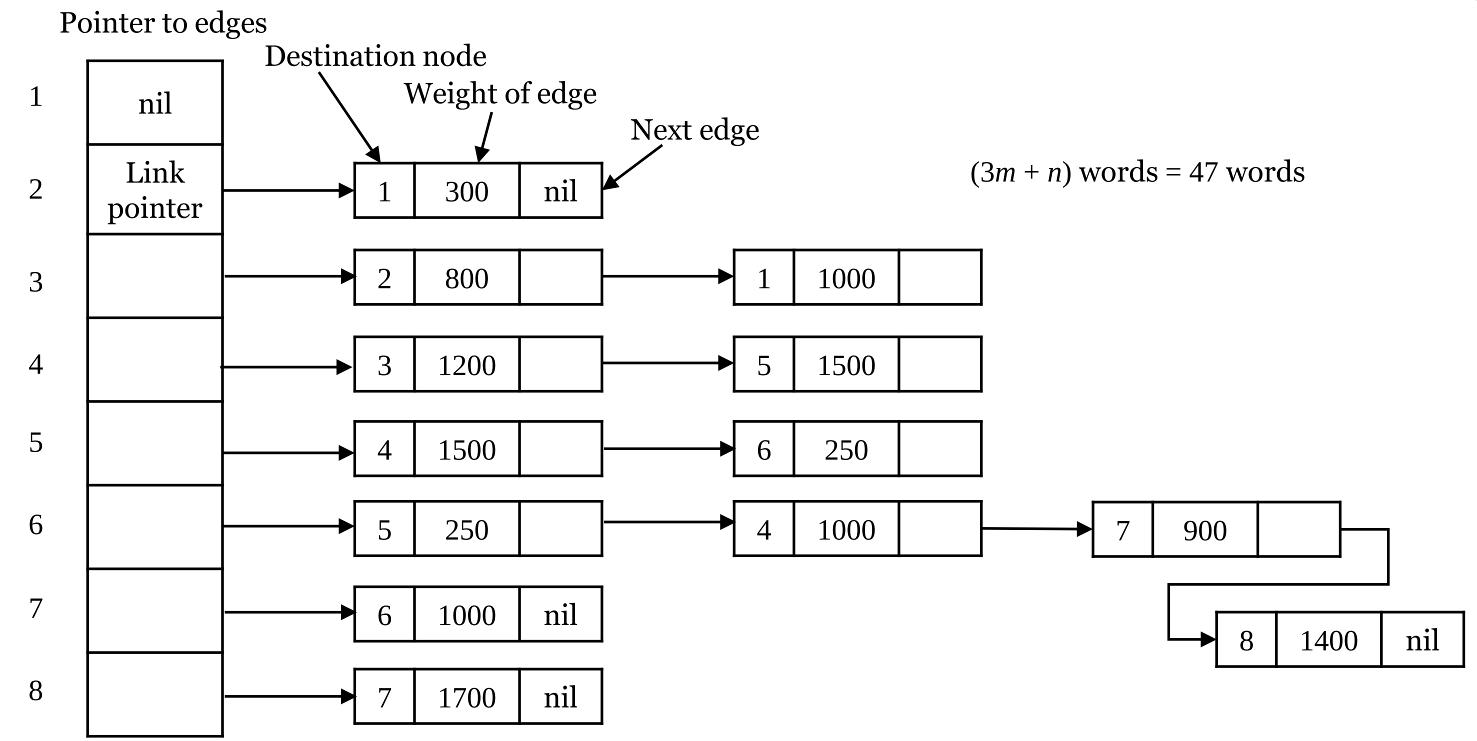
$d = [1, 4, 5, 8, 3, 1, 8, 4, 7, 2, 1, 6, 5]'$

$c = [1700, 1500, 1500, 1400, 1200, 1000, 1000, 1000, 900, 800, 300, 250, 250]'$

  - Note the weights are in descending order
  - You can start ***b***, ***d*** or ***c*** list in any way you want
  - It is convenient to start ***c*** as a heap for the shortest path problems … more on this later!!

Linked adjacency list representation of graphs

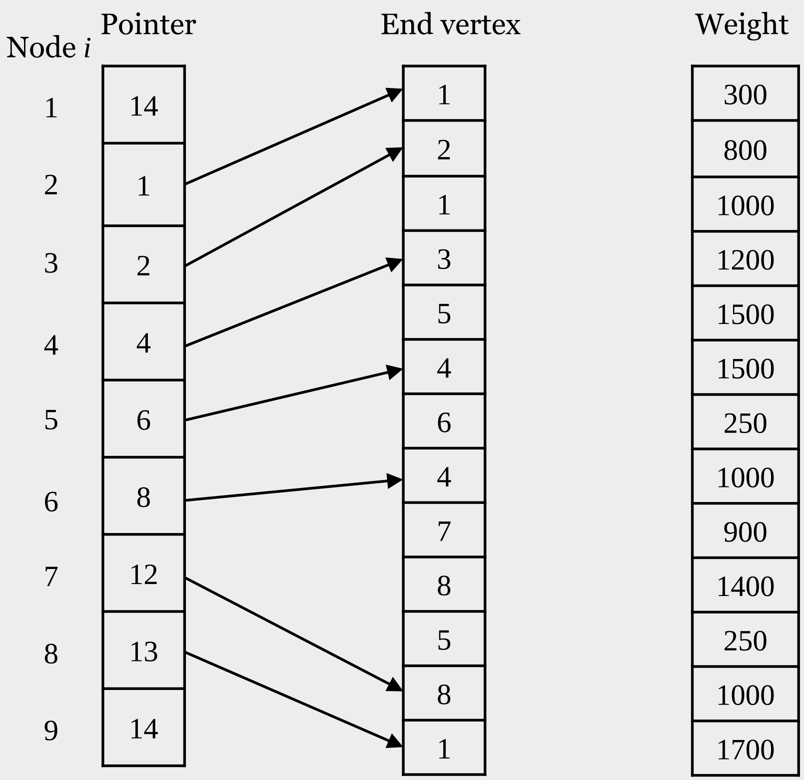
- Linked adjacency list



- Easy to add or delete edges ⇒ change pointers to links
- **Outlists of nodes**
- Can also represent **inlists of nodes**

Forward star representation of graphs

- Forward star (out-list)
  - Useful when edges don't have to be added or deleted
  - It is not easy to add or delete edges



Total words:
$2m + n + 1 = 26 + 8 + 1 = 35$

- Backward star
  - Similar to forward star with in-list (incoming edges to a node)

Shortest path problems

- We can define several path related problems using the above terminology
  - Given any two nodes *s* and *t*, find the shortest path (i.e., minimum length path) from *s* to *t* . . . *single source - single destination shortest path problem*
  - Given a node $v_1 = s$, find the shortest distances to all other nodes. . . *single source - multiple destination shortest path problem*
  - Shortest distance from every node to every other node . . . *all pairs shortest path problem* . . . Lecture 7
- We also distinguish between problems where
  - Edge weights (arc lengths) are nonnegative
  - Edge weights can be negative
- Why do we solve these problems?
- Communication networks
  - $<v_i, v_k>$ in a communication network
  - $c <v_i, v_k>$ = average packet delay to traverse link $<v_i, v_k>$
  - Shortest path ⇒ minimum cost route over which to send data or minimize delay of route
  - Average delay is a function of link traffic … in fact, a nonlinear relationship
  - However, shortest path problem is an integral part of most routing problems

Reliability networks

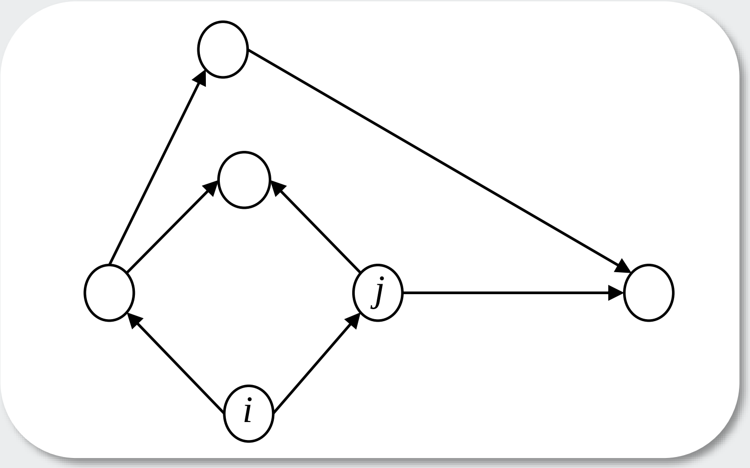
- $c <v_i, v_k> = -\ln p <v_i, v_k>$
- $p <v_i, v_k>$ = probability that a given arc (edge) $<v_i, v_k>$ is usable in the network
- Edges are assumed to be independent
- Most reliable path between s and $t \Rightarrow$ find shortest distance between nodes s and t with edge weights $\{-\ln p <v_i, v_k>\}$
- Note: Reliability of a path π

$$\max \prod_{<v_i, v_{i+1}> \in \pi} p < v_i, v_{i+1} >$$

$$\Rightarrow \min - \sum_{<v_i, v_{i+1}> \in \pi} \ln p < v_i, v_{i+1} >$$

PERT networks (critical path analysis)

- Nodes of subtasks, arcs (edges) ~ dependency
- t_{ij} = time required to complete j after i is completed
- $<i, j>$ denotes precedence constraint that i must be completed before j can begin



- Problem: find the most time consuming path
  - = longest (critical) path .... This is the one you want to monitor!
  - = shortest path with $c(v_i, v_j) = -t_{ij}$
- Viterbi decoding, discrete dynamic programming, etc.

Dual of the shortest path problem

- For simplicity, we denote nodes $\{1, 2, \dots, n\}$ and edges $<i, j>$
  - Source = node 1
  - Destination = node n, for single destination problem
- Dual of the shortest path problem
  - Let us look at the shortest path problem from the viewpoint of the dual
  - If we want shortest path to node n only

$$\begin{aligned} &\max \lambda_n \\ &\text{s.t.} \quad \lambda_1 = 0 \\ &\qquad \lambda_j - \lambda_i \le c_{ij} \Rightarrow \lambda_j \le \lambda_i + c_{ij}, \forall < i, j > \end{aligned}$$

  - If we want to find shortest paths to all nodes from node 1, replace objective function by:

$$\max\{\lambda_2 + \lambda_3 + \cdots + \lambda_n\}$$

  - **CS conditions**
    - If P is the shortest path then
      - $\lambda_j = \lambda_i + c_{ij}$, if $<i, j> \in P$
      - $\lambda_j \le \lambda_i + c_{ij}$, $\forall <i, j> \notin P$
    - $\{\lambda_i\}$ are called **labels** of nodes

Example

$$\max\{\lambda_2+\lambda_3+\lambda_4+\lambda_5\}$$

$$\text{s.t. } \lambda_1 = 0$$

$$\lambda_2-\lambda_1 \le 5$$

$$\lambda_3-\lambda_1 \le 2$$

$$\lambda_3-\lambda_2 \le 3$$

$$\lambda_4-\lambda_2 \le 4$$

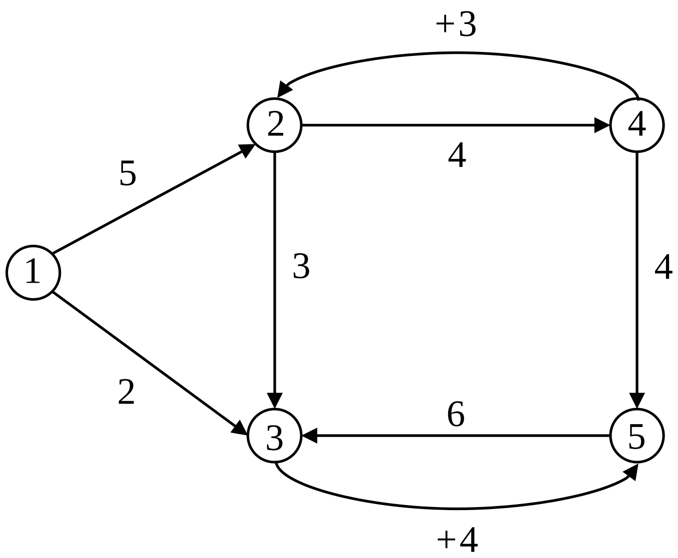
$$\lambda_2-\lambda_4 \le 3$$

$$\lambda_4-\lambda_3 \le 5$$

$$\lambda_5-\lambda_3 \le 4$$

$$\lambda_3-\lambda_5 \le 6$$

$$\lambda_5-\lambda_4 \le 4$$



A generic relaxation (dual) procedure

- **Initialize**:
  - Set $\lambda_1 = 0$
  - $\lambda_i = \infty$ (large #) $\forall i = 2, 3, \ldots, n$
  - $V = \{1\}$ … candidate list
- **Step 1**:
  - If all inequalities are satisfied
    - Stop . . . found an optimal solution
  - Else
    - Remove a node i from the candidate list V
  - End if
- **Step 2**:
  - For each outgoing arc $<i, j>$ with $j \neq 1$,
    - If $\lambda_j - \lambda_i > c_{ij}$
      - Set $\lambda_j = \lambda_i + c_{ij}$ … labeling step
      - Add j to V if it is not already in V
    - End if
  - Go back to Step 1
- Labels $\{\lambda_i\}$ are monotonically nonincreasing
- $\lambda_i < \infty \Leftrightarrow$ node i has entered the candidate list V at least once
- The various implementations differ in the way they select the node from the candidate list V

Dijkstra's way of picking the node to relax

- Pick a node with the minimum label

$$i = \arg\min_j \{\lambda_j\}$$

- Needs non-negativity of $\{c_{ij}\}$ and graph connectivity for convergence!!
- Implementation issues
  - use binary heap to efficiently remove node i from V
  - Dial's "bucket" method … see Bertsekas's book
- A node enters V only once if $c_{ij} \geq 0$
- These implementations are called "**label setting**" methods or "**best-first**" **scanning methods**

BMDP & Threshold Algorithms

- Bellman-Moore-D'Esopo-Pape (BMDP)
  - Maintain a queue of nodes in the candidate list, V
  - A node may enter V more than once!!
  - **Breadth-first scanning** or **label correcting methods**

- Threshold algorithms … see Bertsekas's book
  - Split queue into two queues Q' and Q'', where labels of nodes in Q' are less than a threshold s

Dijkstra's algorithm

- Dijkstra's algorithm … assume $c_{ij} > 0$
- **Step 1:** initialization
  - set $\lambda_1 = 0$
  - pred(1) = $\emptyset$
  - $\lambda_j = c_{1j}$ for $j = 2, \ldots, n$
  - pred(j) = 1 if $c_{ij} < \infty$
  - set $W = \{\emptyset\}$, $V = \{1\}$. . . $W = \{i : \lambda_i < \infty, i \notin V\}$ set of permanently labeled nodes
- **Step 2:** scanning and permanent labeling
  - find $i \in V$, where $\lambda_i = \min\{\lambda_j\}, j \in V$
  - set $V = V - \{i\}$, $W = W \cup \{i\}$
- **Step 3:** revision of tentative labels
  - $\forall$ outgoing arc $<i, j>$ with $j \neq 1$
    - if $\lambda_j > \lambda_i + c_{ij}$

      pred(j) = i

      $\lambda_j = \lambda_i + c_{ij}$

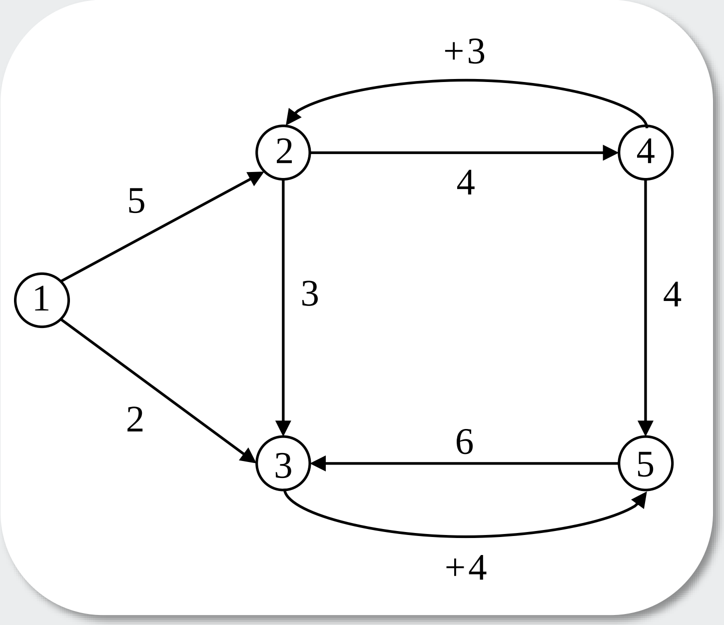
      if ($j \notin V$)

      $V = V \cup \{j\}$

      end if
    - end if
    - if ($V = \emptyset$) stop $\Rightarrow$ computation is completed
    - else go to step 2
    - end if

Illustration of Dijkstra's Algorithm

- Iteration 1
  - Node removed = 1 $\Rightarrow W = \{1\}$
  - Labels: $\lambda_1 = 0,\ \lambda_2 = 5,\ \lambda_3 = 2,\ \lambda_4 = \infty, \lambda_5 = \infty$
  - Node list: $V = \{2, 3\}$
- Iteration 2
  - since $\lambda_3 < \lambda_2$, node removed from $V = 3 \Rightarrow W = \{1, 3\}$
  - labels: $\lambda_1 = 0, \lambda_2 = 5, \lambda_3 = 2, \lambda_4 = \infty, \lambda_5 = 6$
  - node list: $V = \{2, 5\}$
- Iteration 3
  - since $\lambda_2 < \lambda_5$, node removed from $V = 2 \Rightarrow W = \{1, 3, 2\}$
  - labels: $\lambda_1 = 0, \lambda_2 = 5, \lambda_3 = 2, \lambda_4 = 9, \lambda_5 = 6$
  - node list: $V = \{4, 5\}$
- Iteration 4
  - node removed from $V = 5 \Rightarrow W = \{1, 3, 2, 5\}$
  - labels: $\lambda_1 = 0, \lambda_2 = 5, \lambda_3 = 2, \lambda_4 = 9, \lambda_5 = 6$
  - node list: $V = \{4\}$
- Iteration 5 … no need to perform iteration 5 since labels of nodes in W will not change
  - node removed from $V = 4 \Rightarrow W = \{1, 3, 2, 5, 4\}$
  - labels: $\lambda_1 = 0, \lambda_2 = 5, \lambda_3 = 2, \lambda_4 = 9, \lambda_5 = 6$
  - node list: $V = \{\emptyset\}$

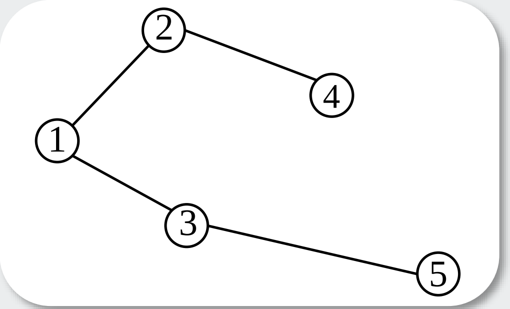


Interpretations and proof of optimality

- Removing from V a minimum label node $\Rightarrow$ W contains nodes with the smallest labels
- At k^{th} step, we have the set W of k closest nodes to node 1 as well as the shortest distances $\{\lambda_i\}_{i\in W}$ from node 1 to each node i of $W \Rightarrow \lambda_i \leq \lambda_j$ if $i \in W$ and $j \notin W$
- At each step, we add the next closest node into the set W
- **Once a node enters W, it stays in W forever and labels of nodes in W do not change $\Rightarrow$ W can be interpreted as the set of permanently labeled nodes**
- **Proof**:
  - Valid initially because node 1 exits and enters W
  - Suppose valid for iteration $(k-1) \Rightarrow \lambda_i \leq \lambda_j$ if $i \in W$ and $j \notin W$
  - Since $c_{pi} \geq 0$, when a node p is removed from V and put in W, then $\forall i \in W$, we have $\lambda_i \leq \lambda_p + c_{pi} \Rightarrow$ node i never enters V if it is already in W
    - $\Rightarrow W$ = set of *permanently labeled* nodes
    - $\Rightarrow$ Any label that changes must be from $j \notin$ W
  - At the end of the iteration, we have $\lambda_j = \lambda_p + c_{pj} \geq \lambda_p \geq \lambda_i,\ \ \forall i \in W \Rightarrow W$ has nodes with "small" labels

Computational load and skim tree

- $(n-1)$ iterations
- Each iteration, need to find minimum label $\Rightarrow$ worst case n operations
- $O(n^2)$ operations
- Label revision: $O(m)$ operations, m = # of arcs
- Since $m \leq n^2$, total computational load $O(n^2)$
- **Can do better with heaps and buckets for sparse graphs**
- Look at shortest paths
  - They form a tree called *shortest path tree* or *skim tree*
  - Spanning tree: tree containing all the vertices



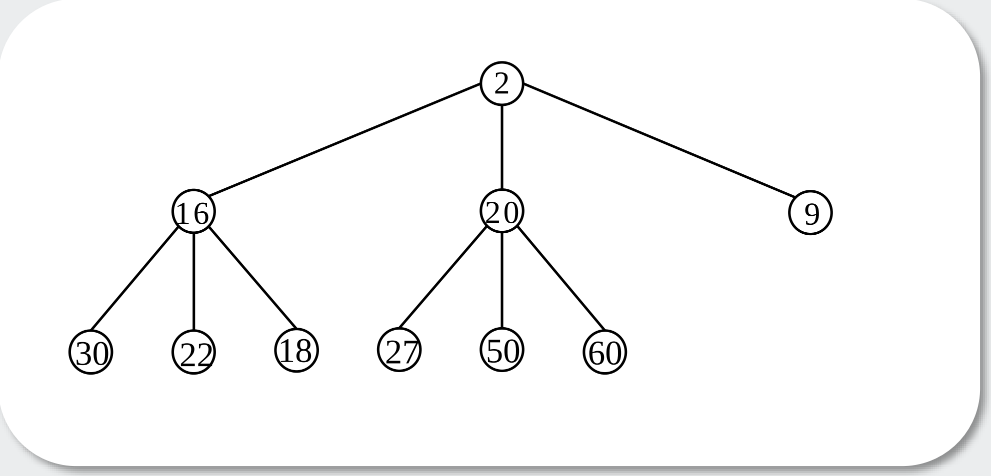
  - **If want to find shortest paths from every node to every other node, invoke the single source algorithm n times**

    $\Rightarrow O(n^3)$ computation time

Heaps

- A heap is a priority queue
- It allows finding the minimum element of a set and insertion (enqueuer)/deletion (dequeuer) of elements is easy
- A *d*-heap is a *d-ary* tree (i.e., with at most *d* children),
  - Each node contains one item
  - Items are arranged in a heap order

    ⇒ value at each node less than values at its children (if they exist)
- Example: 3-*ary* tree

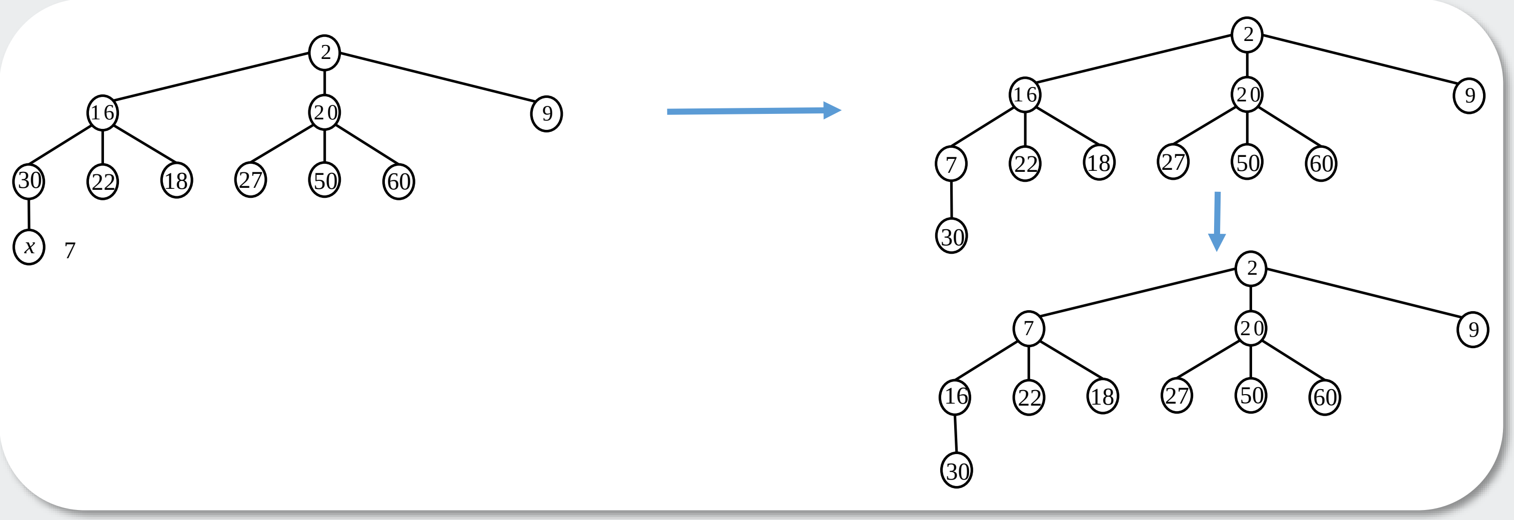


Parent values ≤ Children values

Inserting an element to a *d*-heap

- Easy to insert an element
  - Suppose want to insert 7 into the heap
  - Make a new vacant node *x* to the tree such that *x* is a leaf
  - Storing 7 in *x* may violate heap order
  - Use **SIFT-UP** procedure to place 7 at its proper place

```
DO while parent exceeds child's value
        Move parent to vacant node
        Replace parent node by vacant node value
End DO
```



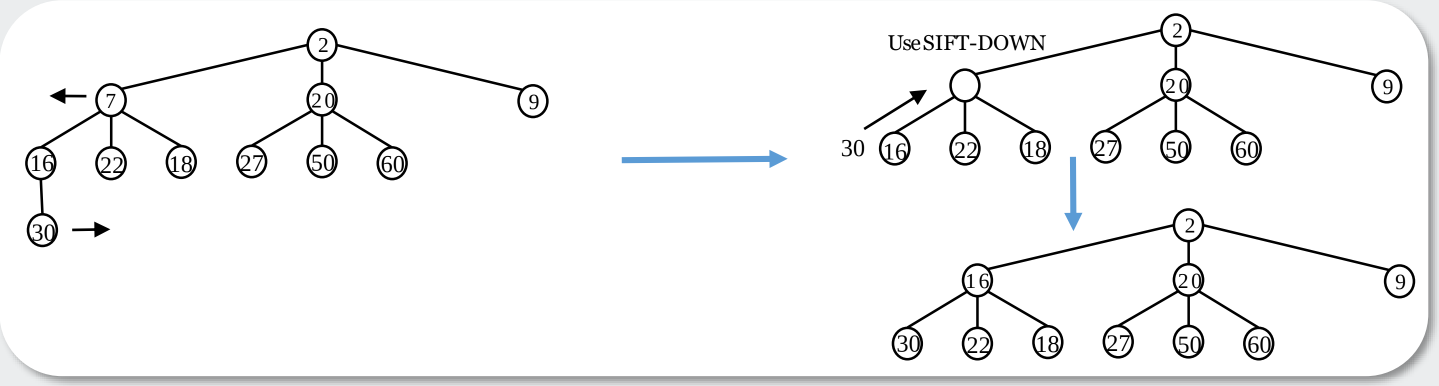
- Note that if inserted at node 9, it takes only one SIFT-UP. This can be done with the so-called **left-complete** *d-ary* tree.

Deleting an element from a *d*-heap

- Easy to delete an element
  - Suppose we want to delete 7
  - Find a node *y* with no children
  - Remove item from the node (say, value is $j = 30$) and delete node *y* from the tree
  - If value $j = 7$ done!!
  - Otherwise remove 7 from the node and attempt to replace it by *j*
  - If ($j < 7$) use SIFT-UP process
  - Otherwise use SIFT-DOWN process
  - SIFT-DOWN

```
If value of parent exceeds the value of a child
        Choose a child with minimum value
        Store child in parent & parent in child
End if
```

  - When deleting an element, choose *y* that was most recently added ~ like stack (LIFO)



Complexity of insert and delete operations in a *d*-heap

- Complexity of insert and delete operations in a *d*-heap
  - Time for SIFT-UP depends on the depth of node at which SIFT-UP starts ⇒ insert = $O(\log_d n)$
  - Time for SIFT-DOWN ∝ total number of child nodes made vacant during SIFT-DOWN

    ⇒ delete = $O(d \log_d n)$
  - Time for minimum of the set of elements: $O(1)$
  - If there are more inserts than deletes (as in shortest path for the set V), use d as large as possible, i.e., use

$$d = \left\lceil 2 + \frac{m}{n} \right\rceil, m = \#\ \text{of edges}, n = \#\ \text{of nodes}$$

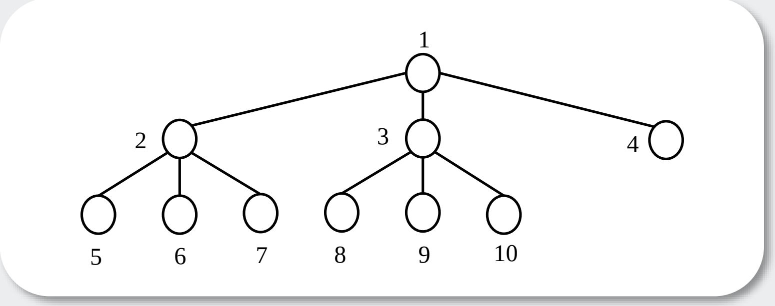
  - Need no explicit pointers, if we number nodes in a breadth-first order
    - Parent of $x = \left\lceil \frac{x-1}{d} \right\rceil$
    - Children of node $x = (d(x-1)+2, \ldots, \min(d(x+1), n)$
    - e.g.,

$x = 4, \quad d = 3 \Rightarrow$ parent $= 1$; children = none

$x = 5, \quad d = 3 \Rightarrow$ parent $= 2$; children = none

$x = 3, \quad d = 3 \Rightarrow$ parent $= 1$; children = 8, 9, 10

Index	1	2	3	4	5	6	7	8	9	10
Key	2	16	20	9	30	22	18	27	50	60



How to make *d*-heaps?

- Q: How to make heaps?
- One of two ways:
  - Use insert n times $\Rightarrow O(n\log_d n)$
  - Create an arbitrary *d-ary* tree and execute SIFT-DOWN

$$\sum_{i=0}^{\lceil \log_d(n) \rceil} \frac{n(i+1)}{d^i} = O(n)$$

- To learn more about heaps, read:
  - J.W.J. Williams, "Algorithm232: Heapsort," CACM, 7, 1964, pp. 347-348
  - D.B. Johnson, "Priority queues with update and finding minimum spanning trees," Inform. Proc. Letters, 4, 1, 1975, pp. 53-57
  - D.B. Johnson, "Efficient algorithms for shortest paths in sparse networks," JACM, vol. 24, pp. 1-13
  - R. Tarjan, Data Structures and Network Algorithms, SIAM, 1983
  - E. Horowitz and S. Sahni, Computer Algorithms, CSP, 1978
- Application to shortest path
  - Let out(i) = set of edges directed away from i
  - n = # of nodes, m = # of edges
  - Node list V is in the form of a heap

Heap implementation of Dijkstra's Algorithm

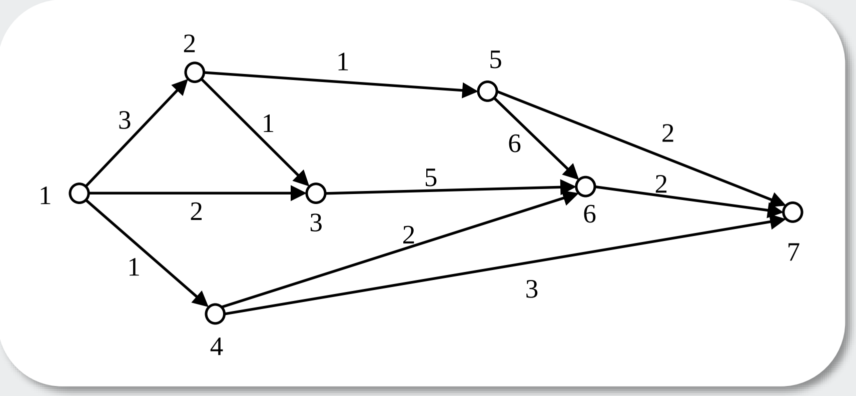
- $\forall i = 2, \ldots, n$
  - parent(i) = *null*
  - $\lambda_i = \infty$
- end $\forall$
- $\lambda_1 = 0$
- parent(1) = *null*
- $V = \{1\}$
- $i = 1$
- while $i \neq null$ do
  - for $(i, j) \in$ out(i) and $j \neq 1$
    - if $(\lambda_j > \lambda_i + c_{ij})$
      - $\lambda_j = \lambda_i + c_{ij}$
      - parent(j) = i
      - if $(j \notin V)$

        insert j into V
      - else

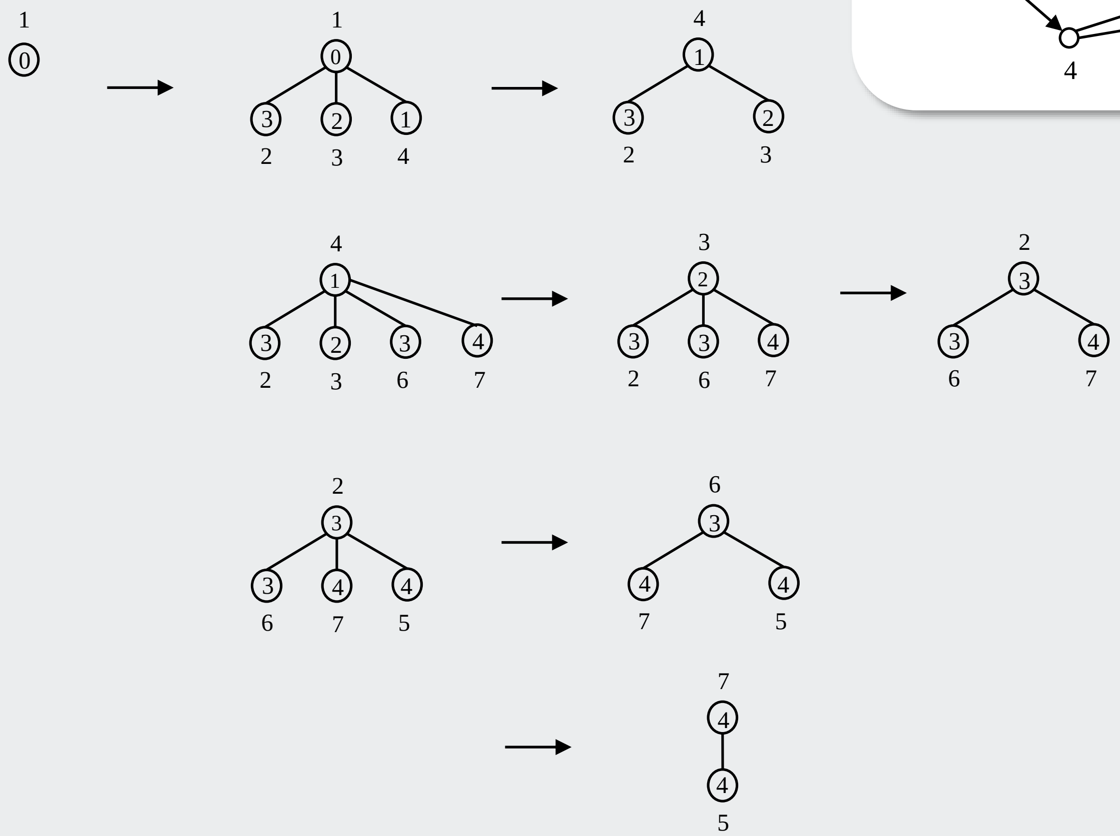
        SIFT-UP j
      - end if
    - end if
  - end for
  - i = delete min$\{V\}$ … finds the next minimum on the list by deleting the current minimum
- end do

Complexity of *d*-heap version of Dijkstra

- $O(m\log_d n)$
- Optimum d, $d = \left\lceil 2 + \frac{m}{n} \right\rceil$
- Considerable savings if $m \approx O(n) \Rightarrow d \approx 4$



Heap



Dial's "bucket" method

- c_{ij} are assumed to be *nonnegative* integers
- No loss in generality: one can always scale real c_{ij} to get integers to a specified accuracy
- The possible label values range from 0 to $(n-1)C$ where

$$C = \max_{i,j} c_{ij}$$

- So, for each possible label value, maintain a bucket and the corresponding nodes with that label value
- Can use doubly-linked lists to maintain the set of nodes in a given bucket
  - List 1: <*bucket b, # of nodes, first node in the bucket*>
  - List 2: <*node #, node label, next node, previous node*>
- Need to maintain only (C +1) buckets because when we are currently searching bucket b, then all buckets beyond ($b + C$) are empty $\lambda_i \le b$ and $c_{ij} \le C \Rightarrow \lambda_j = \lambda_i + c_{ij} \le b + C$

Illustration of Dial's Bucket Method

iteration	*V*	*node labels*	*buckets* 0	1	2	3	4	*V → W* *node*
1	{1}	$(0,\infty,\infty,\infty,\infty,\infty,\infty)$	1	-	-	-	-	1
2	{2, 3, 4}	$(0,3,2,1,\infty,\infty,\infty)$	1	4	3	2	-	4
3	{2, 3, 6, 7}	$(0,3,2,1,\infty,3,4)$	1	4	3	2, 6	7	3
4	{2, 6, 7}	$(0,3,2,1,\infty,3,4)$	1	4	3	2, 6	7	2
5	{6, 7, 5}	$(0,3,2,1,4,3,4)$	1	4	3	2, 6	7, 5	6
6	{7, 5}	$(0,3,2,1,4,3,4)$	1	4	3	2, 6	7, 5	7
7	{5}	$(0,3,2,1,4,3,4)$	1	4	3	2, 6	7, 5	5
	{∅}	$(0,3,2,1,4,3,4)$	1	4	3	2, 6	7, 5	

- Refined versions . . . see references in Bertsekas's book
  - Alternate scanning strategies … label correcting methods
- Recall that Dijkstra's algorithm uses a best-first scanning
- What if we use breadth-first scanning?
  - Scan the one least recently labelled or the first in the queue
  - Idea behind the method was discovered by Moore (1959) and Bellman (1958)
  - Improvements by D'Esopo and Pape (1980)

Bellman-Moore-D'Esopo-Pape (BMDP) algorithm

- $\forall i = 2, \ldots, n$
  - parent(i) = *null*
  - $\lambda_i = \infty$
- end $\forall$
- $\lambda_1 = 0$
- parent(1) = *null*
- *queue* = [1]
- while *queue* ≠ *null* do
  - i = *queue*[1]
  - *queue* = *queue* [2 · · ·] initially *queue* = [∅]
  - for $(i, j) \in$ out(i)
    - if $(\lambda_i + c_{ij} < \lambda_j)$
      - $\lambda_j = \lambda_i + c_{ij}$
      - parent(j) = i
      - if ($j \notin$ *queue*)

        queue = *queue* $\cup\, j$
      - end if
    - end if
  - end for
- end do

BMDP variations

- Unlike Dijkstra, a node may enter and leave the queue several times and may be scanned several times
- Suppose a node that is in the queue (i.e., a labeled node) gets relabeled (i.e., its λ is modified) before it is scanned
- Where should we place it?
  - Leave it where it was, when it first entered the queue
  - Place it at the head of the queue if the node has already been entered, examined and removed from the queue
  - If the node has never entered the queue before (i.e., it was labelled for the first time), put it at the end of the queue
- This is a *hybrid* scanning method, and was found to work very well in practice [Dial *et al*. (1979), and Pape (1980)]
- **Unlike Dijkstra, the algorithm is guaranteed to terminate even in the presence of negative edge weights, as long as there is no cycle with an overall negative weight**
- **If have a cycle of negative weight, you will continue to be in the cycle and distance monotonically decreases ⇒ primal is unbounded and dual is infeasible**
- Each pass requires $O(m)$ computation
- There can be at most $(n-1)$ passes if the network does not have cycles of negative length

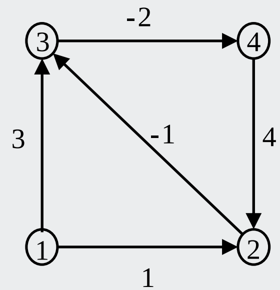
  ⇒ Worst-case computational load $O(mn)$

  ⇒ In practice, they perform much better
- **Detection of negative cycles**
  - If at the end of n passes, queue is not empty ⇒ ∃ a cycle of negative length and can terminate

Illustration of BMDP Algorithm

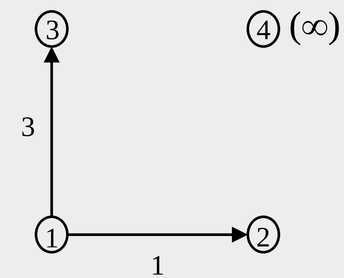
- **Dijkstra won't work for negative edge weight problems!!**
- Example

Problem



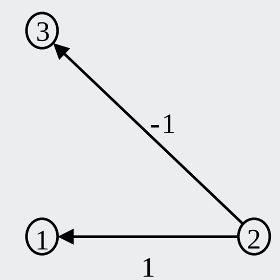
queue = [1]

Iteration 1



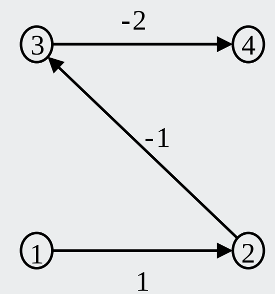
queue = [2, 3]

Iteration 2



queue = [3]

Iteration 3



queue = [4]

- Iteration 4: node 4 goes out ⇒ queue empty ⇒ done!!

Remarks

- Performs very well in practice
- Can devise examples where a node may enter and exit the candidate list an exponential number of times
- See:
  - Kershenbaum, A., “A note on Finding Shortest Path Trees,” Networks, vol. 11, pp. 399-400, 1981
- For variants, see:
  - Bertsekas’s book
  - S. Pallotino, “Shortest path methods: complexity, interrelationships, and new propositions,” Networks, vol. 14, pp. 257-267, 1984
  - G.S. Gallo and S. Pallotino, “Shortest path algorithms,” Annals of Operations Research, vol. 7, pp. 3-79, 1988

Threshold algorithms

- Know that for graphs with positive arc weights, Dijkstra's algorithm ensures that no node is removed more than once
- **Q**: is it possible to emulate the minimum label selection policy of Dijkstra with a much smaller computational effort?
- **One answer**: split V into two queues Q' and Q''
  - Q' = nodes with small labels $\Rightarrow$ nodes with labels $\leq s$
  - Q'' = remaining
- At each iteration
  - Remove a node from Q' and apply generic shortest path algorithm
  - Any node to be added is added to Q''
- When Q' is exhausted, repartition V into Q' and Q'' with a new threshold
- **Key: how to adjust thresholds?**
  - s = current minimum label $\Rightarrow$ Dijkstra
  - s > maximum label $\Rightarrow$ BMDP algorithm
  - Selection of s is an art
  - See:
    - F. Glover, D. Klingman, and N. Phillips, "A new polynomial bounded shortest path algorithm," Operations Research, vol. 33, pp. 65-73, 1985
    - F. Glover, D. Klingman, N. Phillips, and R.F. Schneider, "New polynomial shortest path algorithms and their computational attributes," Management Science, vol. 31, pp. 1106-1128, 1985

Summary

- Graph terminology
- Computer representation of graphs
- A generic shortest path algorithm for single origin-multiple destinations problem
- Dijkstra's algorithm … label setting methods
  - Heap implementation
  - Dial's bucket method
- Label correcting methods
  - Bellman-Moore-D'Esopo-Pape algorithm
  - Threshold algorithm
- Next: all pairs shortest path and distributed shorest path algorithms … Lecture 7