Lecture 7:
Shortest Path Algorithms: (Part II)

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

Outline

- Review of Dijkstra's algorithm
- Bidirectional Dijkstra
- A* algorithm
- Dynamic programming-based algorithms
  - BMDP as a successive approximation to DP recursion
  - Floyd-Warshall algorithm … all pairs
  - Bellman-Ford algorithm
  - Distributed Bellman-Ford algorithm
- Shortest paths on acyclic graphs
- Viterbi decoding
- Martins' algorithm for multi-objective shortest path problems
- Other shortest path algorithms
- Summary

Dijkstra's algorithm

- Single source shortest paths when edge weights $c_{ij} \geq 0$
- Basic idea
  - Find the shortest path to the first nearest node, then to the second nearest node, etc.
    $\Rightarrow 0 = \lambda_{[1]} \leq \lambda_{[2]} \leq \cdots \leq \lambda_{[n]}$, $[i] = i^{\text{th}}$ nearest neighbor
  - If W is the set of nodes for which shortest paths are generated (i.e., W is a permanently labeled list), then at each iteration the nearest neighbor to W, denoted by k, is the one s.t. $\lambda_k = \min\{\lambda_j\}$; $j \in T$; $T = \overline{W}$
- In other words, shortest path to the next nearest neighbor k must necessarily pass through nodes in W
- Shortest path to any other node $j \in T$ will decrease only if the path from $1 \rightarrow k$ and link c_{kj} is less than previous path length:

$$\lambda_j = \min\{\lambda_j, \lambda_k + c_{kj}\}; \forall j \in T$$

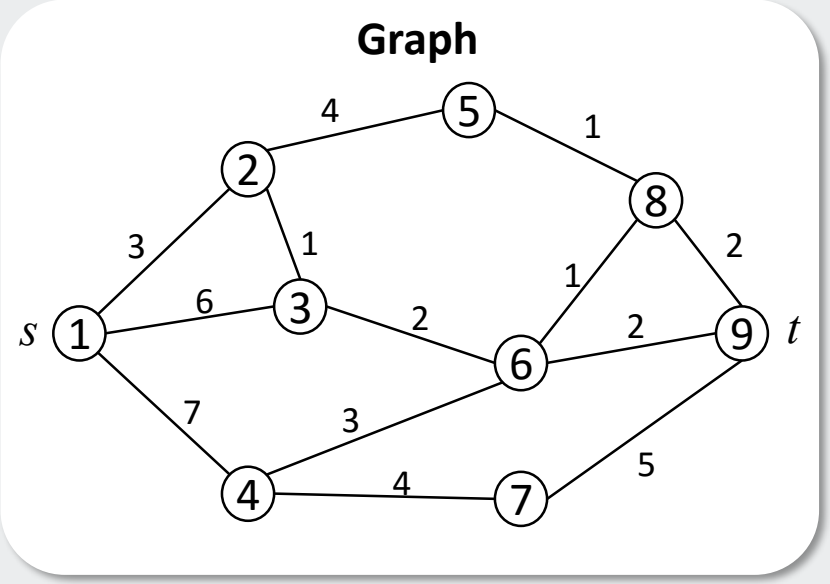
- $O(n^2)$ straightforward implementation
- $O(nC + m)$ Dial's bucket implementation
- $O(m \log_{\lceil 2+\frac{m}{n} \rceil} n)$ heap implementation
- Dijkstra's algorithm is a best-first search strategy

Example

Forward Dijkstra

Node		1	2	3	4	5	6	7	8	9
Initial:	*T*	0	3	6	7					
	P(i)	∅	1	1	1	1	1	1	1	1
Iteration 1:	*T*	0	3	4	7	7				
	P(i)	∅	1	2	1	2	1	1	1	1
Iteration 2:	*T*	0	3	4	7	7	6			
	P(i)	∅	1	2	1	2	3	1	1	1
Iteration 3:	*T*	0	3	4	7	7	6		7	8
	P(i)	∅	1	2	1	2	3	1	6	6
Iteration 4:	*T*	0	3	4	7	7	6	11	7	8
	P(i)	∅	1	2	1	2	3	4	6	6
Iteration 5:	*T*	0	3	4	7	7	6	11	7	8
	P(i)	∅	1	2	1	2	3	4	6	6
Iteration 6:	*T*	0	3	4	7	7	6	11	7	8
	P(i)	∅	1	2	1	2	3	4	6	6
Iteration 7:	*T*	0	3	4	7	7	6	11	7	8
	P(i)	∅	1	2	1	2	3	4	6	6

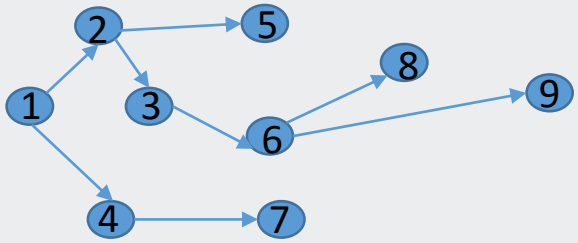
Values in heap are represented by red color



The shortest distance from node 1 to node 9 is therefore 8 units and the shortest route is built up as

- P(9) → 9
- P(6) → 6 → 9
- P(3) →3→ 6 → 9
- 1→2→3→ 6 → 9

Skim tree:



Bidirectional Dijkstra's algorithm

- In the bidirectional Dijkstra, forward search executes from the source node s and backward search executes from destination node t
  - d_{ij} is the distance between node i and node j
  - Q_f is the heap used for forward search from s to i
  - Q_r is the heap used for reverse search from t to i
  - $P(i)$ is the parent node preceding node i in the current optimal route from s to i
  - $R(i)$ is the parent node succeeding i in the current optimal route from i to t
  - *peek*() gives the minimum value in the heap without actually removing the element

Bidirectional Search

Step 1: (Run forward algorithm) forward search from s with labels g. Check termination condition.

- If Q_f.peek() > Q_r.peek(), i.e., minimum label of forward search > minimum label of backward search, go to step 2 (do backward search), else, go to step 1 (do forward search)

Step 2: (Run reverse algorithm) reverse search from t with labels r. Check termination condition.

- If Q_r.peek() > Q_f.peek(), go to step 1, else go to step 2

Termination: Algorithm terminates when some vertex c has g_c+r_c as small as the sum of lengths of the shortest route out of s and the shortest route out of t. Corresponding shortest path can be traced back using $P(c)$ and $R(c)$.

$$g_c + r_c = \min_i (g_i + r_i) \le Q_f.peek() + Q_r.peek()$$

Example

Bi-directional Dijkstra works as follows:

Iteration 1: As $Q_f.peek() = 3 > Q_r.peek() = 2$,

r and $R(i)$ for neighbors of node 6 are updated. Then, for neighbors of node 8, r and $R(i)$ are updated

Then $\min_i(g_i + r_i) = 10 > Q_f.peek() + Q_r.peek() = 3 + 3$

Iteration 2: As $Q_f.peek() = 3 \le Q_r.peek() = 3$,

g and $P(i)$ for neighbors of node 2 are updated

Then $\min_i(g_i + r_i) = 8 > Q_f.peek() + Q_r.peek() = 4 + 3$

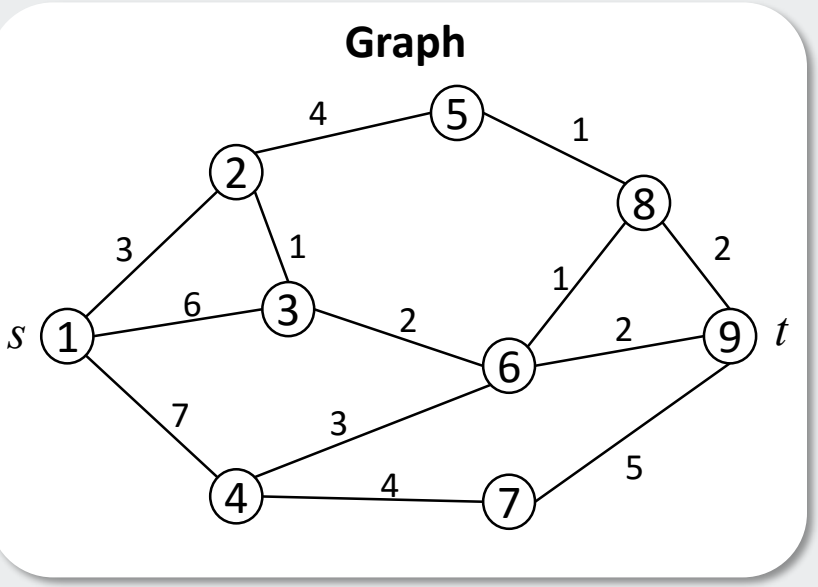
Iteration 3: As $Q_f.peek() = 4 \le Q_r.peek() = 3$,

r and $R(i)$ for neighbors of node 5 are updated.

Then $\min_i(g_i + r_i) = 8 \le Q_f.peek() + Q_r.peek() = 4 + 4$

The shortest distance from node 1 to node 9 is therefore 8 units and shortest route is built up as

- $P(3) \rightarrow 3 \rightarrow R(3)$
- $P(2) \rightarrow 2 \rightarrow 3 \rightarrow 6 \rightarrow R(6)$
- $1 \rightarrow 2 \rightarrow 3 \rightarrow 6 \rightarrow 9$



Node		1	2	3	4	5	6	7	8	9
Initial:	g	0	3	6	7					
	P(i)	∅	1	1	1	-	-	-	-	-
	r						2	5	2	0
	R(i)	-	-	-	-	-	9	9	9	∅
Iteration 1:	r			4	5		2	5	2	0
	R(i)	-	-	6	6	9	9	9	9	∅
	r			4	5	3	2	5	2	0
	R(i)	-	-	6	6	8	9	9	9	∅
Iteration 2:	g	0	3	4	7	7				
	P(i)	∅	1	2	1	2	1	1	1	1
Iteration 3:	r		7	4	5	3	2	5	2	0
	R(i)	9	5	6	6	8	9	9	9	∅

Values in heap are represented by red color

A* algorithm

- A* is a heuristic search version of Dijkstra; it achieves better performance by using cost-to-go heuristics to guide its search
- A* originated in AI. Classic paper: P.E. Hart, N.J. Nilsson, B. Raphael, (1968). "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". *IEEE Transactions on Systems Science and Cybernetics SSC4* **4**(2): 100–107.
- At each iteration of its main loop, A* needs to determine which of its partial paths to expand into one or more longer paths. It does so based on an estimate of the cost (total weight) to go to the goal node. Specifically, A* selects the path that minimizes

$$f(n) = g(n)+h(n)$$

  n := current last node on the path,

  $g(n)$:= cost of the path from the start node to n,

  $h(n)$:= heuristic that estimates the cost of the cheapest path from n to the goal (**a lower bound on the cost-to-go will ensure optimality of A***)
- The heuristics used are problem-specific.
- When $h(n) = 0$, A* algorithm reduces to Dijkstra

A* algorithm

Step 1: INITIALIZE

$g(j) := \infty$ for all $j \in N$

Step 2: COMPUTESHORTESTPATH

```
create heap(H)
create closedSet                                   \\ set containing nodes with permanent labels
g(s):= 0 and parent(s): = ∅
f(s):= g(s)+h(s)
H.insert(s; g(s) + h(s))
while H.peek() ≠ t  do
    i:= H.pop()                                    \\ remove the node with minimum f() value
    closedSet.add(i)
    for each (i; j) that starts at i do
        if j ∉ closedSet then
            tentative_value := g(i) + c_ij
            if g(j) > tentative_value then
                g(j):= tentative_value
                f(j) := g(j)+h(j)
                parent(j):=i
                if j ∉ H then
                    H.insert(j; f(j))
                else
                    H.decreasekey(j, f(j))         \\ sift up node j with new key f(j)
```

DP-based algorithms for single-source shortest paths

- DP recursion-based algorithms
  - Bellman-Ford algorithm … a label-correcting method useful for distributed computations
  - BMDP algorithm – a sophisticated label-correcting method
  - Basic idea
    - Find the shortest path from node 1 to node *j* using only one edge
    - Then find the shortest distance with the constraint that path must contain at most two edges; then one with at most three edges and so on
    - To be precise, let
      - $\lambda_j^{(l)}$ = length of the shortest path from the source node to node *j*, subject to the constraint that the path contains no more than *l* edges
      - Initially, $\lambda_j^{(l)} = c_{1j}; \forall j \neq 1$
      - What is $\lambda_j^{(l+1)}$?

DP recursion-based algorithms

- One of two possibilities
  - Shortest path contains at most l edges

    $\Rightarrow \lambda_j^{(l+1)} = \lambda_j^{(l)}$
  - Shortest path contains (l + 1) edges

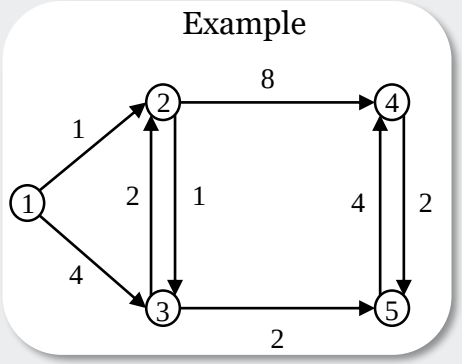
    $\Rightarrow \lambda_j^{(l+1)} = \min_k\{\lambda_k^{(l)} + c_{kj}\}$
- That is, among all nodes k ∋ shortest path from node 1 to node k contains at most l arcs, pick one giving $\min\{\lambda_k^{(l)} + c_{kj}\}$

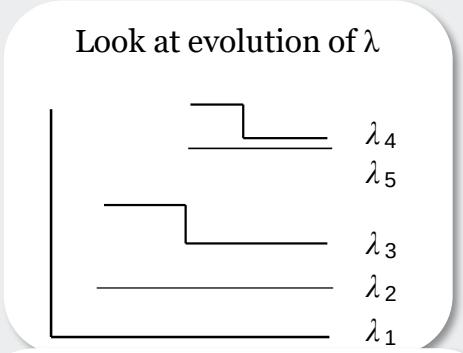
$$\Rightarrow \quad \lambda_j^{(l+1)} = \min\{\lambda_j^{(l)}, \min_{k \neq j}\{\lambda_k^{(l)} + c_{kj}\}\}$$
$$= \min\{\lambda_k^{(l)} + c_{kj}\} \text{ with } c_{jj} = 0 \text{ and } j \neq 1$$

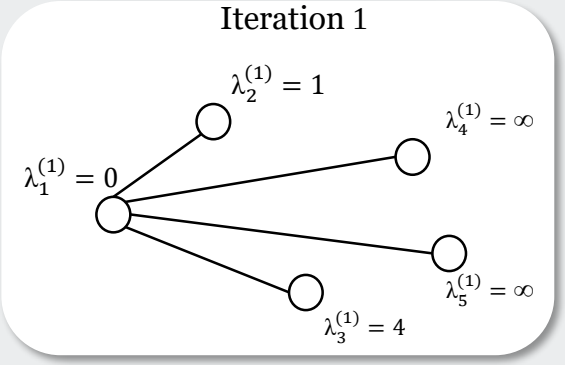
- If ∃ no negative cycles, a path can contain at most (n – 1) nodes ⇒ must iterate at most (n – 1) times
- Each iteration takes approximately m operations

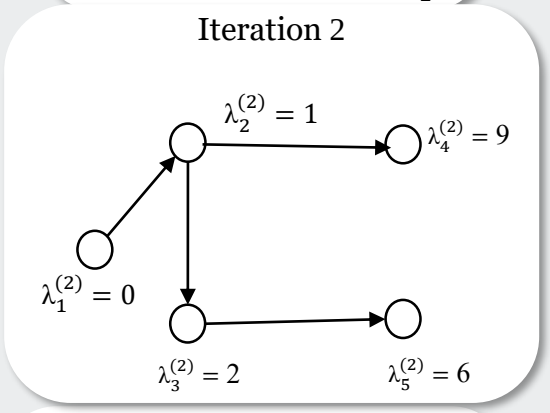
  ⇒ total computation = $O(mn)$(worst case)

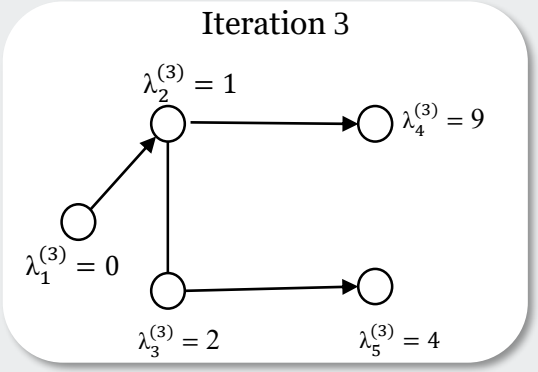
Example

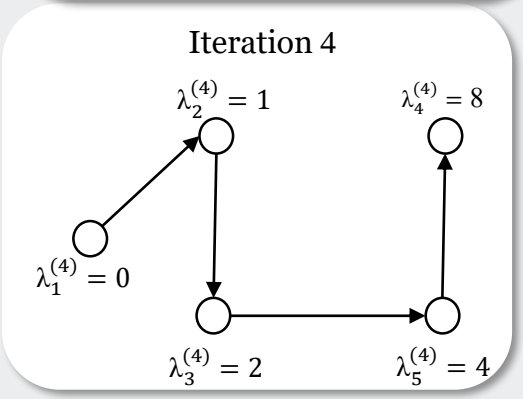








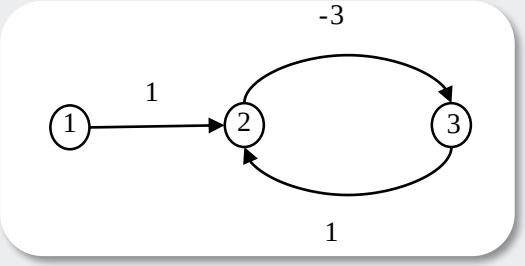




- Shortest path tree

- Can also initialize $\lambda_i = 0$ if $c_{ij} > 0$ (set $c_{jj} = \infty$)

Negative length cycles



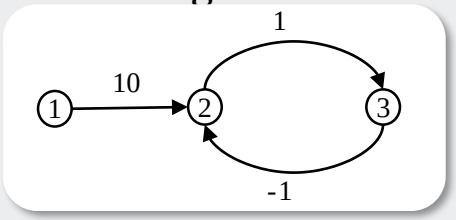
- Note: $\lambda_2^{(3)} = -1$ for path $1 \rightarrow 2 \rightarrow 3 \rightarrow 2$

- Bellman-Ford finds shortest path lengths from node 1 to node j subject to the constraint that node 1 is not repeated
- So, Bellman-Ford algorithm *never converges*, if there are negative length cycles not including node 1

  ⇒ Can test for negative length cycles by comparing $\lambda_j^{(N)}$ with $\lambda_j^{(N-1)}$

  ⇒ If $\lambda_j^{(N)} \neq \lambda_j^{(N-1)}$ ⇒ negative length cycles

- Bellman equation $\lambda_j = \min_k\{\lambda_k + ckj\}$ has *non-unique* solutions, if there are zero length cycles not involving node 1



- $\lambda_1 = 0$
- $\lambda_2 = \min\{10, \lambda_3 - 1\}$
- $\lambda_3 = \lambda_2 + 1$

⇒ $\lambda_2 = \min\{10, \lambda_2\}$

⇒ any solution $\lambda_2 = 1, 2, \ldots, 10$ is OK

- However, Bellman-Ford does converge to correct distances, if you start with ∞ for all nodes, except origin node 1 (not necessarily correct paths)
- Bellman-Ford can be implemented in a distributed fashion … later

BMDP (label-correcting, breadth-first implementation)

- Practically, a much better algorithm
- Suppose had a shortest path from node 1 to node j
- Then, this path must have passed through some node k and used the arc $<k, j>$
- "Principle of optimality"

  $\Rightarrow \lambda_j = \lambda_k + c_{kj}$

  $\Rightarrow \lambda_k$ must have been the shortest path length from node 1 to node k

  $\Rightarrow$ Node k is such that $(\lambda_k + c_{kj})$ is as small as possible, since λ_j is the shortest path from node 1 to node j

  $\Rightarrow$ Shortest paths must satisfy:

  - $\lambda_1 = 0$
  - $\lambda_j = \min_{(k,j) \in \text{In}(j)} \{\lambda_k + c_{kj}\}$
    where $\text{In}(j)$ = set of edges of the form $<k, j>$
- Nonlinear (implicit) functional relationships
- Use successive approximation to solve them

BMDP algorithm employs breadth-first search

- Put node 1 in queue and $\lambda_i = \infty$, $\forall i \neq 1$
- Do until queue is empty or gone through $(n - 1)$ passes & queue is nonempty
  - Pick node i at the head of the queue
  - If $\lambda_i + c_{ij} < \lambda_j$
    - $\lambda_j = \lambda_i + c_{ij}$
    - parent(j) = i
    - If $j \notin$ queue
      - ❖ Insert j into queue
    - Else
      - ❖ Leave it where it was (or) put it at the head of the queue if it was picked before
  - Pass 0 … scanning node 1
  - Pass 1 … scanning list of nodes added to queue by scanned node at pass 0, etc.
  - Gone through $(n - 1)$ passes and queue is nonempty

    ⇒ Negative cycle

Floyd-Warshall method

- Another way of solving DP equation is via Floyd-Warshall method
  - Finds all pairs shortest paths
  - Consider the shortest path from node *i* to node *j*
  - Suppose the shortest path goes through node *l*
  - Then
    - $\lambda_{ij} = \lambda_{il} + \lambda_{lj} \Rightarrow \lambda_{ij} = \min\{\min\{\lambda_{il} + \lambda_{lj}\}, c_{ij}\}$
  - In words
    - Each of λ_{il} and λ_{lj} must also be optimal … "principle of optimality"
  - One way of solving these recursions is as follows
    - Start with single-edge distances c_{ij} (i.e., no intermediate nodes) as starting estimates of shortest path lengths
    - Then, we calculate the shortest distances (and paths) under the constraint that only node 1 is used as an intermediate node, and then with constraint that only nodes 1 and 2 are used as intermediate nodes and so on
    - To be precise, let $\lambda_{ij}^{(k)}$ = shortest path length from node *i* to node *j* under the constraint that only nodes 1, 2, … , *k* can be used as intermediate nodes on the path

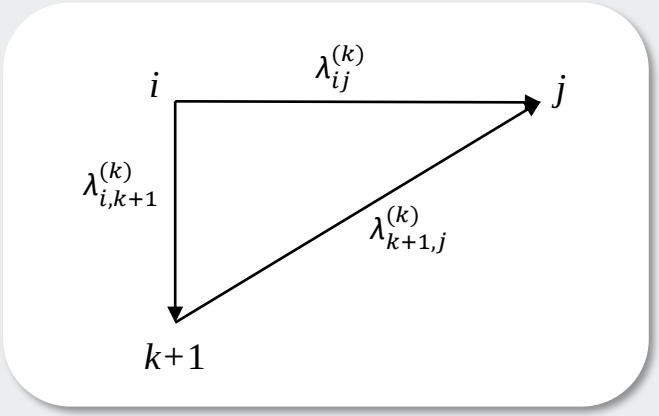
Floyd-Warshall method

- So, the algorithm is
  - $\lambda_{ij}^{(0)} = c_{ij}$, $\forall i, j$
  - for $k = 0, 1, \ldots, n-1$ do
    - $\lambda_{ij}^{(k+1)} = \min\{ \lambda_{ij}^{(k)}, \lambda_{i,k+1}^{(k)} + \lambda_{k+1,j}^{(k)} \}$, $\forall i, j = 1, 2, \ldots, n$
  - End do
- What do these recursion mean?
  - $\lambda_{ij}^{(0)}$ = shortest path lengths under the constraint that no intermediate nodes are involved
  - Now, shortest path between nodes i and j involving nodes $1, 2, \ldots, k+1$ has node $(k+1)$ on its path or not
    - If it does: $\lambda_{ij}^{(k+1)} = \lambda_{i,k+1}^{(k)} + \lambda_{k+1,j}^{(k)}$
    - If it does not: $\lambda_{ij}^{(k+1)} = \lambda_{ij}^{(k)}$

$$\left.\begin{array}{l} \lambda_{ij}^{(k+1)} = \lambda_{i,k+1}^{(k)} + \lambda_{k+1,j}^{(k)} \\ \lambda_{ij}^{(k+1)} = \lambda_{ij}^{(k)} \end{array}\right\} \Rightarrow \lambda_{ij}^{(k+1)} = \min\left\{\lambda_{ij}^{(k)}, \lambda_{i,k+1}^{(k)} + \lambda_{k+1,j}^{(k)}\right\}$$

Floyd-Warshall method

- Computational load
  - n steps, n^2 additions, and n^2 comparisons ⇒ $O(n^3)$
  - Storage: $O(n^2)$
  - Best for dense graphs
- Interpretation as a “triangle” update



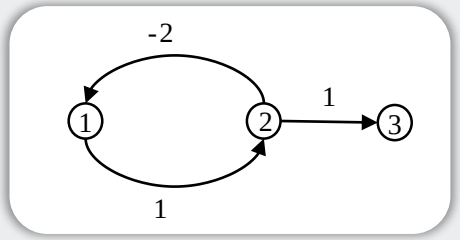
LP interpretation

- Recall all-pairs problem

$$\max \sum_{i=1}^{n}\sum_{j=1}^{n} \lambda_{ij}$$
$$\text{s.t.} \quad \lambda_{ij} \le \lambda_{ik} + \lambda_{kj} \;\forall i, j, k \ne i, j$$
$$\lambda_{ij} \le c_{ij}; i, j = 1, 2, .., n$$

- Relaxation
  - Initially $\lambda_{ij} = c_{ij}$
  - If $\lambda_{ij} > \lambda_{ik} + \lambda_{kj}$
    - Set $\lambda_{ij} = \lambda_{ik} + \lambda_{kj}$
- Can detect negative cycles via $\lambda_{ii} < 0$ and setting $\lambda_{ii} = \infty$

Initially



$$\begin{bmatrix} \infty & 1 & \infty \\ -2 & \infty & 1 \\ \infty & \infty & \infty \end{bmatrix} \xrightarrow{k=0} \begin{bmatrix} \infty & 1 & \infty \\ -2 & -1 & 1 \\ \infty & \infty & \infty \end{bmatrix} \xrightarrow{k=1} \begin{bmatrix} -1 & 1 & 2 \\ -2 & -1 & 1 \\ \infty & \infty & \infty \end{bmatrix}$$

stop: $\lambda_{22} = -1$ stop: $\lambda_{11}, \lambda_{22} = -1$

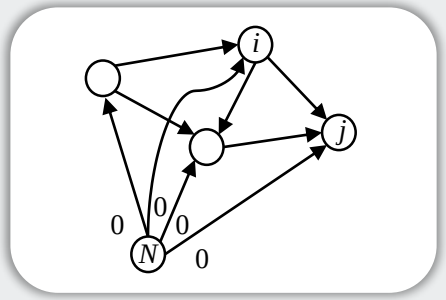
Extension of Dijkstra to all pairs shortest path problem

- If edge weights are non-negative, invoke Dijkstra n times
  - $\Rightarrow O(n^3)$ straightforward implementation
  - $O(n^2C + mn)$ Dial's bucket implementation
  - $O(mn \log_{\lceil 2+\frac{m}{n} \rceil} n)$ heap implementation
- But, what if we have negative edge weights
  - Add a dummy node N to the graph G s.t. $c_{Ni} = 0, \forall i = 1, 2, \ldots, n \Rightarrow$ arcs are of the form $<N, i>$
  - For each i compute shortest distance from N using Bellman-Moore-d'Esopo-Pape $\Rightarrow O(mn)$ time

$$c'_{ij} = \lambda'_i - \lambda'_j + c_{ij} \geq 0$$

$\lambda'_i =$ shortest distance from N to i

$\lambda'_j =$ shortest distance from N to j



- Define
  - Recall LP inequalities: $\lambda'_j \leq \lambda'_i + c_{ij}$
  - Now we can apply Dijkstra n times with total computational load:
    - $O(mn \log_{\lceil 2+\frac{m}{n} \rceil} n) + O(nm)$
  - How do we convert back to original distances: easy!!
    - If p is the path from vertex k to vertex l, then

$$\lambda_{kl} = \lambda'_{kl} + \left(\lambda'_l - \lambda'_k\right)$$

$$\lambda_{kl} = \lambda'_{kl} + \left(\lambda'_l - \lambda'_k\right) = \sum_{<i,j> \in P_{kl}} c'_{ij} + \left(\lambda'_l - \lambda'_k\right)$$

$$= \sum_{<i,j> \in P_{kl}} \left[c_{ij} + \lambda'_i - \lambda'_j\right] + \left(\lambda'_l - \lambda'_k\right) = \sum_{<i,j> \in P_{kl}} c_{ij}$$

Distributed shortest path algorithms

- Idea
  - Each node can compute its distance from other nodes (or symmetrically the distances of other nodes from it) with minimal communication with its neighbor and knowing only the (estimates of) distances from its neighbors
  - Node does not know the topology of network
  - Based on Bellman-Ford algorithm

$$\lambda_j^{(l+1)} = \min_{k \in \ln(j)} \{\lambda_k^{(l)} + c_{kj}\}, \forall j \neq 1$$

$$\text{and } \lambda_1^{(l+1)} = 0$$

- Especially useful in communication networks (e.g., internet)
  - Must continually update in response to changes in link states or link traffic

Distributed shortest path algorithms

- Two types of distributed algorithms:
  - Synchronous
    - Each node executes simultaneously & exchanges data with its neighbors & executes again
    - Will converge in $(N-1)$ iterations, but has two problems:
      - ❖ All nodes must agree to start the algorithm at a certain time (Need a global clock)
      - ❖ Need a mechanism for restarting the algorithm if
        a) c_{ij} changes due to traffic changes (or)
        b) Link status changes
  - Asynchronous
    - Execute the iteration whenever node j receives data from its neighbors
    - Original ARPA network did this
      - ❖ Nodes exchanged their data every 625 *msecs* asynchronously
    - Under what conditions, does such an algorithm converge?
    - What is the speed of convergence? . . . it can be very slow compared to a centralized scheme in some cases!!

A model of asynchronous computation

- Let $\lambda_k^{(j)}(t)$: estimate of m shortest distances of each neighbor node $k \in \text{In}(j)$ communicated to node j
- $\lambda_j(t)$= estimate of the shortest distance of node j which was last computed at node j via

$$\lambda_j(t) = \min_k \{\lambda_k^{(j)}(t) + c_{kj}\}$$

- Assumptions:
  - $c_{kj} \geq 0$
  - Nodes never stop updating their own estimates
  - Initial estimates $\lambda_j(t_0)$ and communicated estimates $\lambda_k(t_0)$ are nonnegative
  - Old distance information is eventually purged from the system ⇒ finite communication time
- Result
  - Algorithm converges to correct shortest distance for sufficiently large time t

Outline of proof

- We use monotonicity property of Bellman-Ford iteration:

$$\lambda_j = \min_{k \in \mathrm{In}(j)} \{\lambda_k^{(j)} + c_{kj}\}$$

- Suppose $\bar{\lambda}_k \geq \underline{\lambda}_k, \forall k \in \mathrm{In}(j)$

$$\Rightarrow \quad \min_{k \in \mathrm{In}(j)} \{\underline{\lambda}_k + c_{kj}\} \leq \min_{k \in \mathrm{In}(j)} \{\bar{\lambda}_k + c_{kj}\}$$

- Suppose we start Bellman-Ford iteration with $\lambda_k^{(0)} = \underline{\lambda}_k = 0$ and $\lambda_k^{(0)} = \bar{\lambda}_k = \infty$
- Then, with the first initialization, the distances converge monotonically up to correct distances

$$\Rightarrow \ \underline{\lambda}_j^{(l)} \leq \underline{\lambda}_k^{(l+1)} \leq \lambda_j$$

- With the second initialization, the distances converge monotonically down to correct distances

$$\Rightarrow \ \lambda_j \leq \bar{\lambda}_j^{(l+1)} \leq \bar{\lambda}_j^{(l)}$$

- Since every bounded monotonic sequence has a limit:

$$\lim_{l \geq \bar{l}} \bar{\lambda}_j^{(l)} \rightarrow \lambda_j; \bar{l} = N - 1$$

$$\& \ \lim_{l \geq \bar{l}} \bar{\lambda}_j^{(l)} \rightarrow \lambda_j; \bar{l} \approx \frac{\max_i \lambda_i}{\min_{(i,j)} c_{ij}}$$

Outline of proof

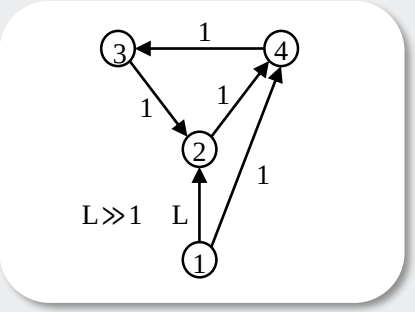
- Now, we show that $\forall l$, $\exists$ a time $t(l)$ such that $\forall t \geq t(l)$

$$\left.\begin{aligned} \underline{\lambda}_j^{(l)} \leq \lambda_j(t) \leq \overline{\lambda}_j^{(l)} \quad & \forall j \\ \underline{\lambda}_k^{(l)} \leq \lambda_k^{(j)}(t) \leq \overline{\lambda}_k^{(l)} \quad & \forall k \end{aligned}\right\} \text{Communication \& computed distances are bounded}$$

- Proof is by induction and trivial
  - If it is valid for l, we can show that it is valid for $(l + 1)$
  - For $l = 0$, it is true due to nonnegative assumption
  - Since the communicated distances satisfy the bounds at iteration l, the computed distances must also satisfy the bounds, since the nodes update their distances using Bellman-Ford recursion
  - Since the nodes eventually communicate, their communicated distances must satisfy the new bounds
- For detailed proof, see:
  - D.P. Bertsekas and R. Gallager, Data Networks, 2nd edition, Prentice-Hall, 1992
  - D.P. Bertsekas and J. Tsitsiklis, Parallel and Distributed Computation, Prentice-Hall, 1990

"Bad news Phenomena"

- Slow reaction time to changes in link status



Before (1, 4) fails

$\lambda_2 = 3$
$\lambda_3 = 2$
$\lambda_4 = 1$

Suppose use synchronized Bellman-Ford

	$l=0$	1	2	3	4	5	6	7	8	9	10
λ_2	3	3	3	6	6	6	9	9	9	12	12
λ_3	2	2	5	5	5	8	8	8	11	11	11
λ_4	1	4	4	4	7	7	7	10	10	10	13 etc.

⇒ Takes as many as L iterations before the algorithm converges to correct shortest path lengths:

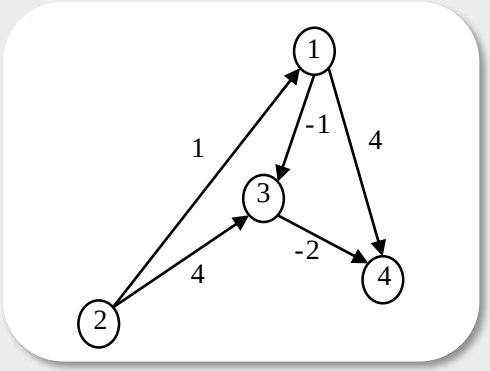
$$L,\ L+2,\ (L+1)$$

⇒ How to make distributed Bellman-Ford algorithm faster? Research Problem

- Starting references
  - See books by Bertsekas
  - D.P. Bertsekas, "Distributed Dynamic Programming," IEEE Trans. AC, pp. 610-616, 1982
  - D.P. Bertsekas, "Dynamic behavior of shortest path routing algorithms for communication networks," IEEE Trans. AC, pp. 60-74, 1982
  - K.M. Chandy and J. Misra, "Distributed computation on graphs", CACM, pp. 833-837, 1982.

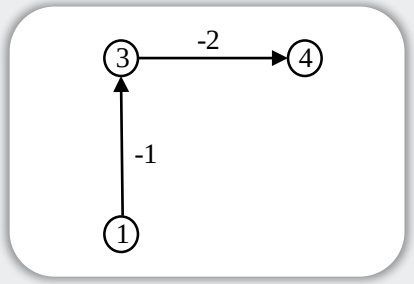
Shortest paths on acyclic graphs

- Applications in PERT networks
- Use depth-first search to topologically order nodes
- Topological order
  - Order nodes from $1 - n$ ∋ if $<i, j>$ is an edge, i appears before j in the order
- Example:



  - Topological order 1 3 4
  - 2 is never considered ⇒ $\lambda_2 = \infty$
  - Iteration 0
    - $\lambda_1 = 0, \lambda_3 = -1, \lambda_4 = 4$
  - iteration 1
    - $\lambda_4 = -3$
  - iteration 2
    - done!!

⇒



Shortest paths on acyclic graphs

- Q: how to get an initial topological ordering?
- A: recursive DFS for topological ordering: call dfs(i)
- How does dfs(i) work?

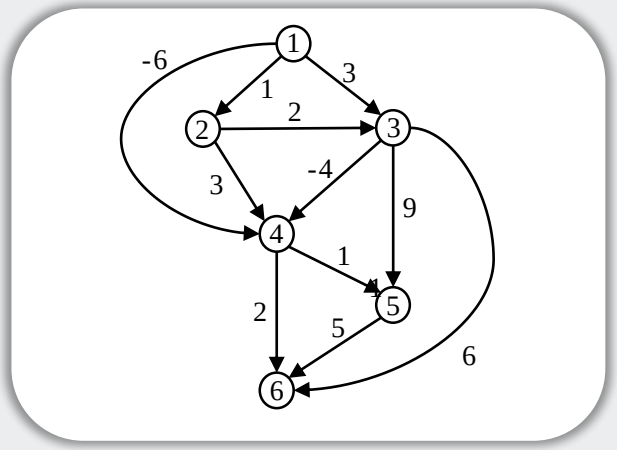
```
Procedure dfs(i)
mark that node i is visited ...pre-visit(i)
for (i, j) ∈ out(i) do
    if j is not visited
        call dfs(j)
    end if
end do
post-visit(i)
```

- $O(m)$ complexity

```
for each i in decreasing post visit ⇒ Topological ordering
    λ_j = min{λ_j, λ_i + c_ij}, ∀ j ∈ out(i)
end i
```

- $O(m)$ complexity

Example



Topological order							
pre visit							
sequence:		1	4	6	5	2	3	nodes: 1 2 3 4 5 6
post visit							
sequence:		6	5	4	3	2	1	Topological order from
Topo. ord:		1	2	3	4	5	6	reversed post visit sequence
Iteration	0:	λ_2	=	1,	λ_3	=	3,	$\lambda_4 = -6$
Iteration	1:	λ_3	=	3,	λ_4	=	–6	
Iteration	2:	λ_4	=	–6,	λ_5	=	12,	$\lambda_6 = 9$
Iteration	3:	λ_5	=	–5,	λ_6	=	-4	
Iteration	4:	λ_6	=	–4	done!!			

Viterbi decoding

- References
  - J. K. Omura, "On the Viterbi decoding algorithm," IEEE T - IT, 1969, 177-179
  - G. D. Forney, The Viterbi algorithm, Proc. IEEE, 1973, pp. 268- 278
- To send binary messages over noisy communication channels, we use coding to enhance the reliability of communication
- A common type of coding is the *convolutional* coding
- Source data sequence: $\{w_1 \;\; w_2 \cdots\}$, $w_k \in \{0, 1\}$
- Coded sequence $\{\underline{y}_1 \;\; \underline{y}_2 \cdots\}$
  - $\underline{y}_k$ is an r-dimensional vector of binary numbers

$$\begin{bmatrix} y_k^1 \\ y_k^2 \\ \vdots \\ y_k^r \end{bmatrix}; \;\; y_k^i \in \{0,1\}$$

- The relationship between $\underline{y}_k$ and w_k

$$\left.\begin{aligned} \underline{y}_k &= C\underline{x}_{k-1} + \underline{d}w_k \\ \underline{x}_k &= A\underline{x}_{k-1} + \underline{b}w_k \end{aligned}\right\} \underline{x}_0 \text{ is given}$$

  - $\underline{x}_k$ is a q-dimensional binary vector (state vector)
  - Products and sums involved in $C\underline{x}_{k-1} + d\underline{w}_k$ and $A\underline{x}_{k-1} + b\underline{w}_k$ are computed using binary arithmetic

Example

- $q = 2,\ r = 3$

$$C = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix}; A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}; \underline{d} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}; \underline{b} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

- Old state → transition → new state
- $\underline{x}_{k-1} = [0\ 1]$ and $w_k = 0 \Rightarrow \underline{x}_k = [1\ 1]$ and $\underline{y}_k = [0\ 1\ 1]$, etc.
- key: given $\underline{x}_0$ and $w_1, \ldots, w_k$ can find $\underline{y}_1, \ldots, y_k$
- Unfortunately, we receive a noisy version of $\underline{y}_k$
- Let the received message be $\underline{z}_k$
- Know $p(\underline{z}_k|\underline{y}_k)$ . . . likelihood function
- So instead of $\underbrace{[\underline{y}_1, \ldots, \underline{y}_N]}_{Y_N}$, we receive $\underbrace{[\underline{z}_1, \ldots, \underline{z}_N]}_{Z_N}$ over N samples
- If assume independent errors, the likelihood function is:

$$p(Z_N \mid Y_N) = \prod_{k=1}^{N} p(\underline{z}_k \mid \underline{y}_k)$$

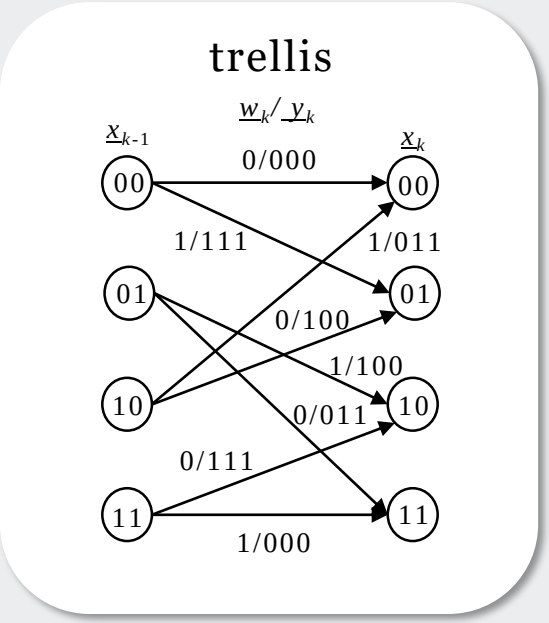
- A maximum likelihood detector converts

$$Z_N \rightarrow \hat{Y}_N = [\hat{\underline{y}}_1, \ldots, \hat{\underline{y}}_N]$$

- Via

$$p(Z_N \mid \hat{Y}_N) = \max_{Y_N} \{p(Z_N \mid Y_N)\}$$

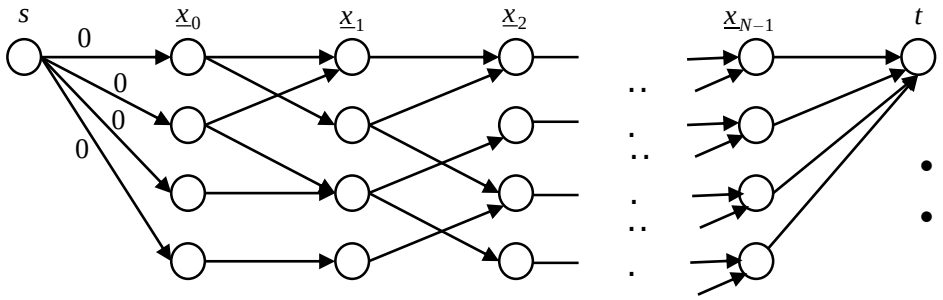
- Given $\hat{Y}_N$, we can find $[\hat{w}_1, \ldots, \hat{w}_N]$. . . Recall trellis



Why is this a shortest path problem?

$$\max_{Y_N}\left\{\mathrm{p}(Z_N \mid Y_N)\right\} = \max_{Y_N}\left\{\prod_{k=1}^{N}\mathrm{p}(\underline{z}_k \mid \underline{y}_k)\right\} \Rightarrow \min_{[\underline{y}_1,\ldots,\underline{y}_N]}\left\{\sum_{k=1}^{N} -\ln \mathrm{p}(\underline{z}_k \mid \underline{y}_k)\right\}$$

- Given $\underline{z}_k$, we can assign to each arc in the trellis, the length $\{-\ln \mathrm{p}(\underline{z}_k|\underline{y}_k)\}$, where $\underline{y}_k$ is the code word associated with the arc



- Get shortest $s \rightarrow t$ path
- Obtain w_k from $\underline{y}_k$ $k = 1,2,\ldots,N$

  - ∃ variations of this idea . . . see references

- For deterministic optimization problem and its connection to the shortest path problem, see
  - D.P. Bertsekas, Dynamic programming: Deterministic and Stochastic Models, Prentice-Hall, 1987

$$\min J = g_N(\underline{x}_N) + \sum_{k=1}^{N-1} g_k(\underline{x}_k, \underline{u}_k)$$

$$\text{s.t.} \quad \underline{x}_{k+1} = f(\underline{x}_k, \underline{u}_k), \underline{x}_k \sim \text{finite set of states}$$

- Some times, we need M shortest paths from a source to a destination
  - E. Lawler, Combinatorial Optimization: Networks and Matroids, Holt, Reinhart, and Whinston, NY, 1976
  - C.C. Skiscism and B.L. Golden, "Computing K-shortest Path Lengths in Euclidean Networks," Networks, Vol. 17, pp. 341-352, 1987

Martins' algorithm

- The multiple objective shortest path problems with n objectives can be described mathematically as

$$\min_{p \in P_{s,t}} (z_1(p), z_2(p), \ldots, z_n(p))$$

  in which $P_{s,t}$ represents the set of all paths between a starting node s and terminal node t, while $z_i(p)$ is the cost of path p with respect to objective i

- Each link (i,j), connecting node i and node j, has a label of values representing the costs for different objectives assigned to it

$$\underline{c}_{ij} = [c_{ij1}, c_{ij2}, \ldots, c_{ijn}]$$

- Martins' algorithm is a label setting algorithm and makes use of labels to indicate the "distance" to a certain node. A label can be represented as $L = [v, \underline{c}, prev_L]$ with v as the node which the label is being assigned to, $\underline{c} = [c_1, c_2, \ldots, c_n]$ and $prev_L$ denoting the reference to the previous label

- **Definition** 1 (dominance) The vector $[a_1,\ldots,a_n]$ dominates the vector $[b_1,\ldots,b_n]$ if and only if

$$(\forall i \in \{1,\ldots,n\} : a_i \leq b_i) \wedge (\exists i \in \{1,\ldots,n\} : a_i < b_i)$$

- **Definition 2** (Pareto optimality) A set of elements S is Pareto optimal if and only if none of the elements of S is dominated by another element

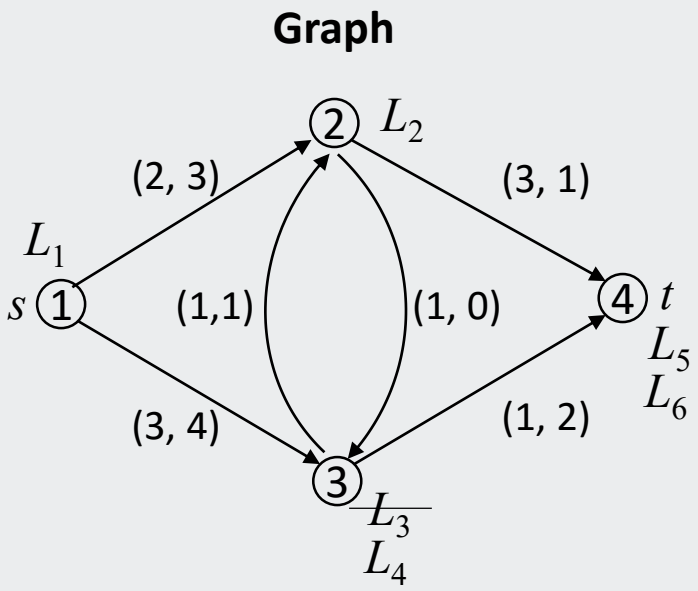
- **Definition 3** (lexicographic ordering) The vector $[a_1,\ldots,a_n]$ is lexicographically smaller (denoted by $<_l$) than the vector $[b_1,\ldots,b_n]$ if

$$\exists k \in \{1,\ldots,n\} : (\forall i < k : a_i = b_i) \wedge (a_k < b_k)$$

Martins' algorithm

```
T := ∅
originLabel := [origin, [0,...,0], ∅]
origin.addLabel(originLabel)
T.add(originLabel)
while (T is not empty) do
    label := T.removeMin()
    owner := label.owner()
    neighbors := owner.neighbors()
    for all (nb in neighbors) do
        link := getLinkBetween(owner, nb)
        newLabel := [nb, label+link.cost, label]
        if (newLabel not dominated by any of nb.labels()) then
            nb.removeDominatedLabel(newLabel)
            nb.addLabel(newLabel)
            T.add(newLabel)
        end if
    end for
end while
```

Example



Iteration #	T	Removed label	Node 1 labels	Node 2 labels	Node 3 labels	Node 4 labels
0	$\{L_1\}$	$\varnothing$	L_1=[1,[0,0], $\varnothing$]	$\varnothing$	$\varnothing$	$\varnothing$
1	$\{L_2, L_3\}$	L_1	L_1=[1,[0,0], $\varnothing$]	L_2=[2,[2,3],L_1]	L_3=[3,[3,4],L_1]	$\varnothing$
2	$\{L_4, L_5\}$	L_2	L_1=[1,[0,0], $\varnothing$]	L_2=[2,[2,3],L_1]	L_4=[3,[3,3],L_2]	L_5=[4,[5,4],L_2]
3	$\{L_5, L_6\}$	L_2	L_1=[1,[0,0], $\varnothing$]	L_2=[2,[2,3],L_1]	L_4=[3,[3,3],L_2]	L_5=[4,[5,4],L_2] L_6=[4,[4,5],L_4]

- In each iteration, we remove the minimum label from T according to lexicographical ordering
- In iteration 2, label L_4 dominates L_3. Hence, L_3 is removed before we add L_4
- In iteration 3, label L_6 is Pareto optimal, while not dominating label L_5. Hence, L_6 was added to node 4 label list without removing L_5

Other shortest path algorithms

- Bidirectional Dijkstra
  - T. Nicholson, J. Alastair, "Finding the shortest route between two points in a network," *The computer journal* 9.3, pp 275-280, 1966
- Martins' algorithm
  - E.Q.V. Martins, "On a multicriteria shortest path problem," *European Journal of Operational Research*, vol. 16, no. 2, pp 236-245, 1984
- Bidirectional Martins' algorithm
  - S. Demeyer, J. Goedgebeur, P. Audenaert, M. Pickavet, P. Demeester, "Speeding up Martins' algorithm for multiple objective shortest path problems," 4OR, pp 323-48, 2013
- D* algorithm
  - Koenig, Sven, and Maxim Likhachev, "Improved fast replanning for robot navigation in unknown terrain," *IEEE International Conference on Robotics and Automation, 2002.*
- Theta star algorithm
  - Nash A, Daniel K, Koenig S, Felner A. "Theta*: Any-Angle Path Planning on Grids," *National Conference on Artificial Intelligence*, vol. 22, no. 2, pp 1177
- Delta-step algorithm
  - U. Meyer, P. Sanders "Δ-stepping: a parallelizable shortest path algorithm," *Journal of Algorithms*, pp 114-52, 2003

Summary

- BMDP as a DP-based algorithm
- Distributed Bellman-Ford algorithm
- Shortest path algorithms for acyclic graphs
- Viterbi decoding