The shortest path tree problem

Combinatorial optimization

Giovanni Righini

UNIVERSITÀ DEGLI STUDI DI MILANO

The Shortest Path Tree Problem

Data:

- a digraph $\mathcal{D} = (\mathcal{N}, \mathcal{A})$ with $|\mathcal{N}| = n$ nodes and $|\mathcal{A}| = m$ arcs;
- a source node $s \in \mathcal{N}$;
- a cost function $c : \mathcal{A} \mapsto \Re$.

Shortest Path Tree Problem.
Find all minimum cost (i.e. shortest) paths from s to all nodes in $\mathcal{N}$.

The problem is called Shortest Path Tree/Arborescence Problem, because of a property of its solution: the set of all shortest paths forms a spanning arborescence rooted in s.

The shortest paths arborescence

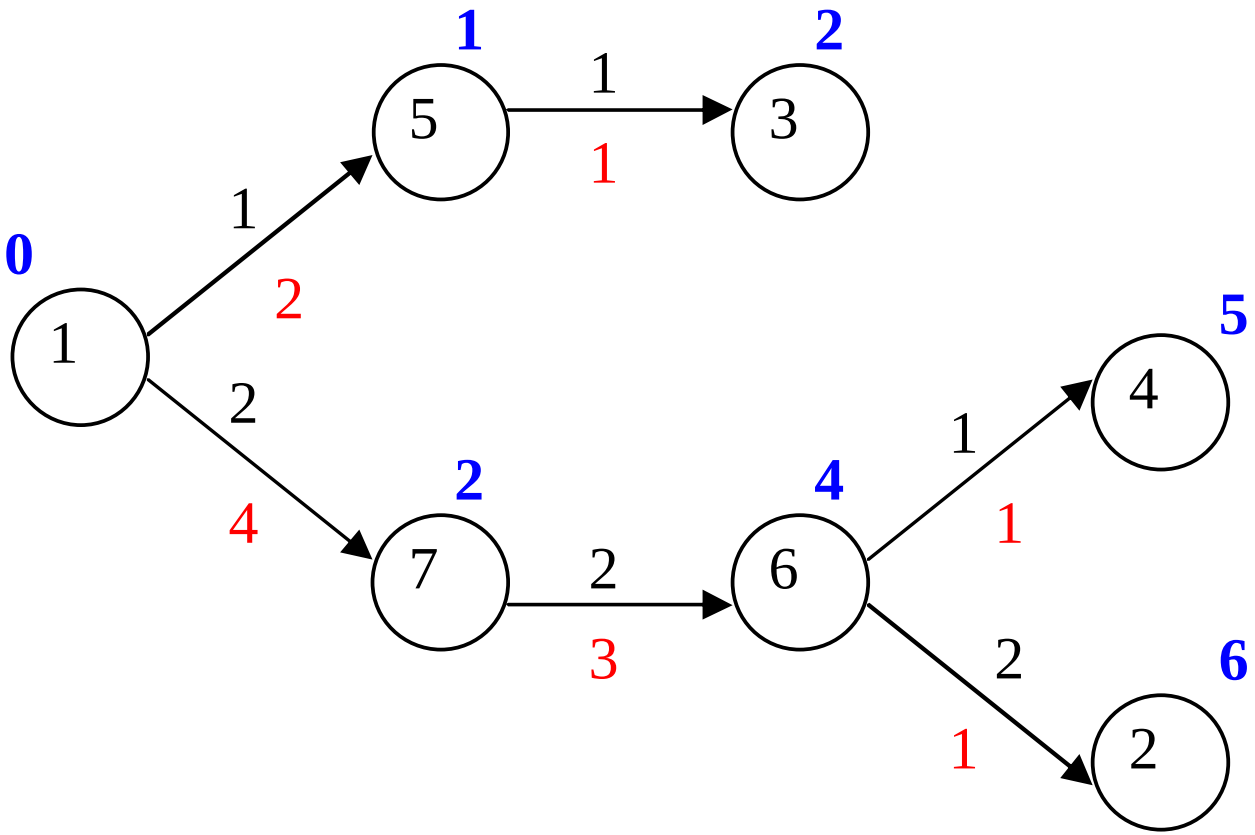

Figure: A shortest paths arborescence ($s = 1$). Costs are black. Flows are red. Distances are blue.

Bellman's optimality principle

Bellman's optimality principle states that every optimal policy is made by optimal sub-policies.

Translating this statement for the SPP:
every shortest path from s to $t \in \mathcal{N}$ visiting $i \in \mathcal{N}$ is made by the shortest path from s to i and the shortest path from i to t.

As a consequence of this principle, the set of all the shortest paths from s to $\mathcal{N}$ forms a spanning arborescence rooted in s.

But this "principle" is indeed a theorem: it can be proved, instead of assumed.

We do *not* assume a priori that we are looking for a spanning arborescence rooted in s.

The mathematical model: variables

Variables. $x_{ij} \in \mathcal{Z}_+ \; \forall (i,j) \in \mathcal{A}$: number of shortest paths that use arc (i,j).

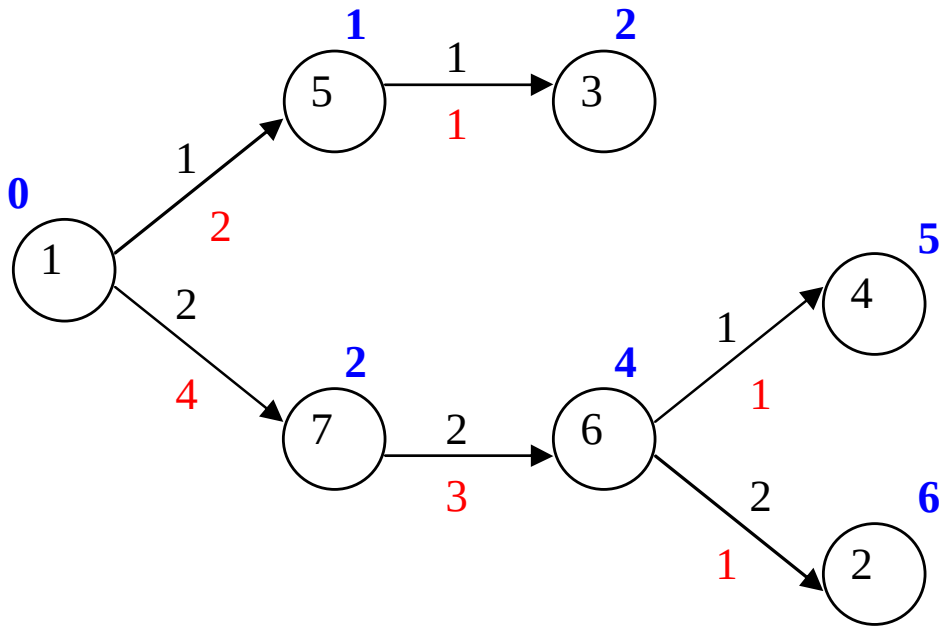

Figure: One unit of flow goes from s to each other node. The flow on each arc equals the number of nodes that are reached through it.

The mathematical model: obj. function

Objective function. Minimize each path from s to t $\forall t \in \mathcal{N}$:

$$\text{minimize} \sum_{(i,j)\in\mathcal{A}} c_{ij}x_{ij}.$$

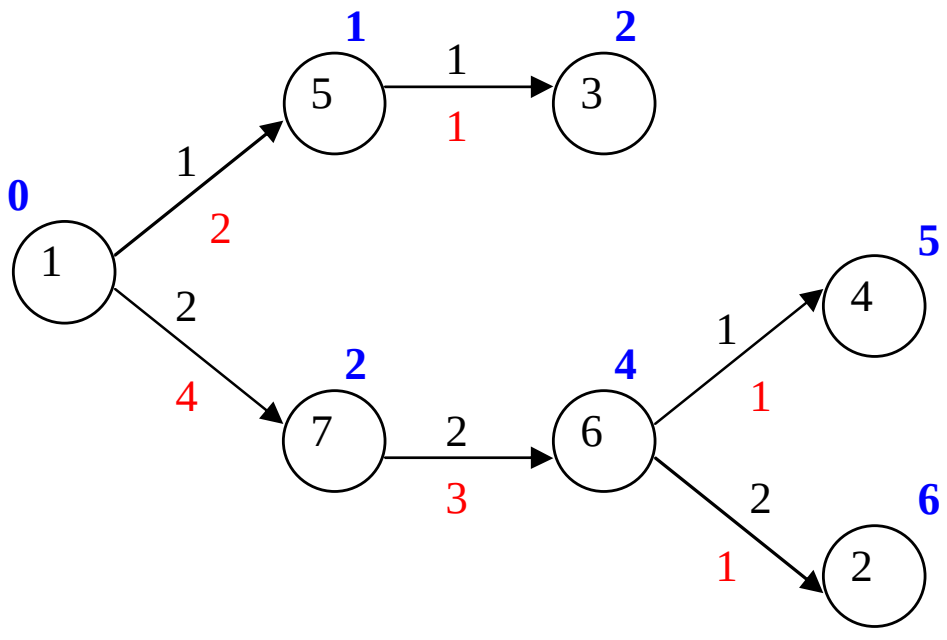

Figure: The sum of the costs times the flow equals the sum of the distances:
$1 \times 2 + 1 \times 1 + 2 \times 4 + 2 \times 3 + 1 \times 1 + 2 \times 1 = 0 + 1 + 2 + 2 + 4 + 5 + 6 = 20$.

The mathematical model: constraints

Constraints. *Flow conservation constraints* for each shortest path from s to $t \in \mathcal{N}$:

$$\sum_{(j,i)\in\delta_i^-} x_{ji} - \sum_{(i,j)\in\delta_i^+} x_{ij} = 0 \quad \forall i \in \mathcal{N}\setminus\{s,t\}$$

$$\sum_{(j,s)\in\delta_s^-} x_{js} - \sum_{(s,j)\in\delta_s^+} x_{sj} = -1$$

$$\sum_{(j,t)\in\delta_t^-} x_{jt} - \sum_{(t,j)\in\delta_t^+} x_{tj} = +1$$

Summing them up for all $t \in \mathcal{N}$:

$$\sum_{(j,i)\in\delta_i^-} x_{ji} - \sum_{(i,j)\in\delta_i^+} x_{ij} = 1 \quad \forall i \in \mathcal{N}\setminus\{s\}$$

$$\sum_{(j,s)\in\delta_s^-} x_{js} - \sum_{(s,j)\in\delta_s^+} x_{sj} = 1 - n$$

SPP: primal formulation (ILP)

$$\hat{P}) \text{ minimize } \sum_{(i,j)\in\mathcal{A}} c_{ij}x_{ij}$$

$$\text{s.t. } \sum_{(j,i)\in\delta_i^-} x_{ji} - \sum_{(i,j)\in\delta_i^+} x_{ij} = 1 \qquad \forall i \in \mathcal{N}\setminus\{s\}$$

$$\sum_{(j,s)\in\delta_s^-} x_{js} - \sum_{(s,j)\in\delta_s^+} x_{sj} = 1 - n$$

$$x_{ij} \in \mathcal{Z}_+ \qquad \forall (i,j) \in \mathcal{A}.$$

Observation 1. The constraint matrix is totally unimodular.

Observation 2. The right-hand-sides of the constraints are all integer numbers.

Therefore every base solution of the continuous relaxation of $\hat{P}$ has integer coordinates.

Reformulation (relaxation) of the primal problem (LP)

Hence we can relax the integrality restrictions:

$$
\begin{aligned}
P)\ \text{minimize} \quad & \sum_{(i,j)\in\mathcal{A}} c_{ij}x_{ij} \\
\text{s.t.} \quad & \sum_{(j,i)\in\delta_i^-} x_{ji} - \sum_{(i,j)\in\delta_i^+} x_{ij} = 1 && \forall i \in \mathcal{N}\setminus\{s\} \\
& \sum_{(j,s)\in\delta_s^-} x_{js} - \sum_{(s,j)\in\delta_s^+} x_{sj} = 1 - n \\
& x_{ij} \geq 0 && \forall (i,j) \in \mathcal{A}.
\end{aligned}
$$

This primal problem P has a dual problem D.
For the primal-dual pair (P, D) the LP duality theorems hold.

SPP: Dual formulation (LP)

$$P) \text{ minimize } \sum_{(i,j)\in\mathcal{A}} c_{ij}x_{ij}$$

$$\text{s.t. } \sum_{(j,i)\in\delta_i^-} x_{ji} - \sum_{(i,j)\in\delta_i^+} x_{ij} = 1 \qquad \forall i \in \mathcal{N}\setminus\{s\}$$

$$\sum_{(j,s)\in\delta_s^-} x_{js} - \sum_{(s,j)\in\delta_s^+} x_{sj} = 1 - n$$

$$x_{ij} \geq 0 \qquad \forall (i,j) \in \mathcal{A}.$$

$$D) \text{ maximize } \sum_{i\in\mathcal{N}\setminus\{s\}} y_i + (1-n)y_s$$

$$\text{s.t. } y_j - y_i \leq c_{ij} \qquad \forall (i,j) \in \mathcal{A}$$

$$y_i \text{ free} \qquad \forall i \in \mathcal{N}.$$

An equivalent dual formulation (LP)

$$
\begin{aligned}
D)\ \text{maximize} \quad & \sum_{i \in \mathcal{N} \setminus \{s\}} y_i + (1-n) y_s & \\
\text{s.t.} \quad & y_j - y_i \leq c_{ij} & \forall (i,j) \in \mathcal{A} \\
& y_i \text{ free} & \forall i \in \mathcal{N}.
\end{aligned}
$$

Observation 1. Adding a constant α to each y variable, nothing changes. Hence we can fix a variable:

$$y_s = 0$$

Observation 2. There are m inequality constraints, $n-1$ original y variables and m slack variables. The LP tableau of the dual problem has m rows and $n-1+m$ columns. Hence, in each base solution of D there should be m basic variables and $n-1$ non-basic (null) variables. For the complementary slackness theorem, there should be $n-1$ basic variables in the primal problem.

An equivalent primal formulation (LP)

$$P) \text{ minimize } \sum_{(i,j)\in\mathcal{A}} c_{ij}x_{ij}$$

$$\text{s.t. } \sum_{(j,i)\in\delta_i^-} x_{ji} - \sum_{(i,j)\in\delta_i^+} x_{ij} = 1 \qquad \forall i \in \mathcal{N}\setminus\{s\}$$

$$\sum_{(j,s)\in\delta_s^-} x_{js} - \sum_{(s,j)\in\delta_s^+} x_{sj} = 1 - n$$

$$x_{ij} \geq 0 \qquad \forall (i,j) \in \mathcal{A}.$$

Observation 3. There are n equality constraints that are not linearly independent: summing up all the rows we obtain $0 = 0$. Hence we can delete a constraint: the flow conservation constraint for s.

Observation 4. There are now $n-1$ equality constraints and m variables. The LP tableau of P has $n-1$ rows and m columns. Hence, in each base solution of P there are $n-1$ basic variables and $m-(n-1)$ non-basic variables.

Complementary slackness conditions (CSC)

$$P') \text{ minimize } z = \sum_{(i,j)\in\mathcal{A}} c_{ij}x_{ij}$$

$$\text{s.t.} \sum_{(j,i)\in\delta_i^-} x_{ji} - \sum_{(i,j)\in\delta_i^+} x_{ij} = 1 \qquad \forall i \in \mathcal{N}\setminus\{s\}$$

$$x_{ij} \geq 0 \qquad \forall (i,j) \in \mathcal{A}.$$

$$D') \text{ maximize } w = \sum_{i\in\mathcal{N}\setminus\{s\}} y_i$$

$$\text{s.t. } y_j - y_i \leq c_{ij} \qquad \forall (i,j) \in \mathcal{A}$$

$$y_i \text{ free} \qquad \forall i \in \mathcal{N}\setminus\{s\}.$$

Primal CSCs: $x_{ij}(c_{ij} + y_i - y_j) = 0$

Basic variables in P' correspond to active constraints in D'.

Only arcs (i,j) for which $y_i + c_{ij} = y_j$ can carry flow x_{ij}.

The Ford-Fulkerson algorithm (1962)

A spanning s-arborescence is completely described by a vector of predecessors, one for each node but s.

---

for $i \in \mathcal{N} \setminus \{s\}$ **do**
  $y_i \leftarrow \infty$
  $\pi_i \leftarrow nil$
$y_s \leftarrow 0$
$\pi_s \leftarrow s$
$V \leftarrow \{(s,j) \in \mathcal{A}\}$
while $V \neq \emptyset$ **do**
  $(i,j) \leftarrow \text{Select}(V)$
  $y_j \leftarrow y_i + c_{ij}$
  $\pi_j \leftarrow i$
  Update(V)

Data structures:

- a predecessor label, $\pi_i \ \forall i \in \mathcal{N}$;
- a cost label, $y_i \ \forall i \in \mathcal{N}$.
- $V = \{(i,j) \in \mathcal{A}\}$ s.t. $y_j - y_i > c_{ij}$ (violated dual constraints).

Different algorithms with different worst-case time complexity are obtained from different implementations of the *Select* function.

Feasibility

After initialization we have neither primal feasibility nor dual feasibility.

Primal viewpoint:
We have $\pi_i = nil$ for all $i \in \mathcal{N}$; hence no flow enters any node.

Dual viewpoint:
We have $y_j = \infty$ for all $j \in \mathcal{N} \setminus \{s\}$; hence all constraints $y_j - y_s \leq c_{sj}$ are violated.

The algorithm *maintains the CSCs* and iteratively *enforces primal and dual feasibility*.

Dual descent

After each iteration one of the dual values y_j is decreased

- from a value such that $y_j - y_i > c_{ij}$
- to a value such that $y_j - y_i = c_{ij}$

so that arc (i,j) becomes *tight* and x_{ij} enters the primal basis.

The update affects the other constraints (dual viewpoint) and arcs (primal viewpoint).

- *Case I:* before the iteration, $y_j = \infty$ and $\pi_j = nil$.
  Then arc (i,j) becomes tight and nothing else changes.
  Flow can now reach j from i (node j has been appended to the arborescence).
- *Case II:* before the iteration, $y_j \neq \infty$ and $\pi_j = k$.
  Then, j was already in the arborescence and was receiving flow from some node k along a tight arc (k,j), i.e. $y_j - y_k = c_{kj}$. After the iteration, arc (k,j) is no longer tight, i.e. $y_j - y_k < c_{kj}$ and cannot carry flow any more.
  Node j now receives flow from i and not from k.

Case I

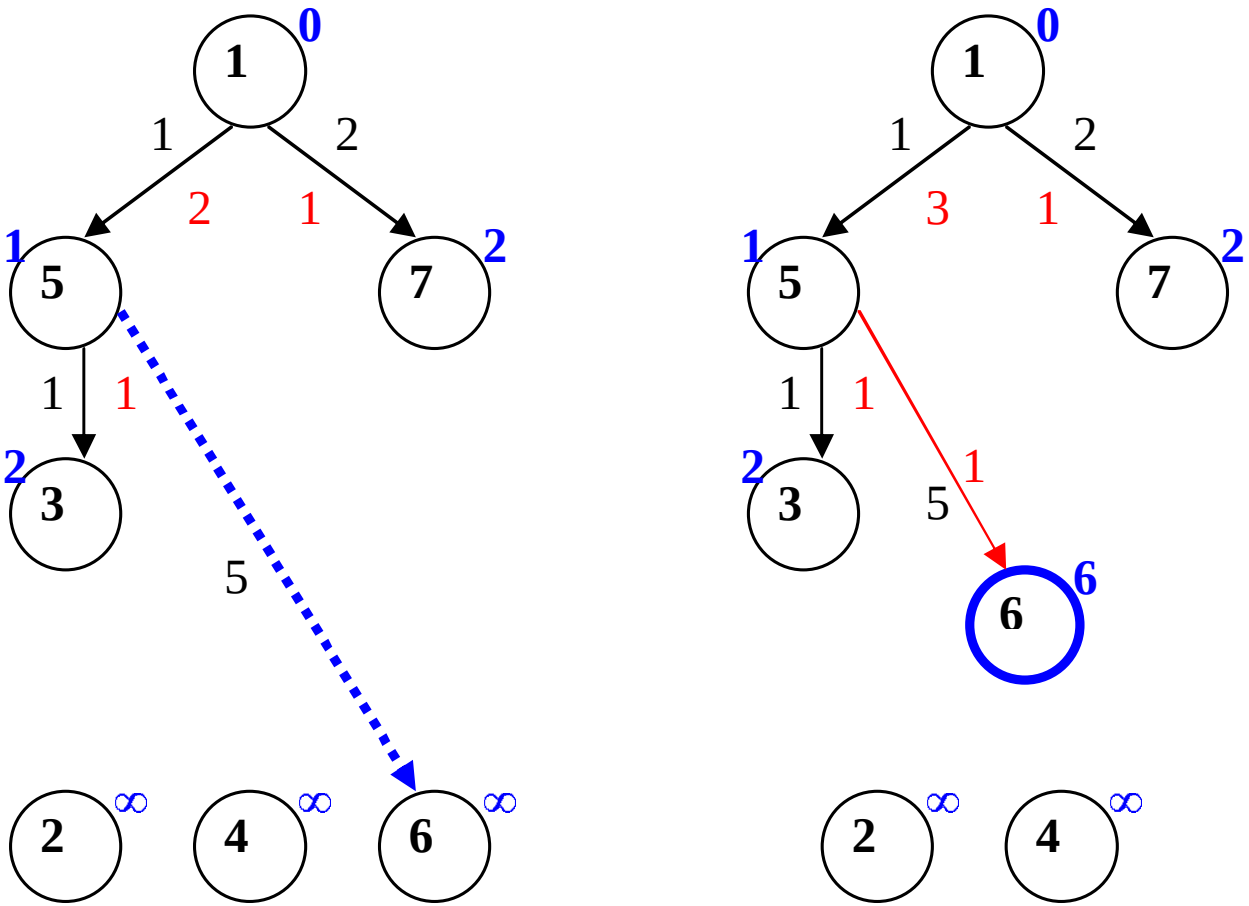

Figure: Arc $(5, 6)$ becomes tight and y_6 takes a finite value.

Case II

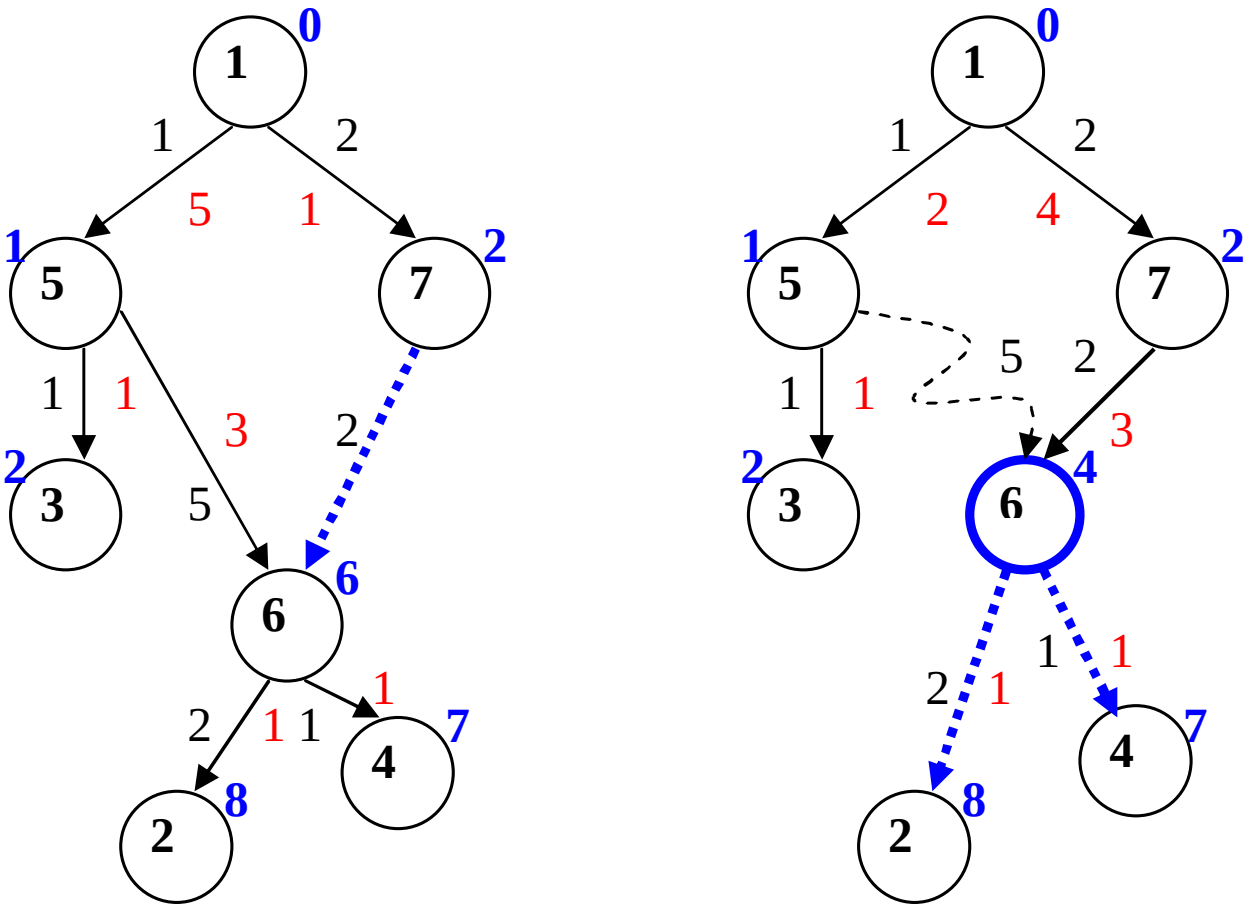

Figure: Arc $(7, 6)$ replaces arc $(5, 6)$. Arcs $(6, 2)$ and $(6, 4)$ become infeasible again.

A mechanical analogy

Assume we have n balls to be linked together by a set of m strings of given lengths. Let ball s to be fixed at the ceiling and let y_i be the distance of ball i from the ceiling ($y_s = 0$).

Initially all balls but s are put on the floor, very far from the ceiling ($y_i = \infty$), and they are not connected to the ceiling ($\pi_i = nil$).

Iteratively take a string (i, j) and append ball j to ball i. Select one for which the distance between the balls exceeds the length of the string (so you will never select i on the floor).
In doing this you can either link a ball j that was on the floor to a ball i hanging from above (Case I) or pull up a ball j already hanging from the ceiling by connecting it to a ball i over it (Case II).

When all strings have been used (all dual constraints have been enforced), there are $n - 1$ tight strings (the spanning arborescence).

This analogy holds for the case in which $c_{ij} \geq 0 \ \ \forall (i, j) \in \mathcal{A}$.

Termination

The algorithm always achieves primal and dual feasibility, but two special cases may occur.

Infeasibility. If there is a node $t \in \mathcal{N}$ not reachable from s, the algorithm does not find any arc $(i, t) \in \mathcal{A}$ corresponding to a violated dual constraint. Hence y_t remains equal to ∞; no arc entering t becomes tight; no flow can reach t: *the primal problem is infeasible* and *the dual problem is unbounded*.

Unboundedness. If there is a negative-cost cycle reachable from s, the algorithm keeps finding a violated dual constraint corresponding to one of the arcs in the cycle. Hence the algorithm enters a never-ending loop in which the y values of the nodes in the cycle are decreased to $-\infty$ and it never finds a feasible dual solution: *the dual problem is infeasible* and *the primal problem is unbounded*.

The two things can also happen independently: *both problems are infeasible*.

Bellman-Ford algorithm (1956,1958)

for $i = 1, \ldots, n$ **do**
  $y[i] \leftarrow c(s, i)$
  $\pi[i] \leftarrow s$
for $k = 1, \ldots, n-1$ **do**
  for $(i, j) \in \mathcal{A}$ **do**
    if $(y[i] + c(i, j) < y[j])$ **then**
      $\pi[j] \leftarrow i$
      $y[j] \leftarrow y[i] + c(i, j)$

The time complexity is $O(nm)$ because it requires $O(n)$ iterations, each one with complexity $O(m)$.

Moore algorithm (1959)

for $i = 1, \ldots, n$ **do**
  $y[i] \leftarrow c(s, i)$
  $\pi[i] \leftarrow s$
$Q \leftarrow \{s\}$
while $Q \neq \emptyset$ **do**
  $Extract(Q, i)$
  for $(i, j) \in \delta^+(i)$ **do**
    if $(y[i] + c(i, j) < y[j])$ **then**
      $y[j] \leftarrow y[i] + c(i, j)$
      $\pi[j] \leftarrow i$
      if $j \notin Q$ **then**
        $Insert(Q, j)$

The worst-case time complexity is still $O(nm)$ but in practice it runs faster than Bellman-Ford, because many operations are skipped.

Moore algorithm (1959)

The performance of Moore's algorithm (also called SPFA, for Shortest Path Faster Algorithm) depends on how Q is implemented.

- Nodes are not ordered in Q.
  Extract and *Insert* take $O(1)$; the complexity remains $O(mn)$.
  No queue is needed; just a binary flag for each node.
- Nodes are sorted according to their value of y.
  A priority queue is used: *Insert* and *Extract* take $O(\log n)$, they are executed at most $n-1$ times for each node: they contribute $O(n^2 \log n)$ to the complexity.
- An approximate order is given to the nodes, using a list.
  Extract always extracts the head of the list in $O(1)$.
  Three *Insert* policies have been tried in practice:
  - *FIFO*: always inserts j at the end of the list (queue) in $O(1)$.
  - *Small Label First*: if $y(j) < y(First(Q))$, then j is inserted as the first element of Q, otherwise as the last one, in $O(1)$.
  - *Large Label Last*: let $\overline{q}$ be the average of the values in Q (it can be updated in $O(1)$ after each operation on Q); all elements larger than $\overline{q}$ are moved at the end of Q in $O(n)$.

Dijkstra's algorithm (1959)

$T \leftarrow \emptyset$
for $i \in \mathcal{N}$ **do**
  $y(i) \leftarrow c(s,i)$
  $\pi(i) \leftarrow s$
  $f(i) \leftarrow (i = s)$
for $k = 1, \ldots, n-1$ **do**
  $i^* \leftarrow \text{argmin}_{i \in \mathcal{N}: \neg f(i)} \{y(i)\}$
  $T \leftarrow T \cup \{(\pi(i^*), i^*)\}$
  $f(i^*) \leftarrow \mathit{true}$
  for $i \in \mathcal{N}$ **do**
    if $(\neg f(i)) \wedge (y(i^*) + c(i^*, i) < y(i))$ **then**
      $\pi(i) \leftarrow i^*$
      $y(i) \leftarrow y(i^*) + c(i^*, i)$

The time complexity is $O(n^2)$ (improvable). It requires $c \geq 0$.

Dijkstra algorithm (dual ascent)

When $c \geq 0$, Dijkstra algorithm can be revisited as a dual ascent algorithm.

Assume to represent the graph as a set of stars (lists of outgoing arcs).

We introduce two node sets:

- O: set of nodes for which a path from s has been found, but the labels π and y are not permanent:

$$y(i) \leq d(s,i) \quad \pi(i) \neq nil \quad \forall i \in O$$

- E: set of nodes with permanent labels:

$$y(i) = d(s,i) \quad y(i) = y(\pi(i)) + c(\pi(i), i) \quad \forall i \in E$$

where $d(s,i)$ is the cost of a shortest path from s to i $\forall i \in N$.

Dijkstra algorithm (dual ascent)

All dual variables y are initialized at 0.
This corresponds to a feasible dual solution.

All primal variables π are not permanent.
This corresponds to an infeasible primal solution.

The algorithm iteratively selects a node whose corresponding y and π are made permanent.

The selected node is used to update y and π for other nodes, keeping dual feasibility and keeping C.S.C. satisfied.

The values of y are non-decreasing (dual ascent procedure).

The algorithm terminates when all labels are permanent (or when t enters E if we are computing an $s - t$ shortest path).

Dijkstra algorithm (dual ascent)

$O \leftarrow \{s\}; \quad E \leftarrow \emptyset; \quad w \leftarrow 0; \quad y(s) \leftarrow 0; \quad \pi(s) \leftarrow s$
while $(O \neq \emptyset) \wedge (t \notin E)$ **do**
  $j \leftarrow \text{argmin}_{v \in O}\{c(\pi(v), v) - y(v) + y(\pi(v))\}$
  $\theta \leftarrow c(\pi(j), j) - y(j) + y(\pi(j))$
  $O \leftarrow O \setminus \{j\}; \quad E \leftarrow E \cup \{j\}; \quad w \leftarrow w + \theta; \quad y(j) \leftarrow w$
  for $k \in O$ **do**
    $y(k) \leftarrow w$
  for $(j, k) \in \delta^+(j) : k \notin E$ **do**
    if $k \in O$ **then**
      if $y(j) + c(j, k) < y(\pi(k)) + c(\pi(k), k)$ **then**
        $\pi(k) \leftarrow j$
    else
      $O \leftarrow O \cup \{k\}; \quad y(k) \leftarrow w; \quad \pi(k) \leftarrow j$

Correctness

Dual feasibility is guaranteed after every iteration.

The rule for selecting the next node to insert in E is equivalent to find an arc from $i \in E$ to $j \in O$ corresponding to a dual constraint with minimum slack, i.e. minimum reduced cost.

Such a dual constraint becomes active (the corresponding arc becomes tight).

The other dual constraints, not corresponding to arcs in the (E, O) cut, are not affected by the increase of the dual variables $y(i)\ \forall i \in O$.

For each node i in E, $y(i) - y(s) = d(s, i)$, and $y(i) = d(s, i)\ \forall i \in E$ because $y(s)$ remains fixed to 0.

Dijkstra algorithm

The computational complexity of the array implementation of Dijkstra algorithm is $O(n^2)$.

However, it can be improved in case of sparse graphs, using suitable data-structures, such as *heaps*.

Initialization

$H \leftarrow \emptyset$
for $i \in \mathcal{N}$ **do**
  $\pi(i) \leftarrow nil$
  if $i = s$ **then**
    $y(i) \leftarrow 0$
  else
    $y(i) \leftarrow +\infty$
  $BuildHeap(H)$

H is a min-heap of nodes, partially sorted according to their associated y value.

Dijkstra algorithm

Inizialization
while $H \neq \emptyset$ **do**
  ExtractMin(H, i, v)
  for $(i,j) \in \delta^+(i)$ **do**
    if $v + c(i,j) < y(j)$ **then**
      DecreaseKey$(j, v + c(i,j), H)$
      $y(j) \leftarrow v + c(i,j)$
      $\pi(j) \leftarrow i$

Here $\delta^+(i)$ indicates the set of arcs outgoing from i, while π and y are the primal and dual variables.

Complexity

- BuildHeap is called once and has $O(n)$ complexity.
- DecreaseKey is called $O(m)$ times (each arc is used only once).
- ExtractMin is called $O(n)$ times (the heap includes only n nodes).

The latter two sub-routines have complexity $O(\log n)$ if the values of non-permanent labels are stored in a binary heap.

Therefore the overall complexity of Dijkstra algorithm implemented in this way is $O(m \log n)$.

d-heaps

Dijkstra algorithm with a d-heap:

- each MoveDn requirs $O(\log_d n)$ executions of Swap.
- the selection of the min cost successor node requires $O(d)$.

In Dijkstra algorithm this occurs up to $O(\log_d n)$ times for each call of ExtractMin and ExtractMin is called $O(n)$ times.

- BuildHeap is called once and its complexity is $O(n)$ (same as binary heaps).
- DecreaseKey is called $O(m)$ times and its complexity is $O(\log_d n)$.
- ExtractMin is called $O(n)$ times and its complexity is $O(d \log_d n)$.

Complexity: $O(nd \log_d n + m \log_d n)$.

d-heaps

The resulting complexity $O(nd \log_d n + m \log_d n)$ depends on d.

Best choice: $d = \lfloor m/n \rfloor$, yielding complexity $O(m \log_{m/n} n) = O(m \frac{\log_n n}{\log_n \frac{m}{n}}) = O(m \frac{1}{\log_n m - 1})$.

Assuming $m = \Omega(n^\epsilon)$ for any fixed $\epsilon > 1$, the complexity is $O(m \frac{1}{\epsilon - 1}) = O(m)$.

The complexity is linear in m for very mild hypothesis on the density of the digraph.

Fibonacci heaps

Using a Fibonacci heap instead of a binary heap:

- BuildHeap is called once and its complexity is $O(n)$.
- DecreaseKey is called $O(m)$ times and its complexity is $O(1)$.
- ExtractMin is called $O(n)$ times and its complexity is $O(\log n)$.

Therefore the overall complexity of Dijkstra algorithm in this implementation is $O(m + n \log n)$.

Data-dependent data-structures

Bucket: array of sets that uses the key values as indices.

It requires two assumptions:

- all values are integer;
- all values are bounded by a known constant C.

In Dijkstra algorithm all the values of non-permanent labels are in the range $[0, \ldots, nC]$, where $C = \max_{(i,j) \in A}\{c_{ij}\}$.

Operations

Initialize.
Initialize an array of $nC + 1$ empty buckets, indexed by $0, 1, \ldots, nC$.
Set an index *MinValue* to 0.
Complexity: $O(nC)$.

Insert(x).
Insert x into *Bucket*[*key*(x)].
Complexity: $O(1)$.

Operations

DecreaseKey(x, v)**.**
Extract x from $Bucket[key(x)]$ and insert it into $Bucket[v]$.
Complexity: $O(1)$.

*ExtractMin***.**
Increase *MinValue* iteratively until a non-empty bucket is found in position p.
Remove an element from $Bucket[p]$.
Complexity: $O(nC)$.
Amortized complexity for all *ExtractMin* operations is $O(C)$ (i.e. $O(1)$), because $n - 1$ iterations are done and *MinValue* never decreases.

Buckets

Dijkstra algorithm using buckets (Dial implementation):

- *Insert* is called $O(n)$ times and its complexity is $O(1)$.
- *DecreaseKey* is called $O(m)$ times and its complexity is $O(1)$.
- *ExtractMin* is called $O(n)$ times and its amortized complexity is $O(C)$.

Complexity: $O(m + nC)$.
This implementation has *pseudo-polynomial* complexity.

Radix heap

A radix heap is made by $1 + \lfloor \log_2(C) \rfloor$ buckets.
Bucket $k = 0$ contains 1 key value;
each bucket $k \geq 1$ contains 2^{k-1} key values, from 2^{k-1} to $2^k - 1$.
An array $\lambda[k]$ indicates the minimum key value in each bucket k.

Initialization.
Allocate the array in $O(\log_2(C))$.
Set $\lambda[0] = 1$; set $\lambda[k] = 2^{k-1}$ in $O(1)$ $\forall k \geq 1$.
Set *MinValue* to 0 in $O(1)$.
Complexity: $O(\log_2(C))$.

Insert(x) (initially).
Insert element x with $key(x) = v$ in bucket $k = 1 + \lfloor \log_2(v) \rfloor$.
Complexity: $O(1)$ for each inserted element.

Radix heap

DecreaseKey(x, v)**.**
Test each bucket k' from k down to 1 until $\lambda[k'] \leq v$ is found in $O(\log_2(C))$.
Extract x from its bucket k and insert it into bucket k' in $O(1)$.
Complexity: $O(\log_2(C))$ (amortized).

*ExtractMin***.**
Starting at *MinValue*, scan the heap until a non-empty bucket k is found.
If $k \geq 2$, replace bucket k by k buckets of size $1, 1, 2, 4, \ldots, 2^{k-2}$.
Set $\lambda[0] = \lambda[k]$ and $\lambda[k'] = \lambda[0] + 2^{k'-1} \ \ \forall k' = 1, \ldots, k-1$ in $O(\log_2(C))$.
For each element of bucket k, find its new bucket $k' < k$ in $O(\log_2(C))$ and insert it.
Complexity: $O(\log_2(C))$ (amortized).

Whenever an element is moved by *DecreaseKey* or *ExtractMin*, it always goes *down* the list of $\log_2(C)$ buckets.

Radix heap

Dijkstra algorithm with a radix heap:

- *Insert* is initially called $O(n)$ times and its complexity is $O(1)$.
- *DecreaseKey* is called $O(m)$ times and its complexity is $O(1)$ for each execution plus the time to move the elements which takes $O(n \log_2(C))$ overall.
- *ExtractMin* is called $O(n)$ times and its amortized complexity is $O(\log_2(C))$.

Complexity: $O(m + n \log_2(C))$.

This implementation has *polynomial* complexity in the input size.

Dijkstra algorithm implementations

Data structure	*Insert*	*DecreaseKey*	*ExtractMin*	Total complexity
Basic	$O(1)$	$O(1)$	$O(n)$	$O(n^2)$
Binary heap	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(m \log n)$
d-heap	$O(\log_d n)$	$O(\log_d n)$	$O(d \log_d n)$	$O(m \log_{m/n} n)$
Buckets	$O(1)$	$O(1)$	$O(nC)_T$	$O(m + nC)$
Radix heap	$O(n \log(C))_T$	$O(n \log(C))_T$	$O(n \log(C))_T$	$O(m + n \log(C))$
Fib. heap	$O(m)_T$	$O(m)_T$	$O(n \log n)_T$	$O(m + n \log n)$

Improved priority queue: $O(m \log \log C)$.

Radix + Fibonacci heaps: $O(m + \sqrt{\log C})$.