Chapter 3: Greedy Method

Chin Lung Lu

Department of Computer Science

National Tsing Hua University

Greedy method

Basic idea of greedy method:

Make a sequence of locally optimal decisions, which finally produces a globally optimal solution.

- Actually, only a few optimization problems can be solved by this greedy method.
- For many problems, however, the greedy method is still useful because it can quickly produce an acceptable solution.

Greedy method (cont'd)

Problem 1:

Given a set of n numbers, pick out k numbers such that the sum of these k numbers is the largest.

Exhausted method of Problem 1:

Test all possible ways of picking k numbers from these n numbers and choose the one with the largest sum.

- The time complexity of this method is $\mathcal{O}(\binom{n}{k}) = \mathcal{O}(n^k)$.

Greedy method (cont'd)

Greedy method of Problem 1:

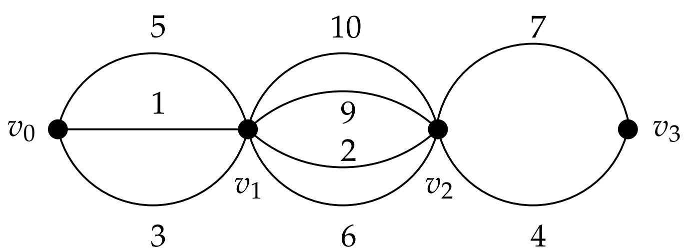
/* Let L be the input */

1. **for** $i = 1$ to k **do**
2. $\quad a[i] =$ the largest number of L;
3. $\quad L = L \setminus a[i]$;
4. **end for**
5. Output $a[1], a[2], \cdots, a[k]$;

- The time complexity of the greedy method is $\mathcal{O}(kn)$.

Greedy method (cont'd)

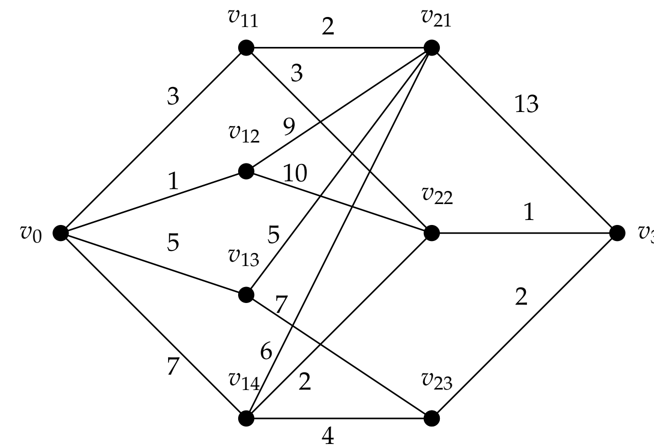
Problem 2:
Find a shortest path from v_0 to v_3 in the following graph.



- Exhausted method: test all possible paths and then choose the smallest one
- Greedy method: find a shortest path between v_i and v_{i+1} for $i = 0$ to 2.

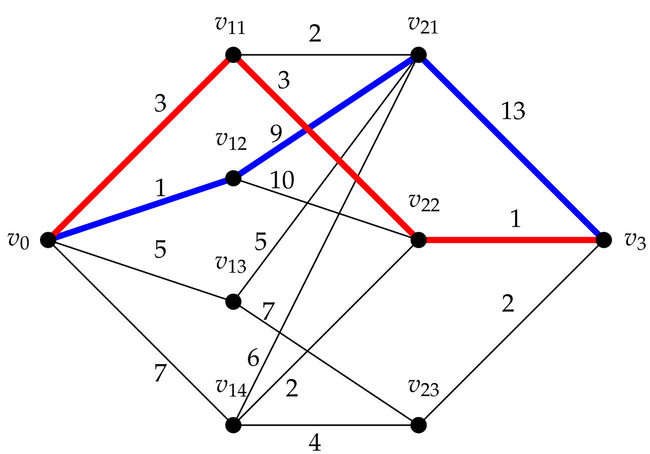
Greedy method (cont'd)

Problem 3:
Find a shortest path from v_0 to v_3 in the following graph.



Greedy method (cont'd)

- Greedy solution: $v_0 \to v_{12} \to v_{21} \to v_3$ with length 23.
- Optimal solution: $v_0 \to v_{11} \to v_{22} \to v_3$ with length 7.

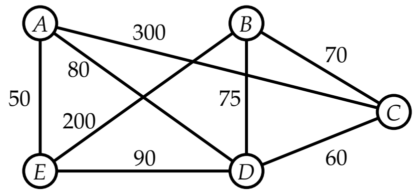


Minimum spanning tree

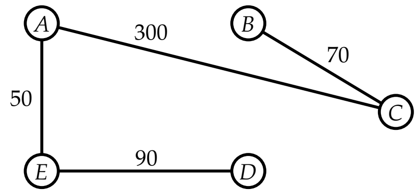
Definition:

- Let $G = (V, E)$ denote an edge-weighted connected undirected graph, where V is the set of vertices and E is the set of edges.
- A spanning tree of G is a connected undirected tree $S = (V, T)$, where $T \subseteq E$ and $|T| = |V| - 1$.
- The total weight of a spanning tree $S = (V, T)$ is the sum of all edge weights of T.
- A minimum spanning tree of G is a spanning tree of G with the smallest total weight.

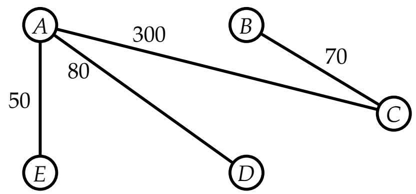
Minimum spanning tree (cont'd)



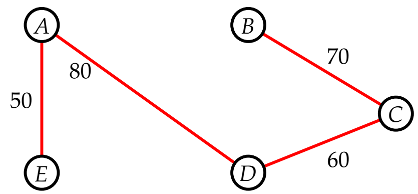
Input graph $G = (V, E)$



$S_1 = (V, T_1)$



$S_2 = (V, T_2)$



$S_3 = (V, T_3) \Rightarrow$ MST

Minimum spanning tree problem

Input:

An edge-weighted connected undirected graph $G = (V, E)$, where $|V| = n$ and $|E| = m$.

Output:

A minimum spanning tree of G, that is, a spanning tree with the minimum weight.

Minimum spanning tree problem (cont'd)

Brute force method:

Enumerate all possible spanning trees and then select the best one among them.

- There are n^{n-2} possible spanning trees for n points.
- The time complexity of the brute force methed is exponential.

Greedy methods:

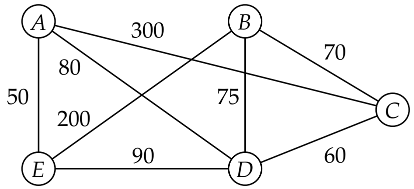
- Kruskal's algorithm: $\mathcal{O}(m \log m)$ time.
- Prim's algorithm: $\mathcal{O}(n^2)$ time.

Kruskal's algorithm

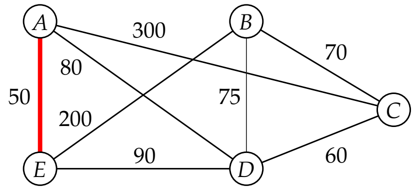
Input: A weighted, connected and undirected graph $G = (V, E)$.
Output: A minimum spanning tree $S = (V, T)$ of G.

$T = \varnothing$.
while T contains less than $n - 1$ edges **do**
    Choose e from E with the smallest weight.
    Delete e from E.
    if adding e to T does not cause a cycle in T **then**
        Add e to T.
    else
        Discard e.
    end if
end while

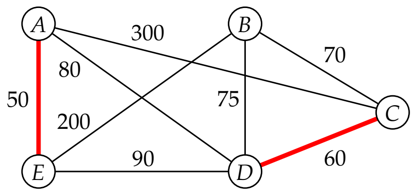
Example of Kruskal's algorithm



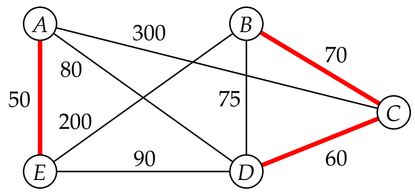
Input graph $G = (V, E)$



Step 1

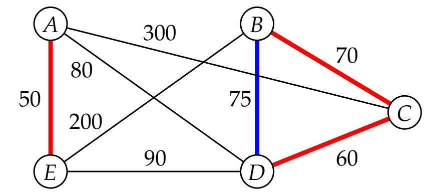


Step 2

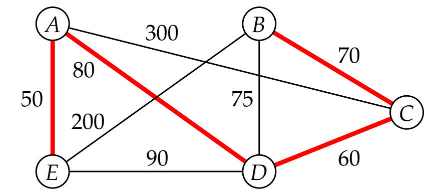


Step 3

Example of Kruskal's algorithm (cont'd)



Step 4



Step 5

Kruskal's algorithm (cont'd)

Question 1:

How to select efficiently the next edge with the smallest weight?

- We can sort all the edges in nondecreasing order of their weights (by using heap sort) and select the next edge according to this order.

Kruskal's algorithm (cont'd)

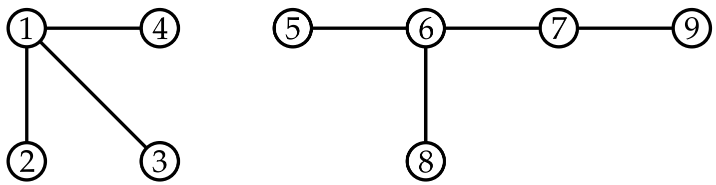
Question 2:

How to determine efficiently whether the added edge $e = (u, v)$ will form a cycle?

- During the process of Kruskal's algorithm, the partially constructed subgraph (i.e., S mentioned in the algorithm) is a spanning forest consisting of many trees.
- For our purpose, we may keep each set of vertices in a tree in an individual set.
- In this way, a cycle will be formed if u and v are in the same set.

Kruskal's algorithm (cont'd)

- For example, consider two subtrees as follows and they can be represented as $S_1 = \{1, 2, 3, 4\}$ and $S_2 = \{5, 6, 7, 8, 9\}$.



- Suppose that the next edge to be added is $(3, 4)$.
- Since both 3 and 4 are in S_1, this will cause a cycle to be formed and hence $(3, 4)$ cannot be added.
- Suppose that the next edge to be added is $(4, 8)$.
- Then no cycle will be formed (since 4 is in S_1, but 8 is in S_2) and hence $(4, 8)$ can be added.

Time complexity of Kruskal's algorithm (cont'd)

Based on the discussion above, we can see that Kruskal's algorithm is dominated by the following four actions:

(1) Sorting:

- Let $m = |E|$ and $n = |V|$.
- Sorting of all edges takes $\mathcal{O}(m \log m)$ time.

Kruskal's algorithm (cont'd)

(2) Creation of a set with one vertex:

- Initially, we need to use an operation, called the make-set operation, to create a set with just one vertex.
- Let makeset(x) denote the make-set operation that can return a set consisting of only x.
- Example Given a vertex 1, we have makeset$(1) = \{1\}$.

Kruskal's algorithm (cont'd)

(3) Finding the set containing an element:

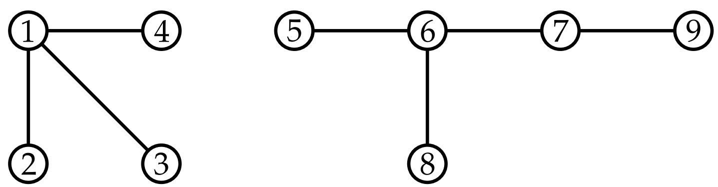
- When checking whether an edge can be added, we must check whether two vertices are in a set or not.
- In this case, we have to perform an operation, called the find operation, to determine which set contains a specific element.
- Let find(x) be the find operation that will return the set containing x.
- Example If $S_1 = \{1, 2, 3, 4\}$ and $S_2 = \{5, 6, 7, 8, 9\}$, then we have find$(4) = S_1$ and find$(8) = S_2$, indicating that 4 and 8 are not in the same set.

Kruskal's algorithm (cont'd)

(4) Union of two sets:

- When we insert an edge linking two subtrees, we are essentially performing the union of two sets.

- Example If we add edge $(4,8)$ into two trees below, then we are merging two sets $S_1 = \{1,2,3,4\}$ and $S_2 = \{5,6,7,8,9\}$ into $\{1,2,3,4,5,6,7,8,9\}$.



$S_1 = \{1,2,3,4\}$ $S_2 = \{5,6,7,8,9\}$

Kruskal's algorithm (cont')

Input: A weighted, connected and undirected graph $G = (V,E)$.
Output: A minimum spanning tree $S = (V,T)$ of G.

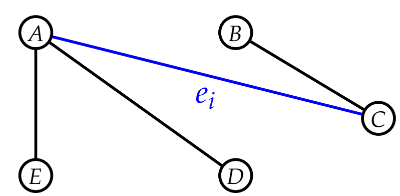
$T = \varnothing$.
for each vertex $v \in V$ **do**
    makeset(v).
end for
Sort the edges in E into nondecreasing order by their weights.
for each edge $(u,v) \in E$ in nondecreasing weight order **do**
    if find$(u) \neq$ find(v) **then**
        $T = T \cup \{(u,v)\}$.
        union(u,v).
    end if
end for

Time complexity of Kruskal's algorithm

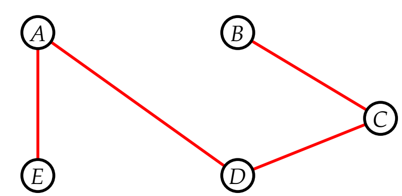
- The number of make-set operations is n.
- The number of find operations is at most $2m$.
- The number of union operations is at most $n-1$.
- Actually, it can be shown that these make-set, find and union operations take a total of $\mathcal{O}(m \cdot \alpha(m,n))$ time (see Chapter 10), where $\alpha(m,n)$ can be considered as a contant not larger than four in practical usage.
- In other words, the total time of Kruskal's algorithm is dominated by sorting, which requires $\mathcal{O}(m \log m)$ time.
- In the worst case, we have $m \leq n^2$, that is, $m = \mathcal{O}(n^2)$.
- As a result, the time complexity of Kruskal's algorithm equals $\mathcal{O}(m \log m) = \mathcal{O}(n^2 \log n)$.

Correctness of Kruskal's algorithm

- Assume that all edge weights are distinct.
- Let T be the spanning tree produced by Kruskal's algorithm.
- Let T_{opt} be the minimum spanning tree.
- Suppose that T is not the same as T_{opt}.
- Let e_i be the edge with the minimum weight in T which does not appear in T_{opt}.



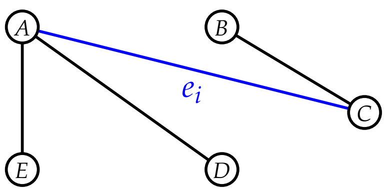
Kruskal spanning tree T



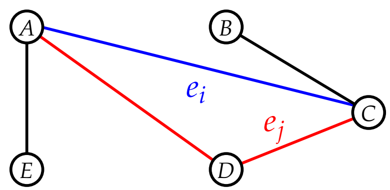
Minimum spanning tree T_{opt}

Correctness of Kruskal's algorithm (cont'd)

- Add e_i to T_{opt}, resulting in a new graph H.
- Clearly, there must be a cycle C in H.



Kruskal spanning tree T

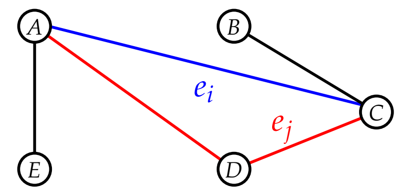


H obtained by adding e_i to T_{opt}

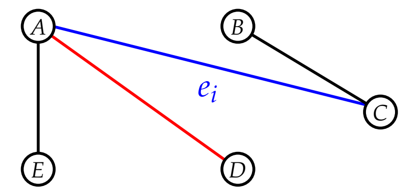
- Let e_j be an edge in C, which is not an edge in T.
- Such e_j must exist.

Correctness of Kruskal's algorithm (cont'd)

- Since $e_j \notin T$, e_j must have a larger weight than e_i according to Kruskal's algorithm.
- Removing e_j from H creates a new spanning tree whose weight is smaller than that of T_{opt}, a contradiction.



H obtained by adding e_i to T_{opt}



New spanning tree

- Hence, T is the same as T_{opt}.

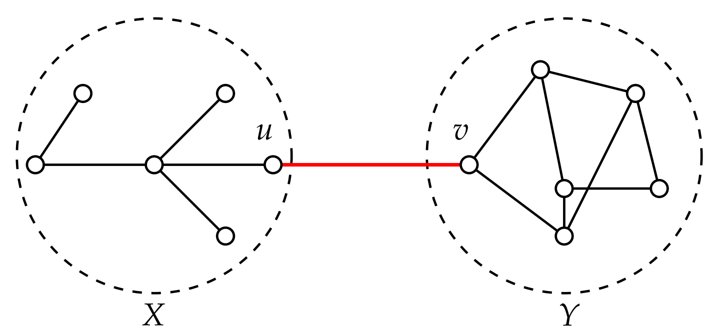
Prim's algorithm

Input: A weighted, connected and undirected graph $G = (V, E)$.
Output: A minimum spanning tree of G.

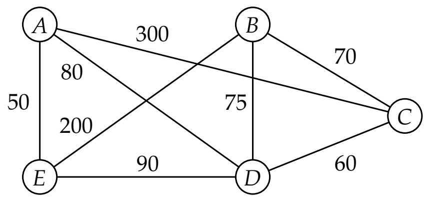
1. Let x be any vertex in V, $X = \{x\}$ and $Y = V \setminus X$.
2. Select an edge (u, v) from E such that $u \in X, v \in Y$ and (u, v) has the smallest weight among those edges between X and Y.
3. Connect u to v and let $X = X \cup \{v\}$ and $Y = Y \setminus \{v\}$.
4. **if** Y is empty **then**
5. The resulting tree is a minimum spanning tree and exit.
6. **else** go to step 2.

Prim's algorithm (cont'd)

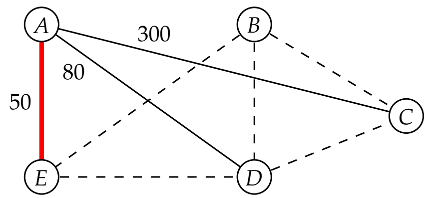
- At each step, X denotes the set of vertices contained in the partially constructed minimum spanning tree.
- Let $Y = V \setminus X$.
- The next edge (u, v) to be added is an edge between X and Y with the smallest weight.



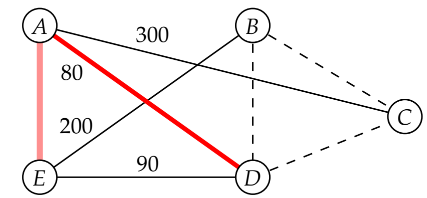
Example of Prim's algorithm



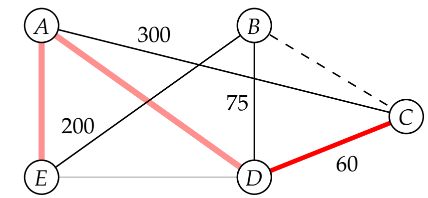
Input $G = (V, E)$ and let $x = A$



Step 1

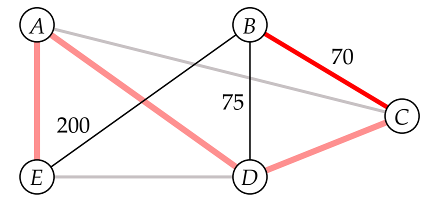


Step 2

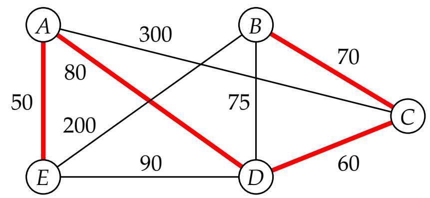


Step 3

Example of Prim's algorithm (cont'd)



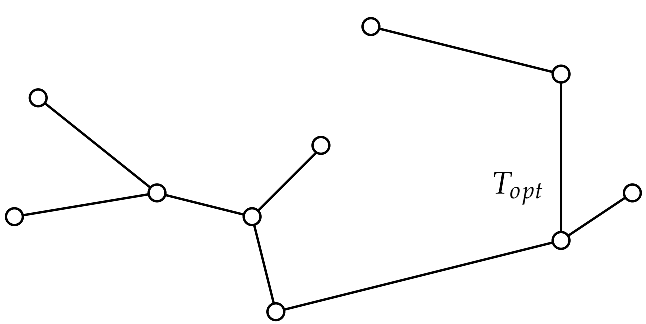
Step 4



Final output

Correctness of Prim's algorithm

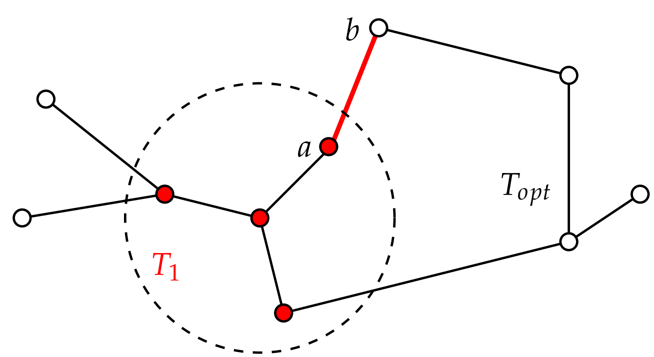
- For simplicity, we assume that the weights of all the edges in G are distict.
- Let T_{opt} be a minimum spanning tree of G.



- Let T be the spanning tree produced by Prim's algorithm.
- Suppose that $T \neq T_{opt}$.

Correctness of Prim's algorithm (cont'd)

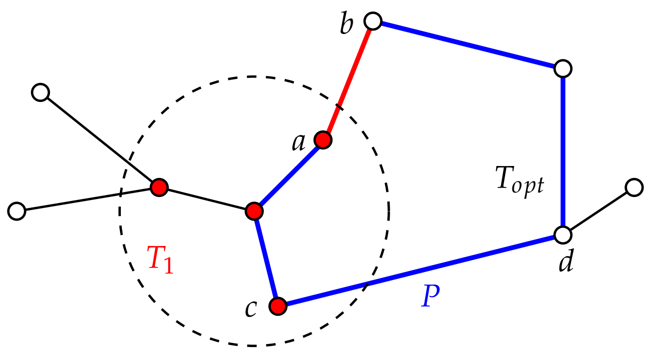
- Let (a, b) be the first edge added into T that is not in T_{opt}.
- Let T_1 be the subtree of T induced by the edges added before (a, b).



- Let V_1 be the set of vertices in T_1 and $V_2 = V \setminus V_1$.

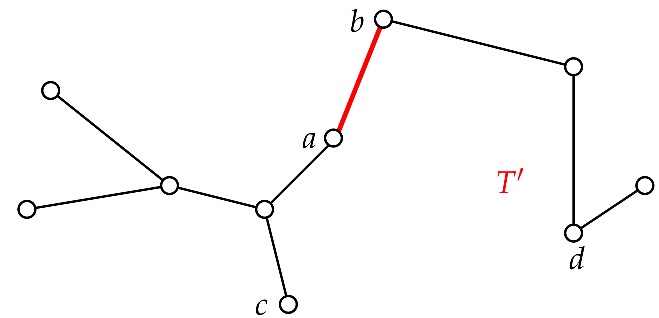
Correctness of Prim's algorithm (cont'd)

- Since T_{opt} is a spanning tree of G, there is a path P from a to b in T_{opt}.
- Let (c, d) be the edge in P such that $c \in V_1$ and $d \in V_2$.



Correctness of Prim's algorithm (cont'd)

- We have $weight(c, d) > weight(a, b)$; otherwise, (c, d) would be chosen, instead of (a, b), by Prim's algorithm.
- In this case, we can create another smaller spanning tree T' by deleting (c, d) and adding (a, b), a contradiction to that T_{opt} is a minimum one.



- In other words, we have $T = T_{opt}$.

Prim's algorithm

How to find the minimum weighted edge between X and Y?

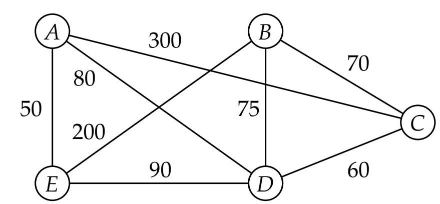
Brute force method:

Examine all the edges incident with some vertices in X and select the minimum weighted one.

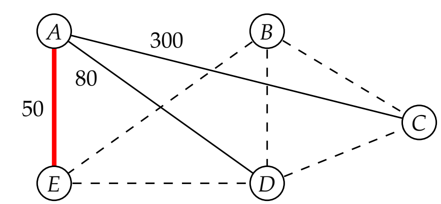
- This brute force method is not efficient because some edges are examined repeatedly.

Prim's algorithm (cont'd)

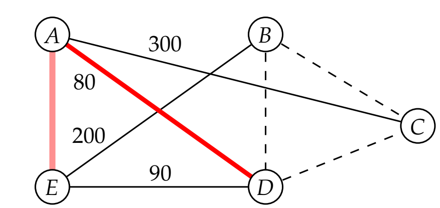
How to find the minimum weighted edge between X and Y?



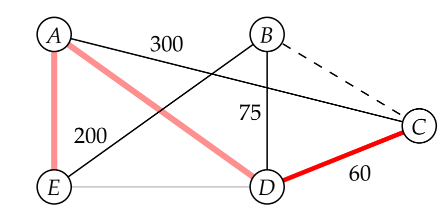
Input $G = (V, E)$ and let $x = A$



Step 1



Step 2

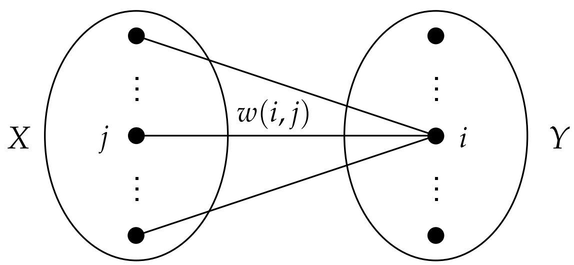


Step 3

Prim's algorithm (cont'd)

How to find the minimum weighted edge between X and Y?

- Let X be the set of vertices in the partially constructed tree in Prim's algorithm.
- Let $Y = V \setminus X$ and i be a vertex in Y.
- Among all edges incident on vertices in X and vertex i in Y, let edge (i, j) be the edge with the smallest weight.



- Let $w(i, j)$ denote the weight of edge (i, j).

Prim's algorithm (cont'd)

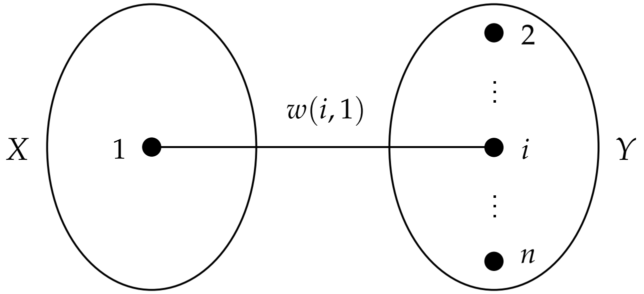
How to find the minimum weighted edge between X and Y?

- We then use vectors C_1 and C_2 to store these two information for each vertex i in Y.
- That is, at any step of Prim's algorithm, we let $C_1(i) = j$ and $C_2(i) = w(i, j)$.
- How can we utilize two vectors C_1 and C_2 to avoid repeatedly examining edges?

Prim's algorithm (cont'd)

How to find the minimum weighted edge between X and Y?

- Without losing generality, let us assume $X = \{1\}$ and $Y = \{2, 3, \ldots, n\}$ initially.
- Obviously, for each vertex i in Y, $C_1(i) = 1$ and $C_2(i) = w(i, 1)$ if edge $(i, 1)$ exists.



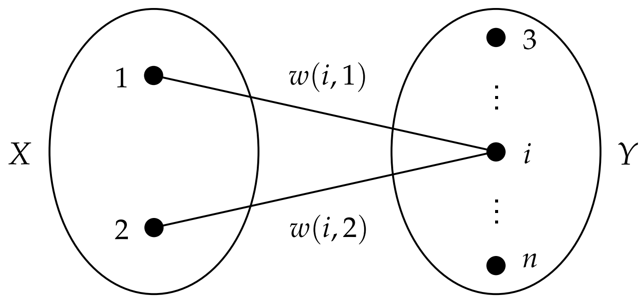
Prim's algorithm (cont'd)

- The smallest $C_2(i)$ then determines the next vertex to be added to X.
- Assume that vertex 2 is selected and added to X, that is, $X = \{1, 2\}$ and $Y = \{3, 4, \ldots, n\}$.
- Prim's algorithm requires to determine the minimum weighted edge between $X = \{1, 2\}$ and $Y = \{3, 4, \ldots, n\}$.
- But, with the help of $C_1(i)$ and $C_2(i)$, we do not need to examine edges incident on vertex 1 any more.

Prim's algorithm (cont'd)

How to find the minimum weighted edge between X and Y?

- Suppose that i is a vertex in Y.
- If $w(i,2) < C_2(i)$, where $C_2(i) = w(i,1)$, we change $C_1(i)$ from 1 to 2 and $C_2(i)$ from $w(i,1)$ to $w(i,2)$.



- If $w(i,2) \geq C_2(i)$, we do nothing.

Prim's algorithm (cont'd)

How to find the minimum weighted edge between X and Y?

- After the above updating is completed for all vertices in Y, we may choose a vertex to be added to X by examining $C_2(i)$ for all vertices i in Y.
- Again, the vertex with smallest $C_2(i)$ is the next vertex to be added.
- In this way, it can be verified that each edge is examined only once and repeatedly examining all edges is avoided.

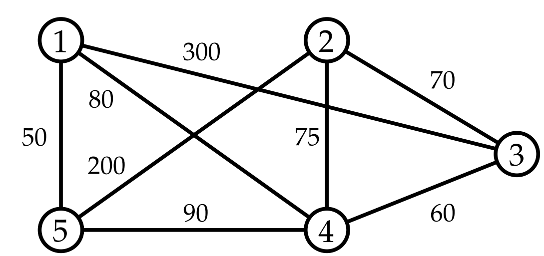
Prim's algorithm (revised)

Input: A weighted, connected and undirected graph $G = (V, E)$.
Output: A minimum spanning tree T of G.

1. Let $X = \{x\}$ and $Y = V \setminus X$, where x is any vertex in V.
2. Set $C_1(y) = x$ and $C_2(y) = \infty$ for every vertex y in Y.
3. **for** every $y \in Y$ **do**
   if $(x, y) \in E$ and $w(x, y) < C_2(y)$ **then**
   Set $C_1(y) = x$ and $C_2(y) = w(x, y)$.
   else do nothing.
4. Let y be in Y such that $C_2(y)$ is minimum and let $z = C_1(y)$.
   Connect y with edge (y, z) to z in partially constructed tree T.
   $X = X \cup \{y\}$, $Y = Y \setminus \{y\}$ and $C_2(y) = \infty$.
5. **if** Y is empty **then** output T and exit.
   else $x = y$ and go to step 3.

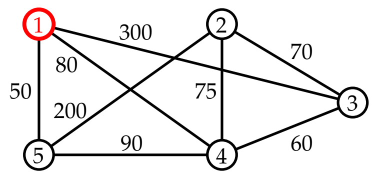
Example of Prim's algorithm

Consider the following graph:



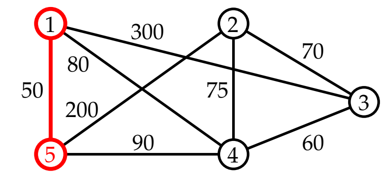
Example of Prim's algorithm (Initialization)

$$\begin{array}{ccccc} & & C_1 & & C_2 \\ x = 1. & \begin{array}{c}1\\2\\3\\4\\5\end{array} & \begin{bmatrix}1\\1\\1\\1\\1\end{bmatrix} & \begin{array}{c}X\\Y\\Y\\Y\\Y\end{array} & \begin{bmatrix}\infty\\\infty\\\infty\\\infty\\\infty\end{bmatrix}\end{array}$$



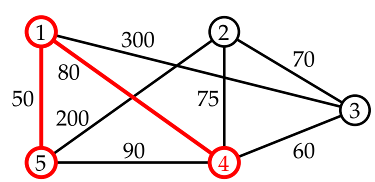
Example of Prim's algorithm (Step 1)

$$\begin{array}{ccccc} & & C_1 & & C_2 \\ x = 1. & \begin{array}{c}1\\2\\3\\4\\5\end{array} & \begin{bmatrix}1\\1\\1\\1\\1\end{bmatrix} & \begin{array}{c}X\\Y\\Y\\Y\\Y\end{array} & \begin{bmatrix}\infty\\\infty\\300\\80\\50\end{bmatrix}\end{array}$$



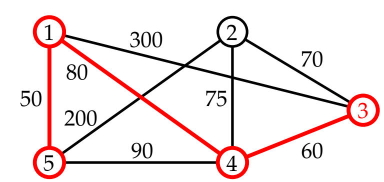
Example of Prim's algorithm (Step 2)

$$\begin{array}{ccccc} & & C_1 & & C_2 \\ x = 5. & \begin{array}{c}1\\2\\3\\4\\5\end{array} & \begin{bmatrix}1\\5\\1\\1\\1\end{bmatrix} & \begin{array}{c}X\\Y\\Y\\Y\\X\end{array} & \begin{bmatrix}\infty\\200\\300\\80\\\infty\end{bmatrix}\end{array}$$



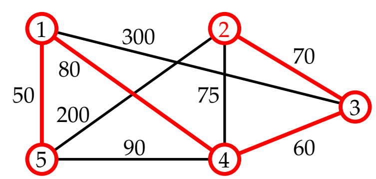
Example of Prim's algorithm (Step 3)

$$\begin{array}{ccccc} & & C_1 & & C_2 \\ x = 4. & \begin{array}{c}1\\2\\3\\4\\5\end{array} & \begin{bmatrix}1\\4\\4\\1\\1\end{bmatrix} & \begin{array}{c}X\\Y\\Y\\X\\X\end{array} & \begin{bmatrix}\infty\\75\\60\\\infty\\\infty\end{bmatrix}\end{array}$$



Example of Prim's algorithm (Step 4)

$$
x = 3. \quad
\begin{array}{c c c}
 & C_1 & \\
1 & \left[\begin{array}{c} 1 \\ 3 \\ 4 \\ 1 \\ 1 \end{array}\right] & \begin{array}{c} X \\ Y \\ X \\ X \\ X \end{array} \\
\end{array}
\begin{array}{c}
\\
2 \\ 3 \\ 4 \\ 5
\end{array}
\qquad
\begin{array}{c}
C_2 \\
\left[\begin{array}{c} \infty \\ 70 \\ \infty \\ \infty \\ \infty \end{array}\right]
\end{array}
$$



Time complexity of Prim's algorithm

- Whenever a vertex is added to the partially constructed tree T, every vertex in Y must be examined.
- Therefore, the time complexity of Prim's algorithm in the worst case is $\mathcal{O}(n^2)$, where n is the number of vertices in V.

Kruskal's algorithm vs Prim's algorithm

Question:

Kruskal's algorithm is always better than Prim's algorithm?

Answer:

- The time complexity of Prim's algorithm is $\mathcal{O}(n^2)$ and the time complexity of Kruskal's algorithm is $\mathcal{O}(m \log m)$.
- When G is a sparse graph (i.e., $m \approx n$), Kruskal's algorithm is better than Prim's algorithm.
- When G is a dense graph (i.e., $m \approx n^2$), Prim's algorithm is better than Kruskal's algorithm.

Single-source shortest path problem

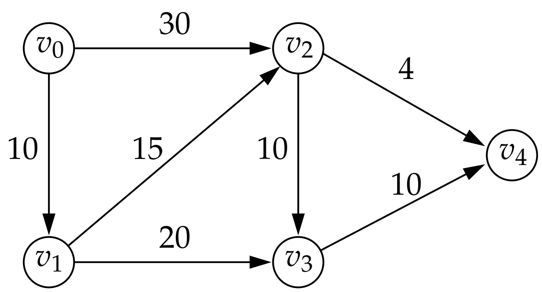
Input:

A directed, edge-weighted graph $G = (V, E)$, where each edge (u, v) has a nonnegative weight (length), denoted by $c(u, v)$, and a source vertex v_0.

Output:

Find all of the shortest paths from v_0 to all other vertices in V.

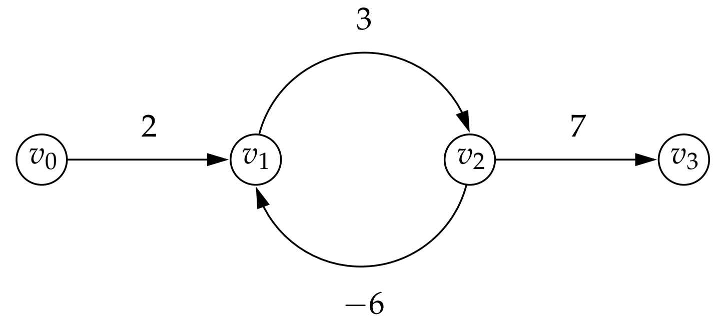
Single-source shortest path problem (cont'd)



Source	v_i	Shortest path from v_0 to v_i	Length
v_0	v_1	$v_0 \to v_1$	10
v_0	v_2	$v_0 \to v_1 \to v_2$	25
v_0	v_3	$v_0 \to v_1 \to v_3$	30
v_0	v_4	$v_0 \to v_1 \to v_2 \to v_4$	29

Single-source shortest path problem (cont'd)

- If there is a negative weight cycle on some path from v_0 to v_i, the shortest path between v_0 and v_i is not defined, because no path from v_0 to v_i can be a shortest path.



Dijkstra's method

- Dijkstra proposed a greedy algorithm for solving the single-source shortest path problem, where all edge weights are assumed to be non-negative.

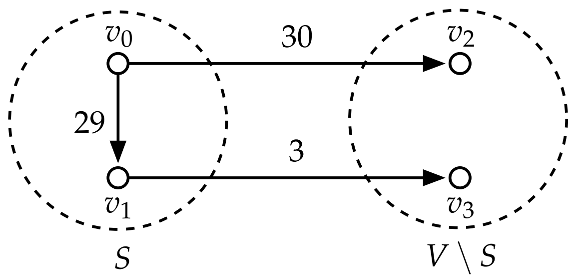
E. W. Dijkstra

Dijkstra's method (cont'd)

- Dijkstra's algorithm divides the set of vertices into two sets S and $V \setminus S$, where S contains all the i nearest neighbors which have been found in the first i steps.
- The $i+1$th step is to find the $i+1$th nearest neighbor of v_0.
- However, it does not mean that the path between v_0 and its $i+1$th nearest neighbor must pass through the ith nearest neighbor of v_0.

Dijkstra's method (cont'd)

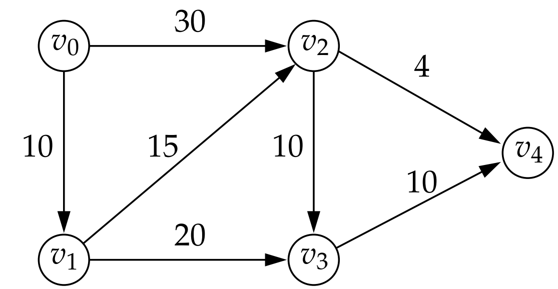
- As shown in the following figure, suppose that we have already found the 1st nearest neighbor of v_0, which is v_1.



- Clearly, in this example, v_2 is the 2nd nearest neighbor of v_0, not v_3.

Dijkstra's method (cont'd)

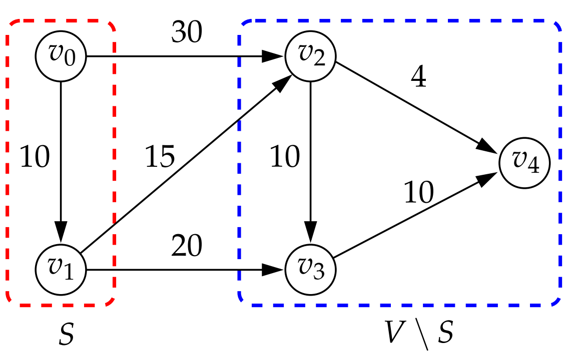
- Let $L(v_i)$ be the shortest distance from v_0 to v_i presently found (i.e., the upper bound of the shortest path from v_0 to v_i).
- For example, consider the following instance:



- In the beginning, we let $S = \{v_0\}$.
- Since v_1 and v_2 are connected to v_0, we have $L(v_1) = 10$ and $L(v_2) = 30$ (currently, $L(v_3) = L(v_4) = \infty$).

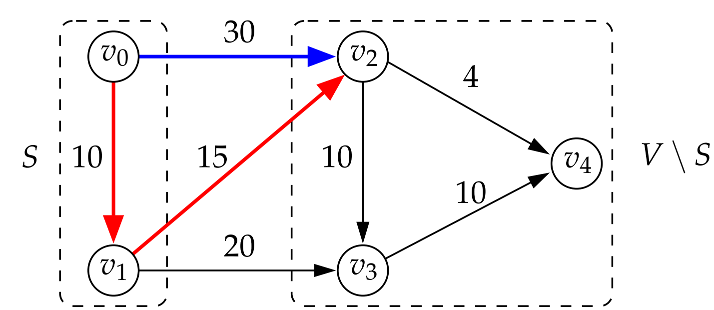
Dijkstra's method (cont'd)

- Since $L(v_1)$ is the shortest, v_1 is the first nearest neighbor of v_0.
- Let $S = \{v_0, v_1\}$.
- Now, only v_2 and v_3 are connected to some vertices in S.



Dijkstra's method (cont'd)

- For v_2, its previous $L(v_2) = 30$.
- However, after v_1 is put into S, we may change it by using the path $v_0 \to v_1 \to v_2$ whose length is $10 + 15 < 30$.



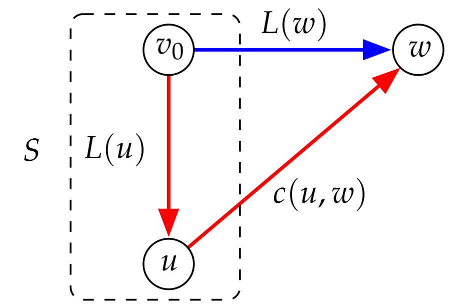
$$
\begin{aligned}
\therefore L(v_2) &= \min\{L(v_2), L(v_1) + c(v_1, v_2)\} \\
&= \min\{30, 10 + 15\} \\
&= 25
\end{aligned}
$$

Dijkstra's method (cont'd)

- The above discussion shows that the shortest distance from v_0 to v_2 presently found may be not short enough because of newly-added vertex v_1.
- If this situation occurs, this shortest distance must be updated.

Dijkstra's method (cont'd)

- Let u be the latest vertex added to S.
- Let $L(w)$ be the presently found shortest distance from v_0 to w



- Then $L(w)$ will need to be updated by the following formula :

$$L(w) = \min\{L(w), L(u) + c(u,w)\}$$

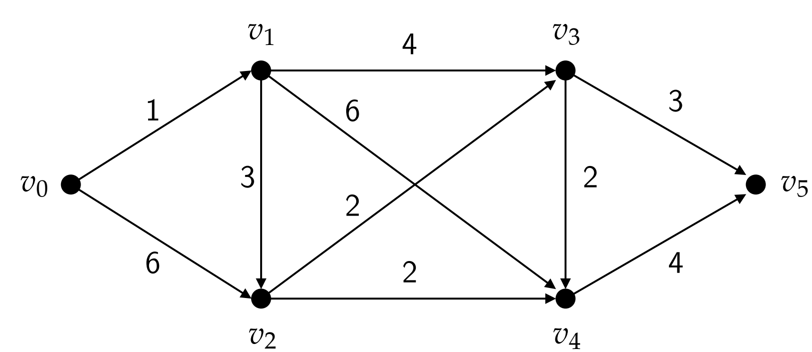
where $c(u,w)$ denotes the length of edge (u,w).

Dijkstra's algorithm

Input: A directed graph $G = (V, E)$ and a source vertex v_0.
Output: For each $v \in V$, the length of a shortest path from v_0 to v.

1. $S = \{v_0\}$ and $L(v_0) = 0$.
2. **for** $i = 1$ to n **do** /* Initialization */
   if $(v_0, v_i) \in E$ **then** $L(v_i) = c(v_0, v_i)$.
   else $L(v_i) = \infty$.
   end for
3. **for** $i = 1$ to n **do** /* Find ith nearest neighbor of v_0 */
   Choose $u \in V \setminus S$ such that $L(u)$ is the smallest.
   $S = S \cup \{u\}$.
   for all $w \in V \setminus S$ **do**
   $L(w) = \min\{L(w), L(u) + c(u,w)\}$. /* Relaxation */
   end for
   end for

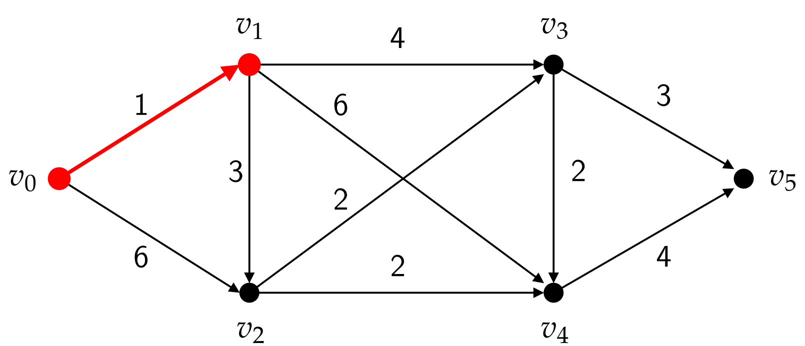
Example of Dijkstra's algorithm



Example of Dijkstra's algorithm (cont'd)

Step 1:

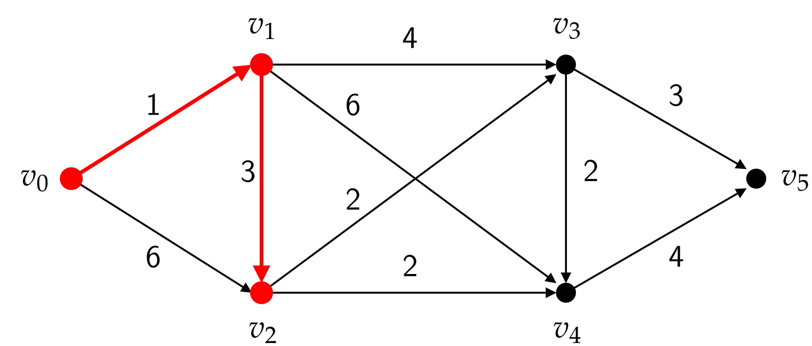
- We have $S = \{v_0\}$, and $L(v_1) = 1$, $L(v_2) = 6$, $L(v_3) = \infty$, $L(v_4) = \infty$ and $L(v_5) = \infty$.
- Since $L(v_1)$ is the smallest, v_1 is the 1st neighbor of v_0.



Example of Dijkstra's algorithm (cont'd)

Step 2:

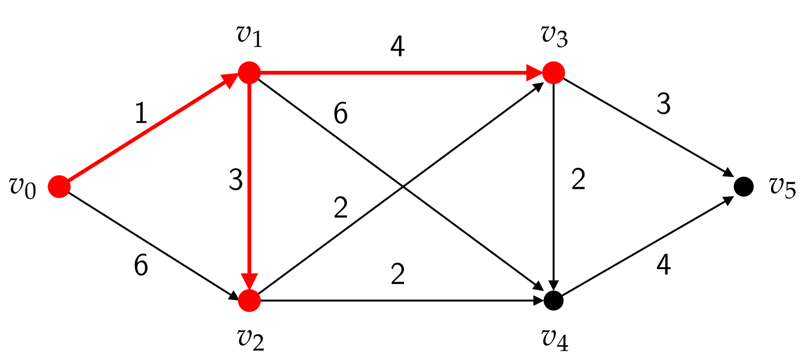
- We have $S = \{v_0, v_1\}$, and $L(v_2) = 4$, $L(v_3) = 5$, $L(v_4) = 7$ and $L(v_5) = \infty$.
- Since $L(v_2)$ is the smallest, v_2 is the 2nd neighbor of v_0.



Example of Dijkstra's algorithm (cont'd)

Step 3:

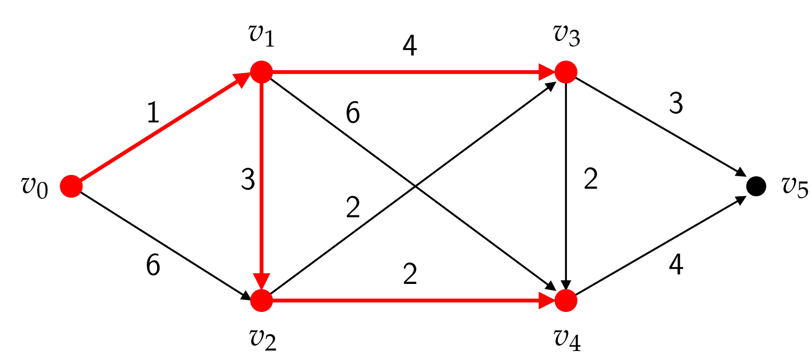
- We have $S = \{v_0, v_1, v_2\}$, and $L(v_3) = 5$, $L(v_4) = 6$ and $L(v_5) = \infty$.
- Since $L(v_3)$ is the smallest, v_3 is the 3rd neighbor of v_0.



Example of Dijkstra's algorithm (cont'd)

Step 4:

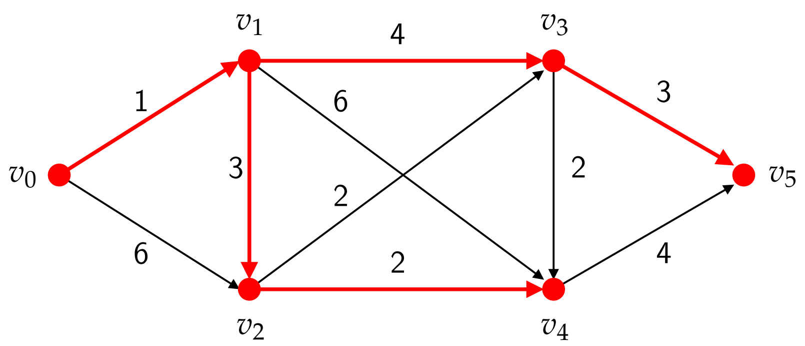
- We have $S = \{v_0, v_1, v_2, v_3\}$, and $L(v_4) = 6$ and $L(v_5) = 8$.
- Since $L(v_4)$ is the smallest, v_4 is the 4th neighbor of v_0.



Example of Dijkstra's algorithm (cont'd)

Step 5:

- We have $S = \{v_0, v_1, v_2, v_3, v_4\}$ and $L(v_5) = 8$.
- Since $L(v_5)$ is the smallest, v_5 is the 5th neighbor of v_0.



Correctness of Dijkstra's algorithm

- Suppose we are given a weighted, directed graph $G = (V, E)$ with weight function $c : E \to R$ mapping edges to real-valued weights.
- Weight $c(p)$ of path $p = (v_0, v_1, \ldots, v_k)$: the sum of the weights of its constituent edges

$$c(p) = \sum_{i=1}^{k} c(v_{i-1}, v_i).$$

- Shortest-path weight $\delta(u, v)$ from u to v:

$$\delta(u,v) = \begin{cases} \min\{c(p) : u \overset{p}{\rightsquigarrow} v\} & \text{if } \exists u \overset{p}{\rightsquigarrow} v \\ \infty & \text{otherwise} \end{cases}$$

Optimal substructure property

Lemma 1: (Subpaths of shortest paths are shortest paths)

- Given a directed graph $G = (V, E)$ with edge weight function $c : E \to R$, let $p = (v_0, v_1, \ldots, v_k)$ be a shortest path from v_0 to v_k.
- For any i and j with $0 \le i \le j \le k$, let $p_{ij} = (v_i, v_{i+1}, \ldots, v_j)$ be the subpath of p from v_i to v_j.
- Then, p_{ij} is a shortest path from v_i to v_j.

Optimal substructure property (cont'd)

Proof of Lemma 1

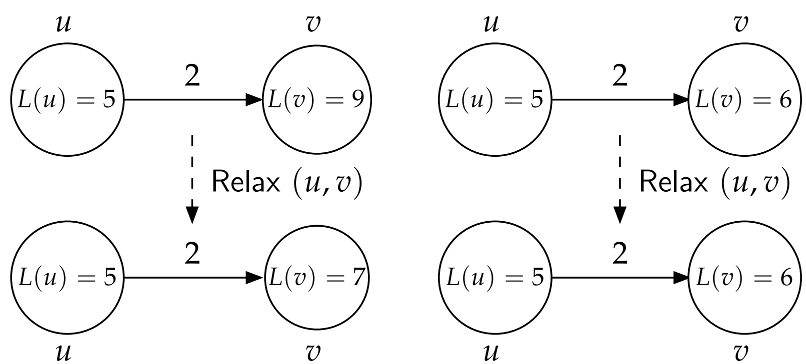
- Recall that $p = v_0 \to v_1 \rightsquigarrow v_{i-1} \to v_i \overset{p_{ij}}{\rightsquigarrow} v_j \to v_{j+1} \rightsquigarrow v_k$.
- Suppose that we decompose path p into three subpaths:

$$v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$$

- Then we have $c(p) = c(p_{0i}) + c(p_{ij}) + c(p_{jk})$.
- Assume that there is a path p'_{ij} from v_i to v_j with weight $c(p'_{ij}) < c(p_{ij})$.
- Then $v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{p'_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$ is a path from v_0 to v_k whose weight $c(p_{0i}) + c(p'_{ij}) + c(p_{jk})$ is less than $w(p)$, a contradiction.

Edge relaxation (for tightening $L(v)$)

Definition:
Relaxing an edge (u,v) is to test whether we can improve the upper bound of the shortest path to v found so far by going through u and, if so, update $L(v)$ using $L(v) = \min\{L(v), L(u) + c(u,v)\}$.



Correctness of Dijkstra's algorithm

Lemma 2:
For each $u \in V$, $L(u) = \delta(v_0, u)$ at the time when u is added to S.

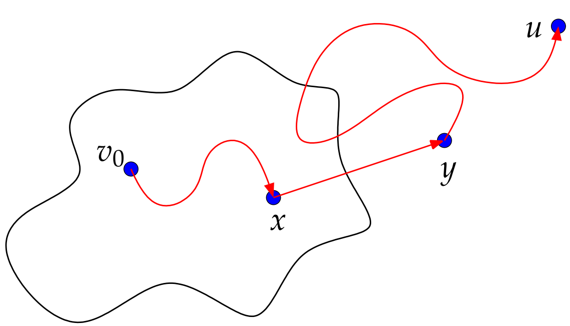
Proof:

- Recall that $\delta(v_0, u)$ is the distance of the shortest path from v_0 to u.
- Suppose that u is the first vertex for which $L(u) \neq \delta(v_0, u)$, when u is added to S.
- It means that currently for all $x \in S$, $L(x) = \delta(v_0, x)$.
- We have $u \neq v_0$, because v_0 is the first vertex added to S and $L(v_0) = 0 = \delta(v_0, v_0)$.

Correctness of Dijkstra's algorithm (cont'd)

Proof of Lemma 2

- There must be some path from v_0 to u.
- Otherwise, we have $L(u) = \infty = \delta(v_0, u)$, a contradiction with the assumption.
- Let p be a shortest path from v_0 to u.
- Let y be the first vertex along p such that $y \in V \setminus S$ and x be the y's predecessor.



Correctness of Dijkstra's algorithm (cont'd)

Proof of Lemma 2

Claim:
$L(y) = \delta(v_0, y)$ when u is added to S.

- $v_0 \rightsquigarrow x \to y$ is the shortest path from v_0 to y (by Lemma 1).
- By assumption, we have $L(x) = \delta(v_0, x)$.
- Note that edge (x, y) is relaxed when x is added to S.
- After relaxing (x, y), we then have:

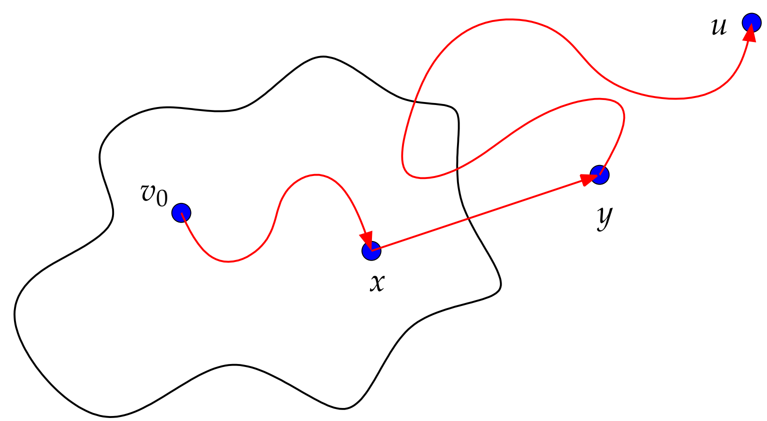
$$L[y] \leq L[x] + c(x,y) = \delta(v_0, x) + c(x,y) = \delta(v_0, y)$$

- Since $L[y]$ is an upper bound on the length of a shortest path from v_0 to y, we have $L[y] \geq \delta(v_0, y)$.
- As a result, we have $L[y] = \delta(v_0, y)$.

Correctness of Dijkstra's algorithm (cont'd)

Proof of Lemma 2

- Path p can be decomposed as $v_0 \overset{p_1}{\rightsquigarrow} x \to y \overset{p_2}{\rightsquigarrow} u$.

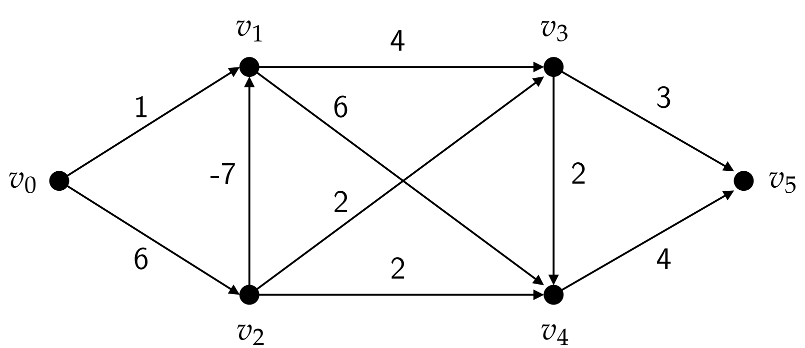


Correctness of Dijkstra's algorithm (cont'd)

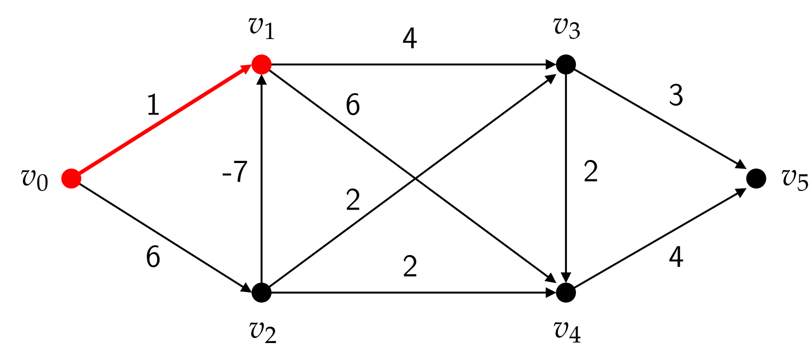
Proof of Lemma 2

- We have $L(y) = \delta(v_0, y) \le \delta(v_0, u) \le L(u)$.
- The reason is that y is in the path p from v_0 to u and all edge weights are nonnegative, and moreover $L(u)$ is an upper bound of $\delta(v_0, u)$.
- Both u and y were in $V \setminus S$ when u was chosen to be added to S, implying $L(u) \le L(y)$.
- Hence, $L(y) = \delta(v_0, y) = \delta(v_0, u) = L(u)$, which contradicts to the assumption (u is the first vertex with $L(u) \ne \delta(v_0, u)$).

Counterexample of Dijkstra's algorithm



Counterexample of Dijkstra's algorithm (cont'd)



- In the step 1, we have $S = \{v_0\}$ and $L(v_1) = 1$, $L(v_2) = 6$, $L(v_3) = \infty$, $L(v_4) = \infty$ and $L(v_5) = \infty$.
- Then v_1 will be selected as the 1st neighbor of v_0 (since $L(v_1)$ is the smallest), but the 1st shortest path is $v_0 \to v_2 \to v_1$, instead of $v_0 \to v_1$.

Time complexity of Dijkstra's algorithm

- The time complexity of Dijkstra's algorithm is $\mathcal{O}(n^2)$.

Lower bound of single-source shortest path problem:

The minimum number of steps to solve the single-source shortest path problem is $\Omega(|E|) = \Omega(n^2)$.

- Because every edge in the graph has to be examined.
- Hence, Dijkstra's algorithm is optimal.

Linear merge problem

- Two sorted lists $L_1 = (a_1, \ldots, a_{n_1})$ and $L_2 = (b_1, \ldots, b_{n_2})$, can be merged into one sorted list using the linear merge algorithm.

Linear merge algorithm:

1. $i = 1$ and $j = 1$.
2. **do**
   Compare a_i and b_j.
   if $a_i > b_j$ **then** output b_j and $j = j + 1$.
   else output a_i and $i = i + 1$.
   while $i \leq n_1$ and $j \leq n_2$
3. **if** $i > n_1$ **then** output $b_j, b_{j+1}, \ldots, b_{n_2}$.
   else output $a_i, a_{i+1}, \ldots, a_{n_1}$.

- Number of comparisons requires $n_1 + n_2 - 1$ in the worst case.

2-way merge problem

- If more than two sorted lists are to be merged, then we can still apply the above linear merge algorithm by merging two sorted lists repeatedly.
- These merging processes are called 2-way merge because each merging step only merges two sorted lists.

2-way merge problem (cont'd)

Example 1:

Suppose that we have (L_1, L_2, L_3) with sizes $(50, 30, 10)$.

1. Merge L_1 and L_2 into L_4 with $50 + 30 - 1 = 79$ comparisons.
2. Merge L_4 and L_3 into L_5 with $80 + 10 - 1 = 89$ comparisons.

- The total number of comparisons is 168.

Example 2:

Suppose that we have (L_1, L_2, L_3) with sizes $(50, 30, 10)$.

1. Merge L_2 and L_3 into L_4 with $30 + 10 - 1 = 39$ comparisons.
2. Merge L_4 and L_1 into L_5 with $40 + 50 - 1 = 89$ comparisons.

- The total number of comparisons is 128.

2-way merge problem (cont'd)

Input:

There are m sorted lists, each of which consists of n_i elements.

Output:

Find an optimal sequence of merging process to merge these m sorted lists by using the minimum number of comparisons.

2-way merge problem (cont'd)

- To simplify the discussion, we use $n_i + n_j$, instead of $n_i + n_j - 1$, as the number of comparisons needed to merge two lists with sizes n_i and n_j, respectively.

Example:

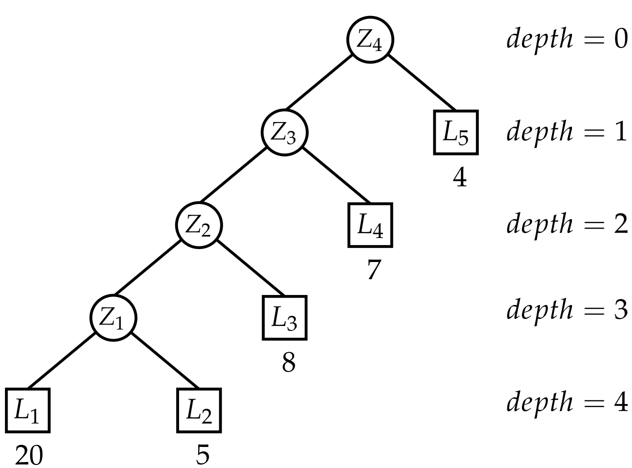
Suppose that we have $(L_1, L_2, \ldots, L_5)$ with sizes $(20, 5, 8, 7, 4)$.

1. Merge L_1 and L_2 to produce Z_1 $(20 + 5 = 25)$.
2. Merge Z_1 and L_3 to produce Z_2 $(25 + 8 = 33)$.
3. Merge Z_2 and L_4 to produce Z_3 $(33 + 7 = 40)$.
4. Merge Z_3 and L_5 to produce Z_4 $(40 + 4 = 44)$.

- Therefore, the total number of comparisons is 142.

Binary tree of merging pattern

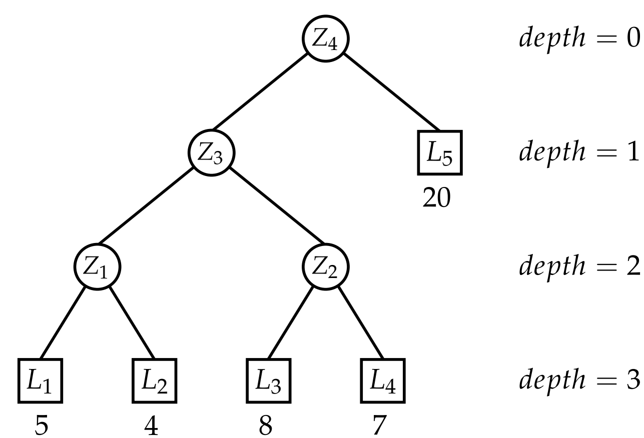
- The above merging process can be represented by a binary tree.



- The total number of comparisons is $\sum_{i=1}^{5} depth_i \times n_i = 142$.

Binary tree of merging pattern (cont'd)

- Suppose that we utilize a greedy method in which we always merge two presently shortest lists, as shown below:



- As a result, the total number of comparisons is 92.

2-way merge problem

Greedy algorithm

Input: m sorted lists $L_1, \ldots L_m$, each L_i consisting of n_i elements.
Output: An optimal 2-way merge tree.

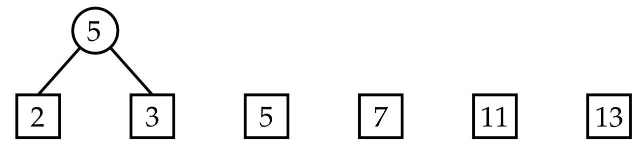
1. Generate m trees, where each tree has exactly one external node with weight n_i.
2. Choose two trees T_1 and T_2 with minimal weights.
3. Create a new tree T whose root has T_1 and T_2 as its subtrees and weight equals to $w(T_1) + w(T_2)$.
4. Replace T_1 and T_2 by T.
5. **if** there is only one tree left **then** stop **else** go to step 2.

2-way merge problem (cont'd)

An example of the greedy algorithm

Step 1:

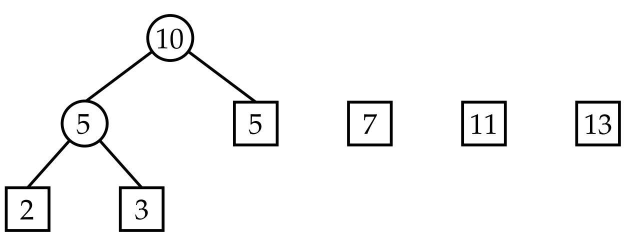
Step 2:



2-way merge problem (cont'd)

An example of the greedy algorithm

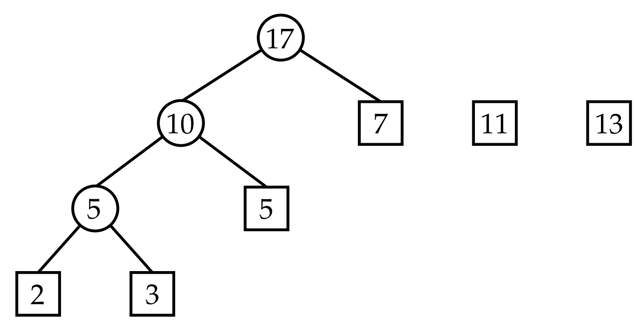
Step 3:



2-way merge problem (cont'd)

An example of the greedy algorithm

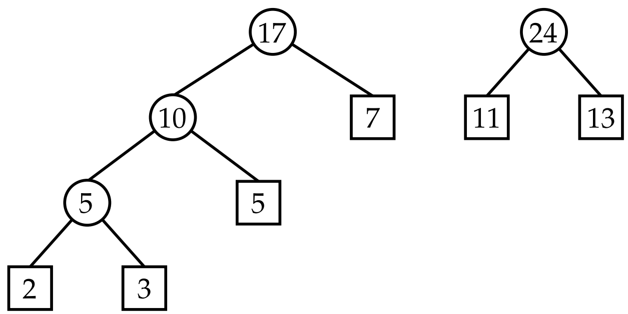
Step 4:



2-way merge problem (cont'd)

An example of the greedy algorithm

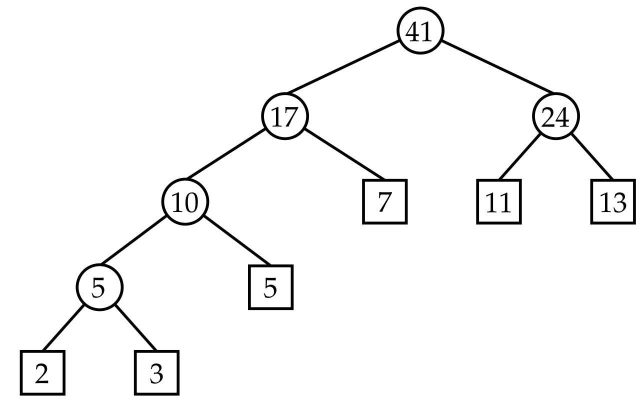
Step 5:



2-way merge problem (cont'd)

An example of the greedy algorithm

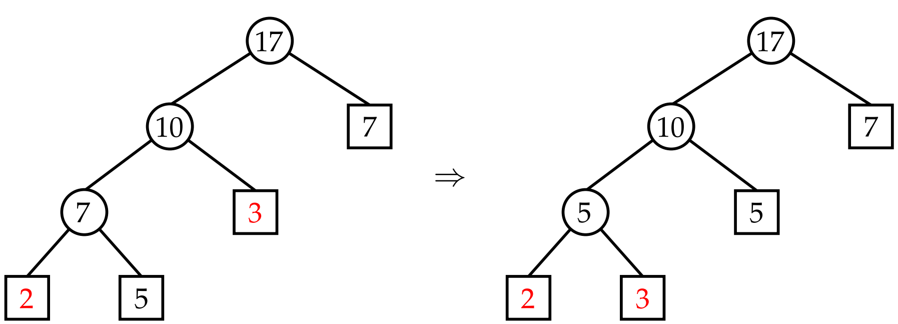
Step 6:



2-way merge problem (cont'd)

Correctness of the greedy algorithm

Observation:
There is an optimal 2-way merge tree in which the two leaf nodes with minimum sizes are assigned to be brothers (note that their parent is an internal node of maximum distance from the root).



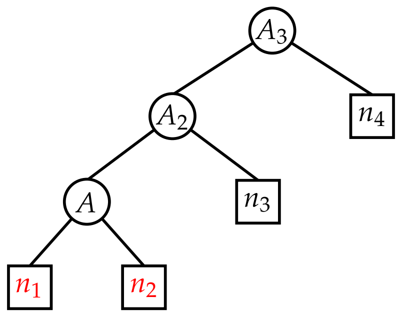
2-way merge problem (cont'd)

Correctness of the greedy algorithm

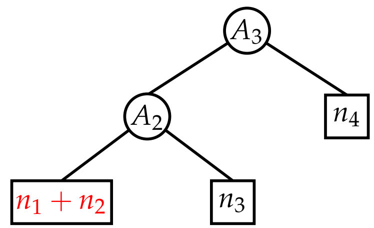
- Let T be an optimal 2-way merge tree for $L_1, L_2, \ldots, L_m$ with lengths $n_1, n_2, \ldots, n_m$ respectively in which the two lists of the shortest lengths, say L_1 and L_2, are brothers.
- Assume that $n_1 \leq n_2 \leq \ldots \leq n_m$.
- Let A be the parent of L_1 and L_2.
- Let T_1 denote the tree where A is replaced by a list with length $n_1 + n_2$.
- Let $W(X)$ denote the weight of a 2-way merge tree X.
- Then, we have $W(T) = W(T_1) + n_1 + n_2$.
- In fact, T_1 can be considered as a 2-way merge tree for $m - 1$ lists with lengths $n_1 + n_2, n_3, n_4, \ldots, n_m$, respectively.

2-way merge problem (cont'd)

Correctness of the greedy algorithm



Optimal 2-way merge tree T for $n_1, n_2, \ldots, n_m$



A 2-way merge tree T_1 for $n_1 + n_2, n_3, \ldots, n_m$

2-way merge problem (cont'd)

Correctness of the greedy algorithm

- We continue to prove the correctness of the greedy method by induction on m.
- (Basis step) The greedy algorithm produces an optimal 2-way merge tree for $m = 2$.
- (Hypothesis step) Assume that the greedy algorithm produces an optimal 2-way merge tree for $m - 1$ lists.
- (Induction step) For the instance with m lists, we combine lists L_1 and L_2 to obtain a new instance.
- Then we apply the greedy algorithm to the new instance and let the resulting tree be T_2.
- In T_2, there is a leaf node X with length $n_1 + n_2$.

2-way merge problem (cont'd)

Correctness of the greedy algorithm

- Split X of T_2 to obtain a new tree T_3 so that T_3 has two sons L_1 and L_2 with lengths n_1 and n_2, respectively.
- We have $W(T_3) = W(T_2) + n_1 + n_2$.
- If T_3 is not an optimal 2-way merge tree, then we have:

  $$W(T_3) > W(T)$$

  which implies $W(T_2) > W(T_1)$.
- However, it is impossible since T_2 is an optimal 2-way merge tree for $m - 1$ lists by the induction hypothesis.
- In other words, T_3 is an optimal 2-way merge tree.

2-way merge problem (cont'd)

Time complexity of the greedy algorithm

- For the given m numbers $n_1, n_2, \ldots, n_m$, we can construct a min-heap to represent these numbers, where the value of a node is smaller than or equal to the values of its sons.
- The root then has the smallest value.
- After removing the root, the reconstruction of the heap can be done in $\mathcal{O}(\log m)$ time.
- Actually, the insertion of a new node into the heap also can be done in $\mathcal{O}(\log m)$ time.
- Since the main loop in the greedy algorithm is executed $m - 1$ times, the total time of the greedy algorithm is $\mathcal{O}(m \log m)$.

2-way merge problem (cont'd)

Application on Huffman codes

Telecommunication problem:

We want to represent a set of messages by a sequence of 0's and 1's so that we can send these messages by transmitting their corresponding strings of 0's and 1's and the transmission cost is minimized.

Example:

- Assume that there are 6 messages whose access frequencies are 2, 3, 5, 7, 11 and 13.
- Their Huffman codes then are:
  - $2 \Rightarrow 0000$
  - $3 \Rightarrow 0001$
  - $5 \Rightarrow 001$
  - $7 \Rightarrow 01$
  - $11 \Rightarrow 10$
  - $13 \Rightarrow 11$

2-way merge problem (cont'd)

Application on Huffman codes