Unit 5.2 The Greedy Method, II

Algorithms

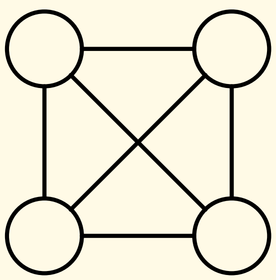
EE3980

Apr. 22, 2019

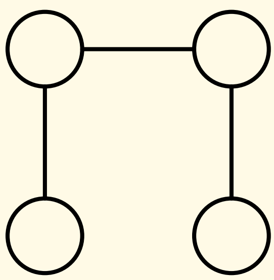
Minimum-Cost Spanning Trees

Definition 5.2.1.

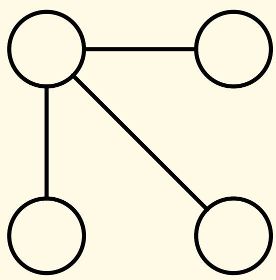
Let $G = (V, E)$ be an undirected connected graph. A sub-graph $T = (V, E')$ with $E' \subseteq E$ is a spanning tree of G if and only if T is a tree.



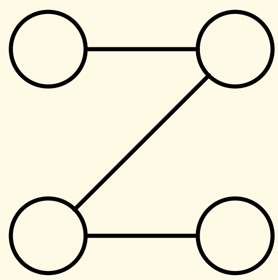
Undirected graph G.



Spanning tree T_1.



Spanning tree T_2.

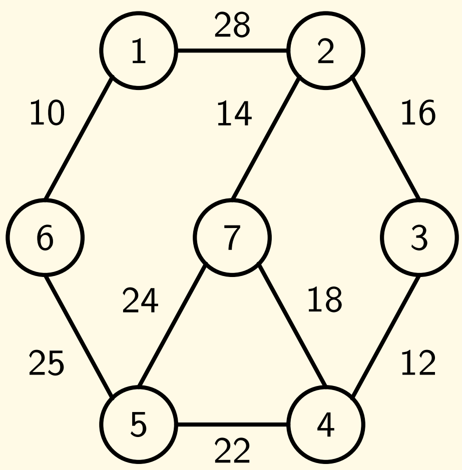


Spanning tree T_3.

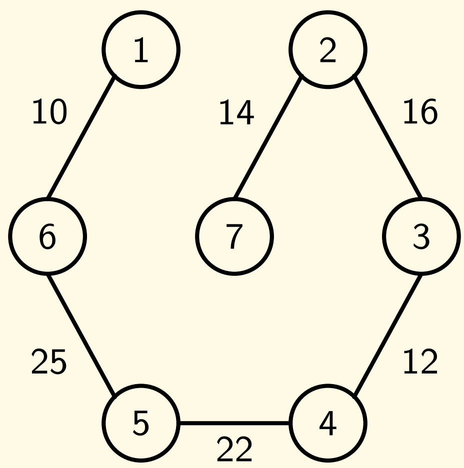
- Notes
  - Spanning tree is not unique.
  - Spanning trees have $n - 1$ edges ($n = |V|$.)

Minimum-Cost Spanning Tree, Example

- In addition, there is a cost function associated with each edge, $w: E \to \mathbb{R}$.
- The cost of a tree is the sum of the costs of the tree edges.
- A feasible solution of the minimum-cost spanning tree of a undirected graph G is any spanning tree T of G.
- The optimal solution is a spanning tree with the minimum cost.



An undirected graph, G.



Minimum-cost spanning tree, T.

Minimum-Cost Spanning Tree, Generic Algorithm

- Using the greedy methodology, let T be a subset of a spanning tree, at each step an edge (u, v) is added to T to maintain the feasibility of the solution.
- An edge, (u, v), is safe to a set of edges T if $T \cup \{(u, v)\}$ is still a subset of a spanning tree.
- The generic algorithm for the minimum-cost spanning tree then is:

Algorithm 5.2.2. Generic minimum-cost spanning tree

```
   // Given a graph G(V, E) with cost function w find minimum cost spanning tree.
   // Input: V, E, n, w; Output: minimum cost tree T.
 1 Algorithm MCST(V, E, n, w, T)
 2 {
 3     T := ∅;
 4     while (|T| < n − 1) do {
 5         select an edge (u, v) ∈ E {
 6             if (u, v) is safe to T then T := T ∪ (u, v);
 7             E := E − {(u, v)};
 8         }
 9     }
10 }
```

- The key is in line 5, how to select an edge.

Minimum-Cost Spanning Tree, Prim's Algorithm

Algorithm 5.2.3. Prim

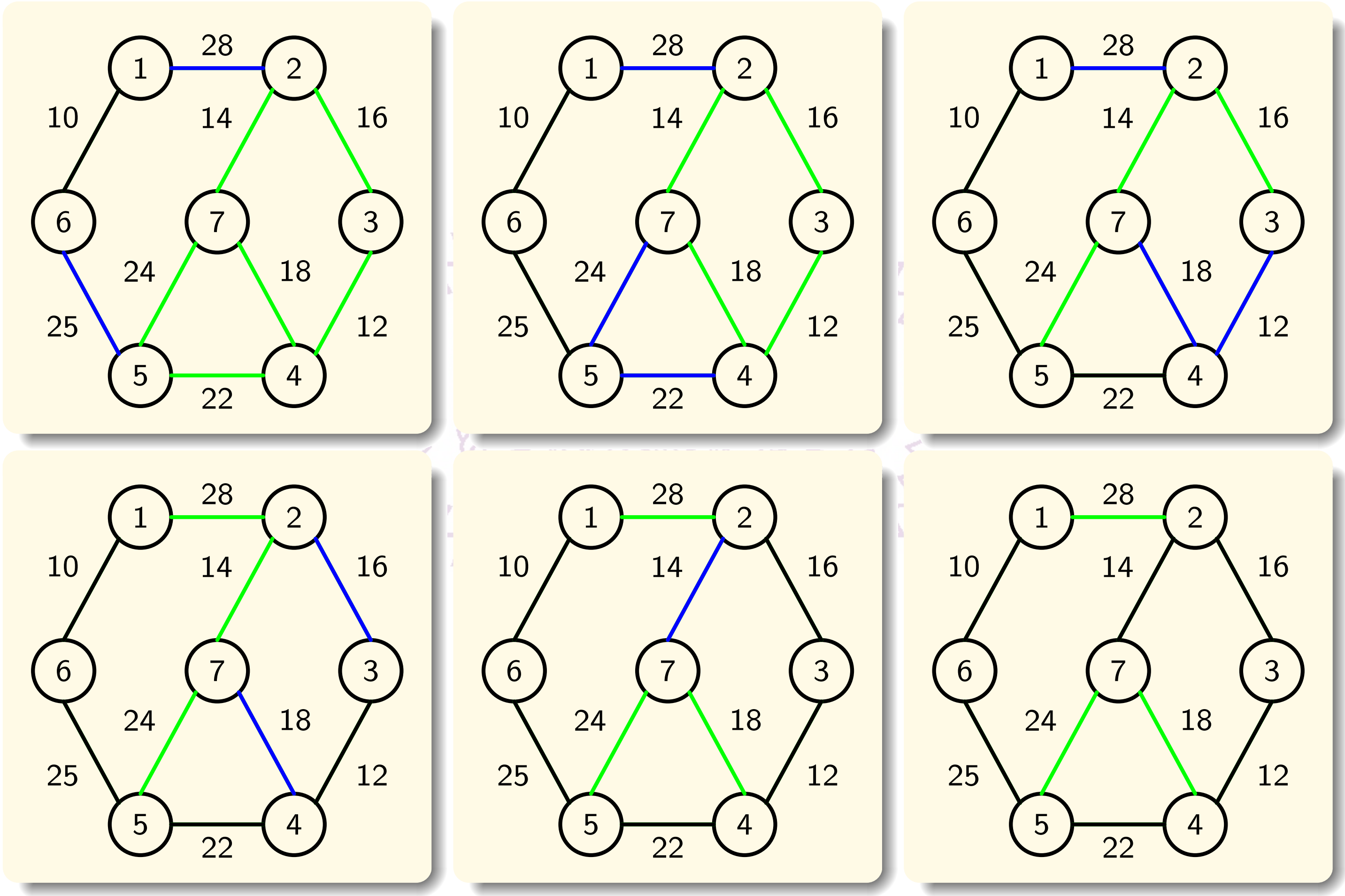
// Given a graph $G(V, E)$ with cost function w find minimum cost spanning tree.
// Input: V, E, n, w; Output: minimum cost tree T and *mincost*.

1 **Algorithm** Prim(V, E, n, w, T)
2 {
3   Find edge $(k, \ell) \in E$ with the minimum cost ;
4     $mincost := w[k, \ell]$; // *mincost* set to minimum edge cost.
5     $T[1, 1] := k$; $T[1, 2] := \ell$; // Add (k, ℓ) to spanning tree.
6   **for** $i := 1$ **to** n **do** // Init *near* array for every vertices.
7   **if** $(w[i, \ell] < w[i, k])$ **then** $near[i] := \ell$; **else** $near[i] := k$;
8     $near[k] := near[\ell] := 0$; // Vertices already in the spanning tree.
9   **for** $i := 2$ **to** $(n - 1)$ **do** {
10   Find j such that $near[j] \neq 0$ **and** $w[j, near[j]]$ is minimum ;
11         $T[i, 1] := j$; $T[i, 2] := near[j]$; // Add minimum cost *near* edge to tree.
12         $mincost := mincost + w[j, near[j]]$; // Update *mincost*.
13         $near[j] := 0$; // Reset *near* array for selected vertex.
14   **for** $k := 1$ **to** n **do** // update *near* array for the other unselected vertices.
15   **if** $((near[k] \neq 0)$ **and** $(w[k, near[k]] > w[k, j]))$ **then** $near[k] := j$;
16   }
17   **return** *mincost*;
18 }

Minimum-Cost Spanning Tree, Prim's Algorithm II

- In Algorithm Prim
  1. The edge with the minimum cost is first selected as the initial tree
  2. The array near keeps the node already selected in the tree with the smallest single-edge cost for each node
  3. Among the all the near edges, the minimum is selected and the node added to the tree
  4. Array near is then updated and go back to step 3 until all nodes have been selected
- The time complexity is dominated by
  - Finding the minimum-cost edge on line 3, $\mathcal{O}(|E|) \approx \mathcal{O}(n^2)$
  - Loop on lines 6-7, $\mathcal{O}(n)$
  - Loop on lines 9-16
    - Inner loops line 10 and lines 14-15
    - Complexity $\mathcal{O}(n^2)$
  - Overall complexity is $\mathcal{O}(n^2)$
- The time complexity can be improved to $\mathcal{O}((n + |E|) \lg n)$
  - If the non-selected vertices are stored in a red-black tree

Minimum-Cost Spanning Tree, Prim's Algorithm Example



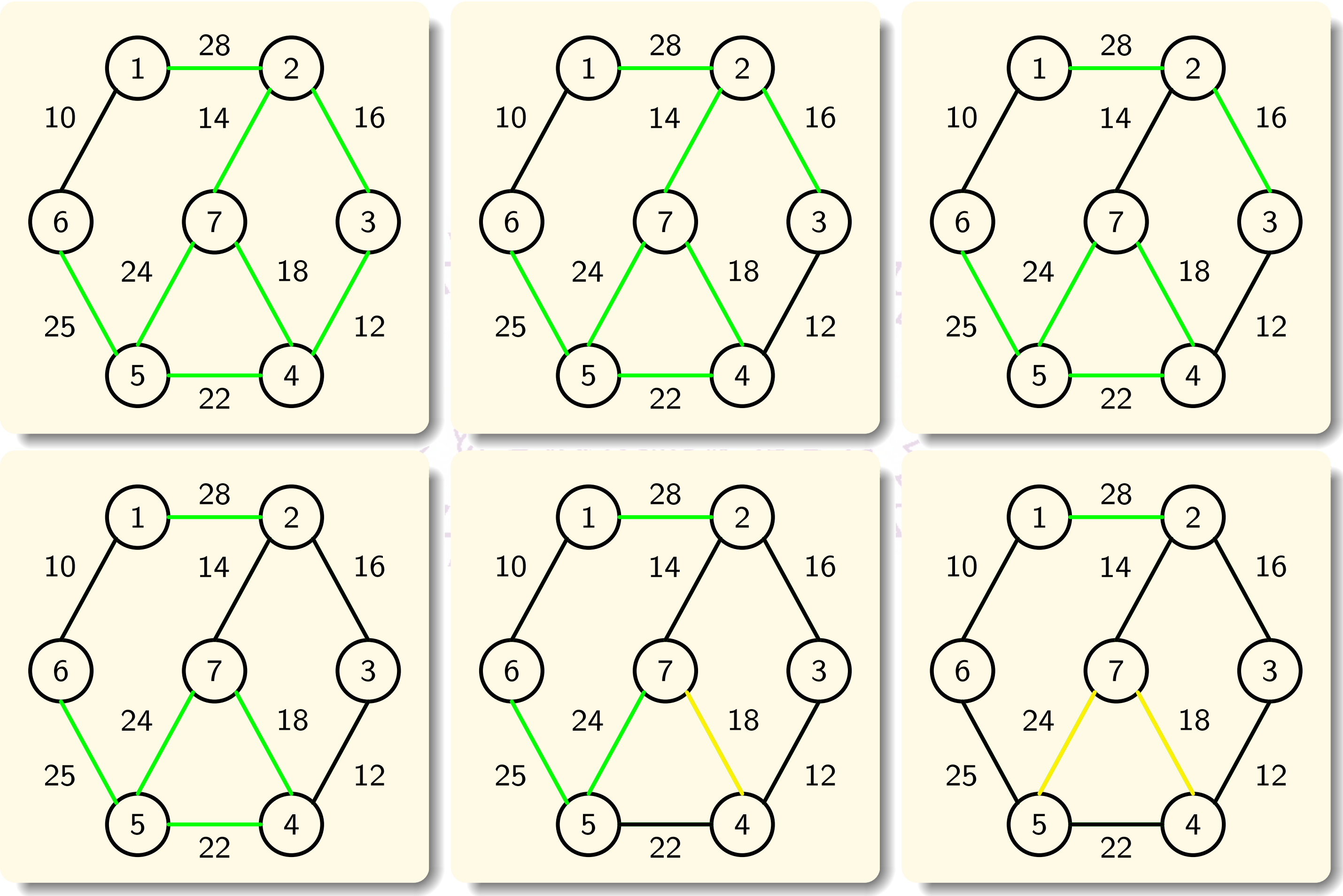
Kruskal's Algorithm – High Level

- A different approach to finding the minimum-cost spanning tree
- High level description of the algorithm

Algorithm 5.2.4. Kruskal's Algorithm

```
    // Given a graph G(V, E) with cost function w find minimum cost spanning tree.
    // Input: V, E, n, w; Output: minimum cost tree T.
Algorithm KruskalH(V, E, n, w, T)
{
    T := ∅;
    while ((T has less than (n − 1) edges ) and (E ≠ ∅)) do {
        Find the edge (u, v) ∈ E with the minimum cost ;
        Delete(u, v) from E;
        if (u, v) does not create a cycle in T then T := T ∪ (u, v) ;
        else discard (u, v) ;
    }
}
```

Kruskal's Algorithm – Example



Kruskal's Algorithm

Algorithm 5.2.5. Kruskal's Algorithm

// Given a graph $G(V,E)$ with cost function w find minimum cost spanning tree.
// **Input**: V, E, n, w; **Output**: minimum cost tree T and $mincost$.

Algorithm Kruskal(V, E, n, w, T)
{
    Construct a min heap from the edge costs using Heapity ;
    for $i := 1$ **to** n **do** $parent[i] := -1$; // Enable cycle checking
    $i := 0$; $mincost := 0$;
    while $((i < n-1)$ **and** (heap not empty)) **do** {
        delete a minimum cost edge (u, v) from the heap ;
        Adjust the heap ;
        $j :=$ Find(u) ; $k :=$ Find(v) ; // using $parent$ array
        if $(j \neq k)$ **then** {
            $i := i + 1$; $T[i,1] := u$; $T[i,2] := v$;
            $mincost := mincost + w[u, v]$;
            Union(j, k) ; // modify $parent$ array
        }
    }
    if $(i \neq n-1)$ **then** write("*No spanning tree*") ; **else return** $mincost$;
}

Kruskal's Algorithm – Complexity and Optimality

- The time complexity of `Kruskal` algorithm is dominated by the `while` loop, lines 6-15, – $\mathcal{O}(|E|)$
  - Line 7 finding minimum cost edge, $\mathcal{O}(1)$
  - Line 8 `Adjust` the heap, $\mathcal{O}(\lg |E|)$
  - Overall complexity $\mathcal{O}(|E| \lg |E|)$.

Theorem 5.2.6.

Kruskal's algorithm (Algorithm 5.2.5) generates a minimum-cost spanning tree for every undirected connected graph G.

- Proof please see textbook [Horowitz], p. 244.

Minimum-Cost Spanning Tree, Properties

- A different approach to prove Kruskal's algorithm.
- We define the following terms.
  - A cut $(S, V-S)$ of an undirected graph $G=(V,E)$ is a partition of V, i.e., $S \in V$.
  - An edge $(u,v) \in E$ is said to cross the cut $(S, V-S)$ if one of its end points is in S and the other in $V-S$.
  - A cut is said to respect a set T of edges if no edges in T crosses the cut.
  - An edge is said to be a light edge crossing a cut if its cost is the minimum of any edge crossing the cut.

Theorem 5.2.7.

Let $G=(V,E)$ be a connected, undirected graph with a cost function w defined on E. Let T be a subset of E that is subset of a spanning tree of G, let $(S, V-S)$ be any cut of G that respects T, and let (u,v) be a light edge crossing $(S, V-S)$. Then, edge (u,v) is safe for T.

- Proof please see textbook [Cormen], pp. 627-628.

Minimum-Cost Spanning Tree, Properties, II

Corollary 5.2.8.

Let $G = (V, E)$ be a connect, undirected graph with cost function w defined on E. Let T be a subset of E that is included in a minimum spanning tree of G, and let $C = (V_C, E_C)$ be a connected component (tree) in the forest $G_T = (V, T)$. If (u, v) is a light edge connecting C to some other component in G_T, then (u, v) is safe for T.

- Proof please see textbook [Cormen], pp. 629.
- Algorithm `Prim` can be shown to be a special case of Theorem (5.2.7), and it also returns an optimal solution.

The Matroid

Definition 5.2.9. Matroid

A matroid is an ordered pair $M = (\mathcal{S}, \mathcal{I})$ satisfying the following conditions.

1. $\mathcal{S}$ is a finite set.
2. $\mathcal{I}$ is a nonempty family of subsets of $\mathcal{S}$, called the independent subsets of $\mathcal{S}$, such that if $B \in \mathcal{I}$ and $A \subseteq B$, then $A \in \mathcal{I}$. We say that $\mathcal{I}$ is hereditary if it satisfies this property. Note that the empty set $\emptyset$ is necessary is a member of $\mathcal{I}$.
3. if $A \in \mathcal{I}$, $B \in \mathcal{I}$ and $|A| < |B|$, then there exists some element $x \in B - A$ such that $A \cup \{x\} \in \mathcal{I}$. We say that M satisfies the exchange property.

- References
  – Textbook [Cormen], pp. 437 - 442.
  – Bernhard Korte and Jens Vygen, *Combinatorial Optimization – theory and algorithms*, 4th edition, Springer, 2008.
  - Chapter 13. Matroids
- Example: Given a matrix, $\mathcal{S}$ is the set of columns of the matrix, $\mathcal{I}$ is the set formed by independent columns.
  - All three conditions are met.

Graph Matroid

- Graphic matroid $M_G = (\mathcal{S}_G, \mathcal{I}_G)$ defined in terms of a given undirected graph $G = (V, E)$ as follows:
  - The set $\mathcal{S}_G$ is defined to be E, the set of edges of G.
  - If A is a subset of E, the $A \in \mathcal{I}_G$ if and only if A is acyclic. That is, a set of edges A is independent if and only if the subgraph $G_A = (V, A)$ forms a forest.

Theorem 5.2.10. Graph matroid.

If $G = (V, E)$ is an undirected graph, then $M_G = (\mathcal{S}_G, \mathcal{I}_G)$ is a matroid.

- Proof please see textbook [Cormen], p. 438.
- Exchange property of M_G can be shown as: if no such x can be found then $|B| \le |A|$ that contradicts to the assumption.

Definition 5.2.11. Extension.

Given a matroid $M = (\mathcal{S}, \mathcal{I})$, we call an element $x \notin A$ an extension of $A \in \mathcal{I}$ if we can add x to A while preserving the independence; that is, x is an extension of A if $A \cup \{x\} \in \mathcal{I}$.

- Graphic matroid: if $A \in \mathcal{I}$, then an edge e is an extension of A if $e \notin A$ and there is no cycle in $A \cup \{e\}$.

Graph Matroid – Spanning Trees

Definition 5.2.12. Maximal independent set.

If A is an independent set in a matroid M, we cay that A is maximal if is has no extensions. That is, A is maximal if it is not contained in any larger independent subset of M.

Theorem 5.2.13.

All maximal independent subsets in a matroid have the same size.

- Proof please see textbook [Cormen], p. 439.
- Note that
  - There can be more than one maximal independent subset.
  - All of them are of the same size.
- Example
  - For a graphic matroid M_G for a connected, undirected graph G, every maximal independent subset of M_G must be a free tree with exactly $|V| - 1$ edges that connects all the vertices of G.
  - These trees are the spanning tree of G.

Weighted Graph Matroid

Definition 5.2.14. Weighted matroid

A matroid $M = (\mathcal{S}, \mathcal{I})$ is weighted if it is associated with a weight function w that assigns a strictly positive weight $w(x)$ for each element $x \in \mathcal{S}$. The weight function w extends to subsets of $\mathcal{S}$ by summation:

$$w(A) = \sum_{x \in A} w(x) \qquad \text{for any } A \in \mathcal{S}. \tag{5.2.1}$$

- For example, if $w(e)$ is the weight of an edge e in a graphic matroid M_G, then $w(A)$ is the total weights of the edges in A.
- The minimum-spanning-tree problem can be formulated using weighted graph matroid.
  Given a connect undirected graph $G = (V, E)$ and a weight function w such that $w(e)$ is the weight of an edge $e \in E$. The minimum-spanning-tree problem is to find a subset of the edges that connects all of the vertices together and has the minimum total weight.

Greedy MST Algorithm

- Given a undirected graph $G = (V, E)$ and weight function w. Let $M_G = (\mathcal{S}, \mathcal{I})$ where $\mathcal{S}$ is the set of all edges and $\mathcal{I}$ is the set of all acyclic edges in G.

Algorithm 5.2.15. Greedy Minimum Spanning Tree

```
   // Given a graph G(V,E) and the matroid M_G(S,I) find minimum spanning tree.
   // Input: S, I, w; Output: minimum spanning tree T.
Algorithm GreedyMST(S, I, w)
{
    T := ∅; // Initialize empty tree.
    Sort S into monotonically increasing order by w ;
    for each minimum x ∈ S do { // Try all edges.
        if (T ∪ {x} ∈ I) then { // Maintain independency then add.
            T := T ∪ {x};
        }
        S := S − {x}; // Delete x from S
    }
    return T;
}
```

Greedy MST Algorithm, II

- Let n be the number of edges in G, i.e., $n = |\mathcal{S}|$.
- Line 4 takes $\mathcal{O}(n \lg n)$ time to execute.
- Lines 5-10 execute n times.
- Let $f(n)$ be the time that line 6 takes to check the condition
- The execution time for the `GreedyMST` is then $\mathcal{O}(n \lg n + n \cdot f(n))$.

- The optimality of the algorithm comes from the following theorems.

Lemma 5.2.16.

Suppose that $M = (\mathcal{S}, \mathcal{I})$ is a weighted matroid with weight function w and that $\mathcal{S}$ is sorted into monotonically increasing order by weight. Let x be the first element of $\mathcal{S}$ such that $\{x\}$ is acyclic. If x exists the there exists an optimal subset $A \subseteq \mathcal{S}$ and $x \in A$.

- Proof uses maximum size Theorem (5.2.13) and the exchange property. Please see textbook [Cormen], p. 441.

Greedy MST Algorithm, III

Lemma 5.2.17.

Let $M = (\mathcal{S}, \mathcal{I})$ be any matroid. If x is an element of $\mathcal{S}$ that is an extension of some independent subset A of $\mathcal{S}$, then x is also an extension of $\emptyset$.

- Proof please see textbook [Cormen], p. 441.

Corollary 5.2.18.

Let $M = (\mathcal{S}, \mathcal{I})$ be any matroid. If x is an element of $\mathcal{S}$ such that x is not an extension of $\emptyset$, then x is not an extension of any independent subset A of $\mathcal{S}$.

- Proof please see textbook [Cormen], p. 441.

- This corollary says that if x is discarded by line 9 it should not be included in the optimal solution.

Greedy MST Algorithm, IV

Lemma 5.2.19.

Let x be the first element of $\mathcal{S}$ chosen by Algorithm GreedyMST for the weighted matroid $M = (\mathcal{S}, \mathcal{I})$. The remaining problem of finding a minimum-weight independent subset containing x reduces to finding a minimum-weight independent subset of weighted matroid $M' = (\mathcal{S}', \mathcal{I}')$, where

$$\mathcal{S}' = \{y \in \mathcal{S} | \{x, y\} \in \mathcal{I}\}, \tag{5.2.2}$$

$$\mathcal{I}' = \{B \subseteq \mathcal{S} - \{x\} | B \cup \{x\} \in \mathcal{I}\}. \tag{5.2.3}$$

and the weight function for M' is the weight function for M restricted to $\mathcal{S}'$. (M' is called the contraction of M by the element x.)

- Proof please see textbook [Cormen], p. 442.

Theorem 5.2.20.

If $M = (\mathcal{S}, \mathcal{I})$ is a weighted matroid with weight function w, then GreedyMST returns an optimal subset.

- Proof please see textbook [Cormen], p. 442.

Job Sequencing with Deadlines

- Given a set of n jobs to be processed on one machine.
  - Each job takes 1 time unit to process.
  - Associated with job i, $1 \le i \le n$, there is a deadline d_i and profit p_i.
  - If job i is completed by d_i then p_i is earned.
- A feasible solution is a subset J of jobs that each job in J can be completed by its deadline.
  - The value of the subset J is $\sum_{i \in J} p_i$.
- An optimal solution is a feasible solution with the maximum value.

- Example, $n = 4$, $\{p_1, p_2, p_3, p_4\} = \{100, 10, 15, 27\}$, $\{d_1, d_2, d_3, d_4\} = \{2, 1, 2, 1\}$.
- Feasible solutions are

	Feasible solution	Processing sequence	Value
1	$\{1, 2\}$	2,1	110
2	$\{1, 3\}$	1,3 or 3,1	115
3	$\{1, 4\}$	4,1	127
4	$\{2, 3\}$	2,3	25
5	$\{3, 4\}$	4,3	42
6	$\{1\}$	1	100
7	$\{2\}$	2	10
8	$\{3\}$	3	15
9	$\{4\}$	4	27

- Solution 3 is optimal.

Job Sequencing with Deadlines – Algorithm

Alrogithm 5.2.21. Job Sequencing

```
// Solve job scheduling problem with jobs sorted in non-increasing order.
// Input: int n, deadline d, profit p; Output: Optimal sequence J[1 : k].
Algorithm JS(n, d, p, J)
{
    d[0] := J[0] := 0; // initialize.
    J[1] := 1;
    k := 1;
    for i := 2 to n do {
        r := k;
        while ((d[J[r]] > d[i]) and (d[J[r]] ≠ r)) do r := r − 1;
        if ((d[J[r]] ≤ d[i]) and (d[i] > r)) then { // insert i into J
            for q := k to (r + 1) step −1 do J[q + 1] := J[q];
            J[r + 1] := i; k := k + 1;
        }
    }
}
```

- The worst-case time complexity of JS algorithm is $\Theta(n^2)$.
- The space complexity of JS algorithm is $\mathcal{O}(n)$ for arrays J and d.

Job Sequencing with Deadlines – Example

- Example: $n = 5$, $(p_1, p_2, p_3, p_4, p_5) = (20, 15, 10, 5, 1)$, and $(d_1, d_2, d_3, d_4, d_4) = (2, 2, 1, 3, 3)$. Then, the execution sequence of the algorithm is as following.

i	J	d	action	profit
–	$\{1, , , , \}$	$\{2, , , , \}$	accept 1	20
2	$\{1, 2, , , \}$	$\{2, 1, , , \}$	accept 2	35
3	$\{1, 2, , , \}$	$\{2, 1, , , \}$	reject 3	35
4	$\{1, 2, 4, , \}$	$\{2, 1, 3, , \}$	accept 4	40
5	$\{1, 2, 4, , \}$	$\{2, 1, 3, , \}$	reject 5	40

Theorem 5.2.22.

Let J be a set of k jobs and $\sigma = i_1, i_2, \cdots, i_k$ a permutation of jobs in J such that $d_{i_1} \le d_{i_2} \le \cdots \le d_{k_i}$. Then J is a feasible solution if and only if the jobs in J can be processed in the order σ without violating any deadline.

- Proof please see textbook [Horowitz], p. 229.

Job Sequencing with Deadlines – Matroid Formulation

- The job sequencing with deadline can be shown to be a matroid.
  The set $\mathcal{S}$ contains all the jobs, and a set A of jobs are independent if there is a schedule such that all jobs in A are done before their deadlines.

Lemma 5.2.23.

For any set of jobs A, the following statements are equivalent.

1. The set A is independent.
2. Let $N_t(A)$ denote the number of jobs completed before time t, then for $t = 0, 1, 2, \ldots, n$, we have $N_t(A) \le t$.
3. If the tasks in A are scheduled in order of monotonically increasing deadlines, the all jobs in A are completed before their deadlines.

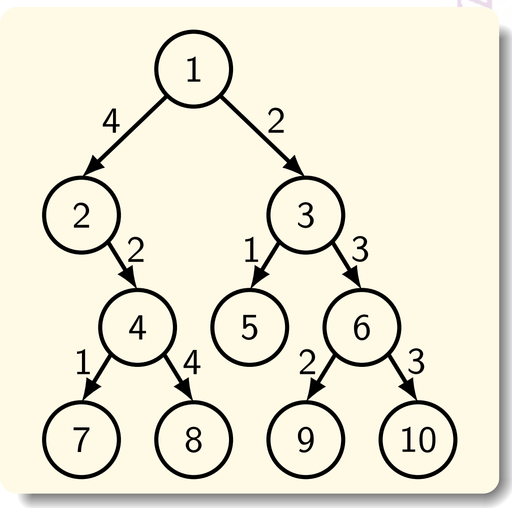
Theorem 5.2.24.

If $\mathcal{S}$ is a set of unit-time jobs with deadlines, and $\mathcal{I}$ is the set of all independent sets of tasks, then the corresponding system $(\mathcal{S}, \mathcal{I})$ is a matroid.

- Since the job sequencing problem is a matroid, the greedy algorithm can be applied and it results in an optimal solution.

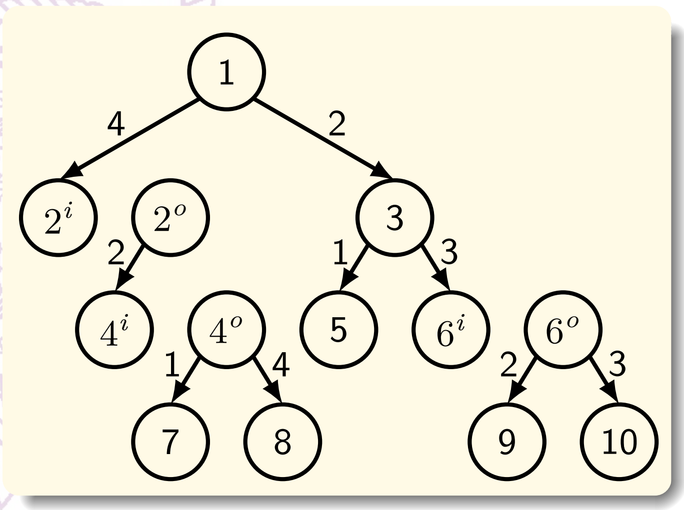
Tree Vertex Splitting Problem

Original tree T



$d(T) = 10.$

Tree with vertices splitted T/X



$d(T/X) = 5.$

Tree Vertex Splitting Problem – Definition

- $T = (V, E, w)$ is weighted directed tree.
  - V is the vertex set, E is the edge set, and w is weight function for the edges.
  - $w(i, j)$ is defined if the edge $\langle i, j\rangle \in E$; $w(i, j)$ is undefined if $\langle i, j\rangle \notin E$.
  - A source vertex is a vertex with in-degree 0.
  - A sink vertex is a vertex with out-degree 0.
  - For any path P in the tree, its delay, $d(P)$, is defined to be the sum of the weights on the path.
  - The delay of the tree, $d(T)$, is the maximum of all the path delays.
- T/X is the forest resulted from splitting every vertex u in $X \subseteq V$ into two nodes u^i and u^o such that all the edges $\langle i, u\rangle$ are replaced by $\langle i, u^i\rangle$ and all the edges $\langle u, j\rangle$ are replaced by $\langle u^o, j\rangle$.
- The Tree Vertex Splitting Problem (TVSP) is to find a set $X \subseteq V$ with minimum cardinality for which $d(T/X) \le \delta$ for some specified tolerance δ.
  - Note that a TVSP has solution only if the maximum edge weight is less than or equal to δ.
  - Any $X \subseteq V$ with $d(T/X) \le \delta$ is a feasible solution.
  - The optimal solution is the feasible X with the minimum number of vertices.

Tree Vertex Splitting Problem – Algorithm

Algorithm 5.2.25. TVS

```
   // Find the minimum set X for vertex splitting.
   // Input: tree T, maximum edge weight δ ; Output: solution X.
Algorithm TVS(T, δ, X)
{
    if (T ≠ ∅) then {
        d[T] := 0 ;
        for each child v of T do {
            TVS(v, δ, X) ;
            d[T] := max(d[T], d[v] + w(T, v)) ;
        }
        if ((T is not the root ) and (d(T) + w(parent(T), T) > δ)) then {
            X := X ∪ {T} ; d[T] := 0 ;
        }
    }
}
```

- Note that d is a global array that stores the *delay* for each vertex.

Tree Vertex Splitting Problem – Algorithm II

Algorithm 5.2.26. TVS1

```
// Tree vertex splitting with tree stored in an array tree[1 : n].
// Input: root i, maximum edge weight δ; Output: solution X.
Algorithm TVS1(i, δ, X)
{
    if (tree[i] ≠ 0) then {
        if (2×i > N) then d[i] := 0; // i is a leaf.
        else {
            TVS1(2×i, δ, X);
            d[i] := max(d[i], d[2×i] + w[2×i]);
            if (2×i + 1 ≤ N) then {
                TVS1(2×i + 1, δ, X);
                d[i] := max(d[i], d[2×i + 1] + w[2×i + 1]);
            }
        }
        if ((i ≠ 1) and (d[i] + w[i] > δ)) then {
            X := X ∪ {i}; d[i] := 0;
        }
    }
}
```


Tree Vertex Splitting Problem – Complexity and Optimality

- In this version the directed binary tree is stored in an array $tree$
- The weight is stored in array w and $w[i]$ is the weight of the parent of vertex i to vertex i.
- Array d is still the $delay$ of each vertex.

- The time complexity of Algorithm TVS is $\Theta(n)$.
  - Every vertex of T is traversed once.

Theorem 5.2.27.

Algorithm TVS finds a minimum cardinality set X such that $d(T/X) \leq \delta$ on any tree T, provided that no edge of T has weight greater than δ.

- Proof please see textbook [Horowitz], pp. 225 - 226.

Summary

- Minimum-cost spanning tree problem.
- The theory of Matroid.
- Job sequencing with deadlines.
- Tree vertex splitting problem.