CS 260
Design and Analysis of Algorithms
5. Greedy Algorithms

Mikhail Moshkov

Computer, Electrical and Mathematical Sciences & Engineering Division
King Abdullah University of Science and Technology

Greedy Algorithms

Greedy algorithms are algorithms that use the following meta-heuristic: to make the locally optimal choice at each stage with the hope of finding the global optimum.

Binary Heaps

To make algorithms considered in this section more efficient we will use a priority queue (implemented via binary heap) which maintains a set of elements (nodes) with associated numeric key values and supports the following operations:

- *Insert*. Add a new element to the set.
- *Decrease-key*. Accommodate the decrease in key value of an element.
- *Delete-min*. Return the element with the smallest key, and remove it from the set.
- *Make-queue*. Build a priority queue for the given elements, with the given key values.

Binary Heaps

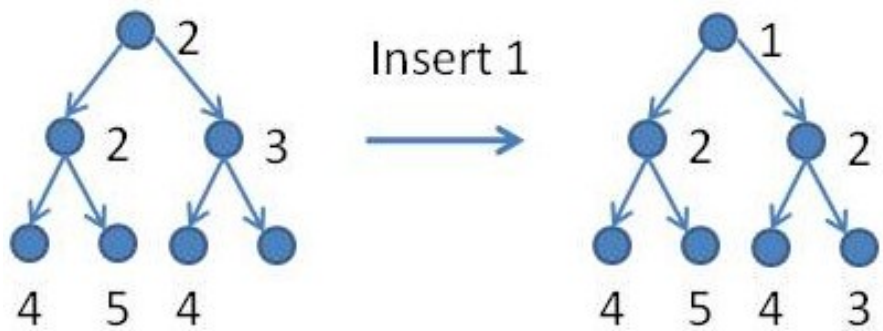
In a *binary heap* elements are stored in a complete binary tree, namely, a binary tree in which each level is filled from the left to the right, and must be full before the next level is started.

In addition, the key value of any node of the tree is less than or equal to key values of its children. In particular, the root of the tree always contains an element with the smallest key.

Binary Heaps

To *Insert* a new element, we should place it at the bottom of the tree (in the first available position). If the key of new element is less than the key of its parent, then we should swap these elements, etc.

The number of swaps is at most the depth of the tree, which is $\lfloor \log_2 n \rfloor$ where n is the number of elements.



Binary Heaps

The *Decrease-key* is similar to *Insert* with the only difference that the element is already in the tree.

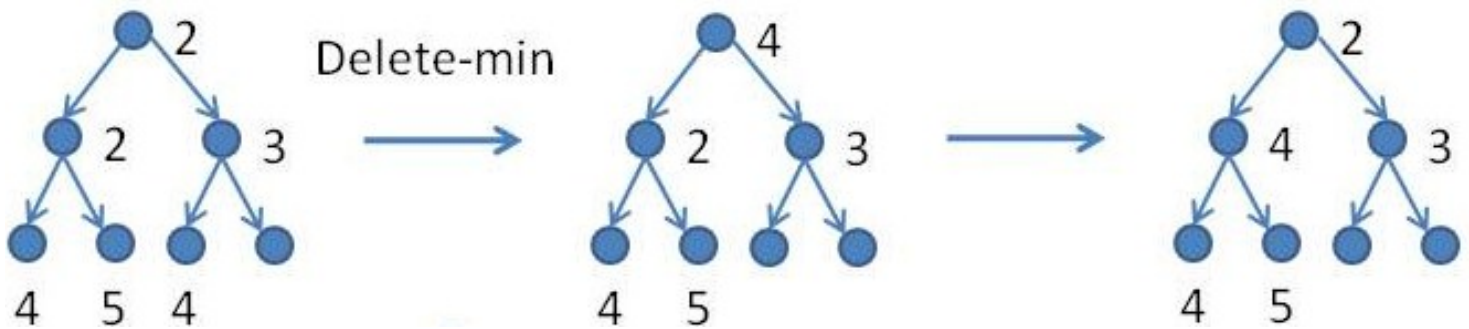
The *Make-queue* takes at most n *Insert* operations.

Binary Heaps

To *Delete-min* we should return the element attached to the root. Then we should remove this element from the heap, take the element attached to the last node in the tree (in the rightmost position in the bottom level) and place it at the root.

If the key of this element is bigger than for a child, we should swap these elements, etc.

The number of swaps is at most $\lfloor \log_2 n \rfloor$.



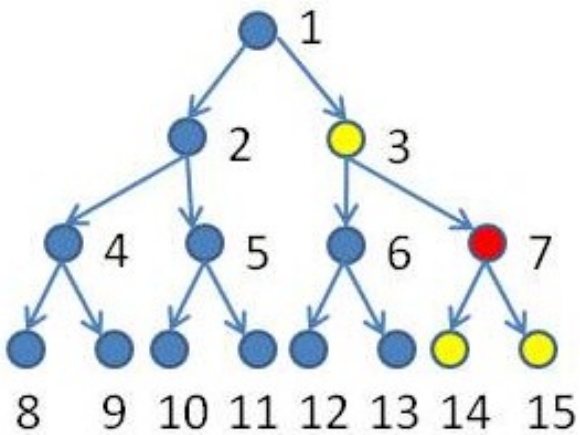
Binary Heaps

Thus, *Insert*, *Decrease-key* and *Delete-min* require $O(\log n)$ time, and *Make-queue* takes $O(n \log n)$ time.

Binary Heaps

The considered binary tree can be easily represented as an array. The nodes of tree have a natural ordering: row by row, starting at the root and moving from the left to the right within each level.

The array has positions $1, \ldots, n$ corresponding to the considered nodes. One can show that the node number j has parent with number $\left\lfloor \frac{j}{2} \right\rfloor$ and children with numbers $2j$ and $2j + 1$.



Numeration of nodes

$7 \rightarrow \lfloor 7/2 \rfloor = 3$

$7 \rightarrow 2 \times 7$ & $2 \times 7 + 1 = 14$ & 15

Dijkstra's Algorithm

Let $G = (V, E)$ be a directed graph in which each edge e has a length $l_e \geq 0$. For a path P, the length of P (denoted by $l(P)$) is the sum of lengths of all edges in P.

Let s be a node of G (start node). We assume that s has a path to every other node in G.

Our goal is to find the shortest path from s to every other node in G.

Dijkstra's Algorithm

The algorithm constructs a set S of nodes u for which we have determined the length of the shortest path $d(u)$ from s. Initially, $S = \{s\}$ and $d(s) = 0$.

Now for each node $v \in V \setminus S$, we determine the length $d'(v)$ of a shortest path of the following kind: the considered path passes through nodes of S until some node $u \in S$ and then passes through an edge $(u, v) \in E$.

It is clear that

$$d'(v) = \min_{e=(u,v)\in E, u\in S} d(u) + l_e.$$

If there is no $(u, v) \in E$ such that $u \in S$, then $d'(v) = +\infty$.

Dijkstra's Algorithm

We choose $v \in V \setminus S$ for which $d'(v) = \min\{d'(v') : v' \in V \setminus S\}$ and add v to S.

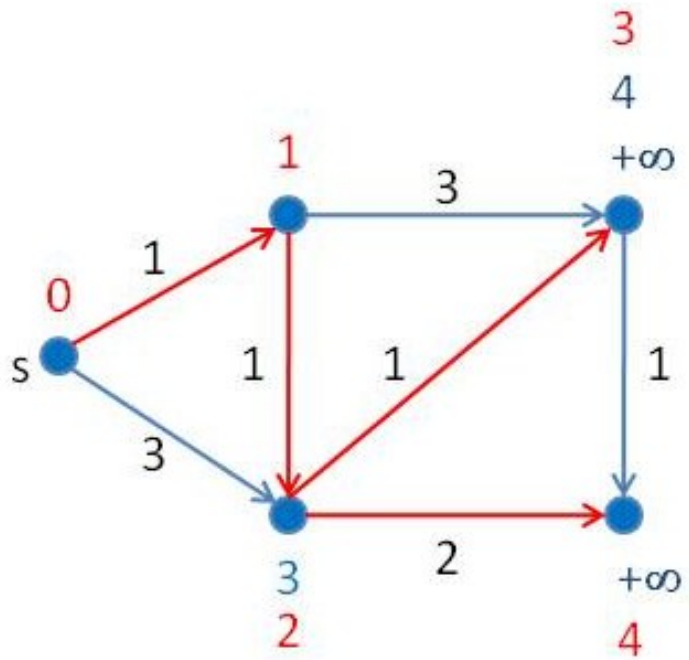
We set $d(v) = d'(v)$ and $predecessor(v) = u$, where $u \in S$ and there exists an edge $e = (u, v) \in E$ such that $d'(v) = d(u) + l_e$.

The value $predecessor(v)$ for each $v \in V$, $v \neq s$, will allow us to restore the shortest paths from s to all other nodes in G.

The algorithm will finish the work when $S = V$.

Dijkstra's Algorithm

Example 5.1. Dijkstra's algorithm



Dijkstra's Algorithm

During each step the algorithm adds new node v to the set S, computes the value $d(v)$ for this node and forms the path P_v from s to v with the help of values $predecessor(.)$.

Let us show by induction on $|S|$ that P_v is the shortest path from s to v. From here it follows that $d(v)$ is the minimum length of a path from s to v.

Dijkstra's Algorithm

Let $|S| = 1$. In this case $S = \{s\}$ and $d(s) = 0$. It is clear that the considered statement holds in this case.

Let us assume that the considered statement holds when $|S| = k$ for some value of $k \geq 1$.

Let during the step number $k + 1$ the algorithm choose the node v, and $e = (u, v)$ be the final edge in the path P_v.

Dijkstra's Algorithm

Let us consider an arbitrary path P from s to v. Since $v \notin S$, there is an edge $e' = (x, y)$ in this path such that $x \in S$ and $y \notin S$.

By the inductive hypothesis, the length of any path from s to x is at least $d(x)$. Therefore $l(P) \geq d(x) + l_{e'} \geq d(u) + l_e = l(P_v)$.

The inequality $d(x) + l_{e'} \geq d(u) + l_e$ follows from the description of the algorithm. Therefore $l(P) \geq l(P_v)$, and P_v is the shortest path from s to v.

Dijkstra's Algorithm

To make the considered algorithm more efficient, we will use binary heap to state values $d'(v)$ for nodes $v \in V \setminus S$.

Before the first step of the algorithm we have $d'(s) = 0$ and $d'(v) = +\infty$ for each $v \in V \setminus \{s\}$.

By the operation *Make-queue* we build a binary heap for nodes from V using values $d'(v)$ as keys.

Later step by step we will modify this heap to have current set of nodes $V \setminus S$ and current values $d'(v)$.

Dijkstra's Algorithm

Let after the step number k we have in the heap all nodes v from $V \setminus S$ with keys $d'(v)$.

During the step number $k+1$, by operation *Delete-min* we choose in the heap the node v with the minimum value $d'(v)$ and delete v from the heap.

We set $d(v) = d'(v)$. Now we should modify values of $d'(w)$ for each node $w \in V \setminus (S \cup \{v\})$.

Dijkstra's Algorithm

If (v, w) is not an edge then we remain $d'(w)$ untouched.

Let $e' = (v, w)$ be an edge. Then the new value of the key for w is

$$\min\{d'(w), d(v) + l_{e'}\}.$$

If $d'(w) > d(v) + l_{e'}$ then it is necessary to set $\mathit{predecessor}(w) = v$ and to use the *Decrease-key* operation to decrease the key of node w up to $d(v) + l_{e'}$.

Dijkstra's Algorithm

To decrease the key for w we should find w in the heap (in the corresponding array).

To this end, we will assume that all n nodes in G are numbered by numbers $1, \ldots, n$ and we have an array in which in the t-th position we have the current value of the position of t-th node in the heap.

We can store in this array values $d(u)$ and $predecessor(u)$.

Dijkstra's Algorithm

The *Decrease-key* operation can be used at most once per edge, when the initial node of the edge is added to S.

Let $|V| = n$ and $|E| = m$. During the work the algorithm makes one operation *Make-queue* ($O(n \log n)$), at most n operations *Delete-min* ($O(n \log n)$), and at most m operations *Decrease-key* ($O(m \log n)$).

Thus, the overall time for the implementation is $O((m + n) \log n)$.

Minimum Spanning Tree

Let we have a connected undirected graph $G = (V, E)$ in which each edge e is labeled with a cost $c_e \geq 0$. We will call a subset $T \subseteq E$ a *spanning tree* of G if (V, T) is a tree.

A spanning tree for which the total cost $\sum_{e \in T} c_e$ is as small as possible is called a *minimum* spanning tree. Our goal is to construct a minimum spanning tree.

Minimum Spanning Tree

Let us describe Prim's Algorithm for this problem solving.

We start with a root node s. We maintain a set $S \subseteq V$ on which a spanning tree is constructed. Initially, $S = \{s\}$.

During each iteration, we add to S one node $v \in V \setminus S$ which minimizes the value

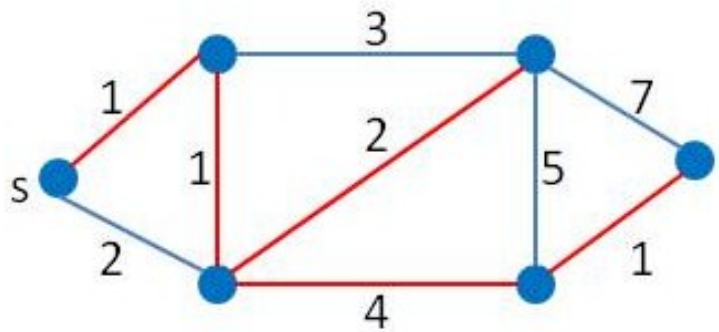
$$a(v) = \min_{e=(u,v)\in E, u\in S} c_e,$$

and include the edge $e = (u, v)$ that achieves this minimum in the constructed spanning tree T. We will finish the work when $S = V$.

It is clear that the constructed graph is connected and has $|V| - 1$ edges. So this graph is a spanning tree.

Minimum Spanning Tree

Example 5.2. Prim's algorithm



Minimum Spanning Tree

Let us prove by induction on $|S|$ that partial spanning tree constructed during the step number $|S|$ is a part of minimum spanning tree.

If $|S| = 1$ then the partial spanning tree is empty, and the considered statement holds.

Let for some $k \geq 1$ this statement hold: $|S| = k$ and the constructed partial spanning tree T_k is a part of some minimum spanning tree T.

Minimum Spanning Tree

During the step number $k+1$ we add to S an edge $e=(u,v)$ such that $u \in S$, $v \in V \setminus S$ and e has minimum cost c_e among all edges that join S and $V \setminus S$.

If $e \in T$ then the statement under consideration holds.

Let now $e \notin T$. In this case we add e to T. As a result we obtain a graph K with a cycle. This cycle must contain another edge e' which joins S and $V \setminus S$.

Minimum Spanning Tree

If we remove e' from K, we obtain a connected graph with $|V|$ nodes and $|V| - 1$ edges (since T is a spanning tree, $|T| = |V| - 1$). Therefore, we obtain a tree.

So $T' = (T \setminus \{e'\}) \cup \{e\}$ is a spanning tree. According to the choice of e, the total cost of T' is at most the total cost of T. Thus, T' is a minimum spanning tree. It is clear, that $T_k \subseteq T'$ and $e \in T'$. Therefore the considered statement holds.

From this statement it follows that Prim's algorithm constructs a minimum spanning tree.

Minimum Spanning Tree

We can implement Prim's algorithm almost in the same way as Dijkstra's algorithm. By analogy with Dijkstra's algorithm we should be able do decide which node $v \in V \setminus S$ will be added to S.

To this end we will consider the attachment cost

$$a(v) = \min_{e=(u,v)\in E, u\in S} c_e$$

for each node $v \in V \setminus S$.

As before, we keep the nodes in a priority queue with $a(v)$ as the key. We build a binary heap with a *Make-queue* operation. We select a node with a *Delete-min* operation, and update the attachment cost using *Decrease-key* operation.

Minimum Spanning Tree

We perform *Make-queue* operation one time. The are $n-1$ iterations in which we perform *Delete-min* operation, and we perform *Decrease-key* operation at most once for each edge. So the overall running time is $O((n+m)\log n)$ where $n = |V|$ and $m = |E|$.

At the beginning of the algorithm we have $S = \{s\}$. We form values of $a(v)$ for $v \in V \setminus \{s\}$ in the following way. If $(s, v) \in E$ then $a(v) = c_{(s,v)}$. Otherwise, $a(v) = +\infty$.

Let during some step we add to S a new node v. Then for each node $w \in V \setminus (S \cup \{v\})$ for which $(v, w) \in E$, instead of old value $a(w)$ we should use the new value which is equal to $\min(a(w), c_{(v,w)})$.

Huffman Codes

Let we have n symbols $a_1, \ldots, a_n$ (alphabet) and frequencies $f_1, \ldots, f_n$ ($f_1 > 0, \ldots, f_n > 0$, $\sum_{i=1}^{n} f_i = 1$) of these symbols in a text. We will encode symbols $a_1, \ldots, a_n$ by words $\alpha_1, \ldots, \alpha_n$ (codewords) in the alphabet $\{0, 1\}$.

To avoid problems with decoding we will consider only prefix-free codes in which no codeword can be a prefix of another codeword.

Our goal is to find a prefix-free code $\alpha_1, \ldots, \alpha_n$ for which the cost of encoding $\sum_{i=1}^{n} |\alpha_i| f_i$ has minimum value, where $|\alpha_i|$ is the length of α_i. The cost of encoding is the average number of bits per symbol from $\{a_1, \ldots, a_n\}$.

Huffman Codes

One can show that to this end it is enough to consider only prefix-free codes which can be represented by full binary trees. In such a tree every node has either zero of two children, leaves are symbols $a_1, \ldots, a_n$ and each codeword is generated by a path from the root to leaf, interpreting left as 0 and right as 1.

The process of decoding is simple. The word in the alphabet $\{0, 1\}$ is decoded by starting at the root. We read the word and move from the root to a leaf. When a leaf is reached we output the symbol attached to the leaf, and return to the root.

Huffman Codes

The cost of encoding (cost of the tree) is equal to

$$\sum_{i=1}^{n} f_i \cdot (\text{depth of } a_i \text{ in the tree}).$$

This value can be represented also in another way. The frequency of the leaf a_i is f_i. We can define the frequency of any internal node. This frequency is equal to the sum of frequencies of descendant leaves for the considered node. One can show that the cost of encoding is equal to

the sum of frequences of all leaves and internal nodes, except the root.

Huffman Codes

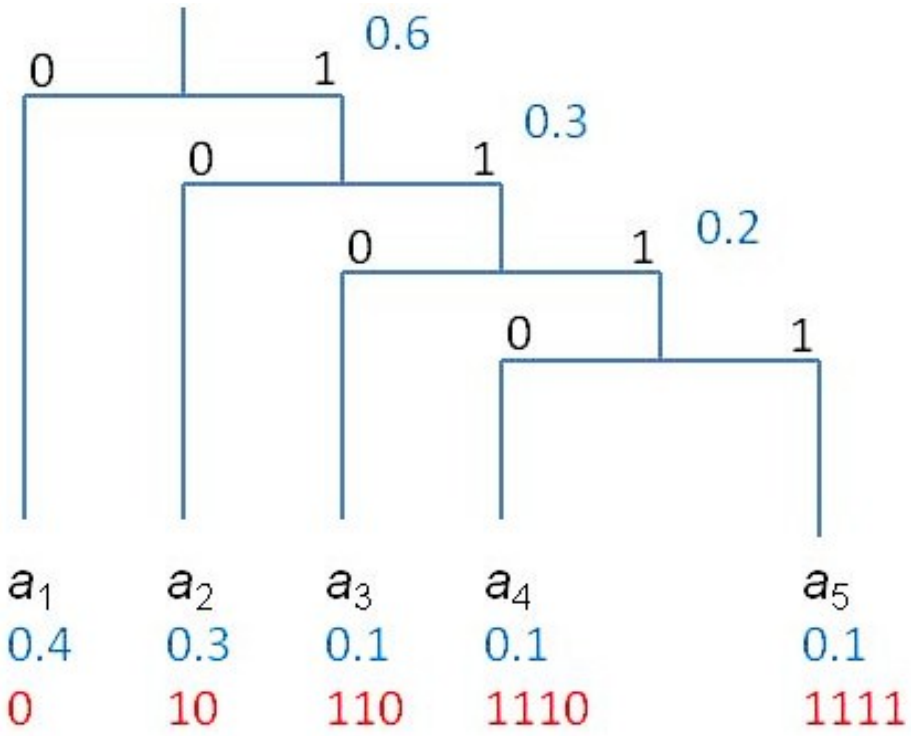
The first formulation of the cost function tells us that the two symbols with the smallest frequencies must be at the bottom of the optimal tree (tree with minimum cost), as children of the lowest internal node (this node has two children since the tree is full). Otherwise swapping these two symbols with symbols that are lower in the tree we can improve the cost of the tree.

We will construct the tree greedily: find two symbols with the smallest frequencies (let for simplicity it will be a_1 and a_2) and make then children of a new node which has frequency $f_1 + f_2$.

By the second formulation of the cost function, any tree in which a_1 and a_2 are sibling-leaves, has cost $f_1 + f_2$ plus the cost for a tree with $n - 1$ leaves of frequencies $(f_1 + f_2), f_3, \ldots, f_n$, etc.

Huffman Codes

Example 5.3. Huffman code



Huffman Codes

The resulting algorithm can be described in the terms of priority queue operations. We will use a binary heap for symbols $a_1, \ldots, a_n$ and keys $f_1, \ldots, f_n$ respectively.

We use one *Make-queue* operation to create the binary heap, during each iteration (the whole number of iterations is $n - 1$) we use two *Delete-min* operations to choose two lowest-frequency symbols a_i and a_j, we create new symbol with frequency $f_i + f_j$ and use one operation *Insert* to add this symbol to the heap. The overall time is $O(n \log n)$.

Matroids

We describe an approach to the study of greedy algorithms which is based on the notion of matroid and covers many cases when greedy algorithms give optimal solutions.

It is not universal, in particular, it does not cover algorithms of Dijkstra and Huffman.

Matroids

A *matroid* is a pair $M = (S, I)$ where S is a finite nonempty set and I is a nonempty family of subsets of S (called *independent* subsets of S) satisfying the following conditions:

1. If $B \in I$ and $A \subseteq B$ then $A \in I$. We say that I is *hereditary* if it satisfies this property. It is clear that $\emptyset \in I$.
2. If $A, B \in I$ and $|A| < |B|$ then there is an element $x \in B \setminus A$ such that $A \cup \{x\} \in I$. We say that M satisfies the *exchange property*.

Matroids

The *rank* of matroid is the maximum cardinality of an independent set.

The word "matroid" is due to Whitney who studied *matric matroids* in which S is the set of rows of a given matrix, and a set of rows is independent if they are linearly independent in the usual sense.

Matroids

Let $G = (V, E)$ be a connected undirected graph. We correspond to G the *graphic matroid* $M_G = (S_G, I_G)$ where $S_G = E$ and I_G is the family of acyclic subsets of of the set E.

A subset A of the set E is called *acyclic* if the graph (V, A) has no cycles (is a forest).

Theorem 5.4. *If $G = (V, E)$ is a connected undirected graph, then $M_G = (S_G, I_G)$ is a matroid.*

Matroids

Proof. It is clear that I_G is hereditary: a subset of an acyclic set of edges is an acyclic set.

Let $C \in I_G$. Then the graph (V, C) is a forest with $|V| - |C|$ trees.

At the beginning we have forest with $|V|$ trees, each of which is a single node without edges.

Each edge from C that is added to the forest reduces the number of trees by one.

Matroids

Let us show that M_G satisfies the exchange property. Let $A, B \in I_G$ and $|A| < |B|$.

The forest (V, B) contains less trees than the forest (V, A). Therefore a tree in the forest (V, B) contains nodes of two different trees from (V, A).

From here it follows that the set B contains an edge x which connects nodes from two different trees in (V, A). It is clear that $A \cup \{x\} \in I_G$. □

Matroids

Another example is a *uniform* matroid $U_{k,n} = (S, I)$ with rank k and with n elements.

For this matroid, S is a set with n elements, and a subset A of S is independent if and only if $|A| \leq k$.

Matroids

We will say that an independent subset A in a matroid M is *maximal* if there is no independent subset B in M such that $A \subset B$.

Theorem 5.5. *All maximal independent subsets in a matroid $M = (S, I)$ have the same cardinality.*

Proof. Let us assume the contrary: A, B are maximal independent subsets and $|A| < |B|$. Then, by the exchange property, the subset A is not maximal. □

Matroids

Let $G = (V, E)$ be a connected undirected graph.

Then, for the matroid $M_G = (S_G, I_G)$, the cardinality of each maximal independent subset from I_G is equal to $|V| - 1$, i.e., the rank of M_G is equal to $|V| - 1$.

A subset $A \subseteq E = S_G$ is a maximal independent set if and only if the graph (V, A) is a tree.

Matroids

We denote by $\mathbb{R}_{\geq 0}$ the set of all nonnegative real numbers.

A *weight function* for a matroid $M = (S, I)$ is a function $w : S \to \mathbb{R}_{\geq 0}$.

For a subset A of S, let $w(A) = \sum_{x \in A} w(x)$.

Matroids

We consider the following *optimization problem*: for a given matroid $M = (S, I)$ and a weight function $w : S \to \mathbb{R}_{\geq 0}$, we should find a subset $A \in I$ such that $w(A) = \max\{w(B) : B \in I\}$.

We will call such independent subsets *optimal* subsets.

Since M satisfies the exchange property and w has only nonnegative values, for each optimal subset A, there is a maximal independent subset B which is optimal and for which $A \subseteq B$.

Matroids

To solve the optimization problem, we consider the following *greedy algorithm*:

Let $a_1, \ldots, a_n$ be all elements from S ordered such that $w(a_1) \geq \ldots \geq w(a_n)$.

The algorithm constructs the sets $C_0, C_1, \ldots, C_n$, where $C_0 = \emptyset$, and, for $i = 1, \ldots, n$, $C_i = C_{i-1} \cup \{a_i\}$ if $C_{i-1} \cup \{a_i\} \in I$ and $C_i = C_{i-1}$, otherwise.

The algorithm returns the set $C = C_n$.

Matroids

Theorem 5.6. *For any matroid $M = (S, I)$ and any weight function $w : S \to \mathbb{R}_{\geq 0}$, the subset C constructed by the greedy algorithm is an optimal maximal independent subset of S.*

Proof. It is clear that C is an independent subset. Let us show that C is maximal.

Let us assume the contrary: there exists an independent set B such that $C \subset B$. Since M satisfies the exchange property, there exists $a_i \in B \setminus C$ such that $C \cup \{a_i\} \in I$.

Since I is hereditary, $C_{i-1} \cup \{a_i\} \in I$ but this is impossible: $C_{i-1} \cup \{a_i\} \notin I$ since $a_i \notin C$. Therefore C is a maximal independent subset of S.

Matroids

Let $C = \{a_{l_1}, \ldots, a_{l_k}\}$ where $l_1 < \ldots < l_k$. It is clear that $w(a_{l_1}) \geq \ldots \geq w(a_{l_k})$. Let $B = \{a_{t_1}, \ldots, a_{t_k}\}$ be an optimal maximal independent subset of S where $t_1 < \ldots < t_k$. It is clear that $w(a_{t_1}) \geq \ldots \geq w(a_{t_k})$.

Let us assume that $w(a_{t_i}) > w(a_{l_i})$ for some i, $1 \leq i \leq k$. We consider the sets $C' = \{a_{l_1}, \ldots, a_{l_{i-1}}\}$ (if $i = 1$ then $C' = \emptyset$) and $B' = \{a_{t_1}, \ldots, a_{t_i}\}$. Note that $C' = C_{l_i - 1}$.

Since M satisfies the exchange property, there is j, $1 \leq j \leq i$, such that $a_{t_j} \notin C'$ and $C' \cup \{a_{t_j}\} \in I$. We have $w(a_{t_j}) \geq w(a_{t_i}) > w(a_{l_i})$ but this is impossible since, by the description of the greedy algorithm, $w(a_{l_i}) = \max\{w(a_m) : m \in \{1, \ldots, n\}, C' \cup \{a_m\} \in I\}$.

Therefore, $w(a_{l_i}) \geq w(a_{t_i})$ for $i = 1, \ldots, k$, and C is an optimal maximal independent subset of S. □

Matroids

Theorem 5.7. *If $M = (S, I)$ is not a matroid then there exists a weight function $w : S \to \mathbb{R}_{\geq 0}$ such that the subset C constructed by the greedy algorithm is not an optimal independent subset of S.*

Proof. 1). Let I be not hereditary, i.e., there are nonempty subsets A and B of the set S such that $A \subset B$, $B \in I$ and $A \notin I$. Let us consider a weight function w such that, for any $a \in S$,

$$w(a) = \begin{cases} 2, & a \in A, \\ 1, & a \in B \setminus A, \\ 0, & a \in S \setminus B. \end{cases}$$

It is clear that B is an optimal independent subset. One can show that the set C constructed by the greedy algorithm will not include at least one element from A. Therefore C is not an optimal independent subset.

Matroids

2). Let I be hereditary but M do not satisfy the exchange property, i.e., there are nonempty subsets $A, B \in I$ such that $|A| < |B|$ but $A \cup \{a\} \notin I$ for any $a \in B \setminus A$. Let $|B| = m$.

We consider a weight function w such that, for any $a \in S$,

$$w(a) = \begin{cases} 1 + 1/m, & a \in A, \\ 1, & a \in B \setminus A, \\ 0, & a \in S \setminus (A \cup B). \end{cases}$$

The set C constructed by the greedy algorithms contains all elements from A and no elements from $B \setminus A$.

Since $|A| \leq m - 1$, we have $w(C) \leq m - 1 + (m-1)/m < m \leq w(B)$. Therefore, C is not optimal. □

Matroids

Theorem 5.6 and Theorem 5.7 give us a statement which is known as Theorem of Rado and Edmonds.

Matroids

Let we have a connected undirected graph $G = (V, E)$ in which each edge $e \in E$ is labeled with a cost $c_e \geq 0$.

We call a subset $T \subseteq E$ a spanning tree of G if (V, T) is a tree.

A spanning tree for which the total cost $\sum_{e \in T} c_e$ is as small as possible is called a minimum spanning tree.

Our goal is to construct a minimum spanning tree.

Matroids

We consider the graphic matroid $M_G = (S_G, I_G)$ where $S_G = E$ and the weight function $w : E \to \mathbb{R}_{\geq 0}$ such that $w(e) = p - c_e$ for any $e \in E$, and $p = \max\{c_e : e \in E\}$.

If we apply the greedy algorithm to the matroid M_G and to the weight function w, we obtain a minimum spanning tree.

This algorithm is very close to Kruskal's algorithm. We begin from the set of nodes V and empty set of edges. During each step, we add edge e which has minimum cost c_e among all edges that keep the constructed set of edges acyclic.