Greedy algorithms

Peter Hajnal

Bolyai Institute, University of Szeged, Hungary

2023 Fall

Basic Idea

Basic Idea

- We consider optimization problems. In F, the set of feasible solutions we must find an optimal element, i.e. an element where $c : F \to \mathbb{R}$, the objective function takes minimal/maximal value.

Basic Idea

- We consider optimization problems. In F, the set of feasible solutions we must find an optimal element, i.e. an element where $c : F \to \mathbb{R}$, the objective function takes minimal/maximal value.
- Very often the feasible solutions are subsets of a universe.

Basic Idea

- We consider optimization problems. In F, the set of feasible solutions we must find an optimal element, i.e. an element where $c : F \to \mathbb{R}$, the objective function takes minimal/maximal value.

- Very often the feasible solutions are subsets of a universe.

- An easy algorithm paste the elements of U (or the "surviving" elements of U). The MOST PROMISING element will be chosen as the next element of the output.

Basic Idea

- We consider optimization problems. In F, the set of feasible solutions we must find an optimal element, i.e. an element where $c : F \to \mathbb{R}$, the objective function takes minimal/maximal value.

- Very often the feasible solutions are subsets of a universe.

- An easy algorithm paste the elements of U (or the "surviving" elements of U). The MOST PROMISING element will be chosen as the next element of the output. After pasting all elements of U we will have an output.

Basic Idea: Why greedy?

Basic Idea: Why greedy?

- Taking the most promising element is a natural decision. The essence of greediness is that the above algorithm never overrules previous decisions. In spite of being a promising element at some point, later on we might realize that choosing that element is not a wise decision. A greedy algorithm do not step back.

Basic Idea: Why greedy?

- Taking the most promising element is a natural decision. The essence of greediness is that the above algorithm never overrules previous decisions. In spite of being a promising element at some point, later on we might realize that choosing that element is not a wise decision. A greedy algorithm do not step back.

- Greedy algorithms are very simple to implement. They are very fast. Unfortunately very often they are not able to guarantee that the output is optimal.

Basic Idea: Why greedy?

- Taking the most promising element is a natural decision. The essence of greediness is that the above algorithm never overrules previous decisions. In spite of being a promising element at some point, later on we might realize that choosing that element is not a wise decision. A greedy algorithm do not step back.

- Greedy algorithms are very simple to implement. They are very fast. Unfortunately very often they are not able to guarantee that the output is optimal.

- The above description is not a mathematical definition. It is a scheme, that very often leads to good algorithms. Sometimes (rarely) greediness makes us to be able to find the optimal solution.

Basic Idea: Why greedy?

- Taking the most promising element is a natural decision. The essence of greediness is that the above algorithm never overrules previous decisions. In spite of being a promising element at some point, later on we might realize that choosing that element is not a wise decision. A greedy algorithm do not step back.

- Greedy algorithms are very simple to implement. They are very fast. Unfortunately very often they are not able to guarantee that the output is optimal.

- The above description is not a mathematical definition. It is a scheme, that very often leads to good algorithms. Sometimes (rarely) greediness makes us to be able to find the optimal solution.

- Let us see an example.

Break

The minimal cost spanning tree problem

The problem

Given a connected graph, for each edge we have a positive cost ($c : E(G) \to \mathbb{R}_{++}$). This cost function can be naturally extended to subsets of $E(G)$ (the cost of an edge set is the sum of the costs of its elements).

The minimal cost spanning tree problem

The problem

Given a connected graph, for each edge we have a positive cost ($c : E(G) \to \mathbb{R}_{++}$). This cost function can be naturally extended to subsets of $E(G)$ (the cost of an edge set is the sum of the costs of its elements).
Find a cheapest spanning tree of the input graph (the tree is considered as a set of edges).

The following algorithm, first described by Kruskal is a „prototype" of the greedy algorithm design.

Kruskal's algorithm

Kruskal's algorithm (1956)

(SORTING STEP) Sort the edges of the input graph in ascending order of cost. Let $E(G) : e_1, e_2, \ldots, e_m$, i.e. e_1 is the cheapest edge, e_m is the most expensive edge $(c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m))$.

Kruskal's algorithm

Kruskal's algorithm (1956)

(SORTING STEP) Sort the edges of the input graph in ascending order of cost. Let $E(G) : e_1, e_2, \ldots, e_m$, i.e. e_1 is the cheapest edge, e_m is the most expensive edge ($c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$).
(INITIALIZATION) Let F be the set of edges selected so far. At the beginning of the algorithm $F = \emptyset$.

Kruskal's algorithm

Kruskal's algorithm (1956)

(SORTING STEP) Sort the edges of the input graph in ascending order of cost. Let $E(G) : e_1, e_2, \ldots, e_m$, i.e. e_1 is the cheapest edge, e_m is the most expensive edge ($c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$).
(INITIALIZATION) Let F be the set of edges selected so far. At the beginning of the algorithm $F = \emptyset$. // During the algorithm we only take care that F is an edge set without cycle.

Kruskal's algorithm

Kruskal's algorithm (1956)

(SORTING STEP) Sort the edges of the input graph in ascending order of cost. Let $E(G): e_1, e_2, \ldots, e_m$, i.e. e_1 is the cheapest edge, e_m is the most expensive edge $(c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m))$.

(INITIALIZATION) Let F be the set of edges selected so far. At the beginning of the algorithm $F = \emptyset$. // During the algorithm we only take care that F is an edge set without cycle.

(TESTS) In the i^{th} step, we examine e_i. If $F \cup \{e_i\}$ is cycle-free, then we extend F: $F \leftarrow F \cup \{e_i\}$. If $F \cup \{e_i\}$ contains a cycle, then we do not change F.

Kruskal's algorithm

Kruskal's algorithm (1956)

(SORTING STEP) Sort the edges of the input graph in ascending order of cost. Let $E(G) : e_1, e_2, \ldots, e_m$, i.e. e_1 is the cheapest edge, e_m is the most expensive edge ($c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$).
(INITIALIZATION) Let F be the set of edges selected so far. At the beginning of the algorithm $F = \emptyset$. // During the algorithm we only take care that F is an edge set without cycle.
(TESTS) In the i^{th} step, we examine e_i. If $F \cup \{e_i\}$ is cycle-free, then we extend F: $F \leftarrow F \cup \{e_i\}$. If $F \cup \{e_i\}$ contains a cycle, then we do not change F.
(OUTPUT) After examining the last edge, we announce the current F as the output.

The main theorem

The main theorem

One can say: all our choices are the best decision at the time. Later some of the edges have been discarded. After that it is possible that we need to throw away an edge. This is a questionable choice. It was based on the fact that the previously selected edges form a part of the output. If our previous decisions are overruled, then we could have chosen to use the currently discarded edge (the cheapest edge of the remaining edge set). The cost of the calculated spanning tree cannot simply be compared to the tree calculated above.

The main theorem

One can say: all our choices are the best decision at the time. Later some of the edges have been discarded. After that it is possible that we need to throw away an edge. This is a questionable choice. It was based on the fact that the previously selected edges form a part of the output. If our previous decisions are overruled, then we could have chosen to use the currently discarded edge (the cheapest edge of the remaining edge set). The cost of the calculated spanning tree cannot simply be compared to the tree calculated above.

Despite the huge question mark above, the calculated tree is optimal.

The main theorem

One can say: all our choices are the best decision at the time. Later some of the edges have been discarded. After that it is possible that we need to throw away an edge. This is a questionable choice. It was based on the fact that the previously selected edges form a part of the output. If our previous decisions are overruled, then we could have chosen to use the currently discarded edge (the cheapest edge of the remaining edge set). The cost of the calculated spanning tree cannot simply be compared to the tree calculated above.

Despite the huge question mark above, the calculated tree is optimal.

Theorem (Kruskal's Theorem)

The output of the above algorithm is a minimum cost spanning tree of the input graph.

Initial notations

Initial notations

- $n = |V(G)|$.

Initial notations

- $n = |V(G)|$.
- Let $e_1, e_2, \ldots, e_\ell$ be the elements of the output in the order of their selection. Specifically

$$c(e_1) \leq c(e_2) \leq \ldots \leq c(e_{\ell-1}) \leq c(e_\ell).$$

Initial notations

- $n = |V(G)|$.

- Let $e_1, e_2, \ldots, e_\ell$ be the elements of the output in the order of their selection. Specifically

$$c(e_1) \le c(e_2) \le \ldots \le c(e_{\ell-1}) \le c(e_\ell).$$

- Let F be an arbitrary spanning tree. We list the edges in increasing order of cost: $f_1, \ldots, f_{n-1}$, i.e.

$$c(f_1) \le c(f_2) \le \ldots \le c(f_{n-1}).$$

Initial notations

- $n = |V(G)|$.

- Let $e_1, e_2, \ldots, e_\ell$ be the elements of the output in the order of their selection. Specifically

$$c(e_1) \leq c(e_2) \leq \ldots \leq c(e_{\ell-1}) \leq c(e_\ell).$$

- Let F be an arbitrary spanning tree. We list the edges in increasing order of cost: $f_1, \ldots, f_{n-1}$, i.e.

$$c(f_1) \leq c(f_2) \leq \ldots \leq c(f_{n-1}).$$

- Based on the connectivity of the input graph it is easy to see that Kruskal's algorithm computes a spanning tree, i.e. $\ell = n - 1$.

Strong form of Kruskal's Theorem

Strong form of Kruskal's Theorem

Theorem (Strong form of Kruskal's Theorem)

For $i = 1, 2, \ldots, n-1$ we have

$$c(e_i) \le c(f_i).$$

The Main Lemma

The Main Lemma

Main Lemma

Let F, F' be two cycle-free edge sets over V. Suppose that $|F| < |F'|$. Then there is an edge e in $F' - F$ such that $F \cup \{e\}$ is also cycle-free.

The Main Lemma

Main Lemma

Let F, F' be two cycle-free edge sets over V. Suppose that $|F| < |F'|$. Then there is an edge e in $F' - F$ such that $F \cup \{e\}$ is also cycle-free.

- Let G_F be the cycle-free graph (forest) with vertex set V and edge set F.

The Main Lemma

Main Lemma

Let F, F' be two cycle-free edge sets over V. Suppose that $|F| < |F'|$. Then there is an edge e in $F' - F$ such that $F \cup \{e\}$ is also cycle-free.

- Let G_F be the cycle-free graph (forest) with vertex set V and edge set F.
- We know that G_F has $n - |F|$ components:

The Main Lemma

Main Lemma

Let F, F' be two cycle-free edge sets over V. Suppose that $|F| < |F'|$. Then there is an edge e in $F' - F$ such that $F \cup \{e\}$ is also cycle-free.

- Let G_F be the cycle-free graph (forest) with vertex set V and edge set F.
- We know that G_F has $n - |F|$ components: $c(G_F) = n - |F|$.

The Main Lemma

Main Lemma

Let F, F' be two cycle-free edge sets over V. Suppose that $|F| < |F'|$. Then there is an edge e in $F' - F$ such that $F \cup \{e\}$ is also cycle-free.

- Let G_F be the cycle-free graph (forest) with vertex set V and edge set F.
- We know that G_F has $n - |F|$ components: $c(G_F) = n - |F|$.
- Similarly let $G_{F'}$ be the cycle-free graph (forest) with vertex set V and edge set F'.

The Main Lemma

Main Lemma

Let F, F' be two cycle-free edge sets over V. Suppose that $|F| < |F'|$. Then there is an edge e in $F' - F$ such that $F \cup \{e\}$ is also cycle-free.

- Let G_F be the cycle-free graph (forest) with vertex set V and edge set F.
- We know that G_F has $n - |F|$ components: $c(G_F) = n - |F|$.
- Similarly let $G_{F'}$ be the cycle-free graph (forest) with vertex set V and edge set F'.
- We know that $G_{F'}$ has $n - |F'|$ components:

The Main Lemma

Main Lemma

Let F, F' be two cycle-free edge sets over V. Suppose that $|F| < |F'|$. Then there is an edge e in $F' - F$ such that $F \cup \{e\}$ is also cycle-free.

- Let G_F be the cycle-free graph (forest) with vertex set V and edge set F.
- We know that G_F has $n - |F|$ components: $c(G_F) = n - |F|$.
- Similarly let $G_{F'}$ be the cycle-free graph (forest) with vertex set V and edge set F'.
- We know that $G_{F'}$ has $n - |F'|$ components: $c(G_{F'}) = n - |F'| < n - |F| = c(G_F)$.

The Main Lemma

Main Lemma

Let F, F' be two cycle-free edge sets over V. Suppose that $|F| < |F'|$. Then there is an edge e in $F' - F$ such that $F \cup \{e\}$ is also cycle-free.

- Let G_F be the cycle-free graph (forest) with vertex set V and edge set F.
- We know that G_F has $n - |F|$ components: $c(G_F) = n - |F|$.
- Similarly let $G_{F'}$ be the cycle-free graph (forest) with vertex set V and edge set F'.
- We know that $G_{F'}$ has $n - |F'|$ components: $c(G_{F'}) = n - |F'| < n - |F| = c(G_F)$.
- This can only be imagined if there is an edge $e \in F'$ connecting two different components of G_F.

The Main Lemma

Main Lemma

Let F, F' be two cycle-free edge sets over V. Suppose that $|F| < |F'|$. Then there is an edge e in $F' - F$ such that $F \cup \{e\}$ is also cycle-free.

- Let G_F be the cycle-free graph (forest) with vertex set V and edge set F.
- We know that G_F has $n - |F|$ components: $c(G_F) = n - |F|$.
- Similarly let $G_{F'}$ be the cycle-free graph (forest) with vertex set V and edge set F'.
- We know that $G_{F'}$ has $n - |F'|$ components: $c(G_{F'}) = n - |F'| < n - |F| = c(G_F)$.
- This can only be imagined if there is an edge $e \in F'$ connecting two different components of G_F.
- Hence $e \notin F$

The Main Lemma

Main Lemma

Let F, F' be two cycle-free edge sets over V. Suppose that $|F| < |F'|$. Then there is an edge e in $F' - F$ such that $F \cup \{e\}$ is also cycle-free.

- Let G_F be the cycle-free graph (forest) with vertex set V and edge set F.
- We know that G_F has $n - |F|$ components: $c(G_F) = n - |F|$.
- Similarly let $G_{F'}$ be the cycle-free graph (forest) with vertex set V and edge set F'.
- We know that $G_{F'}$ has $n - |F'|$ components: $c(G_{F'}) = n - |F'| < n - |F| = c(G_F)$.
- This can only be imagined if there is an edge $e \in F'$ connecting two different components of G_F.
- Hence $e \notin F$ and $F \cup \{e\}$ is cycle-free.

Strong form of Kruskal's Theorem: The proof

Strong form of Kruskal's Theorem: The proof

- Induction on i.

Strong form of Kruskal's Theorem: The proof

- Induction on i.
- $c(e_1) \leq c(f_1)$.

Strong form of Kruskal's Theorem: The proof

- Induction on i.
- $c(e_1) \leq c(f_1)$.
- Assume that for $i(< n-1)$

$$c(e_1) \leq c(f_1), c(e_2) \leq c(f_2), \ldots, c(e_i) \leq c(f_i)$$

Strong form of Kruskal's Theorem: The proof

- Induction on i.
- $c(e_1) \leq c(f_1)$.
- Assume that for $i(< n-1)$

$$c(e_1) \leq c(f_1), c(e_2) \leq c(f_2), \ldots, c(e_i) \leq c(f_i)$$

- $F := \{e_1, \ldots, e_i\}$, $F' := \{f_1, \ldots, f_i, f_{i+1}\}$. Apply the Main Lemma.

Strong form of Kruskal's Theorem: The proof

- Induction on i.
- $c(e_1) \leq c(f_1)$.
- Assume that for $i(< n-1)$

$$c(e_1) \leq c(f_1), c(e_2) \leq c(f_2), \ldots, c(e_i) \leq c(f_i)$$

- $F := \{e_1, \ldots, e_i\}$, $F' := \{f_1, \ldots, f_i, f_{i+1}\}$. Apply the Main Lemma. $\rightarrow f \in F'$.

Strong form of Kruskal's Theorem: The proof

- Induction on i.
- $c(e_1) \leq c(f_1)$.
- Assume that for $i(< n-1)$

$$c(e_1) \leq c(f_1), c(e_2) \leq c(f_2), \ldots, c(e_i) \leq c(f_i)$$

- $F := \{e_1, \ldots, e_i\}$, $F' := \{f_1, \ldots, f_i, f_{i+1}\}$. Apply the Main Lemma. $\to f \in F'$.
- The Kruskal's algorithm tests e_{i+1} not later than f.

Strong form of Kruskal's Theorem: The proof

- Induction on i.
- $c(e_1) \leq c(f_1)$.
- Assume that for $i(< n-1)$

$$c(e_1) \leq c(f_1), c(e_2) \leq c(f_2), \ldots, c(e_i) \leq c(f_i)$$

- $F := \{e_1, \ldots, e_i\}$, $F' := \{f_1, \ldots, f_i, f_{i+1}\}$. Apply the Main Lemma. $\to f \in F'$.
- The Kruskal's algorithm tests e_{i+1} not later than f.
- $$c(e_{i+1}) \leq c(f) \leq c(f_{i+1}).$$

The analysis of Kruskal's algorithm

The analysis of Kruskal's algorithm

- The cost of the sorting step is

$$\mathcal{O}(|E|\log|E|).$$

The analysis of Kruskal's algorithm

- The cost of the sorting step is

$$\mathcal{O}(|E| \log |E|).$$

- The remaining part of algorithm is $|E|$ testing on an edge set.

The analysis of Kruskal's algorithm

- The cost of the sorting step is

$$\mathcal{O}(|E|\log|E|).$$

- The remaining part of algorithm is $|E|$ testing on an edge set. In each case the size of the edge set is is at most $|V|$.

The analysis of Kruskal's algorithm

- The cost of the sorting step is

$$\mathcal{O}(|E|\log|E|).$$

- The remaining part of algorithm is $|E|$ testing on an edge set. In each case the size of the edge set is is at most $|V|$. The cost of a test is $\mathcal{O}(|V|)$.

The analysis of Kruskal's algorithm

- The cost of the sorting step is

$$\mathcal{O}(|E|\log|E|).$$

- The remaining part of algorithm is $|E|$ testing on an edge set. In each case the size of the edge set is is at most $|V|$. The cost of a test is $\mathcal{O}(|V|)$.

- Cost of the complete run

$$\mathcal{O}(|E|\log|E|)+|E|\cdot\mathcal{O}(|V|)=\mathcal{O}(|E|\cdot|V|).$$

The analysis of Kruskal's algorithm

- The cost of the sorting step is

$$\mathcal{O}(|E|\log|E|).$$

- The remaining part of algorithm is $|E|$ testing on an edge set. In each case the size of the edge set is is at most $|V|$. The cost of a test is $\mathcal{O}(|V|)$.

- Cost of the complete run

$$\mathcal{O}(|E|\log|E|)+|E|\cdot\mathcal{O}(|V|)=\mathcal{O}(|E|\cdot|V|).$$

- We have performed an analysis of a naive implementation. There are more clever solutions.

Break

The basic question

The basic question

The computational problem

The basic question

The computational problem

Given

The basic question

The computational problem

Given

(i) $\overrightarrow{G}$ directed graph,

The basic question

The computational problem

Given

(i) $\overrightarrow{G}$ directed graph,

(ii) $\ell : E(\overrightarrow{G}) \to \mathbb{R}_{++}$ length function,

The basic question

The computational problem

Given

(i) $\overrightarrow{G}$ directed graph,

(ii) $\ell : E(\overrightarrow{G}) \to \mathbb{R}_{++}$ length function,

(iii) s, t two distinguished.

The basic question

The computational problem

Given

(i) $\overrightarrow{G}$ directed graph,

(ii) $\ell : E(\overrightarrow{G}) \to \mathbb{R}_{++}$ length function,

(iii) s, t two distinguished.

Determine the distance of s and t.

Refreshing memory

Refreshing memory

Definition: Walk in a directed graph

$\overrightarrow{uv}$-walk in $\overrightarrow{G}$:

$$\overrightarrow{\mathcal{S}} : u = w_0, \overrightarrow{e}_1, w_1, \overrightarrow{e}_2, \ldots, w_{L-1}, \overrightarrow{e}_L, w_L = v,$$

Refreshing memory

Definition: Walk in a directed graph

$\overrightarrow{uv}$-walk in $\overrightarrow{G}$:

$$\overrightarrow{\mathcal{S}} : u = w_0, \overrightarrow{e}_1, w_1, \overrightarrow{e}_2, \ldots, w_{L-1}, \overrightarrow{e}_L, w_L = v,$$

where $w_i \in V$ $(i = 0, 1, \ldots, L)$,

Refreshing memory

Definition: Walk in a directed graph

$\overrightarrow{uv}$-walk in $\overrightarrow{G}$:

$$\overrightarrow{\mathcal{S}} : u = w_0, \overrightarrow{e}_1, w_1, \overrightarrow{e}_2, \ldots, w_{L-1}, \overrightarrow{e}_L, w_L = v,$$

where $w_i \in V$ $(i = 0, 1, \ldots, L)$, $\overrightarrow{e}_i \in E$ $(i = 1, \ldots, L)$,

Refreshing memory

Definition: Walk in a directed graph

$\overrightarrow{uv}$-walk in $\overrightarrow{G}$:

$$\overrightarrow{\mathcal{S}} : u = w_0, \overrightarrow{e}_1, w_1, \overrightarrow{e}_2, \ldots, w_{L-1}, \overrightarrow{e}_L, w_L = v,$$

where $w_i \in V$ $(i = 0, 1, \ldots, L)$, $\overrightarrow{e}_i \in E$ $(i = 1, \ldots, L)$, e_i is an outgoing edge from w_{i-1}, and ingoing edge in w_i $(i = 1, \ldots, L)$.

Refreshing memory

Definition: Walk in a directed graph

$\overrightarrow{uv}$-walk in $\overrightarrow{G}$:

$$\overrightarrow{\mathcal{S}} : u = w_0, \overrightarrow{e}_1, w_1, \overrightarrow{e}_2, \ldots, w_{L-1}, \overrightarrow{e}_L, w_L = v,$$

where $w_i \in V$ $(i = 0, 1, \ldots, L)$, $\overrightarrow{e}_i \in E$ $(i = 1, \ldots, L)$, e_i is an outgoing edge from w_{i-1}, and ingoing edge in w_i $(i = 1, \ldots, L)$.

In the case of the existence of an $\overrightarrow{uv}$-walk we say that v is reachable from u.

Refreshing memory

Definition: Walk in a directed graph

$\overrightarrow{uv}$-walk in $\overrightarrow{G}$:

$$\overrightarrow{\mathcal{S}} : u = w_0, \overrightarrow{e}_1, w_1, \overrightarrow{e}_2, \ldots, w_{L-1}, \overrightarrow{e}_L, w_L = v,$$

where $w_i \in V$ $(i = 0, 1, \ldots, L)$, $\overrightarrow{e}_i \in E$ $(i = 1, \ldots, L)$, e_i is an outgoing edge from w_{i-1}, and ingoing edge in w_i $(i = 1, \ldots, L)$.

In the case of the existence of an $\overrightarrow{uv}$-walk we say that v is reachable from u.

The graph theoretical length of $\overrightarrow{\mathcal{S}}$, an $\overrightarrow{uv}$-walk is L.

Refreshing memory (cont'd)

Refreshing memory (cont'd)

Length of a walk in a weighted graph

The (weighted) length of $\overrightarrow{\mathcal{S}}$, an $\overrightarrow{uv}$-walk is

$$\sum_{i=1}^{L} \ell(\overrightarrow{e}_i).$$

Refreshing memory (cont'd)

Length of a walk in a weighted graph

The (weighted) length of $\overrightarrow{\mathcal{S}}$, an $\overrightarrow{uv}$-walk is

$$\sum_{i=1}^{L} \ell(\overrightarrow{e}_i).$$

Definition: The distance of two vertices in a weighted graph

$d(u,v)$ denotes the distance of two vertices u and v, that is the minimal length among the $\overrightarrow{uv}$-walks.

Refreshing memory (cont'd)

Length of a walk in a weighted graph

The (weighted) length of $\overrightarrow{\mathcal{S}}$, an $\overrightarrow{uv}$-walk is

$$\sum_{i=1}^{L} \ell(\overrightarrow{e}_i).$$

Definition: The distance of two vertices in a weighted graph

$d(u, v)$ denotes the distance of two vertices u and v, that is the minimal length among the $\overrightarrow{uv}$-walks. // $d(u, v) = \infty$ is possible!

Refreshing memory (cont'd)

Length of a walk in a weighted graph

The (weighted) length of $\overrightarrow{\mathcal{S}}$, an $\overrightarrow{uv}$-walk is

$$\sum_{i=1}^{L} \ell(\overrightarrow{e}_i).$$

Definition: The distance of two vertices in a weighted graph

$d(u, v)$ denotes the distance of two vertices u and v, that is the minimal length among the $\overrightarrow{uv}$-walks. // $d(u, v) = \infty$ is possible!

Observation

The shortest $\overrightarrow{uv}$-walk will be a path.

Initial remarks

Initial remarks

- We assume the our graph has no loop.

Initial remarks

- We assume the our graph has no loop. We assume that for any two vertices there is at most one edge from u to v.

Initial remarks

• We assume the our graph has no loop. We assume that for any two vertices there is at most one edge from u to v. I.e. we assume that $\overrightarrow{G}$ is simple (in directed sense).

The case of graph theoretical distance: unweighted case

The case of graph theoretical distance: unweighted case

- We solve a harder problem.

The case of graph theoretical distance: unweighted case

- We solve a harder problem. The input will be $(\overrightarrow{G}, s)$.

The case of graph theoretical distance: unweighted case

• We solve a harder problem. The input will be $(\overrightarrow{G}, s)$. We determine S, the set of vertices that are reachable from s.

The case of graph theoretical distance: unweighted case

- We solve a harder problem. The input will be $(\overrightarrow{G}, s)$. We determine S, the set of vertices that are reachable from s.
- The set of reachable vertices will be $S = S_0 \dot\cup S_1 \dot\cup \ldots \dot\cup S_L$,

The case of graph theoretical distance: unweighted case

• We solve a harder problem. The input will be $(\overrightarrow{G}, s)$. We determine S, the set of vertices that are reachable from s.

• The set of reachable vertices will be $S = S_0 \dot\cup S_1 \dot\cup \ldots \dot\cup S_L$, where S_i contains exactly those vertices that are at distance i from s.

The case of graph theoretical distance: unweighted case

- We solve a harder problem. The input will be $(\overrightarrow{G}, s)$. We determine S, the set of vertices that are reachable from s.
- The set of reachable vertices will be $S = S_0 \dot\cup S_1 \dot\cup \ldots \dot\cup S_L$, where S_i contains exactly those vertices that are at distance i from s.
- L denotes the length of longest path starting at s.

The case of graph theoretical distance: unweighted case

- We solve a harder problem. The input will be $(\overrightarrow{G}, s)$. We determine S, the set of vertices that are reachable from s.
- The set of reachable vertices will be $S = S_0 \dot\cup S_1 \dot\cup \ldots \dot\cup S_L$, where S_i contains exactly those vertices that are at distance i from s.
- L denotes the length of longest path starting at s.
- Each edge between S and $\overline{S} = V(G) - S$ are oriented from $\overline{S}$ to S.

The case of graph theoretical distance: unweighted case

- We solve a harder problem. The input will be $(\overrightarrow{G}, s)$. We determine S, the set of vertices that are reachable from s.
- The set of reachable vertices will be $S = S_0 \dot\cup S_1 \dot\cup \ldots \dot\cup S_L$, where S_i contains exactly those vertices that are at distance i from s.
- L denotes the length of longest path starting at s.
- Each edge between S and $\overline{S} = V(G) - S$ are oriented from $\overline{S}$ to S. This property will be a proof of the fact that the elements of $\overline{S}$ are not reachable from s.

Breadth first search algorithm

Breadth first search

Breadth first search algorithm

Breadth first search

(I) // Inicialization // Let $S_0 = \{s\}$.

Breadth first search algorithm

Breadth first search

(I) // Inicialization // Let $S_0 = \{s\}$.

(E) // Extension cycle // While $S_i \neq \emptyset$

- $$S_{i+1} = \{x \in V(G) - (S_0 \cup \ldots \cup S_i) : \text{ there is } \sigma \in S_i, \text{ such that } \overrightarrow{\sigma x} \in E(G)\}$$
- $i \leftarrow i + 1$.

end-while

The correctness of breadth first search algorithm

The correctness of breadth first search algorithm

Theorem

The above algorithm is correct. I.e.

The correctness of breadth first search algorithm

Theorem

The above algorithm is correct. I.e.

(i) In the case of $x \in S_i$ the graph theoretical distance of s and x is i.

The correctness of breadth first search algorithm

Theorem

The above algorithm is correct. I.e.

(i) In the case of $x \in S_i$ the graph theoretical distance of s and x is i.

(ii) If $x \notin S$, then x is not reachable from s.

Breadth first search tree

See the following modification of the algorithm:

Breadth first search tree

See the following modification of the algorithm:

Breadth first search tree (modification)

(I*) $F := \emptyset$

Breadth first search tree

See the following modification of the algorithm:

Breadth first search tree (modification)

(I*) $F := \emptyset$

(E$^+$) In cycle (E), when we insert x into S_{i+1} ($i \geq 0$), then the algorithm search for a suitable $\sigma \in S_i$ and find one: $\sigma_x \in S_i$.

Breadth first search tree

See the following modification of the algorithm:

Breadth first search tree (modification)

(I*) $F := \emptyset$

(E$^+$) In cycle (E), when we insert x into S_{i+1} ($i \geq 0$), then the algorithm search for a suitable $\sigma \in S_i$ and find one: $\sigma_x \in S_i$.
// Suitable means that $\overrightarrow{\sigma_x x}$ is an edge.

Breadth first search tree

See the following modification of the algorithm:

Breadth first search tree (modification)

(I*) $F := \emptyset$

(E$^+$) In cycle (E), when we insert x into S_{i+1} ($i \geq 0$), then the algorithm search for a suitable $\sigma \in S_i$ and find one: $\sigma_x \in S_i$.
// Suitable means that $\overrightarrow{\sigma_x x}$ is an edge.

(Extending F) $F \leftarrow F \cup \{\overrightarrow{\sigma_x x}\}$

Breadth first search tree

See the following modification of the algorithm:

Breadth first search tree (modification)

(I*) $F := \emptyset$

(E$^+$) In cycle (E), when we insert x into S_{i+1} ($i \geq 0$), then the algorithm search for a suitable $\sigma \in S_i$ and find one: $\sigma_x \in S_i$.
// Suitable means that $\overrightarrow{\sigma_x x}$ is an edge.

(Extending F) $F \leftarrow F \cup \{\overrightarrow{\sigma_x x}\}$

Theorem

In the graph $G|_S$ the edge set F will be the edge set of a spanning tree $\mathcal{F}$. $(\mathcal{F}, s)$ is a rooted tree.

Breadth first search tree

See the following modification of the algorithm:

Breadth first search tree (modification)

(I*) $F := \emptyset$

(E$^+$) In cycle (E), when we insert x into S_{i+1} ($i \geq 0$), then the algorithm search for a suitable $\sigma \in S_i$ and find one: $\sigma_x \in S_i$.
// Suitable means that $\overrightarrow{\sigma_x x}$ is an edge.

(Extending F) $F \leftarrow F \cup \{\overrightarrow{\sigma_x x}\}$

Theorem

In the graph $G|_S$ the edge set F will be the edge set of a spanning tree $\mathcal{F}$. $(\mathcal{F}, s)$ is a rooted tree.

For each vertex $x \in S$ there is exactly one $\overrightarrow{sx}$-path in $\mathcal{F}$. This one of the (graph theoretically) shortest $\overrightarrow{sx}$-path.

Weighted case: The basic idea

Weighted case: The basic idea

- We also assume that all vertices are reachable from s.

Weighted case: The basic idea

- We also assume that all vertices are reachable from s.
- Assume that for a vertex set $S \subset V(\overrightarrow{G})$ we have

Weighted case: The basic idea

- We also assume that all vertices are reachable from s.
- Assume that for a vertex set $S \subset V(\overrightarrow{G})$ we have

(o) $s \in S$,

Weighted case: The basic idea

- We also assume that all vertices are reachable from s.
- Assume that for a vertex set $S \subset V(\overrightarrow{G})$ we have

(o) $s \in S$,

(i) for each $v \in S$ we have the shortest $\overrightarrow{sv}$ path, and that is inside S.

Weighted case: The basic idea

- We also assume that all vertices are reachable from s.
- Assume that for a vertex set $S \subset V(\overrightarrow{G})$ we have

(o) $s \in S$,

(i) for each $v \in S$ we have the shortest $\overrightarrow{sv}$ path, and that is inside S.

- The greedy algorithm chooses the most promising vertex from $\overline{S} := V(G) \setminus S$. For this we assume that the following information is also computed:

Weighted case: The basic idea

- We also assume that all vertices are reachable from s.
- Assume that for a vertex set $S \subset V(\overrightarrow{G})$ we have

(o) $s \in S$,

(i) for each $v \in S$ we have the shortest $\overrightarrow{sv}$ path, and that is inside S.

- The greedy algorithm chooses the most promising vertex from $\overline{S} := V(G) \setminus S$. For this we assume that the following information is also computed:

Weighted case: The basic idea

- We also assume that all vertices are reachable from s.
- Assume that for a vertex set $S \subset V(\overrightarrow{G})$ we have

(o) $s \in S$,

(i) for each $v \in S$ we have the shortest $\overrightarrow{sv}$ path, and that is inside S.

- The greedy algorithm chooses the most promising vertex from $\overline{S} := V(G) \setminus S$. For this we assume that the following information is also computed:

(ii) For each $v \in \overline{S}$ we have the shortest $\overrightarrow{sv}$ path, and that is inside S except the last $\overrightarrow{v^- v}$ step ($v^- \in S$).

Weighted case: The basic idea

- We also assume that all vertices are reachable from s.
- Assume that for a vertex set $S \subset V(\overrightarrow{G})$ we have

(o) $s \in S$,

(i) for each $v \in S$ we have the shortest $\overrightarrow{sv}$ path, and that is inside S.

- The greedy algorithm chooses the most promising vertex from $\overline{S} := V(G) \setminus S$. For this we assume that the following information is also computed:

(ii) For each $v \in \overline{S}$ we have the shortest $\overrightarrow{sv}$ path, and that is inside S except the last $\overrightarrow{v^- v}$ step ($v^- \in S$).

- One should think about the promised information as a labeling, $c : V(\overrightarrow{G}) \to \mathbb{R} \cup \{\infty\}$, of the vertices.

Dijkstra's algorithm

Dijkstra's algorithm

Dijkstra's algorithm

Dijkstra's algorithm

Dijkstra's algorithm

(I) // Inicialization

Dijkstra's algorithm

Dijkstra's algorithm

(I) // Inicialization
$S = \{s\}$, $c(s) = 0$.

Dijkstra's algorithm

Dijkstra's algorithm

(I) // Inicialization
$S = \{s\}$, $c(s) = 0$.
In the case of $v \notin S$, $\overrightarrow{sv} \in E$ we have $c(v) = \ell(\overrightarrow{sv})$.

Dijkstra's algorithm

Dijkstra's algorithm

(I) // Inicialization
$S = \{s\}$, $c(s) = 0$.
In the case of $v \notin S$, $\overrightarrow{sv} \in E$ we have $c(v) = \ell(\overrightarrow{sv})$.
In the case of $v \notin S$, $\overrightarrow{sv} \notin E$ we have $c(v) = \infty$.

Dijkstra's algorithm

Dijkstra's algorithm

(I) // Inicialization
$S = \{s\}$, $c(s) = 0$.
In the case of $v \notin S$, $\overrightarrow{sv} \in E$ we have $c(v) = \ell(\overrightarrow{sv})$.
In the case of $v \notin S$, $\overrightarrow{sv} \notin E$ we have $c(v) = \infty$.

(E) // Extension (until $\overline{S} \neq \emptyset$)

Dijkstra's algorithm

Dijkstra's algorithm

(I) // Inicialization
$S = \{s\}$, $c(s) = 0$.
In the case of $v \notin S$, $\overrightarrow{sv} \in E$ we have $c(v) = \ell(\overrightarrow{sv})$.
In the case of $v \notin S$, $\overrightarrow{sv} \notin E$ we have $c(v) = \infty$.

(E) // Extension (until $\overline{S} \neq \emptyset$)
$m \in \overline{S}$ with smallest label in $\overrightarrow{S}$.

Dijkstra's algorithm

Dijkstra's algorithm

(I) // Inicialization
$S = \{s\}$, $c(s) = 0$.
In the case of $v \notin S$, $\overrightarrow{sv} \in E$ we have $c(v) = \ell(\overrightarrow{sv})$.
In the case of $v \notin S$, $\overrightarrow{sv} \notin E$ we have $c(v) = \infty$.

(E) // Extension (until $\overline{S} \neq \emptyset$)
$m \in \overline{S}$ with smallest label in $\overrightarrow{S}$.
$S \leftarrow S \cup \{m\}$.

Dijkstra's algorithm

Dijkstra's algorithm

(I) // Inicialization
$S = \{s\}$, $c(s) = 0$.
In the case of $v \notin S$, $\overrightarrow{sv} \in E$ we have $c(v) = \ell(\overrightarrow{sv})$.
In the case of $v \notin S$, $\overrightarrow{sv} \notin E$ we have $c(v) = \infty$.

(E) // Extension (until $\overline{S} \neq \emptyset$)
$m \in \overline{S}$ with smallest label in $\overrightarrow{S}$.
$S \leftarrow S \cup \{m\}$.
// $\overline{S} \leftarrow \overline{S} \setminus \{m\}$.

Dijkstra's algorithm

Dijkstra's algorithm

(I) // Inicialization
$S = \{s\}$, $c(s) = 0$.
In the case of $v \notin S$, $\overrightarrow{sv} \in E$ we have $c(v) = \ell(\overrightarrow{sv})$.
In the case of $v \notin S$, $\overrightarrow{sv} \notin E$ we have $c(v) = \infty$.

(E) // Extension (until $\overline{S} \neq \emptyset$)
$m \in \overline{S}$ with smallest label in $\overrightarrow{S}$.
$S \leftarrow S \cup \{m\}$.
// $\overline{S} \leftarrow \overline{S} \setminus \{m\}$.

(U) // Update

Dijkstra's algorithm

Dijkstra's algorithm

(I) // Inicialization
$S = \{s\}$, $c(s) = 0$.
In the case of $v \notin S$, $\overrightarrow{sv} \in E$ we have $c(v) = \ell(\overrightarrow{sv})$.
In the case of $v \notin S$, $\overrightarrow{sv} \notin E$ we have $c(v) = \infty$.

(E) // Extension (until $\overline{S} \neq \emptyset$)
$m \in \overline{S}$ with smallest label in $\overrightarrow{S}$.
$S \leftarrow S \cup \{m\}$.
// $\overline{S} \leftarrow \overline{S} \setminus \{m\}$.

(U) // Update
We might change the label of $n \in \overline{S}$ in the case of $\overrightarrow{mn} \in E$:

$$c_{\text{new}}(n) = \min\{c_{\text{old}}(n), c(m) + \ell(\overrightarrow{mn})\}.$$

Dijkstra's algorithm

Dijkstra's algorithm

(I) // Inicialization
$S = \{s\}$, $c(s) = 0$.
In the case of $v \notin S$, $\overrightarrow{sv} \in E$ we have $c(v) = \ell(\overrightarrow{sv})$.
In the case of $v \notin S$, $\overrightarrow{sv} \notin E$ we have $c(v) = \infty$.

(E) // Extension (until $\overline{S} \neq \emptyset$)
$m \in \overline{S}$ with smallest label in $\overrightarrow{S}$.
$S \leftarrow S \cup \{m\}$.
// $\overline{S} \leftarrow \overline{S} \setminus \{m\}$.

(U) // Update
We might change the label of $n \in \overline{S}$ in the case of $\overrightarrow{mn} \in E$:

$$c_{\text{new}}(n) = \min\{c_{\text{old}}(n), c(m) + \ell(\overrightarrow{mn})\}.$$

If $t \notin S$ (or $\overline{S} \neq \emptyset$) back to (E).

Correctness of the algorithm

Correctness of the algorithm

Theorem

In each update step Dijkstra's algorithm computes a correct labeling. I.e.

Correctness of the algorithm

Theorem

In each update step Dijkstra's algorithm computes a correct labeling. I.e.

(i) The label of $v \in S$ is the length of shortest $\overrightarrow{sv}$ path. Furthermore this shortest length can be realized by a $\overrightarrow{sv}$-path inside S.

Correctness of the algorithm

Theorem

In each update step Dijkstra's algorithm computes a correct labeling. I.e.

(i) The label of $v \in S$ is the length of shortest $\overrightarrow{sv}$ path. Furthermore this shortest length can be realized by a $\overrightarrow{sv}$-path inside S.

(ii) The label of $v \in \overline{S}$ is the length of shortest $\overrightarrow{sv}$ path, that leaves S only when it makes the last step.

Correctness of the algorithm

Theorem

In each update step Dijkstra's algorithm computes a correct labeling. I.e.

(i) The label of $v \in S$ is the length of shortest $\overrightarrow{sv}$ path. Furthermore this shortest length can be realized by a $\overrightarrow{sv}$-path inside S.

(ii) The label of $v \in \overline{S}$ is the length of shortest $\overrightarrow{sv}$ path, that leaves S only when it makes the last step.

- We prove by mathematical induction on $|S|$.

Correctness of the algorithm

Theorem

In each update step Dijkstra's algorithm computes a correct labeling. I.e.

(i) The label of $v \in S$ is the length of shortest $\overrightarrow{sv}$ path. Furthermore this shortest length can be realized by a $\overrightarrow{sv}$-path inside S.

(ii) The label of $v \in \overline{S}$ is the length of shortest $\overrightarrow{sv}$ path, that leaves S only when it makes the last step.

• We prove by mathematical induction on $|S|$. Case $|S| = 1$ is straight forward.

Correctness of the algorithm

Theorem

In each update step Dijkstra's algorithm computes a correct labeling. I.e.

(i) The label of $v \in S$ is the length of shortest $\overrightarrow{sv}$ path. Furthermore this shortest length can be realized by a $\overrightarrow{sv}$-path inside S.

(ii) The label of $v \in \overline{S}$ is the length of shortest $\overrightarrow{sv}$ path, that leaves S only when it makes the last step.

- We prove by mathematical induction on $|S|$. Case $|S| = 1$ is straight forward.
- For the induction step we need to prove two claims:

Correctness of the algorithm

Theorem

In each update step Dijkstra's algorithm computes a correct labeling. I.e.

(i) The label of $v \in S$ is the length of shortest $\overrightarrow{sv}$ path. Furthermore this shortest length can be realized by a $\overrightarrow{sv}$-path inside S.

(ii) The label of $v \in \overline{S}$ is the length of shortest $\overrightarrow{sv}$ path, that leaves S only when it makes the last step.

- We prove by mathematical induction on $|S|$. Case $|S| = 1$ is straight forward.

- For the induction step we need to prove two claims:

(i) The label of m is correct,

Correctness of the algorithm

Theorem

In each update step Dijkstra's algorithm computes a correct labeling. I.e.

(i) The label of $v \in S$ is the length of shortest $\overrightarrow{sv}$ path. Furthermore this shortest length can be realized by a $\overrightarrow{sv}$-path inside S.

(ii) The label of $v \in \overline{S}$ is the length of shortest $\overrightarrow{sv}$ path, that leaves S only when it makes the last step.

- We prove by mathematical induction on $|S|$. Case $|S| = 1$ is straight forward.

- For the induction step we need to prove two claims:

(i) The label of m is correct,

(ii) The label of $x \in \overline{S}$ is correct.

Analysis of the algorithm

Analysis of the algorithm

- $\overline{S}_{\text{fin}}$ denotes the set of nodes from $\overline{S}$ with finite label.

Analysis of the algorithm

- $\overline{S}_{\text{fin}}$ denotes the set of nodes from $\overline{S}$ with finite label.
- Extension step: Delete the node with minimum label from $\overline{S}_{\text{fin}}$.

Analysis of the algorithm

- $\overline{S}_{\text{fin}}$ denotes the set of nodes from $\overline{S}$ with finite label.

- Extension step: Delete the node with minimum label from $\overline{S}_{\text{fin}}$.

The number of extension steps is at most $|V| - 1$.

Analysis of the algorithm

• $\overline{S}_{\text{fin}}$ denotes the set of nodes from $\overline{S}$ with finite label.

• Extension step: Delete the node with minimum label from $\overline{S}_{\text{fin}}$.
The number of extension steps is at most $|V|-1$.

• Label updates steps: Some vertex with label "∞" enters $\overline{S}_{\text{fin}}$.
Some labels in $\overline{S}_{\text{fin}}$ will be decreased by a value $\delta(>0)$.

Analysis of the algorithm

• $\overline{S}_{\text{fin}}$ denotes the set of nodes from $\overline{S}$ with finite label.

• Extension step: Delete the node with minimum label from $\overline{S}_{\text{fin}}$.
The number of extension steps is at most $|V| - 1$.

• Label updates steps: Some vertex with label "∞" enters $\overline{S}_{\text{fin}}$.
Some labels in $\overline{S}_{\text{fin}}$ will be decreased by a value $\delta(> 0)$.
Computation for label updates is needed for at most $|E|$ times.

Analysis of the algorithm

• $\overline{S}_{\text{fin}}$ denotes the set of nodes from $\overline{S}$ with finite label.

• Extension step: Delete the node with minimum label from $\overline{S}_{\text{fin}}$.
The number of extension steps is at most $|V| - 1$.

• Label updates steps: Some vertex with label "∞" enters $\overline{S}_{\text{fin}}$.
Some labels in $\overline{S}_{\text{fin}}$ will be decreased by a value $\delta (> 0)$.
Computation for label updates is needed for at most $|E|$ times.

• The total number of steps is

$$\mathcal{O}(|V|^2 + |E|).$$

Analysis of the algorithm

- $\overline{S}_{\text{fin}}$ denotes the set of nodes from $\overline{S}$ with finite label.

- Extension step: Delete the node with minimum label from $\overline{S}_{\text{fin}}$.
The number of extension steps is at most $|V| - 1$.

- Label updates steps: Some vertex with label "∞" enters $\overline{S}_{\text{fin}}$.
Some labels in $\overline{S}_{\text{fin}}$ will be decreased by a value $\delta(> 0)$.
Computation for label updates is needed for at most $|E|$ times.

- The total number of steps is

$$\mathcal{O}(|V|^2 + |E|).$$

- The above argument followed a naive implementation.

Analysis of the algorithm

• $\overline{S}_{\text{fin}}$ denotes the set of nodes from $\overline{S}$ with finite label.

• Extension step: Delete the node with minimum label from $\overline{S}_{\text{fin}}$.
The number of extension steps is at most $|V|-1$.

• Label updates steps: Some vertex with label "∞" enters $\overline{S}_{\text{fin}}$.
Some labels in $\overline{S}_{\text{fin}}$ will be decreased by a value $\delta(>0)$.
Computation for label updates is needed for at most $|E|$ times.

• The total number of steps is

$$\mathcal{O}(|V|^2+|E|).$$

• The above argument followed a naive implementation. There are cleverer ways to implement Dijkstra's high level description.

Final remarks

Final remarks

- Every label has a corresponding edge, that is responsible for its value.

Final remarks

- Every label has a corresponding edge, that is responsible for its value.

- Changing the value of the label/updating: changing the "responsible" node too.

Final remarks

• Every label has a corresponding edge, that is responsible for its value.

• Changing the value of the label/updating: changing the "responsible" node too.

• If we keep track of these edges responsible for the actual value we will obtain a rooted, directed spanning tree of the original graph (see the computation of breadth first search tree). This tree maintains/contains for each vertex a shortest path leading to that vertex.

Break

Coding texts

Coding texts

Definition of Text

Let Σ be a finite alphabet. The elements of Σ are called characters.

Coding texts

Definition of Text

Let Σ be a finite alphabet. The elements of Σ are called characters. A *text* is a finite sequence of characters($\in \Sigma^\star$).

Coding texts

Definition of Text

Let Σ be a finite alphabet. The elements of Σ are called characters. A *text* is a finite sequence of characters($\in \Sigma^\star$).

The length of a text is the number of characters in the text.

Coding texts

Definition of Text

Let Σ be a finite alphabet. The elements of Σ are called characters. A *text* is a finite sequence of characters($\in \Sigma^\star$).

The length of a text is the number of characters in the text.

Coding texts

An algorithm/function $c : \Sigma^\star \to \{0, 1\}^\star$

Coding texts

Definition of Text

Let Σ be a finite alphabet. The elements of Σ are called characters.
A *text* is a finite sequence of characters($\in \Sigma^\star$).

The length of a text is the number of characters in the text.

Coding texts

An algorithm/function $c : \Sigma^\star \to \{0,1\}^\star$

$$+$$

decoding algorithm $d : \{0,1\}^\star \to \Sigma^\star$.

Coding texts

Definition of Text

Let Σ be a finite alphabet. The elements of Σ are called characters. A *text* is a finite sequence of characters($\in \Sigma^\star$).

The length of a text is the number of characters in the text.

Coding texts

An algorithm/function $c : \Sigma^\star \to \{0,1\}^\star$

$$+$$

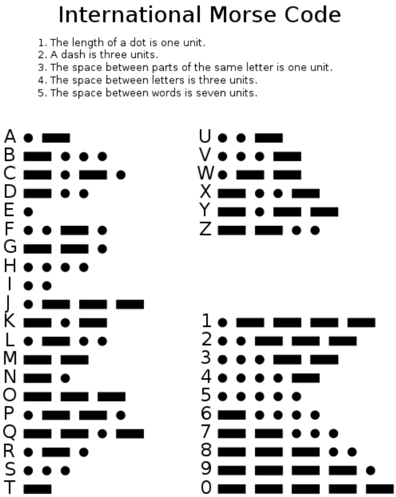
decoding algorithm $d : \{0,1\}^\star \to \Sigma^\star$.

Character based coding: $c_0 : \Sigma \to \{0,1\}^\star$ coding of characters. The code of a text is obtained by "putting together" the codes of its characters.

Fixed-length codes, Example: ASCII (1972) (source: wiki)

Bits $b_7 b_6 b_5$ →					000	001	010	011	100	101	110	111
b_4 ↓	b_3 ↓	b_2 ↓	b_1 ↓	Column → Row ↓	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	´	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	¦
1	1	0	1	13	CR	GS	—	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

Variable-length codes, Example: Morse code (1837–44) (source:wiki)



Variable-length codes without comma: Prefix codes

Variable-length codes without comma: Prefix codes

Definition

Rooted binary plane tree.

Variable-length codes without comma: Prefix codes

Definition

Rooted binary plane tree.
Leaf of a rooted binary plane tree.

Variable-length codes without comma: Prefix codes

Definition

Rooted binary plane tree.
Leaf of a rooted binary plane tree.

Definition: Prefix tree for Σ

Let (T, r) be a rooted binary plane tree. Let L be the set of leaves of (T, r). pause (T, r, ℓ) is a prefix tree for Σ, iff $\ell : \Sigma \to L$ is a bijection.

The coding of the characters based on a prefix tree

$$k(\in \Sigma) \mapsto \text{labels of the } r\text{-}\ell(k) \text{ path in } T$$

The fundamental question

The fundamental question

Problem

Given Σ alphabet and a text τ ($\to$probability distribution over Σ / frequency table($\in \mathbb{N}^{\Sigma}$)).

The fundamental question

Problem

Given Σ alphabet and a text τ ($\rightarrow$probability distribution over Σ / frequency table($\in \mathbb{N}^{\Sigma}$)). Find a prefix tree over Σ, that minimize the lenth of the code of τ.

Huffman's algorithm: Basic idea

Huffman's algorithm: Basic idea

- We consider the the characters as a one-node prefix trees.

Huffman's algorithm: Basic idea

- We consider the the characters as a one-node prefix trees. So we start with $|\Sigma|$ prefix trees.

Huffman's algorithm: Basic idea

- We consider the the characters as a one-node prefix trees. So we start with $|\Sigma|$ prefix trees.
- We choose two trees and merge them.

Huffman's algorithm: Basic idea

- We consider the the characters as a one-node prefix trees. So we start with $|\Sigma|$ prefix trees.

- We choose two trees and merge them.Merging: We introduce a new root-vertex. Its two children are the roots of the two merged tree.

Huffman's algorithm: Basic idea

- We consider the the characters as a one-node prefix trees. So we start with $|\Sigma|$ prefix trees.
- We choose two trees and merge them.Merging: We introduce a new root-vertex. Its two children are the roots of the two merged tree.
- Using merging steps we compute the output (*equiv* greediness).

Huffman's algorithm: Basic idea

- We consider the the characters as a one-node prefix trees. So we start with $|\Sigma|$ prefix trees.
- We choose two trees and merge them.Merging: We introduce a new root-vertex. Its two children are the roots of the two merged tree.
- Using merging steps we compute the output (*equiv* greediness).
- The initial trees have frequency values.

Huffman's algorithm: Basic idea

• We consider the the characters as a one-node prefix trees. So we start with $|\Sigma|$ prefix trees.

• We choose two trees and merge them.Merging: We introduce a new root-vertex. Its two children are the roots of the two merged tree.

• Using merging steps we compute the output (*equiv* greediness).

• The initial trees have frequency values. During the algorithm each tree has an f-vaule: The SUM of the frequencies assigned to its leaves.

Huffman's algorithm: Basic idea

- We consider the the characters as a one-node prefix trees. So we start with $|\Sigma|$ prefix trees.
- We choose two trees and merge them.Merging: We introduce a new root-vertex. Its two children are the roots of the two merged tree.
- Using merging steps we compute the output (*equiv* greediness).
- The initial trees have frequency values. During the algorithm each tree has an f-vaule: The SUM of the frequencies assigned to its leaves.
- ??? HOW TO CHOOSE THE TWO TREES TO MERGE ???

Huffman's algorithm: Basic idea

- We consider the the characters as a one-node prefix trees. So we start with $|\Sigma|$ prefix trees.
- We choose two trees and merge them.Merging: We introduce a new root-vertex. Its two children are the roots of the two merged tree.
- Using merging steps we compute the output (*equiv* greediness).
- The initial trees have frequency values. During the algorithm each tree has an f-vaule: The SUM of the frequencies assigned to its leaves.
- ??? HOW TO CHOOSE THE TWO TREES TO MERGE ???
- The natural idea: choose the two trees with lowest frequencies.

Huffman's algorithm

Huffman's algorithm (1951)

(INITIALIZATION) We construct a rooted binary plane tree for each character.

Huffman's algorithm

Huffman's algorithm (1951)

(INITIALIZATION) We construct a rooted binary plane tree for each character. Each tree has an f-value, the frequency of the corresponding character.

Huffman's algorithm

Huffman's algorithm (1951)

(INITIALIZATION) We construct a rooted binary plane tree for each character. Each tree has an f-value, the frequency of the corresponding character.
// During the algorithm we always have a set of prefix trees with f-values: $\mathcal{T}$.

Huffman's algorithm

Huffman's algorithm (1951)

(INITIALIZATION) We construct a rooted binary plane tree for each character. Each tree has an f-value, the frequency of the corresponding character.
// During the algorithm we always have a set of prefix trees with f-values: $\mathcal{T}$.
(CHOICE) (until $|\mathcal{T}| > 1$) Take the two trees with the lowest frequencies: T_1, T_2.

Huffman's algorithm

Huffman's algorithm (1951)

(INITIALIZATION) We construct a rooted binary plane tree for each character. Each tree has an f-value, the frequency of the corresponding character.
// During the algorithm we always have a set of prefix trees with f-values: $\mathcal{T}$.
(CHOICE) (until $|\mathcal{T}| > 1$) Take the two trees with the lowest frequencies: T_1, T_2.
(Merge) Merge the two chosen trees $\to M(T_1, T_2)$.

Huffman's algorithm

Huffman's algorithm (1951)

(INITIALIZATION) We construct a rooted binary plane tree for each character. Each tree has an f-value, the frequency of the corresponding character.
// During the algorithm we always have a set of prefix trees with f-values: $\mathcal{T}$.
(CHOICE) (until $|\mathcal{T}| > 1$) Take the two trees with the lowest frequencies: T_1, T_2.
(Merge) Merge the two chosen trees $\rightarrow M(T_1, T_2)$.

$$\mathcal{T} \leftarrow \mathcal{T} - \{T_1, T_2\} \cup \{M(T_1, T_2)\}.$$

Huffman's algorithm

Huffman's algorithm (1951)

(INITIALIZATION) We construct a rooted binary plane tree for each character. Each tree has an f-value, the frequency of the corresponding character.
// During the algorithm we always have a set of prefix trees with f-values: $\mathcal{T}$.
(CHOICE) (until $|\mathcal{T}| > 1$) Take the two trees with the lowest frequencies: T_1, T_2.
(Merge) Merge the two chosen trees $\to M(T_1, T_2)$.

$$\mathcal{T} \leftarrow \mathcal{T} - \{T_1, T_2\} \cup \{M(T_1, T_2)\}.$$

The f-value of $M(T_1, T_2)$ is the some of the f-values of the merged two trees.

Huffman's algorithm

Huffman's algorithm (1951)

(INITIALIZATION) We construct a rooted binary plane tree for each character. Each tree has an f-value, the frequency of the corresponding character.
// During the algorithm we always have a set of prefix trees with f-values: $\mathcal{T}$.
(CHOICE) (until $|\mathcal{T}| > 1$) Take the two trees with the lowest frequencies: T_1, T_2.
(Merge) Merge the two chosen trees $\rightarrow M(T_1, T_2)$.

$$\mathcal{T} \leftarrow \mathcal{T} - \{T_1, T_2\} \cup \{M(T_1, T_2)\}.$$

The f-value of $M(T_1, T_2)$ is the some of the f-values of the merged two trees. Back to (CHOICE).

The correctness of Huffman's algorithm: the main idea

The correctness of Huffman's algorithm: the main idea

Huffman's theorem

The output of the Huffman's algorithm is a prefix tree the produces the shortest prefix coding of the input text.

The correctness of Huffman's algorithm: the main idea

Huffman's theorem

The output of the Huffman's algorithm is a prefix tree the produces the shortest prefix coding of the input text.

Let T be a prefix tree. Choose two leaves (ℓ, ℓ') that are siblings on the lowest level of T. Let k and k' be the two characters that are in the first merge step when we run Huffman's coding.

The correctness of Huffman's algorithm: the main idea

Huffman's theorem

The output of the Huffman's algorithm is a prefix tree the produces the shortest prefix coding of the input text.

Let T be a prefix tree. Choose two leaves (ℓ, ℓ') that are siblings on the lowest level of T. Let k and k' be the two characters that are in the first merge step when we run Huffman's coding. Modify T by changing the labels of the leaves. With at most two transpositions we obtain T' where the label of ℓ is k, and the label of ℓ' is k'.

The correctness of Huffman's algorithm: the main idea

Huffman's theorem

The output of the Huffman's algorithm is a prefix tree the produces the shortest prefix coding of the input text.

Let T be a prefix tree. Choose two leaves (ℓ, ℓ') that are siblings on the lowest level of T. Let k and k' be the two characters that are in the first merge step when we run Huffman's coding. Modify T by changing the labels of the leaves. With at most two transpositions we obtain T' where the label of ℓ is k, and the label of ℓ' is k'.

Let β' be the code of τ based on T'. Let β be the code of τ based on T.

The correctness of Huffman's algorithm: the main idea

Huffman's theorem

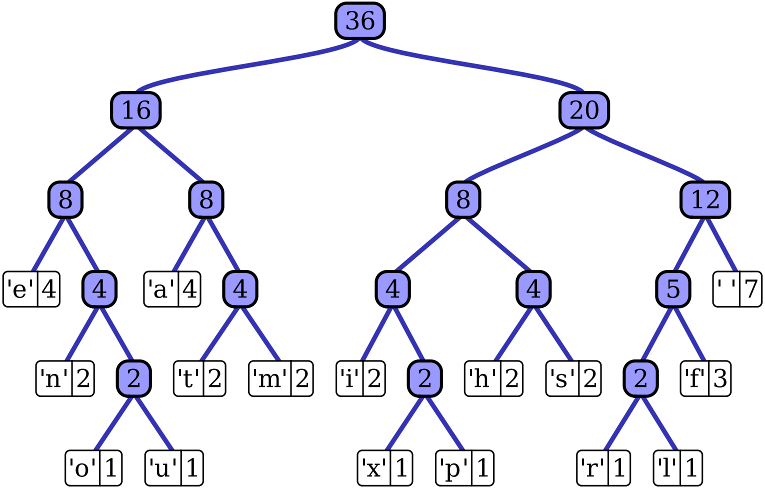
The output of the Huffman's algorithm is a prefix tree the produces the shortest prefix coding of the input text.

Let T be a prefix tree. Choose two leaves (ℓ, ℓ') that are siblings on the lowest level of T. Let k and k' be the two characters that are in the first merge step when we run Huffman's coding. Modify T by changing the labels of the leaves. With at most two transpositions we obtain T' where the label of ℓ is k, and the label of ℓ' is k'.
Let β' be the code of τ based on T'. Let β be the code of τ based on T.

Observation

The length of β' is at most the length of β.

"this is an example of a huffman tree"



Source: wikipedia

Break

The greedy algorithm for matchings

Greedy algorithm for finding large matchings

The greedy algorithm for matchings

Greedy algorithm for finding large matchings

(Initialization) Start with a matching M.

The greedy algorithm for matchings

Greedy algorithm for finding large matchings

(Initialization) Start with a matching M.
WHILE there is an edge $e \in E(G) - M$ such that $M \cup \{e\}$ is a matching too, do
(Greedy extension step) $M \leftarrow M \cup \{e\}$.

The greedy algorithm for matchings

Greedy algorithm for finding large matchings

(Initialization) Start with a matching M.
WHILE there is an edge $e \in E(G) - M$ such that $M \cup \{e\}$ is a matching too, do
(Greedy extension step) $M \leftarrow M \cup \{e\}$.
(Halting) The actual matching is the output.

The greedy algorithm for matchings

Greedy algorithm for finding large matchings

(Initialization) Start with a matching M.
WHILE there is an edge $e \in E(G) - M$ such that $M \cup \{e\}$ is a matching too, do
(Greedy extension step) $M \leftarrow M \cup \{e\}$.
(Halting) The actual matching is the output.
// At halting we have that each edge of $E(G) \setminus M$ is neighbors of an edge from M.

The greedy algorithm for matchings

Greedy algorithm for finding large matchings

(Initialization) Start with a matching M.
WHILE there is an edge $e \in E(G) - M$ such that $M \cup \{e\}$ is a matching too, do
(Greedy extension step) $M \leftarrow M \cup \{e\}$.
(Halting) The actual matching is the output.
// At halting we have that each edge of $E(G) \setminus M$ is neighbors of an edge from M.

- There is no fear of infinite loop.

The greedy algorithm for matchings

Greedy algorithm for finding large matchings

(Initialization) Start with a matching M.
WHILE there is an edge $e \in E(G) - M$ such that $M \cup \{e\}$ is a matching too, do
(Greedy extension step) $M \leftarrow M \cup \{e\}$.
(Halting) The actual matching is the output.
// At halting we have that each edge of $E(G) \setminus M$ is neighbors of an edge from M.

- There is no fear of infinite loop.

- We know that int the case of halting the output can't be augmented by extensions (by adding further edges to the output).

Example

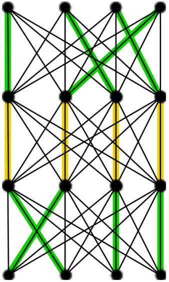


Figure: Our graph has four disjoint levels of equal sized vertex sets (let n be the size of the levels, in our example $n = 4$). Between two adjacent levels all possible edges are present and there are no further edges. It is possible that the greedy algorithm first chooses the yellow edges, matching the two middle levels. Then it halts. The green edges form a perfect matching.

Analysis

Analysis

Theorem

Let $\nu_{\text{greedy}}(G)$ denote the size of the output of the greedy algorithm. Then

$$\frac{\nu(G)}{2} \leq \nu_{\text{greedy}}(G) \leq \nu(G).$$

Analysis

Theorem

Let $\nu_{\text{greedy}}(G)$ denote the size of the output of the greedy algorithm. Then

$$\frac{\nu(G)}{2} \leq \nu_{\text{greedy}}(G) \leq \nu(G).$$

The second inequality is obvious since our algorithm computes a matching.

Analysis: the proof

Analysis: the proof

- Let M_{greedy} denote the output matching of the greedy algorithm.

Analysis: the proof

- Let M_{greedy} denote the output matching of the greedy algorithm.
- $L = V(M_{\text{greedy}})$ is the set of matched vertices.

Analysis: the proof

- Let M_{greedy} denote the output matching of the greedy algorithm.
- $L = V(M_{\text{greedy}})$ is the set of matched vertices.
- It is obvious that L is a covering vertex set, and $|L| = 2\nu_{\text{greedy}}(G)$.

Analysis: the proof

- Let M_{greedy} denote the output matching of the greedy algorithm.
- $L = V(M_{\text{greedy}})$ is the set of matched vertices.
- It is obvious that L is a covering vertex set, and $|L| = 2\nu_{\text{greedy}}(G)$.
- The size of L gives an upper bound on the size of an arbitrary matching, hence $\nu(G) \leq |L| = 2\nu_{\text{greedy}}(G)$.

This is the end!

Thank you for your attention!