

11

Greedy Algorithms

12 January 2026

Prof. Dr. Sebastian Wild

Learning Outcomes

Unit 11: *Greedy Algorithms*

1. Describe informally what greedy algorithms are.
2. Know exemplary problems and their greedy solutions: Change-Making Problem, MSTs, SSSPP, Assignment Problem.
3. Be able to design and proof correctness of greedy algorithms for (simple) algorithmic problems.
4. Be able to explain the matroid properties and its relation to greedy algorithms.

11 Greedy Algorithms

- 11.1 Introduction
- 11.2 How Can Greed Succeed?
- 11.3 Greed in Graphs I: MSTs
- 11.4 Greed in Graphs II: Prim's MST Algorithm
- 11.5 Greed in Graphs III: Shortest Paths
- 11.6 Greedy Schedules
- 11.7 The Essence of Greed: Matroids

11.1 Introduction

Myopic Optimization

- ▶ In a “*greedy*” *algorithm*,
we assemble a solution to an **optimization** problem **step by step**
always picking the next step to maximize **current** gain,
and we **never take back** earlier steps.



“Take what you can, give nothing back!”

- ▶ reminiscent of *gradient-descent* algorithms
but discrete and even more unwilling to undo mistakes

↪ greedy algorithms only yield optimal solutions for certain problems

- ▶ but where they do, their speed is usually unbeatable

↪ it is understanding where they succeed

- ▶ even where they are not optimal, greedy approaches can be efficient heuristics or approximation algorithms

(unknown quality)

↪ c -approximation = at most factor c worse than optimum

Plan for the Unit

- ▶ We will first see a few examples (known and new) of greedy algorithms to make the vague generic description concrete
 - ▶ in particular minimum spanning trees and shortest paths in graphs
- ▶ Unlike other algorithm design techniques, greedy algorithms have a formal basis: *matroids* (and *greedoids*)
 - ▶ The second part will introduce these and how they can unify correctness proofs

A First Example: Reunion With An Old Friend

► We have seen an example of a Greedy Algorithm in Unit 7: *Huffman Codes!*

► Recall the problem:

► **Given:** Set of symbols $\Sigma = [0..\sigma)$, weights $w : \Sigma \rightarrow \mathbb{R}_{\geq 0}$

► **Goal:** prefix code E (= code trie) that minimizes $\sum_{c \in \Sigma} w(c) \cdot |E(c)|$

↪ Since only *code tries* are valid, all solutions consist in repeatedly merging tries (starting from single-leaf tries, until single trie left)

► each merge contributes the subtree's total weight to overall cost (since all leaves in merged tries move one level down / all codewords get one extra bit)

► **Huffman's Algorithm:** Always choose current cheapest merge.

► In the correctness proof, we had to show:


There is always an optimal code trie where the two lowest-weight symbols are siblings.

This is typical: To show that Greedy is optimal, we need a structural insight into optimal solutions.

11.2 How Can Greed Succeed?

Greedy For Change

The Change-Making Problem (a.k.a. Coin-Exchange Problem)

- ▶ **Given:** a set of integer denominations of coins $w_1 < w_2 < \dots < w_k$ with $w_1 = 1$, target value $n \in \mathbb{N}_{\geq 1}$  (we have sufficient supply of all coins ...)
- ▶ **Goal:** “fewest coins to give change n ”, i. e., multiplicities $c_1, \dots, c_k \in \mathbb{N}_{\geq 0}$ with $\sum_{i=1}^k c_i \cdot w_i = n$ minimizing $\sum_{i=1}^k c_i$

For Euro coins, denominations are 1c, 2c, 5c, 10c, 20c, 50c, 1€, and 2€.
formally: $1, 2, 5, 10, 20, 50, 100, \text{ and } 200$.
 $w_1 \quad w_2 \quad w_3 \quad w_4 \quad w_5 \quad w_6 \quad w_7 \quad w_8$

↪ Simple greedy algorithm:
largest coins first

- ▶ optimal time ($O(k)$ if coins sorted)
- ▶ is $\sum c_i$ minimal?

```
1 procedure greedyChange( $w[1..k], n$ ):  
2   // Assumes  $1 = w[1] < w[2] < \dots < w[k]$   
3   for  $i := k, k-1, \dots, 1$ :  
4      $c[i] := \lfloor n / w[i] \rfloor$   
5      $n := n - c[i] \cdot w[i]$   
6   // Now  $n == 0$   
7   return  $c[1..k]$ 
```

Optimality of Greedy Euro-Change

- ▶ **Theorem:** greedyChange computes an optimal $c[1..8]$ for $w[1..8] = [1, 2, 5, 10, 20, 50, 100, 200]$ for every $n \in N_{\geq 1}$.
- ▶ The greedy algorithm can be interpreted as picking one coin at a time, each time choosing the largest possible denomination $\hat{w}(n) = \max\{w[i] : w[i] \leq n\}$.
- ▶ We prove by induction over n : Any optimal solution for n must contain $\hat{w}(n)$.
 - ▶ $n = 1$: can only use $\hat{w}(n) = 1$ ✓
 - ▶ $n \in [2..5]$: Assume we had a solution without $\textcircled{2\text{€}}$ \rightsquigarrow must be $n \times \textcircled{1\text{€}}$ with $n \geq 2$;
 \rightsquigarrow we can make this strictly better by replacing $\textcircled{1\text{€}} \textcircled{1\text{€}}$ by $\textcircled{2\text{€}}$ ⚡
 - ▶ $n \in [5..10]$: Assume solution without $\textcircled{5\text{€}}$ summing to $n \geq 5$.
The solution must fall into one of the following cases:
 - (a) $\geq 3 \times \textcircled{2\text{€}}$ \rightsquigarrow replacing $\textcircled{2\text{€}} \textcircled{2\text{€}} \textcircled{2\text{€}}$ by $\textcircled{5\text{€}} \textcircled{1\text{€}}$ strictly better ⚡
 - (b) $\leq 1 \times \textcircled{2\text{€}}$ \rightsquigarrow value $n - 2 \geq 3$ without $\textcircled{2\text{€}}$ ⚡ by IH
 - (c) $2 \times \textcircled{2\text{€}}$ and $\geq 1 \times \textcircled{1\text{€}}$ \rightsquigarrow $\textcircled{2\text{€}} \textcircled{2\text{€}} \textcircled{1\text{€}} \rightarrow \textcircled{5\text{€}}$ strictly better ⚡
 - (d) $2 \times \textcircled{2\text{€}}$, no $\textcircled{1\text{€}}$ \rightsquigarrow only obtain value $\leq 4 < n$ ⚡
 - ▶ $n \in [10, 20)$: Any solution without $\textcircled{10\text{€}}$ contains
 - (a) $\textcircled{5\text{€}} \textcircled{5\text{€}}$ \rightsquigarrow replace by $\textcircled{10\text{€}}$; or
 - (b) at most one $\textcircled{5\text{€}}$ \rightsquigarrow at least value 5 without $\textcircled{5\text{€}}$ ⚡ by IH

Optimality of Greedy Euro-Change [2]

► ... proof continued

► $n \in [20..50)$ Without $\textcircled{20\text{c}}$, we must have

(a) $\textcircled{10\text{c}} \textcircled{10\text{c}} \rightarrow \textcircled{20\text{c}} \text{ ⚡}$

(b) at most one $\textcircled{10\text{c}}$ \rightsquigarrow value $n - 10 \geq 10$ without $\textcircled{10\text{c}}$ ⚡ by IH

► $n \in [50..100)$ Without $\textcircled{50\text{c}}$, we must have

(a) $\geq 3 \times \textcircled{20\text{c}} \rightsquigarrow \textcircled{20\text{c}} \textcircled{20\text{c}} \textcircled{20\text{c}} \rightarrow \textcircled{50\text{c}} \textcircled{10\text{c}} \text{ ⚡}$

(b) $\leq 1 \times \textcircled{20\text{c}} \rightsquigarrow$ value $n - 20 \geq 30$ without $\textcircled{20\text{c}}$ ⚡ by IH

(c) $2 \times \textcircled{20\text{c}}$ and $\geq 1 \times \textcircled{10\text{c}} \rightsquigarrow \textcircled{20\text{c}} \textcircled{20\text{c}} \textcircled{10\text{c}} \rightarrow \textcircled{50\text{c}} \text{ ⚡}$

(d) $2 \times \textcircled{20\text{c}}$, no $\textcircled{10\text{c}}$ \rightsquigarrow value $n - 40 \geq 10$ without $\textcircled{10\text{c}}$ ⚡ by IH

► $n \in [100..200)$: as for $n \in [10, 20)$, *mutatis mutandis*.

► $n \geq 200$: as for $n \in [20, 50)$.

► The same arguments work for adding coins $1 \cdot 10^m, 2 \cdot 10^m, 5 \cdot 10^m$ for $m = 3, 4, \dots$

That went smoothly!

And we proved a nice structural statement about how optimal solutions look like as a bonus.

Maybe Greedy always works?

Greed Can Mislead

- ▶ *Unfortunately, No.* See $w = (1, 3, 4)$ and $n = 6$.
or $w = (1, 4, 9)$ and $n = 12$

Where/Why does our proof from above fail?

- ▶ Indeed, Greedy can be **arbitrarily bad** compared to the optimal solution:
See $w = (1, 999, 1000)$ and $n = 1998$.

↪ Need to be careful about the details of a correctness argument for greedy algorithms.

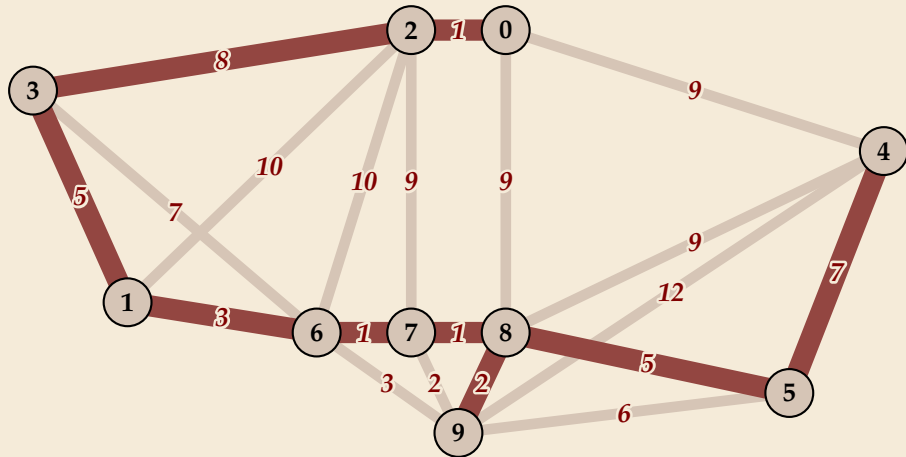
- ▶ The Change-Making problem is still only partially understood.
 - ▶ Exactly characterizing the denomination sequences that are optimally handled by greedyChange is an **open research problem**.
 - ▶ Sufficient criteria for “greed-compatible” denominations found in the literature.
 - ▶ The general problem is (weakly) NP-hard
 - ▶ Yet, for moderate n , we will see a solution for general denomination sequences later!

11.3 Greed in Graphs I: MSTs

Metaphor: Planning an electricity grid

Given: Houses to be connected to central power grid
Possible connections with building costs given

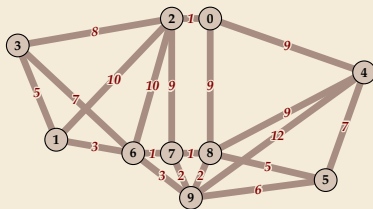
Goal: Cheapest way to get every house connected



The Minimum Spanning Tree (MST) Problem

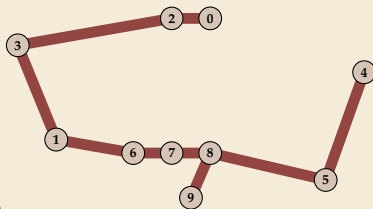
Given: undirected, edge-weighted, simple,
connected graph $G = (V, E, c)$ ↗ no self loops,
no parallel edges

Formally: Recall assumption $V = [0..n)$ (\rightsquigarrow array indices)
edges $E \subseteq \{\{u, v\} : u, v \in V \wedge u \neq v\}$
edge weights (costs) $c : E \rightarrow \mathbb{R}_{\geq 0}$
for all $u, v \in V$ there exists a path $u \rightsquigarrow v$ in (V, E)



Goal: a **spanning tree** (V, T)
with **minimal** total cost $c(T) := \sum_{e \in T} c(e)$

Formally: $T \subseteq E$
 (V, T) is connected and acyclic ("spanning tree")
for every spanning tree (V, T') of G we have $c(T') \geq c(T)$.



Further MST Applications

Direct Applications

- ▶ single-linkage hierarchical clustering
- ▶ Bottleneck-shortest paths
- ▶ Approximation algorithms, e. g.,
 - ▶ Christofides's Metric TSP Approximation
 - ▶ Steiner-tree problem

As a cheap subroutine

- ▶ Routing protocols
- ▶ medical image processing
- ▶ ...

Interlude: On Varieties of Trees



We freely use “tree” to mean different things in different contexts . . . mind the confusion.

- ▶ here: “tree” = *undirected, nonplane tree* = an undirected, connected and acyclic graph

in spanning tree

no order on edges

The digraph flavor is a rooted tree: (hence undirected trees sometimes called *unrooted*)

- ▶ *rooted (nonplane/unordered) tree* = **digraph** (V, E) with *root* $r \in V$ s.t.
 $\forall v \in V \setminus \{r\} : d_{\text{out}}(v) = 1$ and $d_{\text{out}}(r) = 0$

out-degree = #outgoing edges



We draw trees with the single(!) root on top . . .

Other “trees” don’t originate from graphs naturally, but rather from recursion / terms:

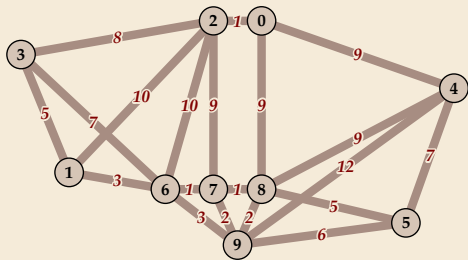
- ▶ *binary tree* = a null pointer or a node with left and right children, each a binary tree
(formally: the set of binary trees is the smallest fixed point of that construction)
- ▶ *ordinal trees* = a node with a sequence of 0 or more children, each ordinal trees
= rooted ordered trees (rooted unordered + total order on children)
- ▶ plus many more variants out there . . . \rightsquigarrow if in doubt, double check definitions!

A Naive Approach

How to start finding an MST?

Using the **cheapest** edge shouldn't hurt ...

```
1 procedure greedyMST( $V, E, c$ ):  
2   // Assume  $(V, E)$  is simple & connected,  $c : E \rightarrow \mathbb{R}_{\geq 0}$   
3    $T := \emptyset$   
4   while  $(V, T)$  not connected  
5      $e :=$  cheapest edge that doesn't close a cycle in  $T$   
6      $T := T \cup \{e\}$   
7   return  $T$ 
```

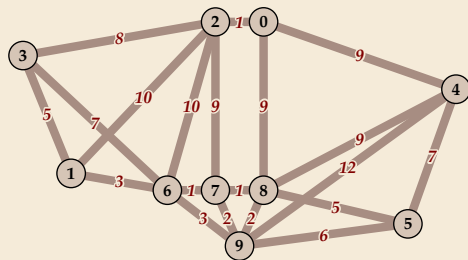


A Naive Approach Works – Kruskal's Algorithm

How to start finding an MST?

Using the **cheapest** edge shouldn't hurt ...

```
1 procedure kruskalMST( $V, E, c$ ):  
2   // Assume  $(V, E)$  is simple & connected,  $c : E \rightarrow \mathbb{R}_{\geq 0}$   
3    $T := \emptyset$   
4   while  $(V, T)$  not connected  
5      $e :=$  cheapest edge that doesn't close a cycle in  $T$   
6      $T := T \cup \{e\}$   
7   return  $T$ 
```



Apart from implementing line 4 and line 5 efficiently, this *is* **Kruskal's Algorithm**!

As so often with greedy algorithms, the hardest bit is the correctness argument. We have:

Theorem: Kruskal's Algorithm finds a minimum spanning tree.

This immediately follows from proving the following invariant:

Kruskal's Invariant: There is some MST T^* with $T \subseteq T^*$.

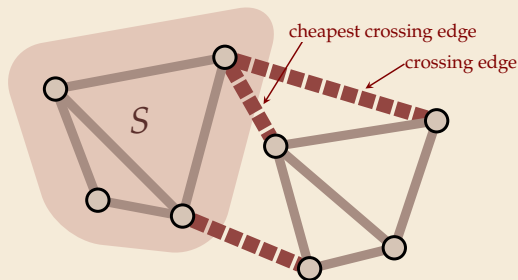
henceforth: identify MST with its edge set

Crossing Edges and the MST-Cut Lemma

To prove the correctness of Kruskal's Algorithm, we need a tool.

Notation:

- **Cut S :**
non-trivial set of vertices $\emptyset \neq S \subsetneq V$
- **crossing edge e wrt. cut S :**
 $e = \{u, v\}$ with $u \in S, v \in \bar{S} := V \setminus S$



The MST-Cut Lemma:

Let T^* be an MST und $W \subseteq T^*$.

For every cut S , not cutting any edges in W , and every *cheapest* crossing edge e wrt. S there is an MST \hat{T}^* that contains $W \cup \{e\}$.

Proof of MST-Cut Lemma

Proof:

- ▶ Case 1: $e \in T^*$.

Then picking $\hat{T}^* = T^*$ proves the claim.

- ▶ Case 2: $e \notin T^*$.

$\rightsquigarrow T^* \cup \{e\}$ contains unique cycle C using e .

- ▶ Since e crosses cut S , C crosses S

\rightsquigarrow There is a second crossing edge $e' \in C$.

- ▶ Since e' is crossing, $e' \notin W$

- ▶ by assumption, $c(e) \leq c(e')$ (we pick cheapest crossing edge)

$\rightsquigarrow \hat{T}^* = T^* \cup \{e\} \setminus \{e'\}$ is a spanning tree, and $W \cup \{e\} \subseteq \hat{T}^*$

- ▶ $c(\hat{T}^*) = c(T^*) + c(e) - c(e') \leq c(T^*)$

$\rightsquigarrow \hat{T}^*$ is an MST.

The MST-Cut Lemma:

Let T^* be an MST und $W \subseteq T^*$.

For every cut S , not cutting any edges in W , and every *cheapest* crossing edge e wrt. S there is an MST \hat{T}^* that contains $W \cup \{e\}$.

Kruskal's Algorithm – Correctness

With these preparations, we can prove

Kruskal's Invariant: There is some MST T^* with $T \subseteq T^*$.

Proof: by induction over the loop iterations

- ▶ IB: initially $T = \emptyset$ and $\emptyset \subseteq T^*$ for every MST T^* .
- ▶ IH: Assume the invariant is after the i th iteration.
- ▶ IS: Let $e = vw$ be the edge considered in iteration $i + 1$.
 - ▶ Let S be the connected component of v in (V, T) (T : before potentially adding e)
 - ▶ Case 1: $w \in S$.
Then e closes a cycle in T and is not added to T .
 \rightsquigarrow invariant still satisfied.
 - ▶ Case 2: $w \notin S$.
Then e is a crossing edge wrt. S ; must be a cheapest crossing edge by choice of e .
 \rightsquigarrow by inv. \exists MST $T^* \supseteq T$ and by MST-Cut Lemma, there is an MST $\hat{T}^* \supseteq T \cup \{e\}$
 \rightsquigarrow Invariant still satisfied.

The MST-Cut Lemma:

Let T^* be an MST und $W \subseteq T^*$.

For every cut S , not cutting any edges in W , and every *cheapest* crossing edge e wrt. S there is an MST \hat{T}^* that contains $W \cup \{e\}$.

Since we only terminate when T is spanning, upon termination $T = T^*$ for an MST T^* .

Kruskal's Algorithm – Data Structures

For an efficient implementation of Kruskal's algorithm, we need to efficiently

1. check whether T is spanning
2. find the next cheapest edge to consider
3. test whether an edge closes a cycle

Each can be supported as follows:

1. Since we maintain T acyclic, checking $|T| = n - 1$ suffices!
2. It suffices to pre-sort E by weight!
 - ▶ We only ever grow T , so if e is closing a cycle now, it will for good.
 - ↪ Once discarded, an edge need not be looked at ever again.
3. Use a **Union-Find data structure** (see Algorithmen & Datenstrukturen!)
 - ▶ dynamically maintain connected components
 - ▶ initially, every vertex has its own id
 - ▶ adding vw to T ↪ call $\text{union}(v, w)$
 - ▶ vw closes a cycle *iff* $\text{find}(v) == \text{find}(w)$

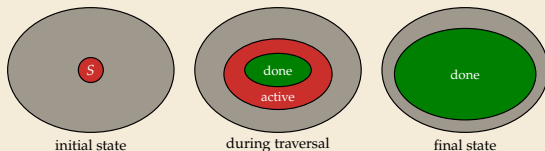
↪ $O(m \log m) = O(m \log n)$ time and $O(m)$ extra space.

11.4 Greed in Graphs II: Prim's MST Algorithm

Prim's Algorithm

- ▶ An alternative greedy approach that tries to consider only crossing edges.
 - ▶ start with $S = \{s\}$ for some vertex s
 - ▶ only consider edges vw for some $v \in S, w \notin S$ (crossing edges)
 - ▶ add cheapest crossing edge vw to T and add w to S
 - ▶ repeat until $|T| = n - 1$
- ↪ T invariably a single tree

↪ a graph traversal with tree edges T !



The MST-Cut Lemma:

Let T^* be an MST und $W \subseteq T^*$.

For every cut S , not cutting any edges in W , and every *cheapest* crossing edge e wrt. S there is an MST \hat{T}^* that contains $W \cup \{e\}$.

↪ Correctness as for Kruskal's algorithm: **Invariant:** \exists MST T^* with $T \subseteq T^*$.

IB: initially true with $T = \emptyset$

IS: whenever we add an edge, it is the cheapest crossing edge w.r.t. cut (S, \bar{S}) .

Prim's Algorithm – Lazy Implementation

How to efficiently find the cheapest crossing edge?

- **Option 1:** Maintain priority queue Q of **edges**, ordered by weight.

```
1 procedure lazyPrimMST( $G$ ):  
2   // Assume  $G = (V, E, c)$  simple & connected,  $c : E \rightarrow \mathbb{R}_{\geq 0}$   
3    $T := \emptyset$ ;  $inS[0..n] := false$   
4    $Q := \text{new MinPQ}()$   
5    $\text{visit}(0)$   
6   while  $|T| < n - 1$ :  
7      $vw := Q.\text{delMin}()$   
8     if  $\neg inS[w]$  then  $\text{visit}(w)$ ;  $T.\text{insert}(vw)$  end if  
9     if  $\neg inS[v]$  then  $\text{visit}(v)$ ;  $T.\text{insert}(wv)$  end if  
10    return  $T$   
11  
12 procedure  $\text{visit}(v)$ :  
13   for  $(w, c) \in G.\text{adj}[v]$  // edge  $vw$  with cost  $c$   
14     if  $\neg inS[w]$  then  $Q.\text{insert}(vw, c)$  //  $w$  now active  
15    $inS[v] := true$  //  $v$  now done
```

- Lazy Prim: check if vw is crossing *lazily*
i. e., only after delMin
- An instance of tricolor graph traversal
 - $v \in \text{done}$ iff $inS[v]$
 - all edges to **active** vertices are in Q
 \rightsquigarrow visit every edge at most once
- size of Q always $\leq m \rightsquigarrow$ **space** $O(m)$
- **Running time:**
 - need m calls to insert
and $n - 1$ delMins
 \rightsquigarrow with binary heaps, total time
 $O(m \log m) = O(m \log n)$
 - with Fibonacci heaps even
 $O(m + n \log n)$ (insert amortized $O(1)$ time)

Easy modification: store parent in tree rooted at vertex 0

Prim's Algorithm – Eager Implementation

We can reduce the extra space to $O(n)$ if we avoid storing multiple edges to the same $w \in \bar{S}$.

- ▶ **Option 2:** Maintain priority queue Q of **vertices** in \bar{S} ,
ordered by **weight of cheapest edge** connecting them to S .
 - ▶ call that weight the *distance*, $dist[w]$, of $w \in \bar{S}$ from S .
($dist[w] = 0$ if $w \in S$, $dist[w] = \infty$ if no single edge to S)
 - ▶ after adding a vertex u to S , distance to w can **shrink** (to $c(uw)$) (but never grow)
- ↪ need a MinPQ that supports decreaseKey
 - ▶ implementation hassle: efficient implementations require a “pointer” into data structure
cleaner design: let data structure handle pointers internally
- ↪ **IndexMinPQ:** (use ST otherwise) (use amortized doubling otherwise)
 - ▶ **Assumption:** stored objects are from $[0..n)$ and n known/fixed at construction time
 - ▶ IndexMinPQ implementations maintain array positions
e. g., for binary heaps, maintain *heapIndex* $[0..n)$, update whenever heap modified
- ↪ easy to support decreaseKey(i, p') and contains(i)
(for a full implementation see Sedgewick & Wayne or Nebel & Wild)

Prim's Algorithm – Eager Implementation Code

```
1 procedure primMST(G):
2   // Assume  $G = (V, E, c)$  is simple & connected,  $c : E \rightarrow \mathbb{R}_{\geq 0}$ 
3    $father[0..n) := \text{NONE}$ ;  $inS[0..n) := \text{false}$ ;  $dist[0..n) := \infty$ 
4    $Q := \text{new IndexMinPQ}(n)$ 
5    $Q.\text{insert}(0, 0)$ 
6   while  $\neg Q.\text{isEmpty}()$ 
7      $\text{visit}(Q.\text{delMin}())$ 
8   return  $\{\{father[v], v\} : v \in [1..n)\}$ 
9
10 procedure visit(v):
11   for  $(w, c) \in G.\text{adj}[v]$  // edge  $vw$  with cost  $c$ 
12     if  $\neg inS[w]$ 
13       if  $c < dist[w]$  //  $vw$  currently cheapest edge to  $w$ 
14          $father[w] := v$ ;  $dist[w] := c$ 
15         if  $Q.\text{contains}(w)$  //  $w$  already active
16            $Q.\text{decreaseKey}(w, c)$ 
17         else //  $w$  now becoming active
18            $Q.\text{insert}(w, c)$ 
19       end if
20     end if
21   end for
22    $inS[v] := \text{true}$ ;  $dist[v] := 0$  //  $v$  now done
```

- ▶ Eager Prim: filter edges eagerly!
 \rightsquigarrow keep only **cheapest edge** to $w \in \bar{S}$
(namely $\{father[w], w\}$)
- ▶ Prototypical tricolor traversal variant
 - ▶ $v \in \text{done}$ iff $inS[v]$
 - ▶ $v \in \text{active}$ iff $Q.\text{contains}(v)$
 - ▶ choose next vertex using PQ Q ,
iterative over its edges
- ▶ size of Q always $\leq n \rightsquigarrow$ **space** $O(n)$
- ▶ **Running time:**
 - ▶ $n \times \text{insert}$, $(n - 1) \times \text{delMin}$,
up to $m \times \text{decreaseKey}$
 - \rightsquigarrow with binary heaps $O(m \log n)$
with Fibonacci heaps $O(m + n \log n)$

Minimum Spanning Trees – Discussion

- 👍 MSTs are a versatile modeling tool
- 👍 very efficient to compute even for arbitrary weights
- 👍 Prim's Algorithm (eager version) give best time and space and is efficient in practice

*Above algorithms are **almost linear-time**, but not quite . . . can we find MSTs in linear time?*

- ▶ Yes, if graph is **dense**, e. g., $m = \Omega(n \log n)$. Then $O(m + n \log n) = O(m)$
 - ▶ stronger results known, as well
- ▶ Yes, for **integer** weights on the word-RAM (Fredman, Willard 1994)
- ▶ Yes, if **randomization** is allowed (Karger, Klein, Tarjan 1995)
 - ▶ uses that linear time suffices to *verify* a given ST as minimal(!)
- ▶ General (deterministic, comparison-based, on sparse graphs)? **Open research problem!**
 - ▶ Best known general time $O(m\alpha(m, n))$ where α is an “inverse Ackermann function”

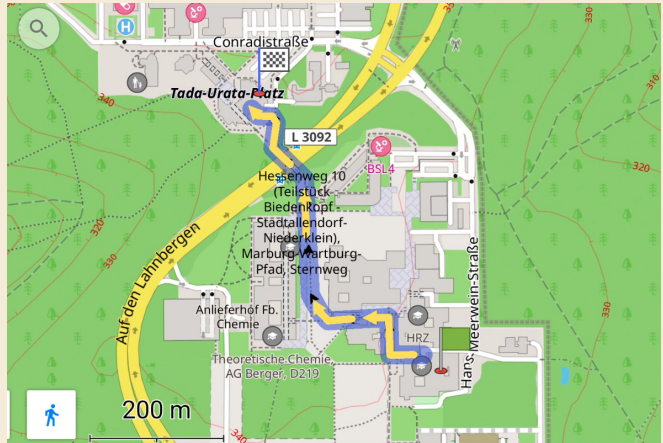
$$\begin{aligned}\alpha(m, n) &= \min\{z \geq 1 : A(z, 4\lceil m/n \rceil) > \lg n\} \\ A(0, x) &= 2x, \quad A(i, 0) = 0, \quad A(i, 1) = 2, \quad (i \geq 1), \\ A(i, x) &= A(i-1, A(i, x-1)); \quad (i \geq 1, x \geq 2)\end{aligned}$$

11.5 Greed in Graphs III: Shortest Paths

Metaphor: Route Planning

Given: Road network (map), current location, target location
crossings = vertices, roads = edges, road length = edge weight

Goal: Find shortest path from current location to target



SSSPP

It turns out that a cleaner algorithmic problem is to find shortest paths to *all* vertices.

Single Source Shortest Path Problem (SSSPP)

- ▶ **Given:** directed, edge-weighted, simple graph $G = (V, E, c)$
with edge costs $c : E \rightarrow \mathbb{R}$, a start vertex $s \in V$
- ▶ **Goal:** a data structure that reports for every $v \in V$:
 $\delta_G(s, v)$: the shortest-path distance from s to v
 $\text{spath}(v)$: a shortest path from s to v (if it exists)

Formally:

- ▶ for a walk $w[0..m]$ in G , we define $c(w) = \sum_{i=0}^{m-1} c(w[i]w[i+1])$
- ▶ $\delta_G(s, v) = \inf \left(\{+\infty\} \cup \{c(w) : w = w[0..m] \text{ a walk in } G \text{ with } w[0] = s \wedge w[m] = v\} \right)$
 - ▶ Note: δ_G defined via all s - v -walks, not only s - v -paths (= vertex-single walks)
 - ▶ But we will see: In relevant scenarios, we can restrict to paths (hence the name)
- ▶ $\text{spath}(v)$ returns a walk w with $c(w) = \delta_G(s, v)$ if such a walk exists

The Trouble with Negative Cycles

- ▶ The complications in the definition all stem from **negative-weight edges**

$$\delta_G(s, v) = \boxed{\inf \left(\{+\infty\} \cup \{c(w) : w \text{ an } s\text{-}v\text{-walk in } G\} \right)}$$

- ▶ In general, $\delta_G(s, v)$ can be

- ▶ $+\infty$ if there is no s - v -walk at all, or (“no-path case” easy to detect and handle)
- ▶ $-\infty$ if there are s - v -walks of arbitrarily small (negative) value

This happens *iff* we reach a negative cycle that we can repeat indefinitely,
always improving the total “cost” of the walk.

- ↪ **Lemma (Shortest Paths):** If w is a shortest s - v -walk in $G = (V, E, c)$,
there is an s - v -path p with $c(p) = c(w)$.

Proof: Suppose w contains a cycle C .

- ▶ If $c(C) < 0$, w is not shortest as we can repeat C and reduce cost ⚡
- ▶ If $c(C) > 0$, w is not shortest as we can remove C and reduce cost ⚡
- ▶ If $c(C) = 0$ for all cycles in w , we can remove them from w to obtain a path p and $c(p) = c(w)$.

- ↪ In the absence of negative cycles, all shortest walks are **shortest paths** (of at most $n - 1$ edges).

Variants of Shortest Path Problems

Important special cases

1. Positive SSSPP

- ▶ $c : E \rightarrow \mathbb{R}_{>0}$
- ▶ most relevant case for many applications \rightsquigarrow focus of this section

2. Unweighted SSSPP

- ▶ $c(e) = 1$ for $e \in E$ \rightsquigarrow $c(w) = \# \text{edges}$ for every walk w
- \rightsquigarrow solved by BFS in linear time

3. Acyclic SSSPP

- ▶ G is a DAG
- ▶ can be solved in linear time based on topological sort (for *arbitrary* c)
- ▶ For the rest of this section, we will assume $c(e) > 0$.
- ▶ But: The general case of cyclic graphs with negative edge weights is also relevant
 - ▶ We will come back to this case in Unit 12!

Dijkstra's Algorithm

- ▶ **Intuition:** Imagine sending out many little pioneers, walking at unit speed from s across all edges in G . The first pioneer to reach a vertex v “claims” v and proclaims the current time (= distance).
Dijkstra's Algorithm is a event-driven simulation of this process!

- ▶ Event: Some pioneer reaches a new vertex.
Can set a “timer” for that as soon as they start walking over an edge.
- ▶ Maintain priority queue of events, sorted by time.
 - ▶ Discard events for vertices that have been claimed already.
 - ▶ Avoid generating events when already clear that they will be discarded.
- ▶ Note: With $c(e) = 1$, this simulates BFS!

- ▶ **Implementation:** Store unclaimed vertices in IndexMinPQ
Priority = earliest time known so far when this vertex will be claimed

- ▶ To claim w at time t , must have claimed some v at time $t - c(vw)$
 - ↪ whenever we claim a vertex v , update successors' claim times (via decreaseKey)
 - ↪ overall process is a graph traversal! claimed = *done*

Dijkstra's Algorithm – Code & Correctness

```
1 procedure dijkstra( $G, s$ ):
2   // Assume  $G = (V, E, c)$  is simple (di)graph,  $c : E \rightarrow \mathbb{R}_{>0}$ 
3    $pred[0..n] := NONE$ ;  $inS[0..n] := false$ ;  $dist[0..n] := +\infty$ 
4    $Q := \text{new IndexMinPQ}(n)$ 
5    $Q.insert(s, 0)$ ;  $dist[s] := 0$ 
6   while  $\neg Q.isEmpty()$ 
7     visit( $Q.delMin()$ )
8   return ( $dist, pred$ )
9
10 procedure visit( $v$ ):
11   for  $(w, c) \in G.adj[v]$  // edge  $vw$  with cost  $c > 0$ 
12     if  $\neg inS[w]$ 
13       if  $dist[v] + c < dist[w]$ 
14         //  $s \rightsquigarrow v \rightarrow w$  new currently cheapest path to  $w$ 
15          $pred[w] := v$ ;  $dist[w] := dist[v] + c$ 
16         if  $Q.contains(w)$  then  $Q.decreaseKey(w, c)$ 
17         else  $Q.insert(w, c)$  end if //  $w$  active
18       end if
19     end if
20   end for
21    $inS[v] := true$  //  $v$  done
```

► Same as primMST except *dist* computation
distance from s , not distance from S

\rightsquigarrow Same running time:

► $n \times \text{insert}$, $(n - 1) \times \text{delMin}$,
up to $m \times \text{decreaseKey}$

\rightsquigarrow with binary heaps $O(m \log n)$
with Fibonacci heaps $O(m + n \log n)$

► Correctness:

1. current “time” = $dist[v]$ in visit(v) calls
strictly increasing over iterations
2. Invariant: $dist[v]$ is cost of some s - v -path
or $dist[v] = +\infty$
3. $dist[u] = \delta_G(s, u)$ for all $u \in \text{done}$

Shortest Paths Discussion



Simple and efficient solution if edge weights are positive



Dijkstra's Algorithm (with Fibonacci heaps) is worst-case optimal

- ▶ (for sorting vertices by distance from s in a comparison-addition model)
 - ▶ another fine example of a greedy algorithm!
-
- ▶ improvements often possible for s - t shortest paths (although worst case remains same)
 - ▶ in SSSPP Dijkstra, can stop once t is *done*
 - ▶ bidirectional Dijkstra (alternatingly work from both ends until we “meet”)
 - ▶ A^* /goal-directed search (use cheap lower bound for $\delta_G(v, t)$ in vertex selection)
 - ▶ we will revisit the general SSSPP (with negative weights)

11.6 Greedy Schedules

Scheduling

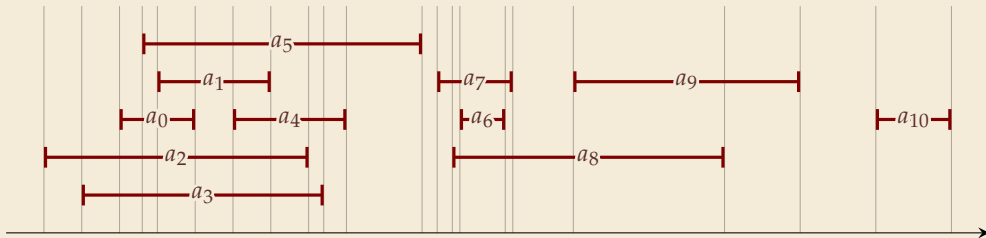
- ▶ A rich class of optimization problems deals with *scheduling*.
 - ▶ **Given:** Jobs (a.k.a. tasks, processes) and machines (a.k.a. workers, processors); optionally: constraints (e.g., order of certain jobs)
 - ▶ **Common Goal:** Find an optimal schedule, i.e.,
decide which machine does which jobs, and when,
such that a given objective is optimized (e.g., shortest makespan)
- ▶ exact properties change computational complexity of scheduling dramatically
 - ▶ can jobs be preempted (paused)?
 - ▶ are all machines equally fast on all jobs?
 - ▶ can we choose to drop certain jobs (at a cost) or must we schedule all?
 - ▶ do jobs have a hard deadline after which they are useless?
 - ▶ ...

~> *Could fill a module of its own ... Here: one exemplary special case*

The Activity selection problem

- ▶ **Activity Selection:** scheduling for *single* machine, jobs with *fixed* start and end times
pick a *subset* of jobs without *conflicts*
Formally:

- ▶ **Given:** Activities $A = \{a_0, \dots, a_{n-1}\}$, each with a start time s_i and finish time f_i
($0 \leq s_i < f_i < \infty$)
- ▶ **Goal:** Subset $I \subseteq [0..n)$ of tasks such that $i, j \in I \wedge i \neq j \implies [s_i, f_i) \cap [s_j, f_j) = \emptyset$
and $|I|$ is maximal among all such subsets
- ▶ We further assume that jobs are sorted by finish time, i. e., $f_0 \leq f_1 \leq \dots \leq f_{n-1}$
(if not, easy to sort them in $O(n \log n)$ time)

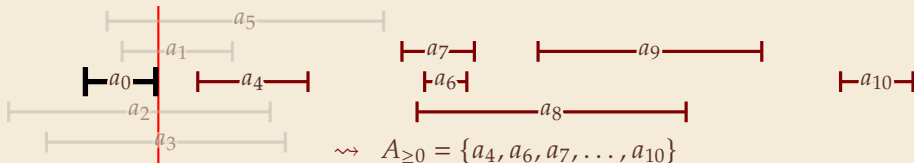


Greedy Activity Selection

```
1 procedure greedyActivitySelection( $s[0..n]$ ,  $f[0..n]$ ):  
2    $I := \{0\}$   
3    $last := 0$   
4   for  $i := 1, \dots, n-1$   
5     if  $s[i] \geq f[last]$  // no conflict, add it!  
6        $I := I \cup \{i\}$   
7        $last := i$   
8   return  $I$ 
```

- ▶ running time $O(n)$ trivial (assumes that tasks already sorted!)
- ▶ **Correctness:** greedyActivitySelection (gAS) is effectively recursive:

$$\begin{aligned} \text{gAS}(A) &= \{0\} \cup \text{gAS}(A_{\geq 0}) \\ \text{for } A_{\geq 0} &= \{a_i : s_i \geq f_0\} \end{aligned}$$



We prove:

1. \exists optimal solution I^* with $0 \in I^*$
 2. I^* with $0 \in I^*$ is an optimal solution iff $I^* \setminus \{0\}$ is an optimal solution for $A_{\geq 0}$.
- \leadsto Correctness of gAS follows by induction on n .

Greedy Activity Selection – Correctness Proof

Proofs:

1. \exists optimal solution I^* with $0 \in I^*$
 - ▶ Let I^* be some optimal solution and let $i = \min I^*$.
 - ▶ If $i = 0$, we are done.
 - ▶ Otherwise, since I^* is conflict-free and a_0 finishes earlier than a_i , $I^* \setminus \{i\} \cup \{0\}$ is also conflict-free.
2. I^* with $0 \in I^*$ is an optimal solution iff $I^* \setminus \{0\}$ is an optimal solution for $A_{\geq 0}$.
 - “ \Rightarrow ” by contraposition.
Let $I_{\geq 0} = I \setminus \{0\}$ be a non-optimal solution for $A_{\geq 0}$, i. e.,
 \exists solution $I_{\geq 0}^*$ for $A_{\geq 0}$ with $|I_{\geq 0}^*| > |I_{\geq 0}|$.
Then also $|I| = |I_{\geq 0} \cup \{0\}| < |I_{\geq 0}^* \cup \{0\}|$.
 - “ \Leftarrow ” by contraposition. Let I be non-optimal for A , i. e., $|I^*| > |I|$ exists.
By Claim 1, we can assume that $0 \in I^*$.
Then $|I \setminus \{0\}| < |I^* \setminus \{0\}|$.

11.7 The Essence of Greed: Matroids

Set Systems

We will now see a formalism to unify the study a whole class of Greedy algorithms.

► **Hereditary Set System:**

(S, \mathcal{I}) for a finite set S and a nonempty set of “*independent*” sets $\mathcal{I} \subseteq 2^S$ is a *hereditary* set system if $B \in \mathcal{I} \wedge A \subseteq B \implies A \in \mathcal{I}$

► **Weighted hereditary set system:**

(S, \mathcal{I}, w) with a hereditary set system (S, \mathcal{I}) and weight $w : S \rightarrow \mathbb{R}_{\geq 0}$

► We extend w from S to 2^S via $w(A) := \sum_{x \in A} w(x)$

\rightsquigarrow Natural *optimization problem* for weighted set system:

$$\max_{A \in \mathcal{I}} w(A)$$

► usually also: find this set A , i. e., $\arg \max_{A \in \mathcal{I}} w(A)$

Canonical Greedy Algorithm

- ▶ Given a weighted set system, we can try to greedily optimize $w(A)$:

```
1 procedure canonicalGreedy( $S, \mathcal{I}, w$ ):  
2   // Assume  $S = \{s_1, \dots, s_n\}$  sorted by weight:  $w(s_1) \geq w(s_2) \geq \dots \geq w(s_n)$   
3    $A := \emptyset$   
4   for  $i := 1, \dots, n$   
5     if  $A \cup \{s_i\} \in \mathcal{I}$   
6        $A := A \cup \{s_i\}$   
7   return  $A$ 
```

↪ When does this greedy algorithm succeed, i. e., find $\arg \max_{A \in \mathcal{I}} w(A)$?

- ▶ Certainly not always:

$$S = \{x, y, z\}, \quad \mathcal{I} = \{\emptyset, \{x\}, \{y\}, \{z\}, \{y, z\}\}$$

$$w(x) = 3, w(y) = w(z) = 2$$

- ▶ Indeed: Greedy succeeds **if and only if** (S, \mathcal{I}) is a **matroid**.

Matroids

► Matroid:

Hereditary set system (S, \mathcal{I}) is a matroid if it satisfies the *exchange property*:

$$A, B \in \mathcal{I} \wedge |A| < |B| \implies \exists x \in B \setminus A : A \cup \{x\} \in \mathcal{I}$$

► Prototypical example (also origin of names):

► S = rows of a given matrix

► \mathcal{I} = set of **linearly independent** rows

↪ (S, \mathcal{I}) is a matroid by *Steinitz exchange lemma* („Austauschlemma der linearen Algebra“)

► Further example: *Graphic Matroid*: Given an undirected graph $G = (V, E)$

► $S = E$

► $A \in \mathcal{I}$ iff (V, A) is **acyclic**

↪ check exchange property:

adding k acyclic edges reduces #connected components by exactly k

if $|B| > |A|$, some edge in $B \setminus A$ does not close a cycle in A

► set $w(e) = W - c(e)$ for c the edge cost and $W > \max c(e)$

↪ a maximum-weight independent set in (S, \mathcal{I}) iff MST of G !

Greedy iff Matroid

Theorem:



Let (S, \mathcal{I}) be a hereditary set system. The following statements are equivalent

1. $\text{canonicalGreedy}(S, \mathcal{I}, w) = \arg \max_{A \in \mathcal{I}} w(A)$ for *all* weights $w : S \rightarrow \mathbb{R}_{\geq 0}$.
2. (S, \mathcal{I}) is a matroid.




Proof:

Discussion

Matroid Theory

-  If we can identify a problem as matroid, Greedy automatically works!
-  unfortunately often necessarily easier than a direct proof

Greedy Algorithms

-  If applicable, Greedy algorithms usually offer linear running time
-  If successful, correctness proof often insightful for problem solved
-  Restricted to “tame” problems