Algorithms and Computability

Lecture 2
Greedy Algorithms

SW6/DVML8 spring 2025

Simonas Šaltenis

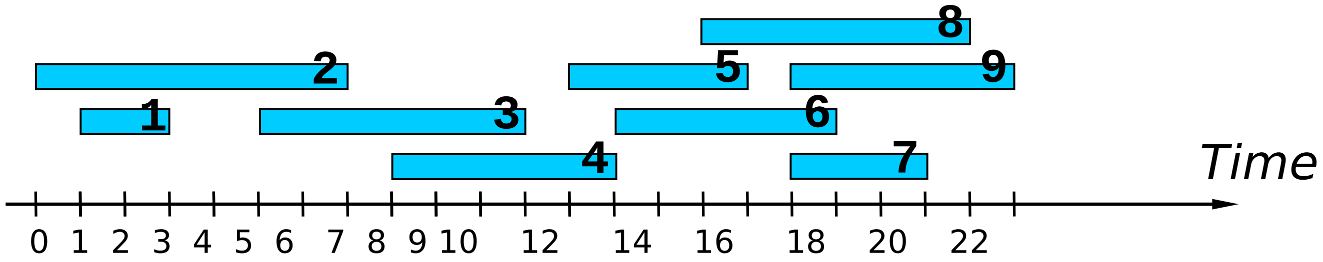
AALBORG UNIVERSITY
DENMARK

Greedy algorithms

- Goals of the lecture:
  - *to understand the **principles** of the greedy algorithm design technique;*
  - *to understand the **example greedy algorithms** for activity selection, Huffman coding, and Prim's algorithm for the minimum spanning tree.*
  - *to be able to **prove** that these algorithms find optimal solutions;*
  - *to be able to **apply** the greedy algorithm design technique.*

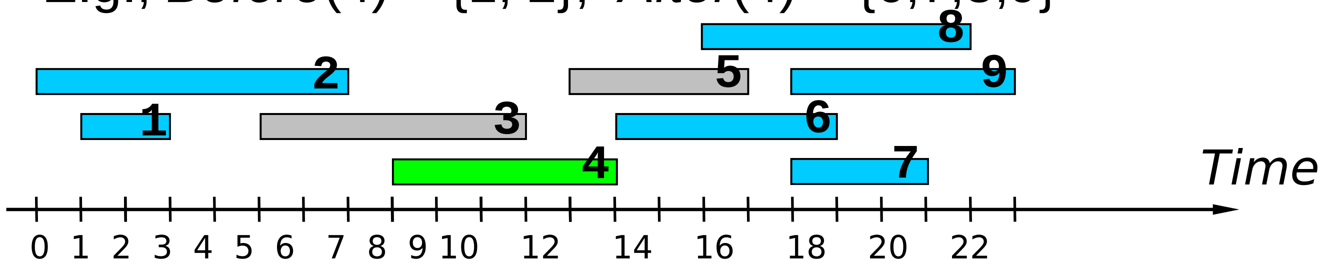
Activity-Selection Problem

- Input:
  - A set of n activities, each with start and end times: $A[i].s$ and $A[i].f$. The activity lasts during the period $[A[i].s, A[i].f)$
- Output:
  - The ***largest*** subset of mutually *compatible* activities
    - Activities are compatible if their intervals do not intersect



"Straight-forward" solution

- Let's just pick (schedule) one activity *A*[*k*]: ***k-nary choice***
  - This generates two set's of activities compatible with it: *Before*(*k*), *After*(*k*)
    - E.g., *Before*(4) = {1, 2}; *After*(4) = {6,7,8,9}



  - Solution:

$$MaxN(A) = \begin{cases} 0 & \text{if } A = \emptyset, \\ \max_{a \in A} \{ MaxN(Before(a)) + MaxN(After(a)) + 1 \} & \text{if } A \neq \emptyset. \end{cases}$$

Dynamic Programming Alg.

- The recurrence results in a dynamic programming algorithm
  - Sort activities on the end time (for simplicity assume also “sentinel” activities $A[0]$ and $A[n+1]$)
  - Let S_{ij} – a set of activities after $A[i]$ and before $A[j]$ and compatible with $A[i]$ and $A[j]$.
  - Let’s have a two-dimensional array, s.t., $c[i, j] = MaxN(S_{ij})$:

$$c[i,j]=\begin{cases}0 & \text{if } S_{ij}=\emptyset\,,\\ \max\limits_{a_k\in S_{ij}}\{c[i,k]+c[k,j]+1\} & \text{if } S_{ij}\neq\emptyset\,.\end{cases}$$

  - $MaxN(A) = MaxN(S_{0,n+1}) = c[0, n+1]$

Pseudocode, analysis

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset, \\ \max\limits_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} & \text{if } S_{ij} \neq \emptyset. \end{cases}$$

```
ActivitySel1m(A, i, j)
if c[i,j] == ∞ then
   c[i,j] = 0
   for k = i+1 to j-1 do
      if not overlaps(A[k], A[i]) and
         not overlaps(A[k], A[j]) then
         sol = ActivitySel1m(A,i,k) +
               ActivitySel1m(A,k,j) + 1
         if sol > c[i,j] then c[i,j] = sol
return c[i,j]
```

- *What is the space used by this algorithm?*
- *What is the running time?*
  - Definitely $\Omega(n^2)$ and $O(n^3)$
  - Can be shown to be $\Omega(n^3)$ and thus $\Theta(n^3)$

Correctness

- Does it really work correctly?
  - We have to prove the ***optimal sub-structure***:
    - *If an optimal solution A to S_{ij} includes A[k], then it also includes optimal solutions to S_{ik} and S_{kj}*
    - To prove use "cut-and-paste" argument

Activity Selection DP Alg. 2.0

- Alternative way of thinking about it – ***binary choice***:
  - Sort activities on the start time (have "sentinel" activity $A[n+1]$ after all the other activities)
  - Let $next(i) = \min\{k \mid k > i \ \wedge \ \neg overlaps(A[i], A[k])\}$
  - The subproblem is then to schedule all the activities starting with i and after.
  - *What is the recurrence?*

$$c[i] = \begin{cases} 0 & \text{if } i > n, \\ \max\left(1 + c[next(i)],\ c[i+1]\right) & \text{otherwise}. \end{cases}$$

  - *MaxN*(*A*) = *c*[1]
  - *What is the running time and space used*?
    - Don't forget *next(i)*...

Greedy choice

- What if we could choose “the best” activity (as of now) and be sure that it belongs to an optimal solution
  - We wouldn’t have to check out all these sub-problems and consider all currently possible choices!

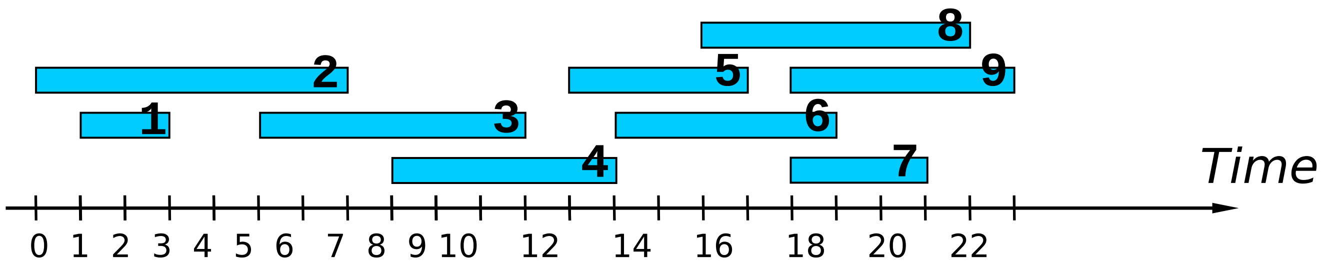
- Idea: Choose the activity that finishes first!
  - Intuition: leave as much time as possible for other activities
  - Then, solve *only one* sub-problem for the remaining compatible activities
  - (Have sentinel activity $A[0].f = 0$ – before all other)

```
MaxN(A[0..n], i)  //returns a set of activities
m = i + 1
while m ≤ n and A[m].s < A[i].f do
   m = m + 1
if m ≤ n then return {A[m]} ∪ MaxN(A, m)
            else return ∅
```

Iterative algorithm

```
MaxNi(A[0..n])  //returns a set of activities
R = {A[1]}
k = 1            // A[k] is always the last added activity
for m = 2 to n
   if A[m].s ≥ A[k].f then  // first compatible A[m]
     R = R ∪ {A[m]}
     k = m
return R
```

- Let’s run it:



- *What is the running time?*

Greedy-choice property

- Does it find an optimal solution?:
  - We have to prove the *optimal sub-structure* property (we did that already)
  - We have to prove the *greedy-choice property*, i.e., that our locally optimal choice a_1 belongs to some globally optimal solution.
    - Let A be an optimal solution and x be an activity with smallest finishing time **in** $\boldsymbol{A}$.
    - $a_1.f \leq x.f$, as a_1 is the activity with the smallest finishing time overall.
    - Thus, we can replace x with a_1 in A to get a valid solution of the same size!
  - *Greedy exchange* proof.

Greedy-choice property

- The challenge is to choose the right interpretation of "the best choice":
  - How about the activity that starts first?
    - Show a *counter-example*
  - The shortest activity?
  - The activity that overlaps the smallest number of the remaining activities?

Greedy choice property

- How about the activity that starts first?

a1	a2	a3
1	2	4
10	3	6

  - {a2, a3}, but not a1 that starts first.

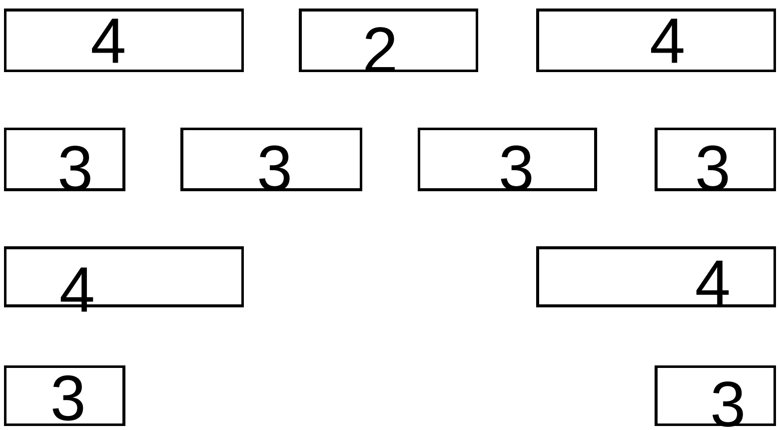
- The shortest activity?

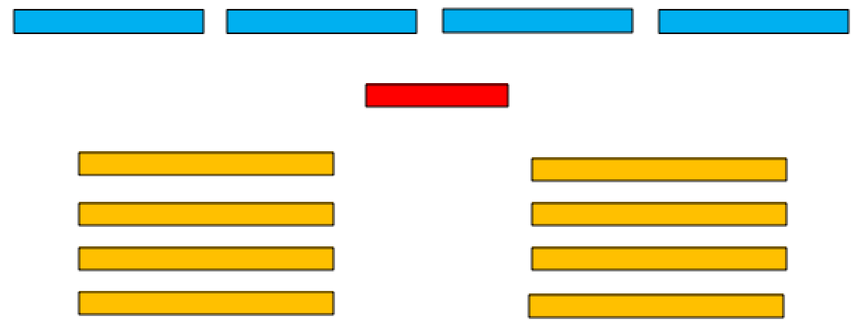
a1	a2	a3	a4
1	11	21	9
10	20	30	12

  - {a1, a2, a3}, but not a4 that is the shortest activity.

Greedy choice property

- The activity that overlaps the smallest number of the remaining activities?





- The second row gives the maximum-size set of mutually compatible activities, but it does not include the activity with the smallest overlaps, i.e., the one with 2.

Data Compression

- *Data compression* problem – strings *S* and *S'*:
  - $S \rightarrow S' \rightarrow S$, such that $|S'| < |S|$

- Text compression by coding with *variable-length* code:
  - Obvious idea – *assign short codes to frequent characters*: "**`abracadabra`**"

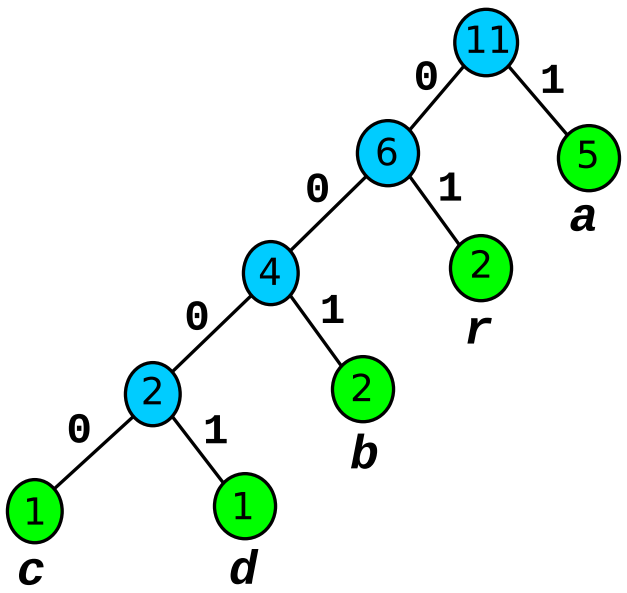
Frequency table:

	a	**b**	**c**	**d**	**r**
Frequency	5	2	1	1	2
Fixed-length code	**000**	**001**	**010**	**011**	**100**
Variable-length code	**1**	**001**	**0000**	**0001**	**01**

- *How much do we save in this case?*

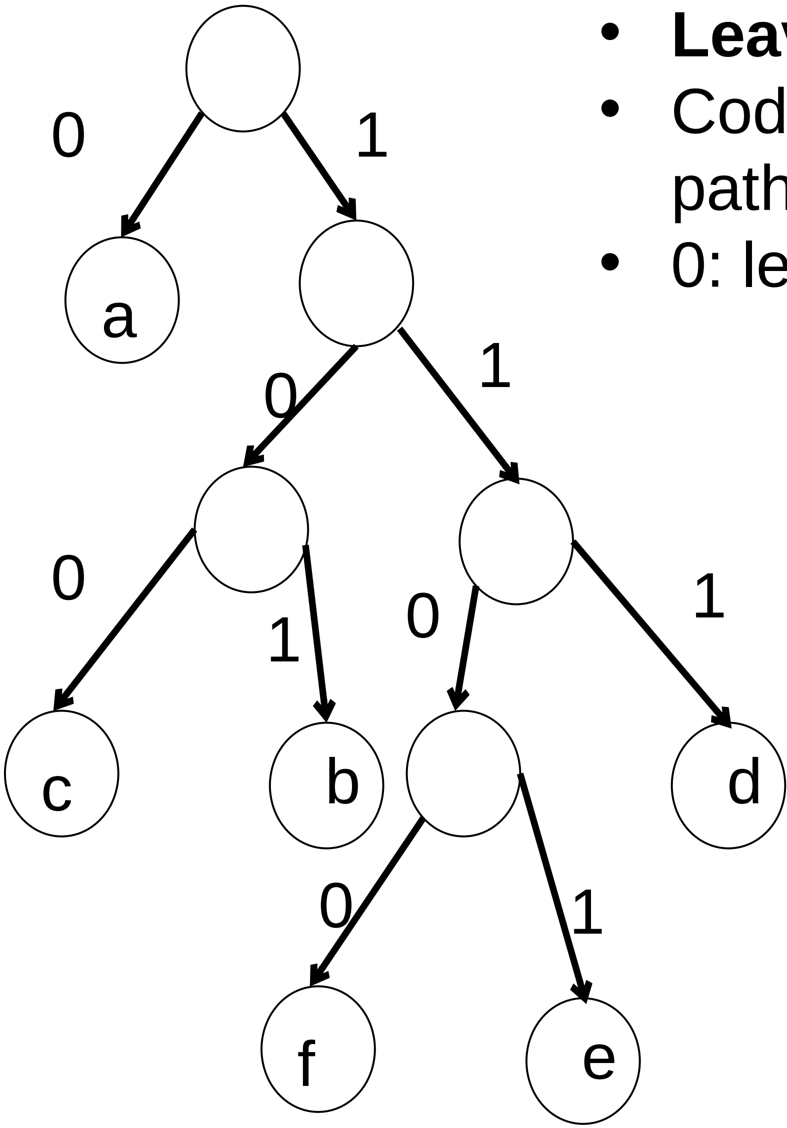
Prefix code

- Optimal code for the given frequencies:
  - Achieves the minimal length of the coded text
- *Prefix code*: no codeword is a prefix of another
  - It can be shown that optimal coding can be done with prefix code



- We can store all codewords in a *binary trie* – very easy to decode
  - Coded characters in leaves
  - Each node contains the sum of the frequencies of all descendants

Decoding using a binary trie



- **Leaves** represent characters.
- Codeword for a character is the simple path from the root to that character.
- 0: left 1:right

Decode: **001011101**

- Go to Socrative and write in your answer.

Optimal Code/Trie

- The *cost* of the coding trie *T*:

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

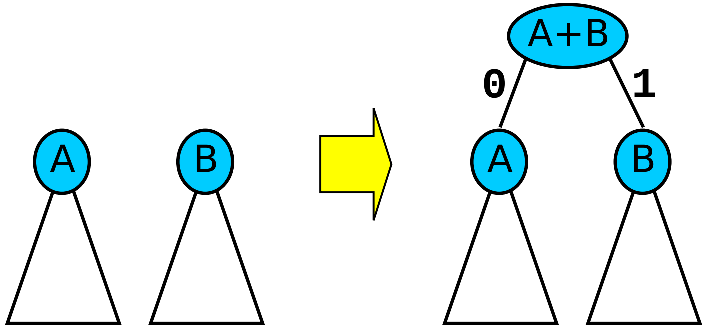
  - *C* – the alphabet,
  - $f(c)$ – frequency of character *c*,
  - $d_T(c)$ – depth of *c* in the trie (length of code in bits)

- Optimal trie – the one that minimizes $B(T)$

- Observation – optimal trie is always full:
  - Every non-leaf node has two children. Why?

Huffman Algorithm - Idea

- Huffman algorithm, builds the code trie bottom up. Consider a forest of trees:
  - Initially – one separate node for each character.
  - In each step – join two trees into a larger tree



  - Repeat this until one tree (trie) remains.
  - Which trees to join? Greedy choice – the trees with the **smallest** frequencies!

Huffman Algorithm

```
Huffman(C)
Q.build(C) // Builds a min-priority queue on frequencies
for i ← 1 to n–1 do
   Allocate new node z
   x ← Q.extractMin()
   y ← Q.extractMin()
   z.setLeft(x)  // corresponding to bit 0
   z.setRight(y) // corresponding to bit 1
   z.setF(x.f() + y.f())
   Q.insert(z)
return Q.extractMin() // Return the root of the trie
```

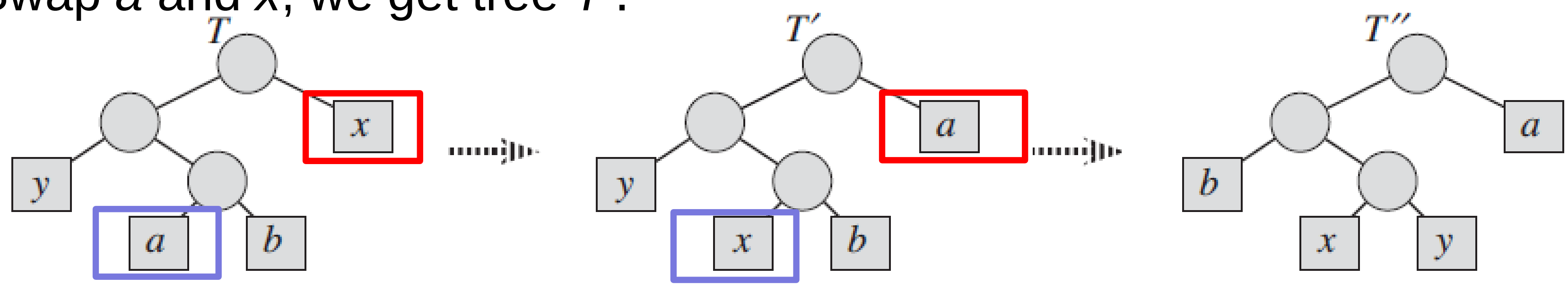
- *What is its running time?*
- Run the algorithm on: "**oho ho, ole**"

Correctness of Huffman

- Greedy choice property:
  - Let *x, y* – two characters with lowest frequencies. Then there exists an optimal prefix code where codewords for *x* and *y* have the same length and differ only in the last bit
  - Let's prove it:
    - Transform an optimal trie *T* into one (*T''*), where *x* and *y* are max-depth siblings. Compare the costs.

Greedy choice property

- Let *x* and *y* be the two characters with lowest frequencies.
- Let's assume that we have an optimal code tree *T*, where leaves a and b are two siblings of the maximum depth.
- Swap *a* and *x*, we get tree *T'*.



$$
\begin{aligned}
B(T) - B(T') &= \sum_{c \in C} c.freq \cdot d_T(c) - \sum_{c \in C} c.freq \cdot d_{T'}(c) \\
&= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot \boxed{d_{T'}(x)} - a.freq \cdot \boxed{d_{T'}(a)} \\
&= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot \boxed{d_T(a)} - a.freq \cdot \boxed{d_T(x)} \\
&= \boxed{(a.freq - x.freq)}\boxed{(d_T(a) - d_T(x))} \\
&\geq 0 ,
\end{aligned}
$$

B(T) ≥ B(T')

Since *x* and *y* are the two characters with lowest frequencies, we have **x.freq ≤ a.freq**

In tree *T, a* and *b* are two siblings of maximum depth. Thus, we have **$d_T(a) \geq d_T(x)$**

Correctness of Huffman

- Optimal sub-structure property:
  - Let x, y – characters with minimum frequency
  - $C' = C - \{x,y\} \cup \{z\}$, such that $f(z) = f(x) + f(y)$
  - Let T' be an optimal code trie for C'
  - Replace leaf z in T' with internal node with two children x and y
  - The result tree T is an optimal code trie for C
- Proof a little bit more involved than a simple “cut-and-paste” argument

Elements of Greedy Algorithms

- Greedy algorithms are used for optimization problems
  - A number of choices have to be made to arrive at an optimal solution.
  - At each step, make the "locally best" choice, without considering all possible choices and solutions to sub-problems induced by these choices (compare to dynamic programming).
  - After the choice, only one sub-problem remains (smaller than the original).

- Greedy algorithms usually sort or use priority queues.

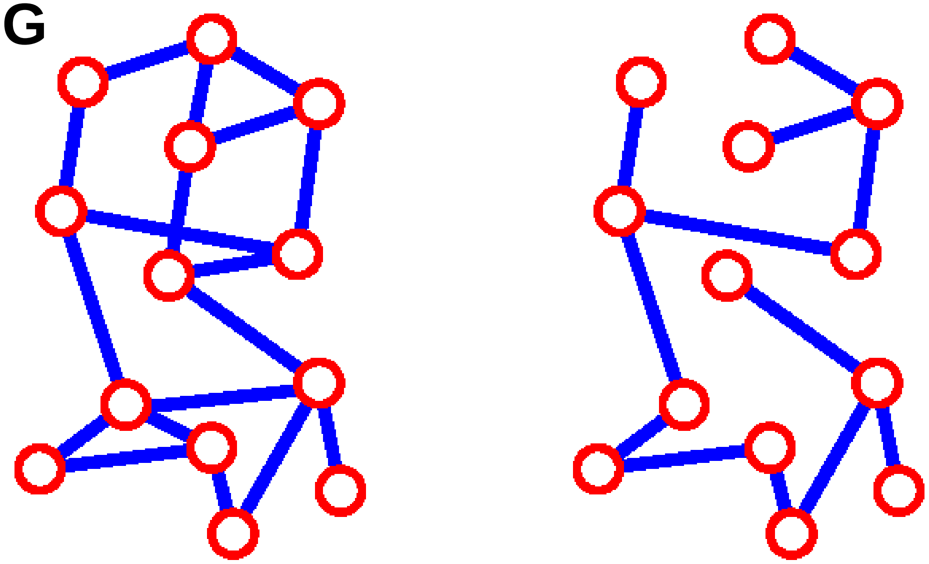
Elements of Greedy Algorithms

- First, one has to prove the *optimal sub-structure* property
  - the simple “cut-and-paste” argument may work
  - Not always possible (sign that it is a hard problem):
    - Longest (vs. shortest) simple unweighted path
    - Maximum clique in a graph (vs. activity selection)

- The main challenge is to decide the interpretation of “the best” so that it leads to a global optimal solution, i.e., you can prove the *greedy choice property*
  - The proof is usually constructive: takes a hypothetical optimal solution without the specific greedy choice and transforms into one that has this greedy choice.
  - Or you find counter-examples demonstrating that your greedy choice does not lead to a global optimal solution.

Spanning Tree

- A **spanning tree** of **G** is a subgraph which
  - is a tree
  - contains all vertices of **G**

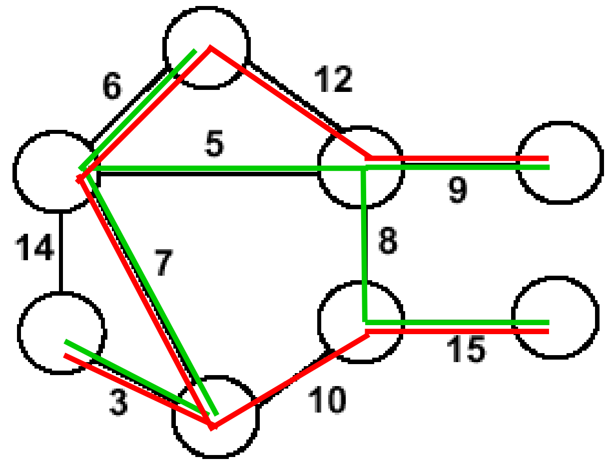


G   spanning tree of **G**

- How many edges are there in a spanning tree, if there are *V* vertices?

Minimum Spanning Trees

- Undirected, connected graph $G = (V,E)$
- **Weight** function $W: E \rightarrow R$ (assigning cost or length or other values to edges)



- Spanning tree: a tree that connects all the vertices
- Optimization problem – **Minimum spanning tree** (MST): tree T that connects all the vertices and minimizes $w(T)=\sum_{(u,v)\in T} w(u,v)$

Idea for an algorithm

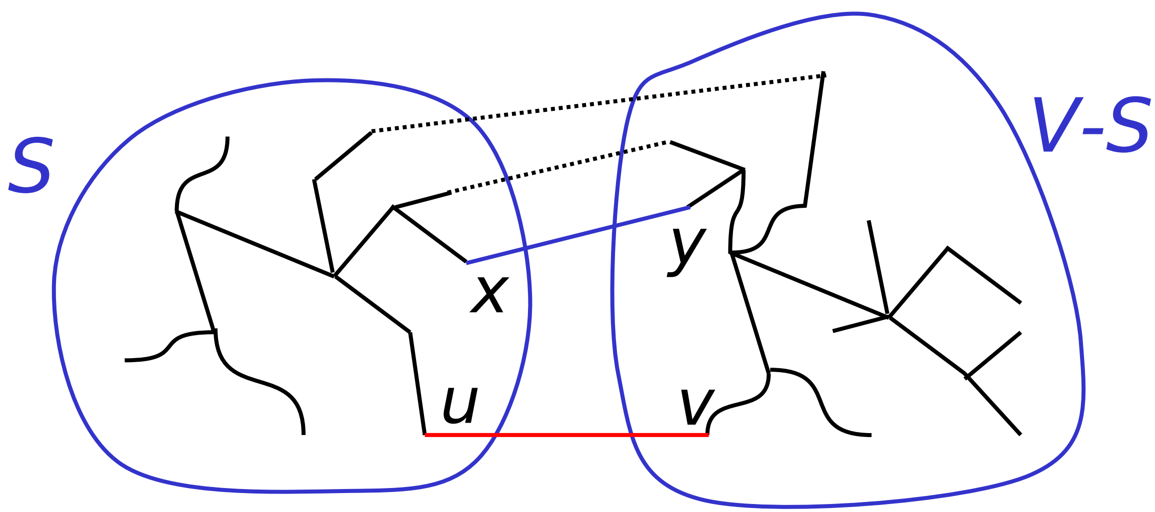
- We have to make V – 1 choices (edges of the MST) to arrive at the optimization goal
- After each choice we have a sub-problem one vertex smaller than the original
  - Dynamic programming algorithm, at each stage, would consider all possible choices (edges)
  - If only we could always guess the correct choice – an edge that definitely belongs to an MST

Greedy Choice

- Greedy choice property: locally optimal (greedy) choice yields a globally optimal solution
- Theorem
  - Let $G=(V, E)$, and let $S \subseteq V$ and
  - let (u,v) be *min*-weight edge in G connecting S to $V - S$: a **light** edge crossing a **cut**
  - Then $(u,v) \in T$ – some MST of G

Greedy Choice (2)

- Proof
  - Suppose (u,v) is light, but (*u*,*v*) $\notin$ any MST
  - look at path from *u* to *v* in some MST *T*
  - Let (*x, y*) – the first edge on path from *u* to *v* in *T* that crosses from *S* to *V – S.* Swap (*x, y*) with (*u,v*) in *T.*
  - this improves *T* – a contradiction (*T* is supposed to be an MST)
    - *if w(x,y) = w(u,v), we get an alternative MST with* (*u,v*) *included*



Generic MST Algorithm

```
Generic-MST(G, w)
1 A←∅    // Contains edges that belong to a MST
2 while A does not form a spanning tree do
3    Find an edge (u,v) that is safe for A
4    A←A∪{(u,v)}
5 return A
```

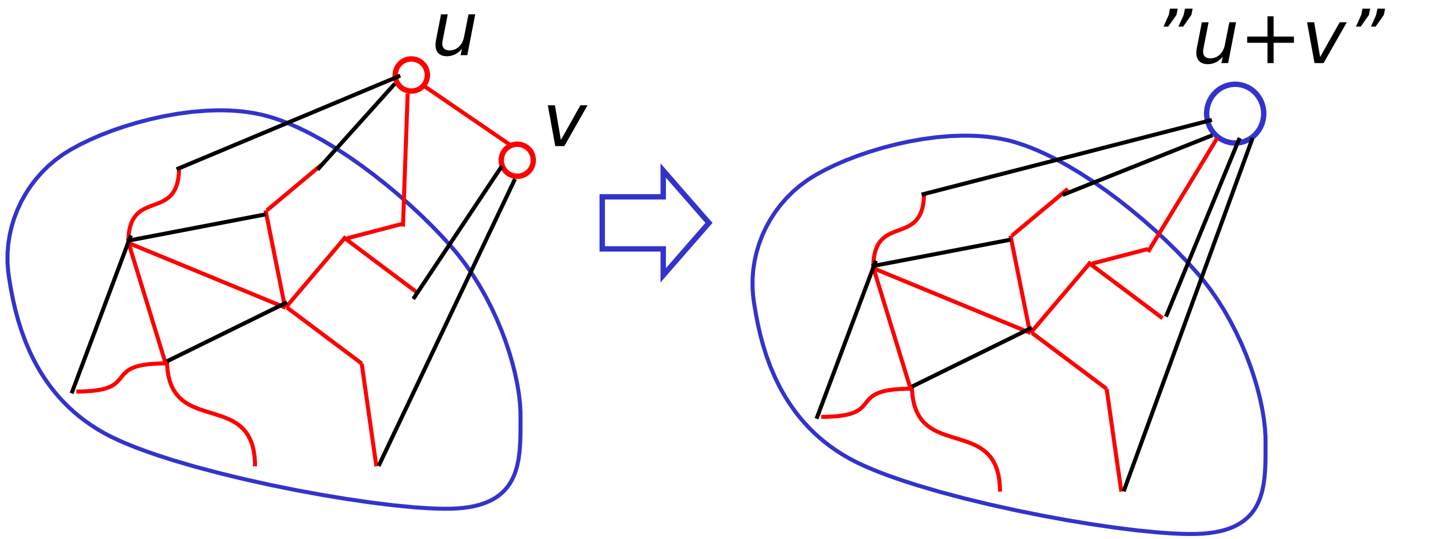
Safe edge – an edge that does not destroy *A*'s property

```
MoreSpecific-MST(G, w)
1   A←∅    // Contains edges that belong to a MST
2   while A does not form a spanning tree do
3.1    Make a cut (S, V-S) of G that respects A
3.2    Take the min-weight edge (u,v) connecting S to V-S
4      A←A∪{(u,v)}
5 return A
```

Greedy graph algorithms

- Prim's and Kruskal's for MST, Dijkstra's for shortest paths: in each step, conceptually merge vertices into larger "supervertices".

$MST(G) = T$ $\qquad\qquad$ $MST(G') = T - (u,v)$



- Optimal substructure: ***If*** (u,v) is in an MST T ***then*** T - (u,v) is an MST of G'
- "Cut and paste" argument
  - If G' would have a cheaper ST T', then we would get a cheaper ST of G: $T' + (u, v)$
- Greedy choice: choose an edge incident on the "supervertex" with a minimum weight