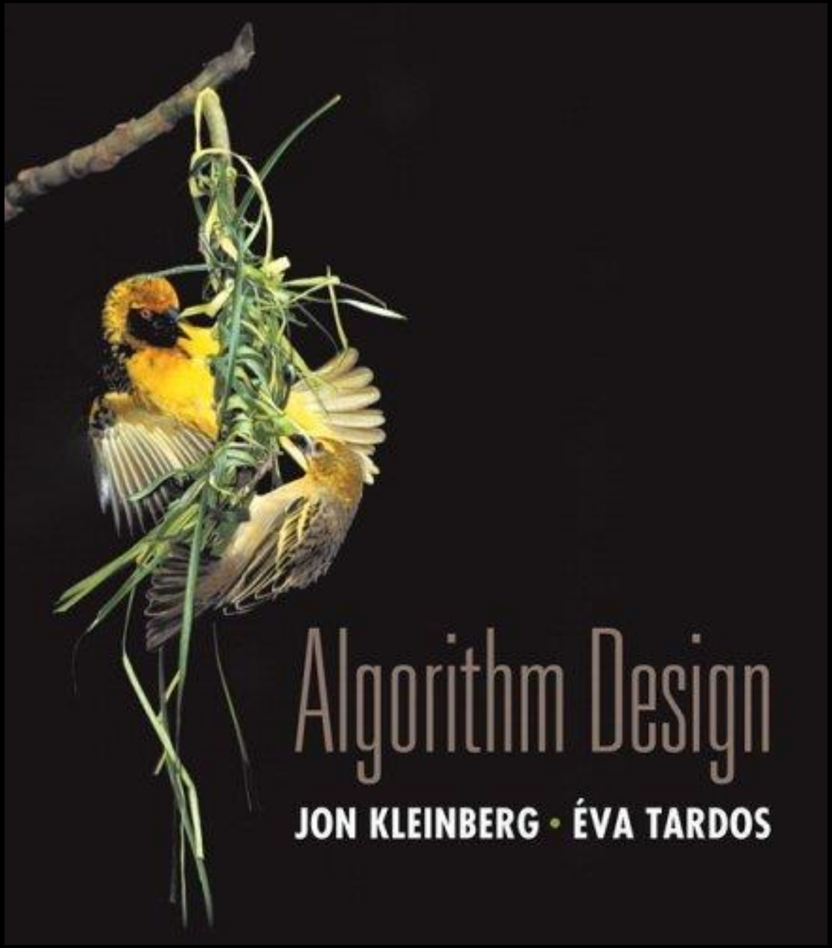


Chapter 4

Greedy Algorithms

Greedy Algorithms

General greedy algorithms. Consider the items in a specific order, makes "greedy" decisions

Pro: Typically very simple to implement and very efficient

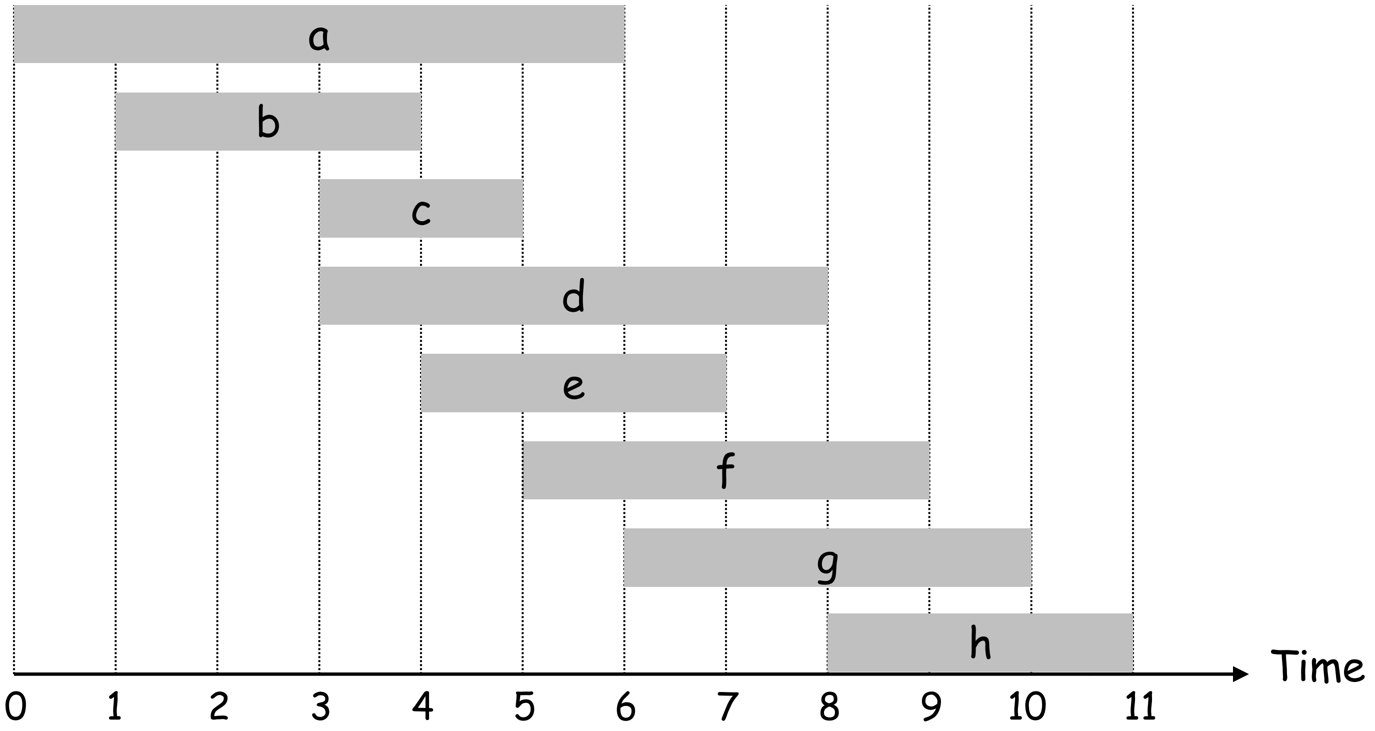
Con: Cannot solve every problem using a greedy algorithm, need to guarantee it works

4.1 Interval Scheduling

Interval Scheduling

Interval scheduling.

- Job j starts at s_j and finishes at f_j.
- Two jobs compatible if they don't overlap.
- Goal: find maximum subset of mutually compatible jobs.



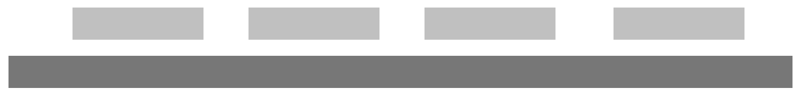
Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some order. Take each job provided it's compatible with the ones already taken.

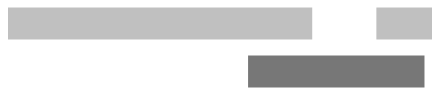
- [Earliest start time] Consider jobs in ascending order of start time s_j.
- [Earliest finish time] Consider jobs in ascending order of finish time f_j.
- [Shortest interval] Consider jobs in ascending order of interval length $f_j - s_j$.
- [Fewest conflicts] For each job, count the number of conflicting jobs c_j. Schedule in ascending order of conflicts c_j.

Interval Scheduling: Greedy Algorithms

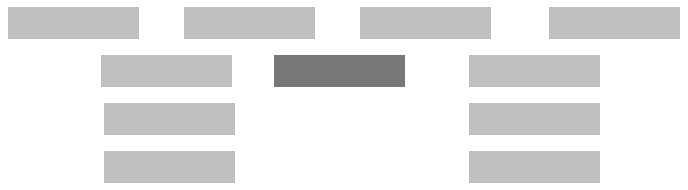
Greedy template. Consider jobs in some order. Take each job provided it's compatible with the ones already taken.



breaks earliest start time



breaks shortest interval



breaks fewest conflicts

Interval Scheduling: Greedy Algorithm

Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it's compatible with the ones already taken.

```
Sort jobs by finish times so that $f_1 \le f_2 \le ... \le f_n$.

  jobs selected
 ↙
A ← ϕ
for j = 1 to n {
   if (job j compatible with A)
      A ← A ∪ {j}
}
return A
```

Implementation.

- Remember job j* that was added last to A.
- Just need to check compatibility with last job: Job j is compatible with A if $s_j \ge f_{j^*}$.

Interval Scheduling: Greedy Algorithm

Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it's compatible with the ones already taken.

```
Sort jobs by finish times so that $f_1 \le f_2 \le ... \le f_n$.

  jobs selected
$A \leftarrow \phi$
for j = 1 to n {
   if (job j compatible with A)
      $A \leftarrow A \cup \{j\}$
}
return A
```

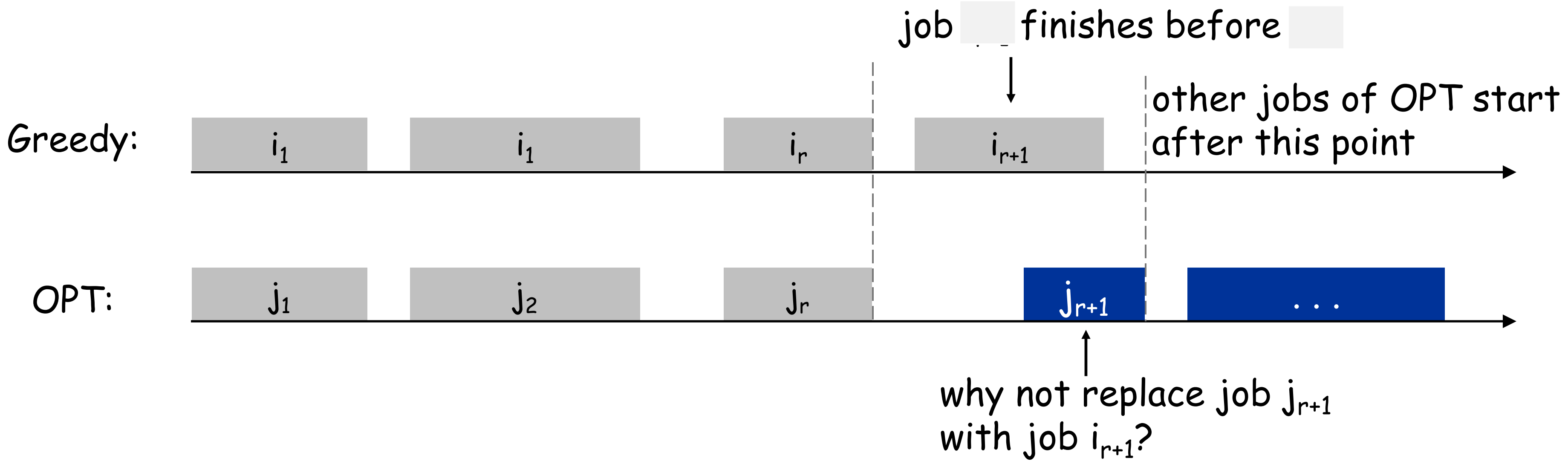
Implementation. O(n log n).

- Remember job j* that was added last to A.
- Just need to check compatibility with last job: Job j is compatible with A if $s_j \ge f_{j^*}$.

Interval Scheduling: Analysis

Theorem. Greedy on finish time algorithm is optimal.

Pf. Consider the Greedy and the Optimal solutions:

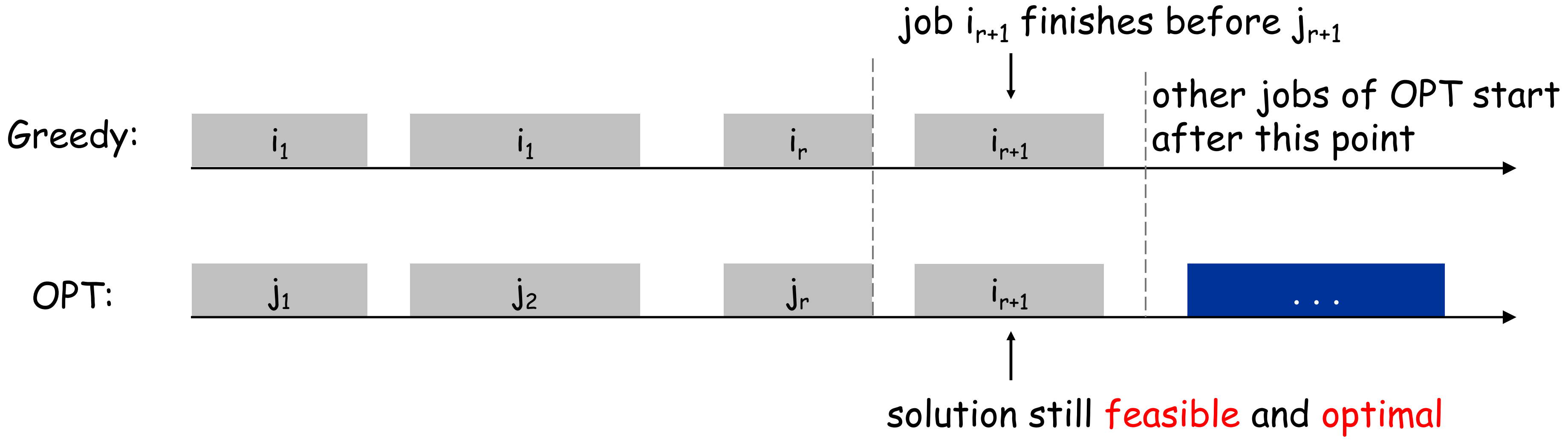

job finishes before
other jobs of OPT start after this point
Greedy:
i_1
i_1
i_r
i_{r+1}
OPT:
j_1
j_2
j_r
j_{r+1}
. . .
why not replace job j_{r+1} with job i_{r+1}?

Interval Scheduling: Analysis

Theorem. Greedy on finish time algorithm is optimal.

Pf. Consider the Greedy and the Optimal solutions:

- We can swap the first job where OPT does something different => gives a solution that is still optimal but more similar to Greedy
- Repeating the swap, we get optimal solutions closer and closer to greedy, until we get it equal to greedy solution => greedy is optimal

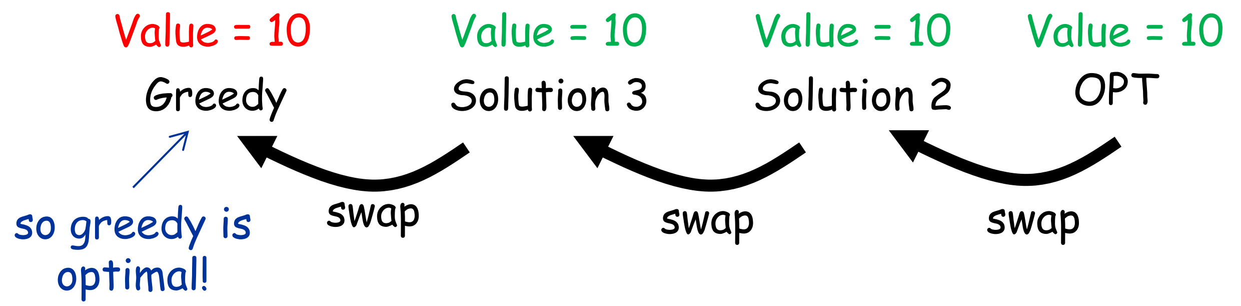

job i_{r+1} finishes before j_{r+1}
other jobs of OPT start after this point
Greedy:
i_1
i_1
i_r
i_{r+1}
OPT:
j_1
j_2
j_r
i_{r+1}
. . .
solution still feasible and optimal

Greedy Algorithms

The hard part of greedy algorithms is to argue it returns the optimal solution

A very comon strategy for analysis is the swap:

- Replace one action of OPT for one action of Greedy, and show value does not get worse
- Repeating we bring OPT closer and closer until we reach Greedy



This is how we analyzed the Interval Scheduling problem

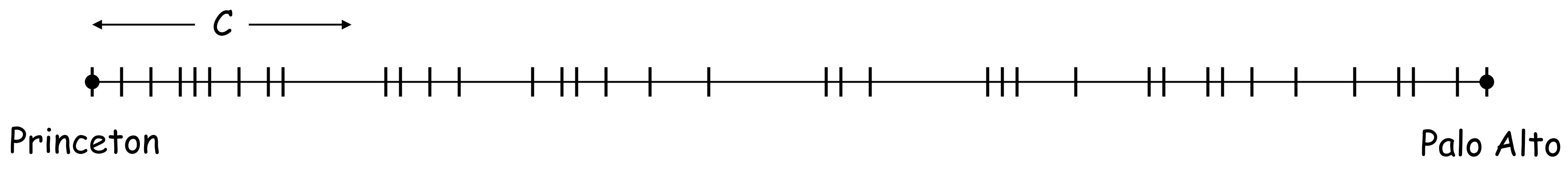
Let's see a few more examples

Example: Selecting fueling points

Selecting fueling points

Selecting fueling points.

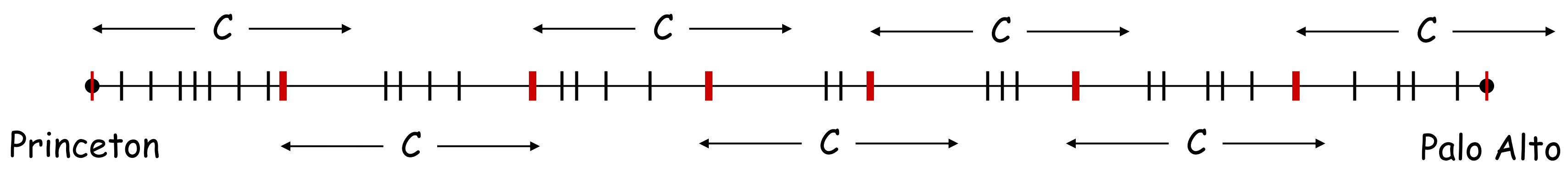
- Road trip from Princeton to Palo Alto along fixed route.
- There are refueling stations at certain points along the way.
- Fuel capacity = *C*.
- Goal: Choose smallest number of refueling stops that make the trip possible



Selecting fueling points

Selecting fueling points.

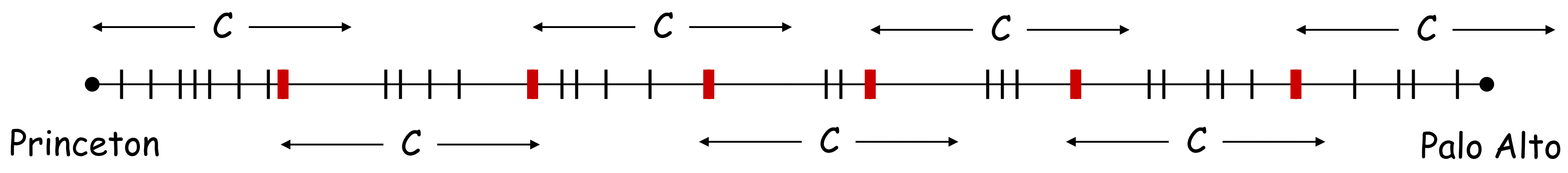
- Road trip from Princeton to Palo Alto along fixed route.
- There are refueling stations at certain points along the way.
- Fuel capacity = C.
- Goal: Choose smallest number of refueling stops that make the trip possible



Task: Design a greedy algorithm for this problem, and give a brief argument for why it returns an optimal solution

Selecting fueling points

Solution: Go as far as you can before refueling.



To argue it returns an optimal solution, think "where could it go wrong/where is another algorithm smater"?

Maybe another algorithm could stop before it is required

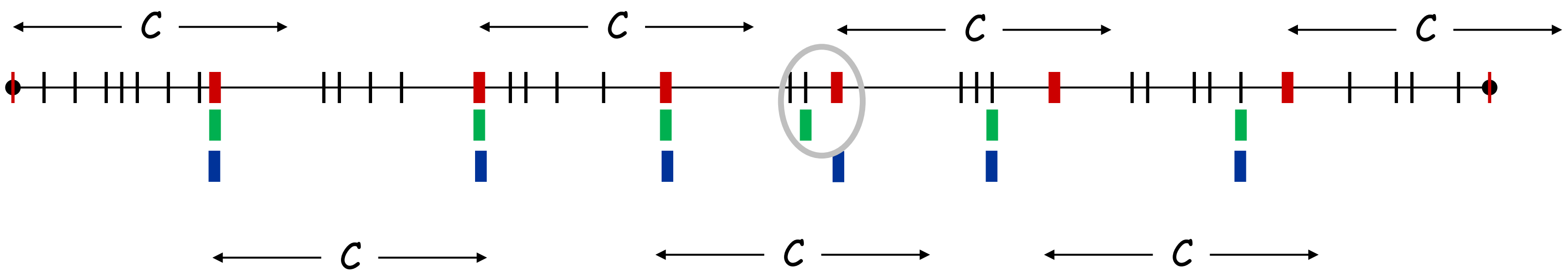
But that does not seem to help...

Selecting Breakpoints: Correctness

Suppose these are the **greedy** solution and the OPT solution

Q: Is there a swap that gives a valid solution of same cost as OPT but brings OPT closer to greedy?

A: Swap first stop where they disagree (swapped solution in **blue**)



Reapeating such swap shows that greedy has same cost as OPT

Theorem. Greedy algorithm is optimal.

Scheduling to Minimize Lateness

Scheduling to Minimizing Lateness

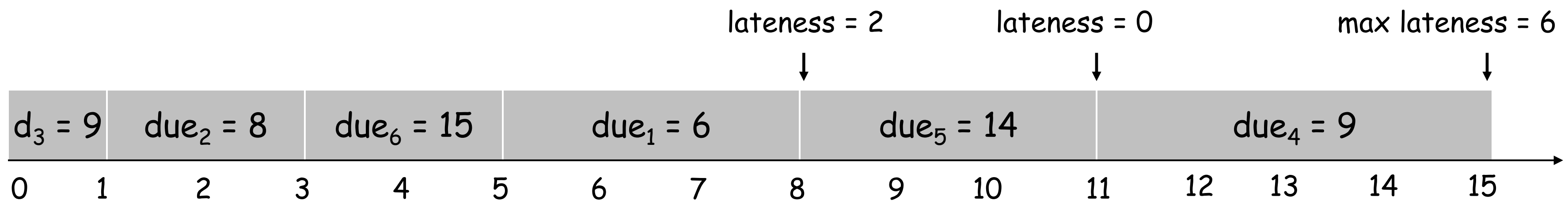
Minimizing lateness problem.

- Machine processes one job at a time.
- Job j requires $time_j$ units of processing time and is due at time due_j.
- If j starts at time $start_j$, it finishes at time $finish_j = start_j + time_j$.
- Lateness: $late_j = 0$ if finished before deadline, otherwise $late_j = due_j - finish_j$
- Goal: schedule all jobs to minimize maximum lateness = max $late_j$.

Aplication: Project management

Ex:

	1	2	3	4	5	6
$time_j$	3	2	1	4	3	2
due_j	6	8	9	9	14	15



[Try to find optimal solution for this example]

Minimizing Lateness: Greedy Algorithms

Greedy template. Process jobs in some order.

- [Shortest processing time first] Consider jobs in ascending order of processing time $time_j$.

- [Earliest deadline first] Consider jobs in ascending order of deadline due_j.

- [Smallest slack] Consider jobs in ascending order of slack $due_j - time_j$.

Minimizing Lateness: Greedy Algorithms

Greedy template. Process jobs in some order.

- [Shortest processing time first] Consider jobs in ascending order of processing time time_j.

	1	2
t_j	1	10
d_j	100	10

counterexample

- [Smallest slack] Consider jobs in ascending order of slack $\text{due}_j - \text{time}_j$.

	1	2
t_j	1	10
d_j	2	10

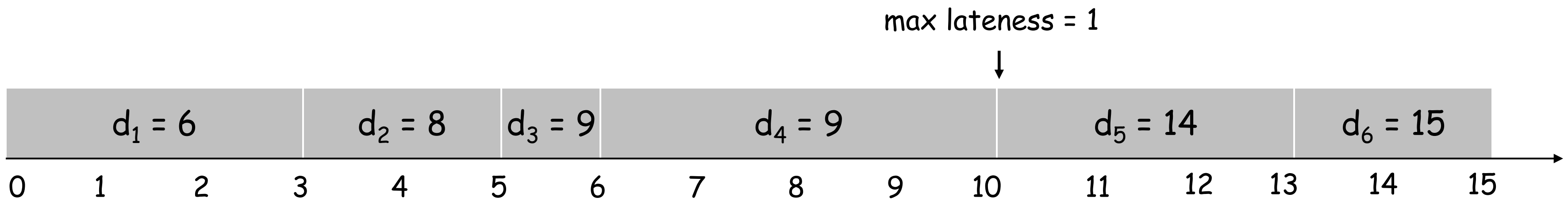
counterexample

Minimizing Lateness: Greedy Algorithm

Greedy algorithm. Earliest deadline first.

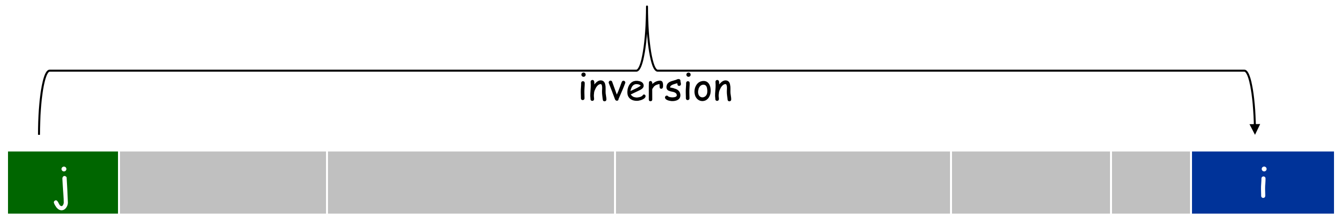
```
Sort n jobs by deadline so that $d_1 \le d_2 \le … \le d_n$

for j = 1 to n
   Assign job j to next available time period
```



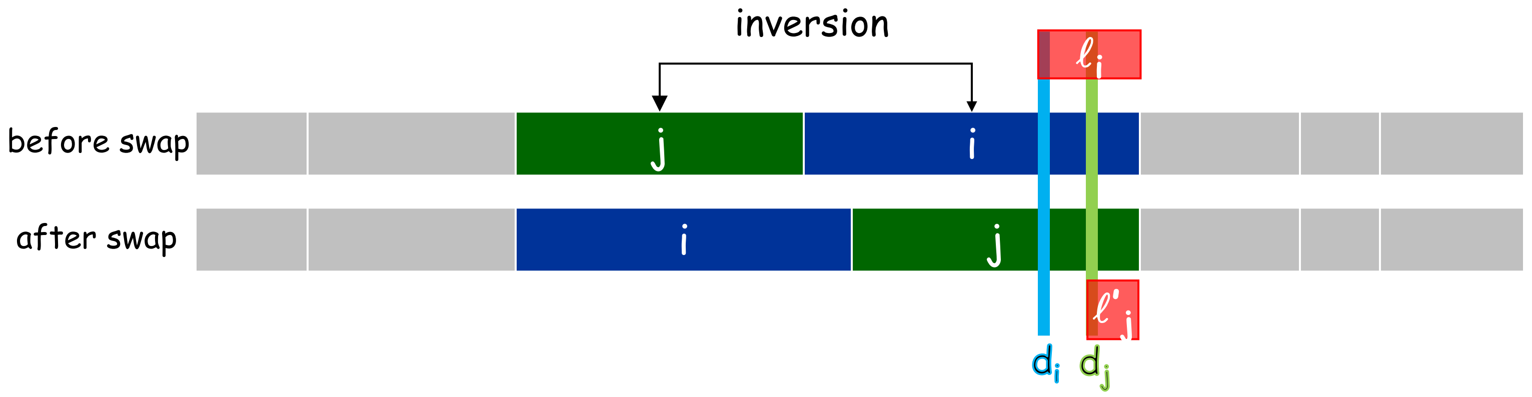
Minimizing Lateness: Idea of analysis

Def. An inversion in schedule S is a pair of jobs i and j such that: $d_i < d_j$ but j scheduled before i.



Observation 1. Greedy schedule has no inversions.

Observation 2. Swapping two adjacent, inverted jobs does not increase the max lateness

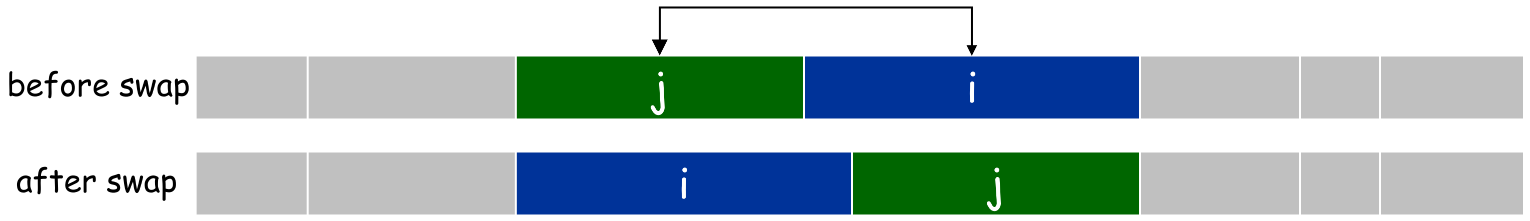


Minimizing Lateness: Idea of analysis

Swaps: Take OPT, keep swapping inverted jobs, brings it closer to greedy without increasing lateness

Repeating this we get a solution without any inversion (even non-adjacent, why?)

Reach solution where all items are in order of deadline => greedy solution (essentially, forget about ties)



Theorem. The Greedy algorithm in increasing order of due date is optimal

Fractional Knapsack Problem

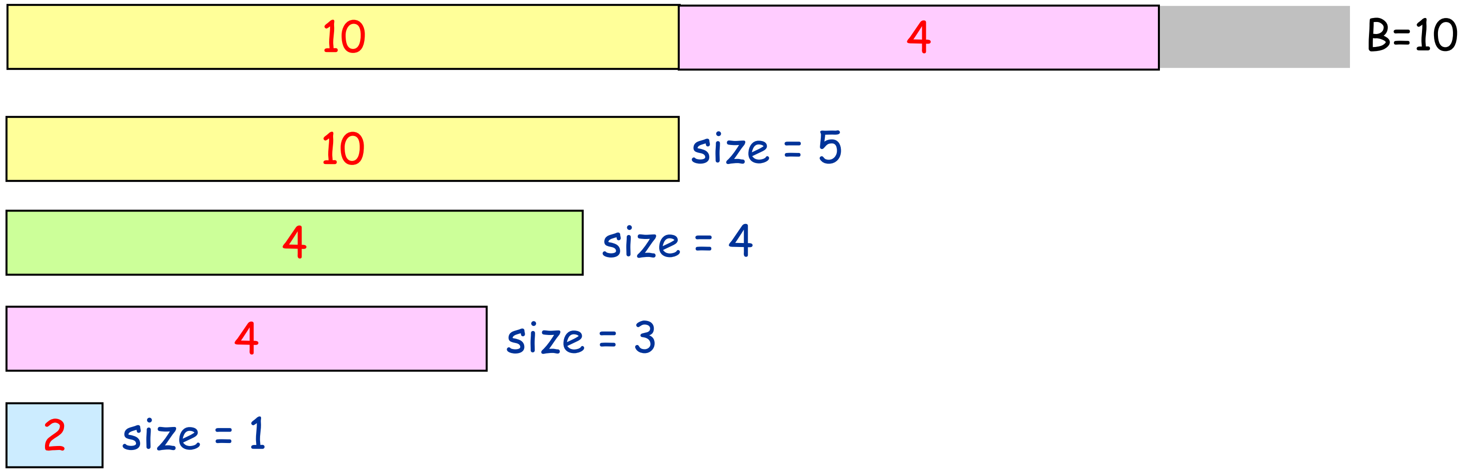
Fractional Knapsack Problem

Knapsack Problem:

- You have a backpack of size B and several items
- Each item i has a size $size_i$ and value $value_i$
- Goal: Select items to put in the backpack that maximizes total value

Application: Project selection. (B is budget, size is cost, value is revenue)

Ex:



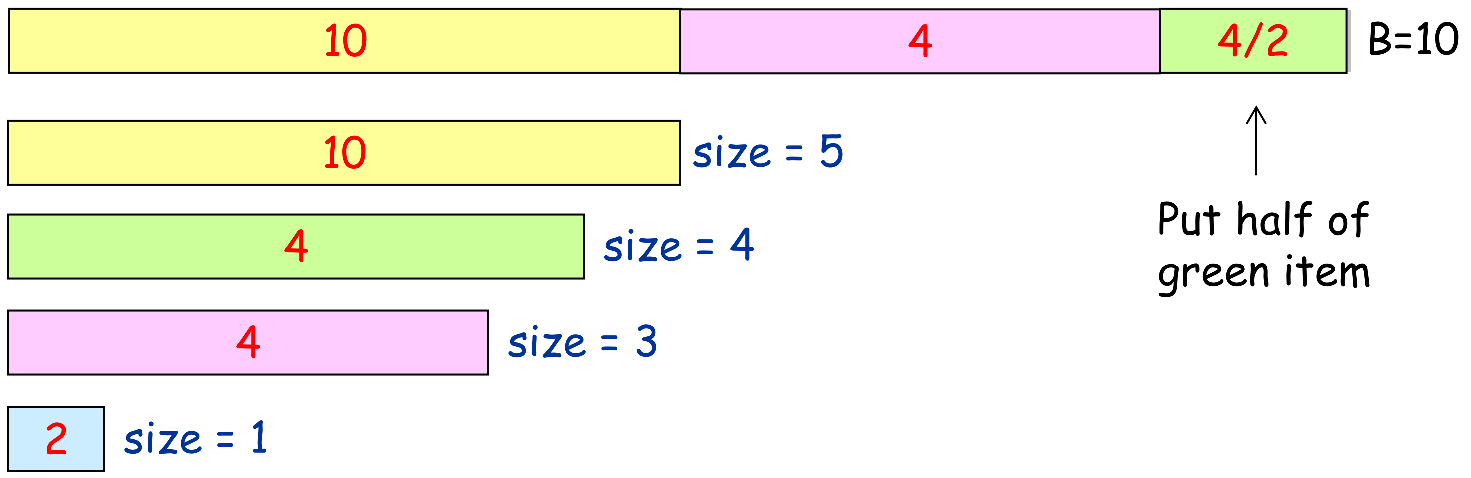
This is a harder problem, we will see in dynamic programming

Fractional Knapsack Problem

Fractional Knapsack Problem:

- You have a backpack of size B and several items
- Each item i has a size $size_i$ and value $value_i$
- Goal: Select items to put in the backpack that maximizes total value, can select just a fraction of an item (e.g. 10%)

Ex:



Q: Can you see how to improve this solution?

A: Replace $\frac{1}{4}$ of green item by the blue item

Fractional Knapsack Problem

Task: Give a greedy algorithm for the Fractional Knapsack Problem [Dantzig '57]

Greedy algorithm:

- sort items in decreasing order of value/size
- Scan items in this order, add as much of the item as you can to the backpack

Why it woks? Swap argument, based on what happened in previous example

If has item different from greedy => has lower value/size

- Swap a little bit of this item with some item in greedy => get closer to greedy solution and does not worsen the value

Repeat to get greedy solution

Theorem: Greedy by value/size is optimal.

[See MST slides]

Greedy Recap

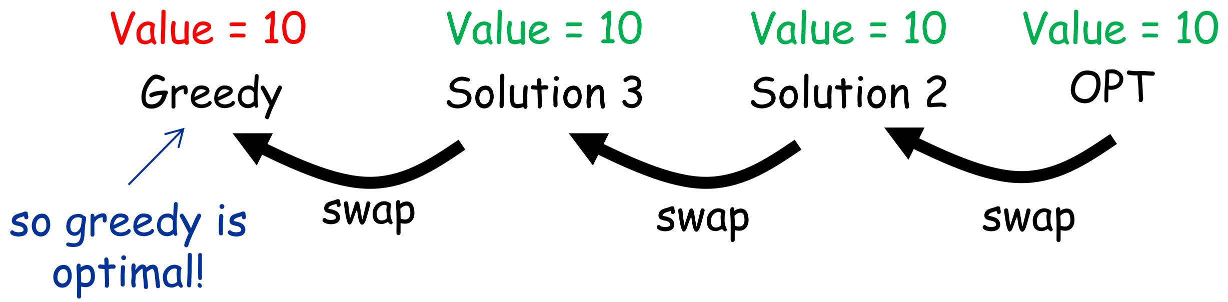
Greedy algorithms. Consider the items in a specific order, makes "greedy" decisions

Which order to use is crucial to guarantee the algorithm gives an optimal solution

Not every problem can be solved with a greedy strategy

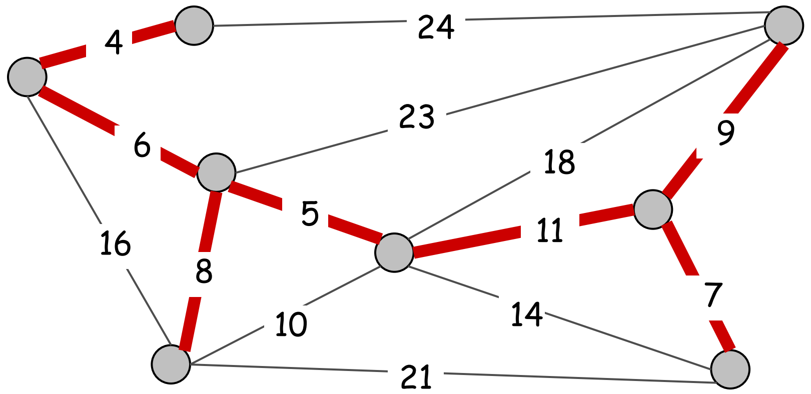
A very comon strategy for analysis is the swap:

- Replace one action of OPT for one action of Greedy, and show value does not get worse
- Repeating we bring OPT closer and closer until we reach Greedy



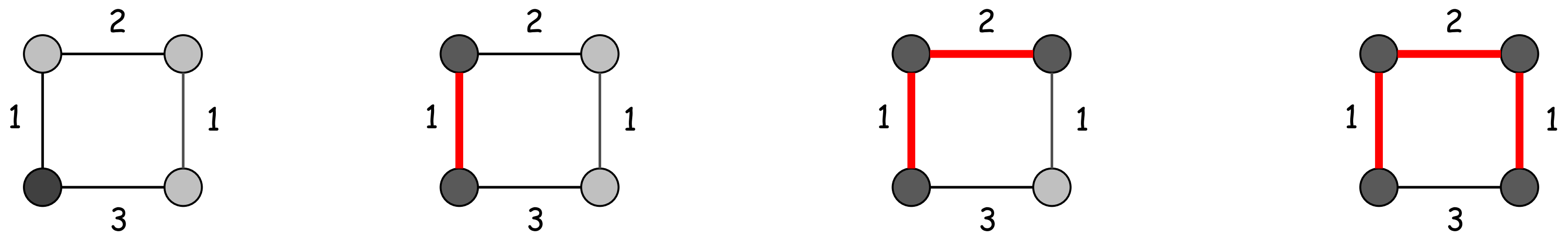
MST Recap

Minimum spanning tree problem (MST). Given a connected graph G with real-valued edge cost c_e, find a tree T in the graph that connects all nodes and has smallest total cost



MST Review: Algorithms

Prim. Grows a connected tree, starting from any node. Adds to the tree node with cheapest connection cost

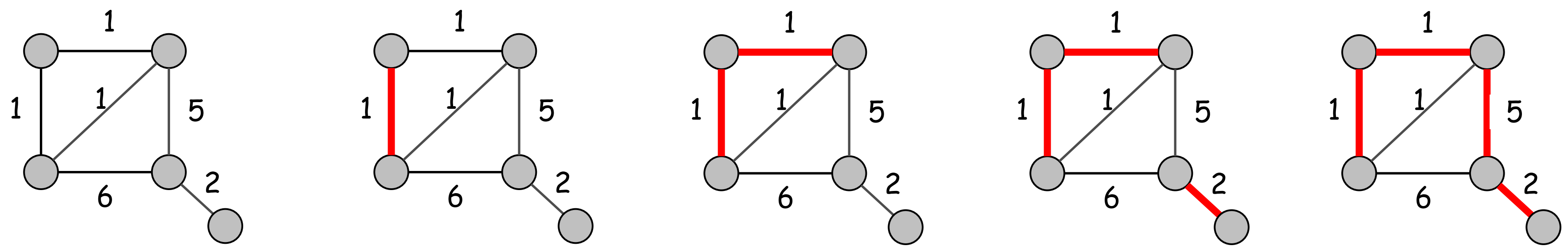


Complexity: O(m log n)

- Maintain connection cost a[v] = cost of cheapest edge v to a node in the tree in a heap

MST Review: Algorithms

Kruskal. Adds cheapest edge that does not form cycle



Complexity: O(m.n)

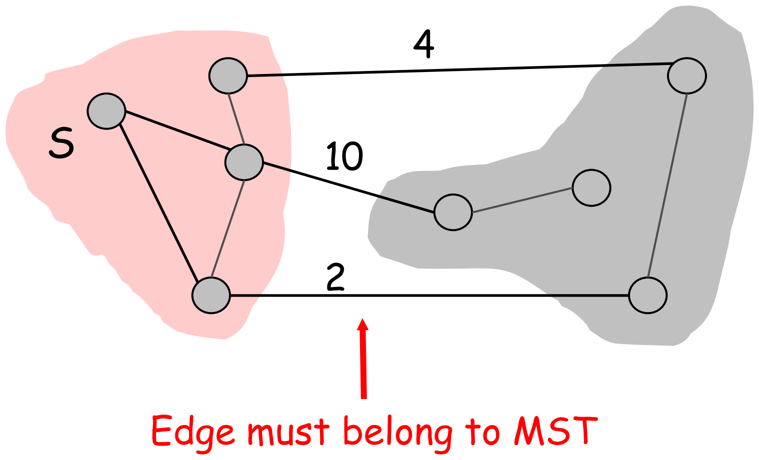
- O(m log m) to sort edges
- O(m.n) in total for checking if edge forms cycle, using BFS over the current tree

Obs: Can run in time O(m log n) if uses union-find data structure

MST Review: Cut property

The main tool we used to prove the correctness of MST algorithms was the cut property. Here is a simplified statement assuming all costs are distinct ← Guarantees there is unique MST

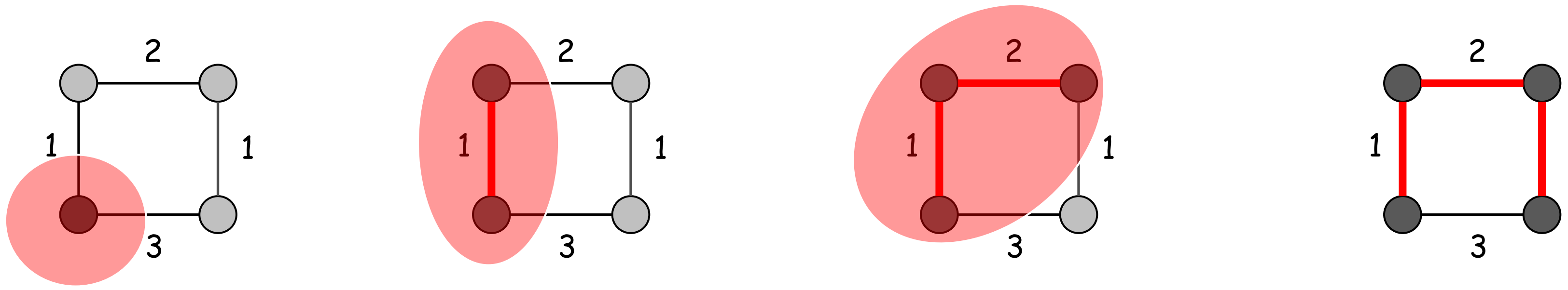
Cut property. Assume that all costs c_e are distinct. For every cut S, the cheapest edge crossing the cut must belong to the MST



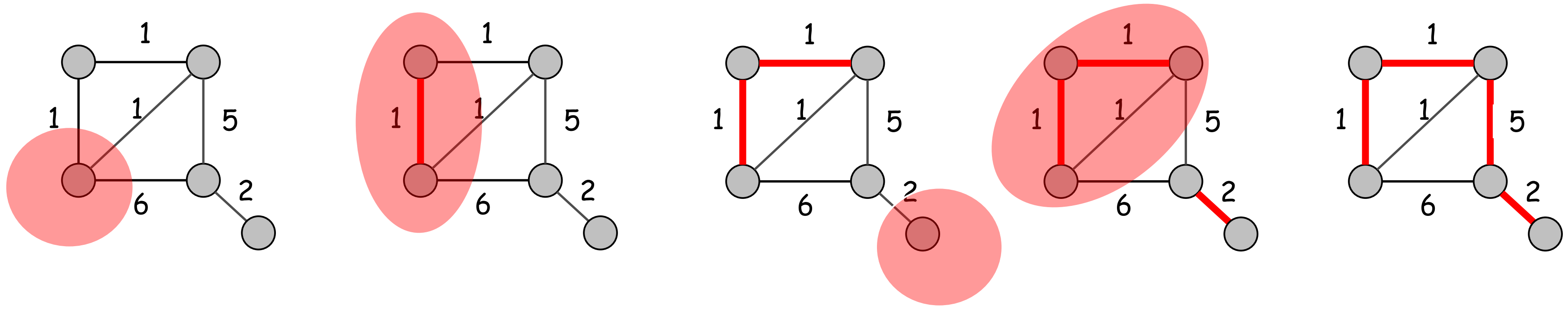
MST Review: Analysis using cut property

Use cut property to assert that all edges picked by the algorithm belong to MST

Prim.



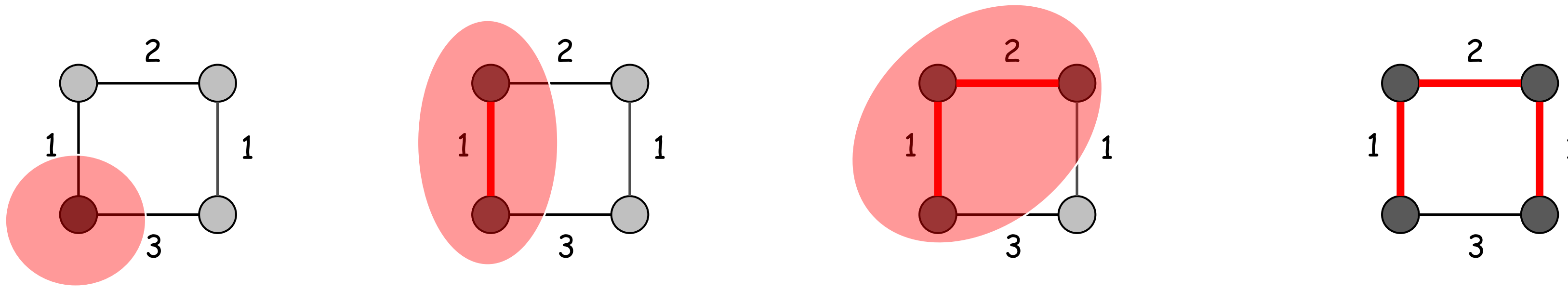
Kruskal. (o[redacted]
form cycle
current tree)



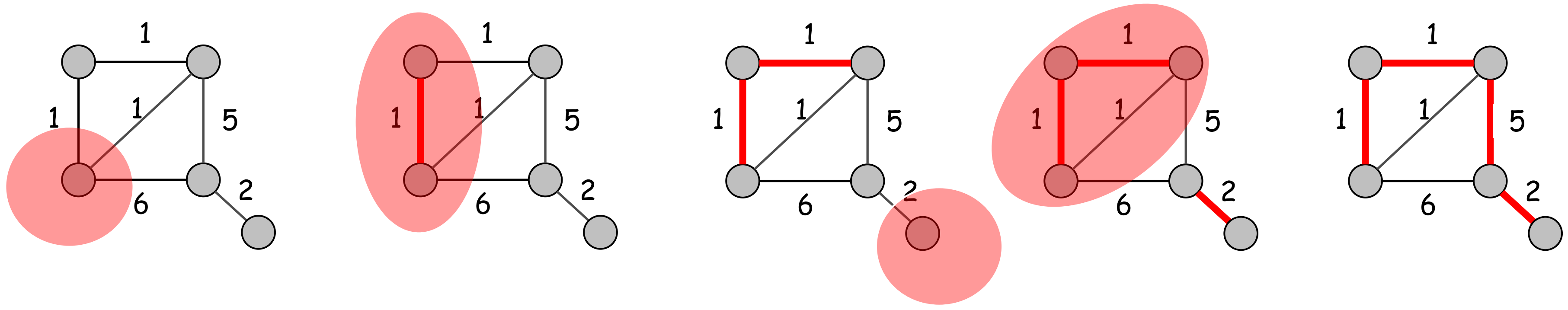
MST Review: Analysis using cut property

Use cut property to assert that all edges picked by the algorithm belong to MST

Prim.



Kruskal.



Results on greedy and MST

Greedy. There is a way of characterizing exactly the class of problems for which the greedy gives optimal solution (for a more formal definition of what "greedy algorithm" means)

Theorem: [Edmonds '71] The greedy algorithm works iff the problem has a matroid structure

Many extensions where "almost greedy" works: matroid intersection, polymatroid, greedoids, jump systems, …

MST. Best algorithms:

- Expected time O(m) [Karger, Klein & Tarjan '95]
- Deterministic O(m $\alpha(m,n)$) [Chazelle '00]
- Optimal, but don't know the complexity (?) [Pettie, Ramachandran '02]

Exercise

Given two strings s, s', find if s is a subsequence of s' in time O(|s| + |s'|).

Ex: abc is subsequence of aadfebgdc, but it is not a subsequence of bca

Solution: (Greedy) Match s[1] to its firs occurrence in s', say s'[i_1]; then match s[2] to its first occurrence in s' after i_1, … If cannot match all of s, return it is NOT a subsequence, else return IS a subsequence

Correctness:

- Just need to show that if s is a subsequence of s', then returns IS
- Suppose s is subsequence of s'; so s[1]=s'[j_1], s[2]=s'[j_2], …for some positions $j_1, < j_2 <$…
- Since s'[j_1] is a candidate for matching s[1], greedy will match s[1] to a position in s' no later than j_1 (that it, $i_1 \leq j_1$)
- But then s'[j_2] is a candidate for matching s[2] (since comes after j_1, and so after i_1), greedy will match s[2] to a position in s' no later than j_2…