

Algorithms and Complexity

Greedy Algorithms

Julián Mestre

School of Information Technologies
The University of Sydney



THE UNIVERSITY OF
SYDNEY

Greedy algorithms are the simplest kind of algorithms. There is no formal definition, but they usually involve making incremental choices by following a simple rule of thumb.

Pros:

- They are easy to design and implement
- Tend to be fast
- Some times work rather well in practice

Cons:

- Not many problems can be solved with greedy strategies
- Usually perform poorly on worst-case instances

Motivation:

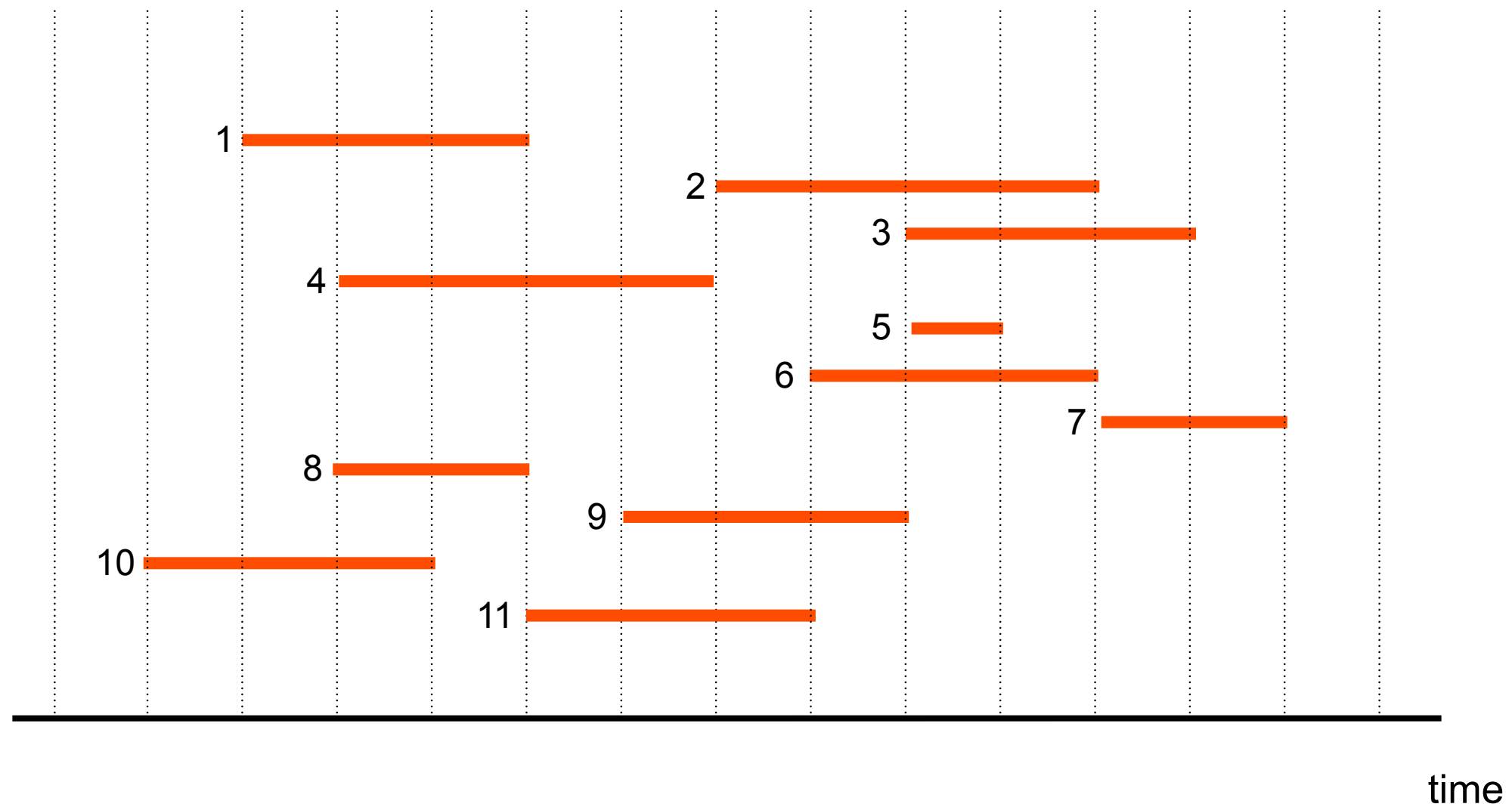
- Users submitting requests to use some common resource (e.g., a classroom)
- Each request has a time window where the resource is needed.
- Users cannot share the resource.

Input:

- Set of intervals $\{I_1, I_2, \dots, I_n\}$ where $I_i = (s(i), f(i))$

Task:

- Find a largest subset of intervals that do not intersect



Possible criteria for sorting the intervals:

- Increasing length
- Decreasing # of intervals in conflict with
- Increasing starting time
- Increasing finishing time

```
def interval_scheduling(intervals, X):
```

```
    “sort intervals according to criterion X”
```

```
    answer = []
```

```
    for I in intervals:
```

```
        if “I can be added to answer”:
```

```
            answer.append(I)
```

```
    return answer
```

Consider the greedy algorithm for interval scheduling where the intervals are sorted in increasing order of finishing time.

Let **OPT** be an optimal solution and **S** the solution we found

- If **OPT** equals **S** then we are done
- Otherwise, we can find another optimal solution that “agrees more with **S**”

Thm.

The greedy algorithm always returns
an optimal solution

```
def interval_scheduling(intervals):  
    sort intervals in increasing finishing time  
    answer = []  
    last_finished = 0  
    for (s,f) in intervals:  
        if s >= last_finished:  
            answer.append( (s,f) )  
            last_finished = f  
    return answer
```

Thm.

The greedy algorithm returns an optimal
solution in $O(n \log n)$ time

Recap: Interval scheduling

Interval scheduling is to select a maximum number of non-conflicting intervals

Greedy picking intervals in increasing finishing time solves the problem optimally

Greedy algorithms are simple, but not every greedy strategy is optimal!

Motivation:

- You want to navigate a road network to go from s to t
- Associated with each link we have a distance
- We would like to find the shortest route to go from s to t

Input:

- Undirected connected graph $G=(V, E)$, and pair of nodes s and t
- length function $\ell : E \rightarrow \mathbb{R}^+$

Task:

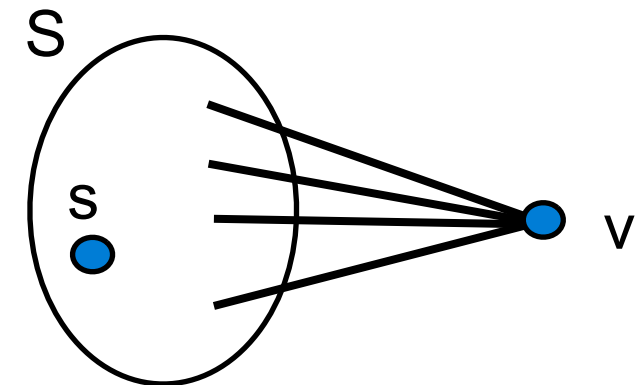
- Find an s - t path P minimizing $\ell(P)$

A greedy algorithm

We compute the distance from s to every other node

- Clearly to s is 0 , that is, $\text{dist}(s,s) = 0$
- Suppose we knew the true distance to some set S of nodes, then we would like to be able to compute the true distance of one more node

Def.: $\text{dist}'(v) = \text{minimum}_{(u,v) \in E : u \in S} \text{dist}(s,u) + \ell(u,v)$



Obs.: If v is a vertex minimizing $\text{dist}'(v)$, then $\text{dist}(v) = \text{dist}'(v)$

This is the basis of a greedy algorithm!

A greedy algorithm

```
def Dijkstra(V,E,l,s):  
  
    d = [  $\infty$  for v in V ]  
    S = set containing s  
    d[s] = 0  
    for “neighbor v of s”:  
        d[v] = “length of (s,v)”  
    while S != V:  
        find u in V \ S minimizing d(v)  
        add u to S  
        for “neighbor v of u”:  
            d[v] = min(d[v], d[u] + length of (u,v))  
    return dist
```

When we transfer some vertex u to S the value $d[u]$ is the true s - u distance in the graph. Otherwise, it is an upper bound

Algorithm terminates because eventually S must equal V

Lem.

Dijkstra's algorithm always returns
a shortest distances from s to every node

Straightforward implementation take $O(n^2)$ time

Instead, a better approach is to use a priority queue that supports a decrease key operation. The priority of u is $d[u]$

- $\text{priority_queue}(A)$: one call with $|A| = n-1$
- $\text{decrease_key}(e)$: at most m calls
- $\text{del_min}()$: $n-1$ calls

Different implementations, yield different times

- If we use a regular heap then Dijkstra runs in $O(m \log n)$ time
- If we use Fibonacci heaps then Dijkstra runs in $O(m + n \log n)$ time

Recap: Shortest path

The shortest **s-t** path problem is to find a minimum length **s-t** path

Dijkstra's algorithm is a greedy algorithm that incrementally find the true distance from **s** to all nodes in the graph

With the right data structure, the algorithm can be implemented to run in $O(m + n \log n)$ time

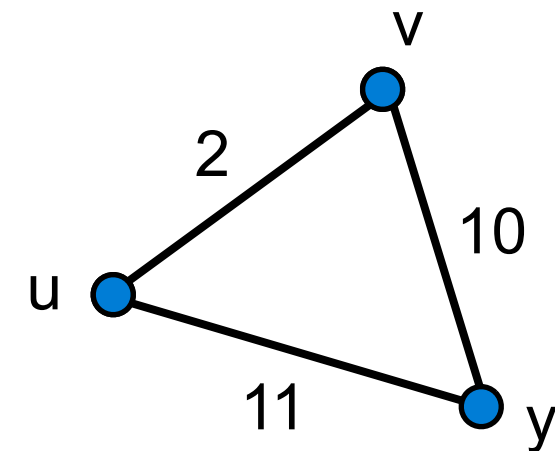
Minimum spanning tree

Motivation:

- You want create a computer network by setting up links between them
- Not all links are possible, and those that are have a cost associated
- We would like to find the cheapest way to connect these computers

Input:

- Undirected connected graph $G=(V, E)$
- cost function $c : E \rightarrow \mathbb{R}^+$



Task:

- Find a X subset of E with minimum cost such that (V, X) is connected

A greedy algorithm

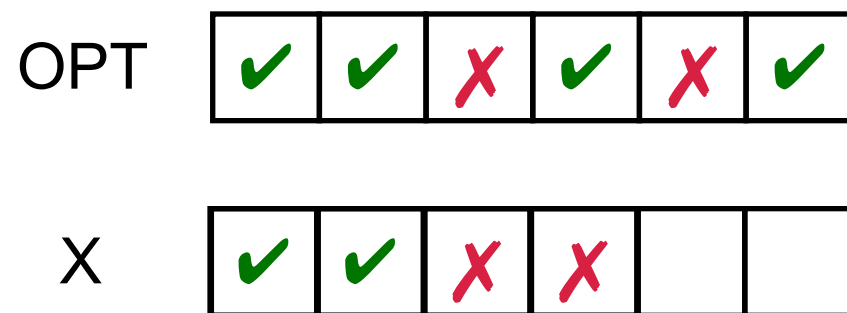
The optimal solution is always a tree

If all weights are different then the edge with the smallest cost has to be part of the optimal tree

```
def Kruskal(V,E,c):  
    sort E by increasing c-value  
    answer = []  
    for e in E:  
        if “adding e to answer does not create a cycle”:  
            answer.append(e)  
    return answer
```


Let **OPT** be an optimal solution and **X** the solution we found

- If **OPT** equals **X** then we are done
- Otherwise, we can find another optimal solution that “agrees more with **X**”



Cannot really happen!



After swap OPT and X
agree on one more edge

Lem.

Kruskal’s algorithm always returns
an optimal MST

Sorting edges takes $\Theta(m \log m)$ time

We need to be able to test if adding a new edge creates a cycle

- We could run DFS in each iteration to see if the number of connected components stays the same. This leads to $\Theta(mn)$ time for the main loop
- Can we do better? Yes, keep track of the connected components with a data structure

Union find data structure:

- `make_sets(A)` : makes singleton sets with elements in A
- `find(a)` : returns an id for the set element a belongs to
- `union(a,b)` : union the sets elements a and b belong to

Implementation with union-find

```
def Kruskal(V,E,c):  
    sort E by increasing c-value  
    answer = []  
    components = make_sets(V)  
    for (u,v) in E:  
        if find(components, u) != find(components, v):  
            answer.append( (u,v) )  
            union(components, u, v)  
    return answer
```

Union find operations:

- make_sets(A) : one call with $|A| = |V|$
- find(a) : $2m$ calls
- union(a,b) : $n-1$ calls

Simplest union find implementation

Store set id for each element in an array

- `make_sets(A)` creates and initialized the array
- `find(u)` is a simple lookup on the array
- `union(u,v)` scans the array changing every entry with `u`'s id to `v`'s

Time complexity:

- `make_sets(A)` takes $\Theta(n)$ time, where $n = |A|$
- `find(u)` takes $\Theta(1)$ time
- `union(u,v)` take $\Theta(n)$ time

Kruskal's algorithm would run in $\Theta(n^2)$ time

Slightly better implementation

In addition to the previous array, explicitly keep track of elements in each set. When taking the union of two sets change the entries of elements in the smaller set.

This way an element can change sets at most $O(\log n)$ time.
A sequence of n union operations takes at most $O(n \log n)$ time.

Thm.

Kruskal's algorithm always returns an optimal MST and runs in $O(m \log n)$ time

Kurkal's algorithm outputs a minimum weight spanning tree.

It can be implemented to run in $O(m \log n)$ time

Union-find is a fundamental data structure providing these primitives: $\text{make_set}(A)$, $\text{find}(u)$, $\text{union}(u,v)$

There are more efficient implementation of union-find

Quiz 3

- 15 minutes long
- during your tutorial session

Problem set 3:

- will be posted later today (Monday 13 August)
- make sure you work on it before the tutorial

Assignment 2:

- Is due tomorrow (Tuesday 14 August)

Assignment 3:

- Out tomorrow, due in a week