Algorithms and Complexity

Greedy Algorithms

Julián Mestre

School of Information Technologies
The University of Sydney

Greedy algorithms

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

Interval scheduling

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.

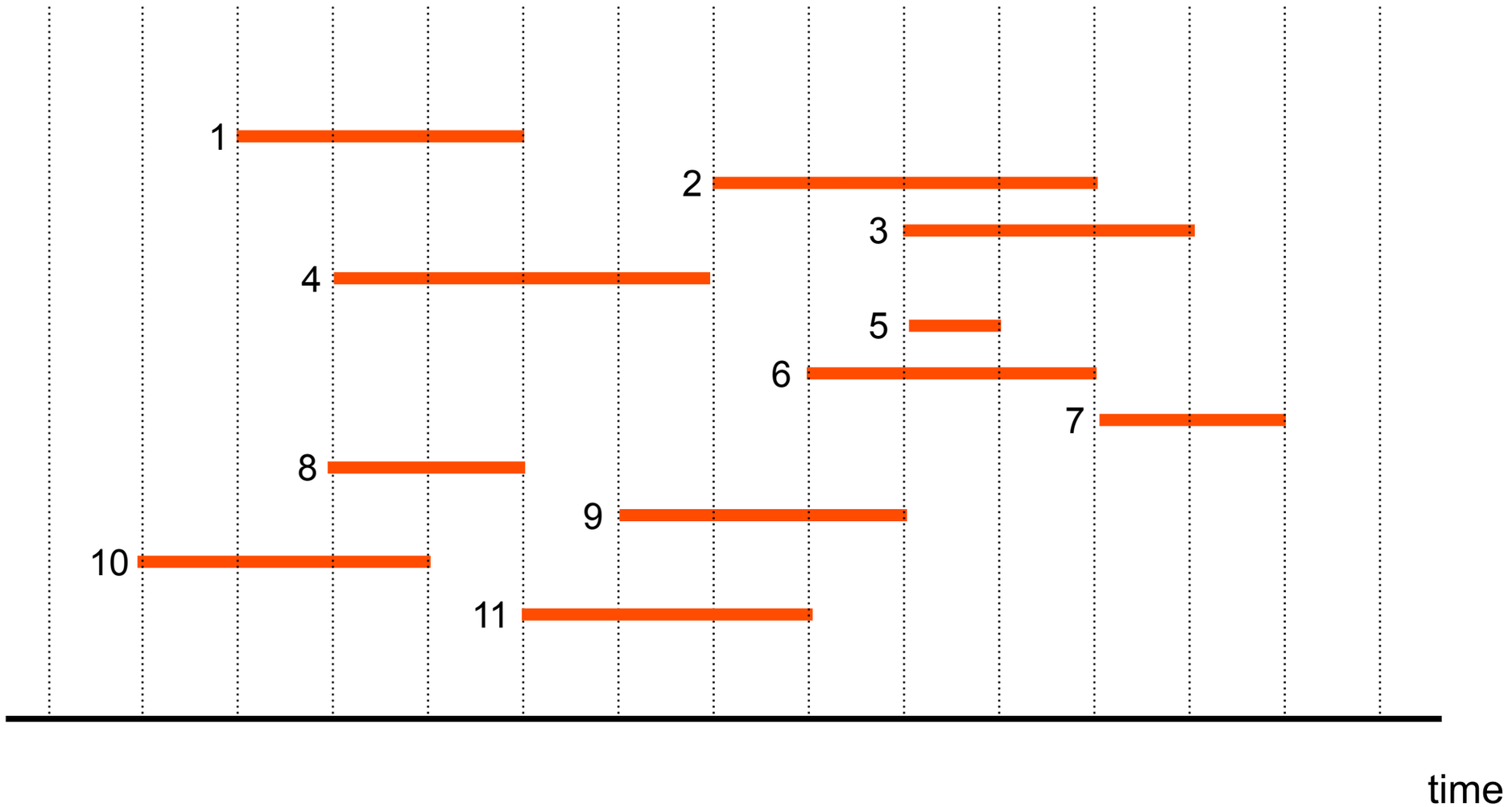
Input:

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

Task:

- Find a largest subset of intervals that do not intersect

Example



Greedy template

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
```

Correctness

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

Time complexity

```
def interval_scheduling(intervals):

  sort intervals in increasing finishing time
  answer = []
  last_finished = 0
  for (s,f) in intervals:
    if s >= last_finish:
      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

Greedily picking intervals in increasing finishing time solves the problem optimally

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

Shortest path

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 $\ell: E \to R^+$

Task:

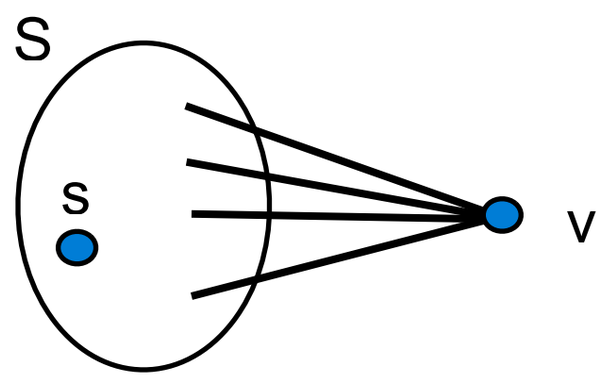
- Find an s-t path P minimizing $\ell(P)$

A greedy algorithm

We compute the distance from s to every other node

- Clearly to s is 0, that is, $dist(s,s) = 0$
- 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

<u>Def.</u>: $estimate(v) = \min_{(u,v) \in E : u \in S} dist(s,u) + \ell(u,v)$



<u>Obs.</u>: If $v \notin S$ is a vertex minimizing $estimate(v)$, then $dist(v) = estimate(v)$

This is the basis of a greedy algorithm!

A greedy algorithm

```
def Dijkstra(G, length, s):

  n = # of vertices in G
  for v in G:
    dist[v] = ∞
  S = { s }
  dist[s] = 0
  for v in G[s]:
    dist[v] = length[(s,v)]
  while |S| < n:
    find u in V \ S minimizing d[v]
    add u to S
    for v in G[u]:
       dist[v] = min(dist[v], dist[u] + length of (u,v)
  return dist
```

Correctness

Throughout the execution, for all u in V, the value dist[u] is an upper bound on the true s-u distance in the graph.

When we transfer some vertex u to S the value dist[u] is the true s-u distance in the graph.

Algorithm terminates because eventually S must equal V

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

Implementation

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]

- priority_queue(A): one call with |A| = n-1
- decrease_key(e): at most m calls
- 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

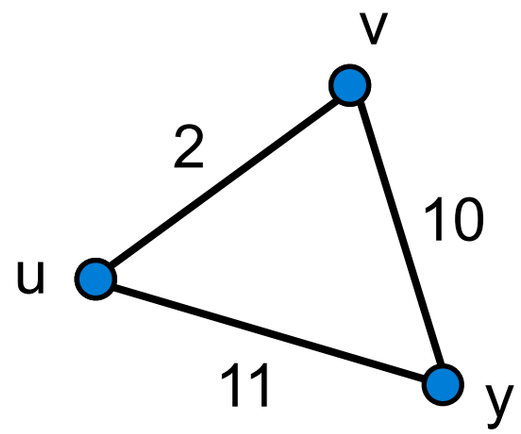
Minimum spanning tree

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 $c : E \to R^+$



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

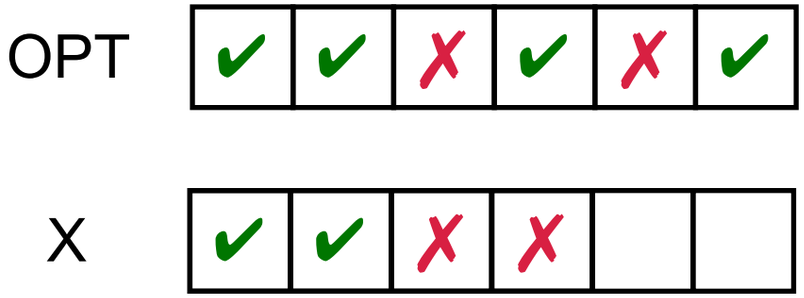
```
def Kurskal(G =(V,E), c):

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

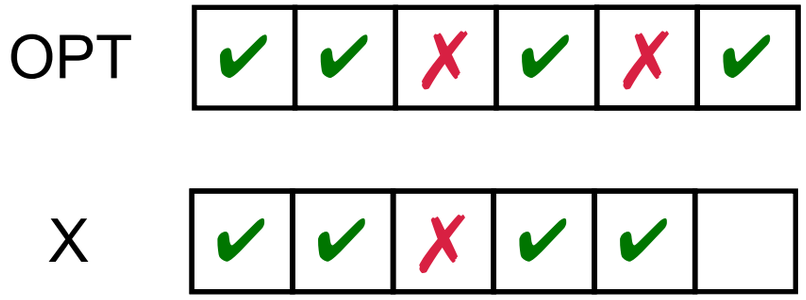
Correctness

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

Time complexity

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 connect components stays the same. This leads to $\Theta(m\ n)$ 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 in 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

Sets are represented with a lists. An array points to the set each element belongs to

- make_sets(A) creates and initialized the array
- find(u) is a simple lookup in the array
- union(u,v) add elements in u's set to v's set

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

Keep track of cardinality of each set. When taking the union of two sets change the smallest.

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

Recap: MST

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: make_set(A), find(u), union(u,v)

There are more efficient implementation of union-find