Algolab 2018 – STL Week 5

Overview

Today's lecture:

- 'Advanced' Techniques
- Greedy Algorithms
  - Example 1: Minimum Spanning Tree
  - Proof Technique: Exchange Argument
  - Example 2: Interval Scheduling
  - Proof Technique: Staying Ahead
- Split & List

Greedy Algorithms

> “*Greed is good.*
> *Greed is right.*
> *Greed works.*
> *Greed clarifies, cuts through, and captures the essence of the evolutionary spirit.*
> *Greed, in all of its forms—greed for life, for money, for love, for knowledge—has marked the upward surge of mankind.*”
>
> — *Wall Street 1987*
> *by Gordon Gekko*

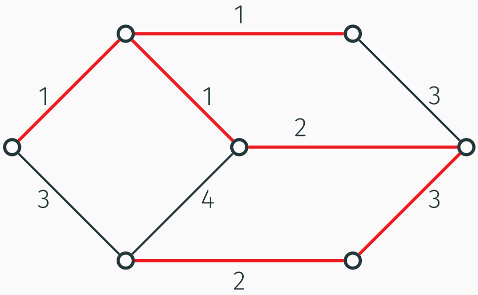
- Often choices that seem best in a particular moment turn out not to be optimal in the long run (e.g. in chess, life, etc.).
- But sometimes locally optimal choices result in a globally optimal solution.
- This is when we can apply greedy algorithms.

A greedy approach typically has the following steps:

1. **Modelling**: realise that your task requires you to construct a set that is in some sense **globally optimal**.
2. **Greedy choice**: given already chosen elements $c_1, \ldots, c_{k-1}$, decide how to choose c_k, based on some **local optimality criterion**.
3. **Prove** that elements obtained in this way result in a **globally optimal** set.
4. **Implement** the greedy choice to be as efficient as possible.

Example: MST

In a graph G with non-negative edge weights, find a minimum weight spanning tree.



Example: MST

Model as an optimisation problem over sets.

In this case, we want to find a set of edges with minimum weight that forms a spanning tree.

Example: MST

Greedy choice

Idea:

- suppose we already have edges $e_1, \ldots, e_{k-1}$
- choose e_k so that
  1. adding e_k to $e_1, \ldots, e_{k-1}$ does not close a cycle (**compatibility**)
  2. e_k has minimum weight among all compatible edges (**local optimality**)

Example: MST

Prove that this yields an optimal solution.

General method: Exchange Argument

- Let A be the choices made by the greedy algorithm.
- Let O be an optimal solution.
- Goal: Assuming A and O are 'not equal', modify O to create O' such that
  1. O' is at least as good as O, and
  2. O' is 'more like' A.

Tip: One good way to do the last bit is to assume O is an optimal solution which 'follows A the longest', that is has the longest common prefix with A.

Look at the first point at which O differs from A and exchange some (further) element to get O' which agrees with A at that point as well.

Example: MST

Prove that this yields an optimal solution.

Proof Sketch

- Let $A = \{e_1, \ldots, e_{n-1}\}$ be the choices made by the greedy algorithm.
- Let $O = \{f_1, \ldots, f_{n-1}\}$ be an optimal solution (which agrees with A the longest, i.e. shares the longest prefix).
- If $A = O$ we are done.
- Let $i \in [n-1]$ be the **smallest index** such that:

$$e_j = f_j \qquad \text{for all } j < i \qquad \text{and} \qquad e_i \neq f_i$$

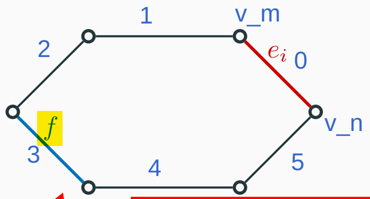
- $A = \{e_1, \ldots, e_{i-1}, e_i, \ldots, e_{n-1}\}$
- $O = \{f_1, \ldots, f_{i-1}, f_i, \ldots, f_{n-1}\}$

Example: MST

Proof Sketch (cont.)

- Let $i \in [n-1]$ be the **smallest index** such that:

$$e_j = f_j \qquad \text{for all } j < i \qquad \text{and} \qquad e_i \neq f_i$$



because let e_i = (v_m, v_n). O is MST to contains a path from v_m to v_n. e_i not in O, so adding e_i to O creates another path between v_m and v_n thus a cycle

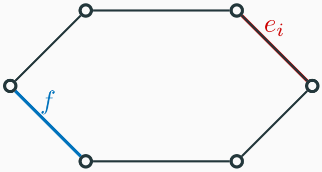
Observations

- $O \cup e_i$ contains a cycle C.
- **Crucial**: there is $f \in C \setminus e_i$ with $w(f) \geq w(e_i)$.

  WHY?! If for all $f \in C \setminus e_i$ we had $w(f) < w(e_i)$ then $C \setminus e_i \subseteq A$ since e_i is the **first edge** in which A and O disagree.

Example: MST

Proof Sketch (cont.)



$$O' := O \setminus f \cup e_i$$

- Let f be an edge $f \in C \setminus e_i$ (thus $f \in O$) with $w(e_i) \leq w(f)$.
- $w(O') = w(O) - w(f) + w(e_i) \leq w(O)$ and is thus still optimal.
- $A = \{e_1, \ldots, e_{i-1}, e_i, \ldots, e_{n-1}\}$
- $O = \{f_1, \ldots, f_{i-1}, f_i, \ldots, f_{n-1}\}$
- $O' = \{f_1, \ldots, f_{i-1}, e_i, \ldots\}$

Contradiction! (With our choice of O.) $\square$

Example: MST

Implement the algorithm efficiently.

1. Sort the edges according to increasing weight.
2. Iterate over the edges in this order.
3. For each edge $\{u, v\}$, if u and v are in the different components formed by the previous edges, add the edge to the MST.

To keep track of the components, use a union find data structure.

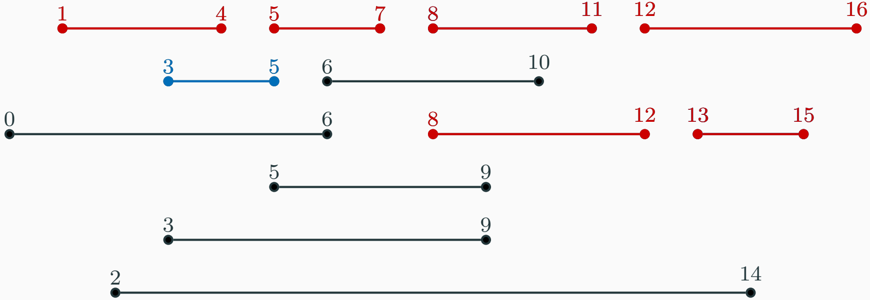
This takes time $O(m \log m)$.

This is Kruskal's algorithm for MST.

Example: Interval scheduling

- Your CPU needs to execute N jobs described by time intervals $[s_i, f_i]$.
- Job i starts at time s_i and ends at time f_i.
- Two jobs are **compatible** if their intervals are disjoint.
- **Goal:** find the maximum number of mutually compatible jobs.

Example: Interval Scheduling



$$A = \{[3,5],[8,11],[13,15]\}$$

Optimal:

$B = \{[1,4],[5,7],[8,11],[12,16]\}$ also $C = \{[1,4],[5,7],[8,12],[13,15]\}$

Example: Interval scheduling

Modelling done for us in the problem description—find the maximum set of compatible jobs.

Example: Interval scheduling

Greedy choice: decide how to choose the job i_k given already chosen jobs $i_1, \ldots, i_{k-1}$.

Natural candidates:

- **Earliest start time** – among compatible jobs, take the one with smallest s_k.
- **Earliest finish time** – among compatible jobs, take the one with smallest f_k.
- **Shortest length** – among compatible jobs, take the one with smallest $f_k - s_k$.
- **Fewest conflicts** – among compatible jobs, take the one which conflicts with the least amount of other compatible jobs.

Example: Interval scheduling

Earliest start time

Earliest finish time

Shortest length

Fewest conflicts

Which one do you think will work?

Example: Interval scheduling

Earliest start time

WRONG

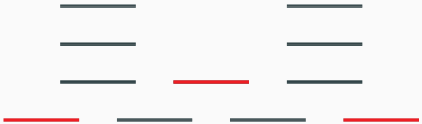
Example: Interval scheduling

Shortest length

WRONG

Example: Interval scheduling

Fewest conflicts



WRONG

Example: Interval scheduling

Earliest finish time

Maybe???

Example: Interval scheduling

Prove that earliest finish time is correct.

General method: Staying Ahead

- Let $A = \{i_1, \ldots, i_n\}$ be the jobs chosen according to earliest finish time.
- Let $O = \{j_1, \ldots, j_m\}$ be an optimal solution (sorted by finish time).
- If $|A| = |O|$ we are done.
- **Goal:** Show that for all $k \leq n$ we have $f_{i_k} \leq f_{j_k}$ (that is, 'stays ahead').

|A| = n and |O| = m. Since O is an optimal solution then A must contains nb of elements (jobs) less of equal to O
A and O are sets of "jobs" sorted by finish time. We can see them as Array's A and O (indexed from 1) where A[k] = i_k and O[k] = j_k
We compare the finish times of A[k].finish_time <= O[k].finish_time for all k <= n = |A|

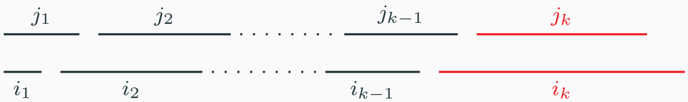
Example: Interval scheduling

Prove that earliest finish time is correct.

Proof Sketch

- **Goal:** Show that for all $k \leq n$ we have $f_{i_k} \leq f_{j_k}$ (that is, 'stays ahead').
- Proof by induction on k.
- **Base case,** $k = 1$: **Clearly holds**!
- Let $k > 1$ and assume it holds for $k-1$ (i.e. $f_{i_{k-1}} \leq f_{j_{k-1}}$).
- Could it happen that $f_{i_k} > f_{j_k}$? **NO!**

  WHY?! $f_{i_{k-1}} \leq f_{j_{k-1}}$ and j_k is compatible with j_{k-1}, thus with i_{k-1} as well. The greedy algorithm would select j_k instead of i_k.



Example: Interval scheduling

Prove that earliest finish time is correct.

Proof Sketch (cont.)

- **Goal:** Show that for all $k \leq n$ we have $f_{i_k} \leq f_{j_k}$ (that is, 'stays ahead').
- For all $k \leq n$, we have $f_{i_k} \leq f_{j_k}$.
- Since $m > n$, there is j_{n+1} in O with:

$$s_{j_{n+1}} > f_{j_n} \qquad \text{and thus} \qquad s_{j_{n+1}} > f_{i_n}.$$

- Therefore, j_{n+1} is **compatible** with $i_1, \ldots, i_n$, but **does not** belong to A.

Contradiction! $\square$

Example: Interval scheduling

Implement the algorithm efficiently.

1. **Sort** the jobs according to increasing finish time.
2. Iterate over the jobs in this order.
3. For each job with interval $[s_i, f_i]$, add the job if s_i is greater than the finish time of the last job that was added.

This takes time $O(N \log N)$.

Example: Checking Change

ATM has bills with values $1, 10$, and 25 and is supposed to give you 42. What is the minimum number of bills used?

Greedy choice

$$1 \times 25 + 1 \times 10 + 7 \times 1 = 42$$

Bills used: 9.

Optimal

$$4 \times 10 + 2 \times 1 = 42$$

Bills used: 6.

Conclusion:

- Some (but not all!) problems can be solved with a greedy approach.
- Deciding how to make the greedy choice can be non-obvious.
- We can check whether the greedy solution works using an **exchange argument** or a **staying ahead** argument.
- Proving that the greedy solution works can be tricky (non-trivial).
- Implementing a greedy solution is usually trivial and quick.

Split & List

Brute Force

Brute force: some problems are hard and we only know how to solve them by trying everything.

However, one can often do it a little bit smarter:

1. Heuristics (important in practice, not in AlgoLab)
2. Improve worst case complexity :)

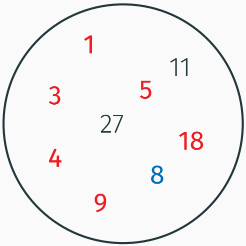
We will see a technique called Split & List.

This technique is why there is ‘DES’ and ‘triple-DES’ but no ‘double-DES’...

Example: Subset Sum

Given a set $S \subseteq \mathbb{N}$, is there a subset $S' \subseteq S$ such that $\sum_{s \in S'} s = k$?

- $S = \{1, 3, 4, 5, 8, 9, 11, 18, 27\}$
- $k = 8$? YES! $S' = \{1, 3, 4\}$ or $S' = \{8\}$
- $k = 1000$? NO!
- $k = 37$? YES! $S' = \{1, 4, 5, 9, 18\}$



NP-Complete :(

n is small: brute force n is small: brute force

Check all subsets!
Recursive/Iterative algorithm

k is small: DP k is small: DP

EXERCISE!

Subset Sum — Recursive

Example: Subset Sum

Given a set $S = \{s_1, \ldots, s_n\} \subseteq \mathbb{N}$, is there a subset $S' \subseteq S$ such that $\sum_{s \in S'} s = k$?

non repeating elements

We want a recursive definition of $f(i,j) :=$ 'is there $S' \subseteq \{s_1, \ldots, s_i\}$ s.t. $\sum_{s \in S'} s = j$'. j (not j', the quote ' is a closing quote)

- Base cases:

$$f(i,0) = \texttt{true}, \text{ for all } i,$$ and we take the empty set as S'

$$f(0,j) = \texttt{false}, \text{ for all } j > 0.$$ i=0 => S' = empty set => can't sum up its elements to j

- $f(i,j) = f(i-1, j-s_i) \vee f(i-1, j)$

Recursive algorithm:

```
bool f(int i, int j) {
    if (j == 0) return true;
    if ((i == 0 && j > 0) || j < 0) return false;
    return f(i - 1, j - elements[i]) || f(i - 1, j);
}
```

then call f(n, k) to find for all j <= k and all i <= n if there is S' in {s_1, ..., s_i} in S with elements of S' summing up to j (note that f() doesn't store results, we can slighly modify it to store them in array f[,]

Time complexity: $O(2^n)$, ok for $n \approx 25$.

Subset Sum — Iterative

How can we iterate over all subsets of an n element set?

Trick: encode the set in an integer.

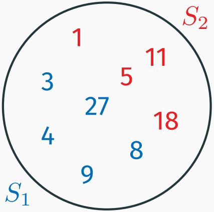
```
bool subsetsum(int k) {
  for (int s = 0; s < 1<<n; ++s) { // Iterate through all subsets
      int sum = 0;
      for (int i = 0; i < n; ++i) {
          if (s & 1<<i) sum += elements[i]; // If i-th element in subset
      }
      if (sum == k) return true;
  }
  return false;
}
```

Time complexity: $O(n \cdot 2^n)$, ok for $n \approx 25$.

Subset Sum — Faster? Split & List

Split S into $S = S_1 \cup S_2$ and $S_1 \cap S_2 = \varnothing$ of size $\approx \frac{n}{2}$.

List all subset sums of S_1 and S_2 into L_1 and L_2



Lemma: The following statements are equivalent:

- There is a $S' \subseteq S$ with $\sum_{s \in S'} s = k$
- There are $S_1' \subseteq S_1$ and $S_2' \subseteq S_2$ such that $\sum_{s \in S_1'} s + \sum_{s \in S_2'} s = k$

Idea: use second statement to check the first.

Algorithm sketch:

- Sort L_2
- For each k_1 in L_1 check if there is k_2 in L_2 (binary search!) such that $k_1 + k_2 = k$.

Time complexity: $O(n \cdot 2^{n/2})$, ok for $n \approx 50$. :)