Greedy approach

Make locally optimal decisions. Ideally, this would create a globally optimal solution. Sometimes, it doesn’t, but it produces a reasonable solution that is computationally tractible.

MST was a greedy algorithm

Both Kruskall’s and Prim’s algorithm made a greedy selection for the next edge to add to the MST.

Interval scheduling

We are given a set of $n$ activities $A = [a_1, a_2, ..., a_n]$ where each activity has a start time $s_i$ and a finish time $f_i$. We would like to schedule as many of these activities for a shared resource. The only constraint is that no two activities can be scheduled at the same time.

Example

A recursive solution:

```plaintext
Algorithm INTERVALSCHEDULE-RECURSIVE(A)
1   if $A = \emptyset$
2      return 0
3   else
4      $max = -\infty$
5      for all $a \in A$
6         $A' \leftarrow A$ minus $a$ and all conflicting activties with $a$
7         $s = \text{INTERVALSCHEDULE-RECURSIVE}(A')$
8         if $s > max$
9            $max = s$
10        return $1 + max$
```
– Is it correct?
  The algorithm tries all possible sets and returns max.
– Runtime
  Worst case, how many recursive calls are made?

\[ O(n!) \]

We'll see next week that we can do better with dynamic programming methods \( O(n^2) \)

Can we do better than this?

Rather than trying to examine all possible solutions, can we make some locally greedy solution and still end up at the optimal solution?

Some ideas:

– Select the activity that starts the earliest, i.e. \( \arg\min \{s_1, s_2, \ldots, s_n\} \)
  counter-example: one long activity that starts the earliest
– Select the shortest activity, i.e. \( \arg\min \{f_1 - s_1, f_2 - s_2, \ldots, f_n - s_n\} \)
  counter-example: one short active that conflicts with two longer
– Select the activity with the smallest number of conflicts
  counter-example?
– Select the activity that finishes first, i.e. \( \arg\min \{f_1, f_2, \ldots, f_n\} \)

**IntervalSchedule-Greedy(A)**

1. sort \( A \) based on finish times \( f_i \)
2. for \( i \leftarrow 1 \) to \( n \)
   3. add \( a_i \) to \( R \)
   4. \( \text{finish} \leftarrow f_i \)
5. while \( s_i < \text{finish} \)
   6. \( i \leftarrow i + 1 \)
7. return \( R \)

– Is it correct?
  Let \( r_1, r_2, r_3, \ldots, r_k \) be the solution produced by **IntervalSchedule-Greedy** in sorted order, with starting time \( s(r_i) \) and end times \( f(r_i) \).
Is there any way we could do better?

Let’s say some other algorithm picks a different starting activity, \( o_1 \). By definition, this activity would have to have a finishing time later than the one selected by \( \text{IntervalSchedule-Greedy} \).

Since \( f(r_1) < f(o_1) \), then the remaining time interval to schedule the other activities must be larger for our algorithm than the alternate algorithm. Since \( o_1 \) is the first selection, then all other activities selected by the alternate algorithm must be after \( o_1 \). Therefore, since our algorithm has a larger time interval to schedule, there is no way the alternative algorithm could do better. This argument can be applied recursively.

- Runtime
  - Sort the algorithm by activities: \( O(n \log n) \)
  - Visit each activity once: \( O(n) \)
  \[ O(n \log n) \]

• Greedy algorithms structure
  Greedy algorithms tend to exhibit the structure that we saw above: a locally optimal decision can be made which results in a subproblem that does not rely on the local decision.

  For proofs, we just need to argue that the combination of the solution to the subproblem and the greedy decision result in an optimal solution.

• Scheduling all intervals
  Same as above, we are given \( n \) activities, however, instead of having only a single resource, we have as many resources as we want. We want to schedule the activities using the minimum number of resources.

Example

What is the minimum number of resources needed? The minimum is the maximum number of overlapping interval at any time.
AllIntervalScheduleCount(A)
1    Sort the start and end times, call this X
2    current ← 0
3    max ← 0
4    for i ← 1 to length[X]
5        if xi is a start node
6            current ++
7        else
8            current --
9        if current > max
10           max ← current
11    return max

– Is it correct?
The algorithm above exactly counts the number of activities occurring during a time interval.

Therefore, the worst case number of labels used is the maximum number of conflicting activities. Since the best case scenario is the maximum number of conflicting activities, this algorithm achieves the optimal.

– runtime
  - sorting the end and start times: 2n log(2n) = O(n log n)
  - Iterate over all start and finish intervals: 2n

O(n log n)

• Horn formulas
**HORN(H)**

1. set all variables to \textit{false}
2. \textbf{for} all implications \(i\)
3. \hspace{1em} \textbf{if} \ \text{EMPTY}(\text{LHS}(i))
4. \hspace{2em} \text{RHS}(i) \leftarrow \text{true}
5. \ \text{changed} \leftarrow \text{true}
6. \textbf{while} \ \text{changed}
7. \hspace{1em} \text{changed} \leftarrow \text{false}
8. \hspace{2em} \textbf{for} all implications \(i\)
9. \hspace{3em} \textbf{if} \ \text{LHS}(i) = \text{true} \ \text{and} \ \neg \text{RHS}(i) = \text{true}
10. \hspace{4em} \text{RHS}(i) \leftarrow \text{true}
11. \hspace{4em} \text{changed} = \text{true}
12. \hspace{2em} \textbf{for} all negative clauses \(c\)
13. \hspace{3em} \textbf{if} \ \text{c} = \text{false}
14. \hspace{4em} \textbf{return} \ \text{false}
15. \hspace{2em} \textbf{return} \ \text{true}

- **Knapsack problems**
  - **0-1 Knapsack problem**
    A thief robbing a store finds \(n\) items worth \(v_1, v_2, \ldots, v_n\) dollars and weigh \(w_1, w_2, \ldots, w_n\) pounds, where \(v_i\) and \(w_i\) are integers. The thief can only carry at most \(W\) pounds in the knapsack. Which items should the thief take if he wants to maximize the value?
  - **Fractional knapsack problem**
    Same as above, but the thief happens to be at the bulk section of the store and can carry fractional portions of the items. For example, the thief could take 20\% of item \(i\) for a weight of 0.2\(w_i\) and a value of 0.2\(v_i\).

Is a greedy approach appropriate for these problems?

- **Huffman coding**
  Data compression - Given a file containing some data of a fixed alphabet \(\Sigma\) (e.g. \(A, B, C, D\)), we would like to pick a binary character code that minimizes the number of bits required to represent the data.
Fixed length codes:

\[A = 00\]
\[B = 01\]
\[C = 10\]
\[D = 11\]

Take the following data:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>70</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>20</td>
</tr>
<tr>
<td>D</td>
<td>37</td>
</tr>
</tbody>
</table>

(adapted from pg. 140 [3])

The number of bits required to encode this using the above encoding is:

\[2 \times 70 + 2 \times 3 + 2 \times 20 + 2 \times 37 = 260\]

Can we do better? We’d like to use a smaller number of bits for the frequently occurring symbols.

Variable length code:

\[A = 0\]
\[B = 01\]
\[C = 10\]
\[D = 1\]

Is this a valid code? What about decoding?

\[A = 0\]
\[B = 100\]
\[C = 101\]
\[D = 11\]
The number of bits required using this encoding:
\[
70 + 3 \times 3 + 3 \times 20 + 2 \times 37 = 213
\]
which is a 20% reduction.

**prefix codes**

A prefix code is a set of codes where no codeword is a prefix of some other codeword. Using prefix codes, we can simply concatenate the bits. To decode, we read them off from the prefix tree.

Show prefix tree

In generaly, a prefix tree encoding can be represented as a *full* binary tree, that is a tree where every node has either 0 or 2 children. Each child of the tree represents an encoding of some symbol. Why?

Given the frequencies of the symbols \( F = f_1, f_2, ..., f_n \), we can calculate the cost of the tree, which is the number of bits required as:

\[
\text{cost}(T) = \sum_{i=1}^{n} f_i \times \text{depth}(i)
\]

Another way of writing this is to consider internal nodes. We can define the cost of the internal nodes as the sum of the frequencies of the descendant leaves. This is the number of times a node is visited during decoding. As we move down the tree, one bit gets read for every nonroot node.

Given this, the total cost of the tree is the sum of the frequencies of all leaves and internal nodes, except the root.
The algorithm greedily selects the two smallest frequency symbols first. These two symbols have to be at the bottom of the tree. Why?

Consider a tree where a lowest frequency symbol was not at the bottom. Swapping this symbol with the lowest frequency symbol would result in a tree with a lower cost, so that tree cannot be optimal.

The algorithm then merges the cost of these nodes and creates a new frequency \( f_1 + f_2 \) (we’ll assume \( f_1 \) and \( f_2 \) are the two smallest frequencies). This can now be viewed as a new encoding problem with frequencies \( F' = (f_1 + f_2), f_3, f_4, ..., f_n \).

- Runtime
  - 1 call to \texttt{MakeHeap} - \( O(n) \)
  - \( 2(n - 1) \) calls to \texttt{ExtractMin} - \( (2n - 2) \log n = O(n \log n) \)

\( O(n \log n) \)

• Set Cover

These notes are adapted from material found in chapter 16 of [1], chapter 4 of [2] and chapter 5 of [3].

References