Tools of AI

Optimisation Problems and Greedy Algorithms

(c) Marcin Sydow

Topics covered by this lecture:

- Optimisation Problems
- Greedy Approach
- Matroids
    - Rado-Edmonds Theorem
- Huffman Code

Example: P2P Network

Consider the following problem:

We are given a peer-to-peer communication network that consists of a set of nodes and a set of bi-directional optical fibre communication channels between them. Assume the graph of communication channels has the property of connectedness (i.e. each node can communicate with each other, if not directly then through a finite number of intermediate nodes). Each connection has a given constant **cost** of maintenance per period of time, expressed with a non-negative number.

The task is to reduce the cost of maintenance of the network to the minimum by removing some communication channels that are not necessary while preserving the connectedness property.

Our example viewed as a graph problem called *MST*

Our problem can be naturally expressed in graph terminology as follows: network nodes correspond to graph nodes and channels correspond to undirected edges with non-negative weights representing the cost.

INPUT: an undirected connected graph $G = (V, E)$ with non-negative weights on edges, given by a weight-function: $w : E \rightarrow R^+$

OUTPUT: a graph G' such that:

1. $G' = (V, E')$ is a connected subgraph of G
   (it connects all the nodes of the original graph G)
2. the sum of weights of its edges $\sum_{e \in E'} w(e)$ is minimum possible

Notice: due to the minimisation constraint, the result must be a tree (why?) (it is called a *spanning tree* of G as it spans all its nodes)

Actually, this problem is widely known as the *minimum spanning tree* problem (MST).

MST is an example of so called **optimisation problem**.

Optimisation Problem

An **instance** of an optimisation problem: a pair (F, c), where F is interpreted as a set of *feasible* solutions and c is the *cost* function: $c : F \rightarrow R$.

The task in *minimisation* is to find $f \in F$ such that $c(f)$ is minimum possible in F. f is called *global optimal solution* to the instance F[1]

An **optimisation problem** is a set I of instances of an optimisation problem.

[1] Another variant is a *maximisation* problem, when we look for a feasible solution f with maximum possible value of $c(f)$ – in this case we can call c a *profit* function

Our example as an optimisation problem

In the MST problem, an *instance* of a problem corresponds to a particular given undirected graph G with non-negative weights on edges, and formally F, the set of feasible solutions, is the set of all spanning trees of G (i.e. a feasible solution is a spanning tree of the graph).
For a given spanning tree $f = (V, E')$, the cost function $c(f)$ is the sum of weights of its edges $c(f) = \sum_{e \in E'} w(e)$. This is a minimisation variant.
The *MST optimisation problem* corresponds to all possible connected, undirected graphs with non-negative weights on edges (i.e. all possible inputs to our problem).

A few more examples of optimisation problems

There are numerous codified, widely known optimisation problems. Below, a few examples (out of dozens):

- given a set of items with specified weights and values select the subset that does not exceed a given total weight and has maximum possible total value (*Knapsack problem*)
  - (a variant of the above) similarly, except the items are divisible (so that you can take fractional parts of items) (continuous *Knapsack problem*)
- given an undirected graph find a minimum subset of its nodes so that each edge is incident with at least one selected node (*Vertex covering problem*)
- given a binary matrix select a minimum subset of rows such that each column has at least one '1' in a selected row (*Matrix covering* problem)

Exercise: for each optimisation problem above, find at least one practical application in engineering, computer science, etc.

Solving Optimisation Problems

Most of the known optimisation problems have very important practical applications.

It may happen that two different problems have seemingly similar formulations, but differ in hardness of finding globally optimal solution (an example very soon).

Some of the optimisation problems are so hard that there is no fast algorithm known to solve it. (if the time complexity of an algorithm is higher than polynomial function of the data size (e.g. exponential function) the algorithm is considered as unacceptably slow)

Brute Force Method

There is a simple, but usually impractical, method that **always guarantees** finding an optimal solution to an instance of any optimisation problem (important assumption: the set of solutions is finite):

1. generate all potential solutions
2. for each potential solution check if it is feasible
3. if solution f is feasible, compute its cost $c(f)$
4. return a solution f^* with minimum (or maximum) cost (or profit).

This method is known as the **brute force** method.

The brute force method is impractical because F is usually huge.

For example, solving an instance of knapsack with merely 30 items with brute force will need to consider *over billion* of potential solutions! (including some “solutions” that turn out to be not feasible, i.e. the weight limit is exceeded).

Greedy Approach

In a greedy approach the solution is viewed as the composition of some parts.

We start with nothing and add to the solution next parts, step by step, until the solution is complete, always adding the part that currently seems to be the best among the remaining possible parts ("greedy choice").

After adding the part to the solution, we never re-consider it again, thus each potential part is considered at most once (what makes the algorithm fast).

However, for some optimisation problems the "greedy choice" does not guarantee that the solution built of locally "best" parts will be globally best (examples on next slides).

Example: continuous Knapsack

Consider the following greedy approach to the continuous knapsack problem: (assume there are n items in total)

1. for each item compute its "profit density" (value/weight) ($O(n)$ divisions)
2. until the capacity is not exceeded, greedily select the item with the maximum profit density ($O(nlog(n))$ for sorting the items)
3. for the "critical" item, take its fractional part to fully use the capacity ($O(1)$)

It can be proven that this greedy algorithm will **always** find a globally optimal solution to any instance of the continuous knapsack problem. Notice that this algorithm is also **fast** – $O(nlog(n))$ (we assume that summation, subtraction and division are constant-time operations)

Exercise: will the algorithm be correct if we greedily select the most valuable items?

Exercise(*): tune the algorithm to work in $O(n)$ time, consider a variant of the Hoare's selection algorithm.

Example: (discrete) Knapsack

Discrete knapsack problem is seemingly almost identical to the continuous variant (you just cannot take fractions of items). It is easy to observe that the greedy approach that is perfect for the continuous variant **will not guarantee global optimum** for the discrete variant:

Example:

capacity: 4, weights: (3,2,2), values: (30,18,17)

greedy solution: item 1 (total value: 30)

optimal solution: items 2,3 (total value: 35)

Even more surprisingly, it can be actually proved that **there is no polynomial-time** algorithm for the discrete knapsack problem! (we assume standard, binary representation of numbers)

Conditions for Optimality of Greedy Approach?

A good question:

How to characterise the optimisation problems for which there is a (fast) greedy approach that guarantees to find the optimal solution?

There is no single easy general answer to this question (one of the reasons is that "greedy approach" is not a precise term)

However, there are some special cases for which a precise answer exists. One of them concerns matroids.

(*)Matroid

Definition

A *matroid* is a pair $M = (E, I)$, where E is a finite set and I is a family of subsets of E (i.e. $I \subseteq 2^E$) that satisfies the following conditions:
(1) $\emptyset \in I$ and if $A \in I$ and $B \subseteq A$ then $B \in I$
(2) for each $A, B \in I$ such that $|B| = |A| + 1$ there exists $e \in B \setminus A$ such that $A \cup \{e\} \in I$ ("exchange property")

We call the elements of I *independent* sets and other subsets of E *dependent* sets of the matroid M.

Matroid can be viewed as a generalisation of the concept of linear independence from linear algebra but has important applications in many other areas, including algorithms.

Maximal Independent Sets

For a subset $C \subseteq E$, we call $A \subseteq C$ its *maximal independent subset* if it is independent and there is no independent B such that $A \subset B \subseteq C$

Theorem

E is a finite set, I satisfies the condition 1 of the definition of matroid. Then, $M = (E, I)$ is a matroid iff it satisfies the following condition:
(3) for each $C \subseteq E$ any two maximal independent subsets of C have the same number of elements

Thus, 1 and 3 constitute an equivalent set of axioms for the matroid.

Proof

$(1) \wedge (2) \Rightarrow (1) \wedge (3)$: (*reductio ad absurdum*[2]) Assume $C \subseteq E$ and $A, B \in I$ are two maximal independent subsets of C with $k = |A| < |B|$. Let B' be a (k+1)-element subset of B. By (1) it is independent. Thus, by (2) there exists $e \in B' \setminus A$ such that $A \cup \{e\}$ is independent. But this contradicts that A is a *maximal* independent set.

$(1) \wedge (3) \Rightarrow (1) \wedge (2)$: (*by transposition*) Assume $|A| + 1 = |B|$ for some independent A, B but there is no $e \in B \setminus A$ such that $A \cup \{e\}$ is independent. Thus, A is a maximal independent subset of $C = A \cup B$. Let $B' \supseteq B$ be an extension of B to a maximal independent subset of C. Thus, $|A| < |B'|$ so that A, B' are two maximal independent subsets of C that have different number of elements.

[2] i.e. by contradiction

Rank of a Set and Basis of a Matroid

The cardinality (i.e. number of elements) of any maximal independent subset $C \subseteq E$ is called the *rank* of C and denoted $r(C)$:

$$r(C) = max\{|A| : A \in I \wedge A \subseteq C\}$$

Notice that C is independent only if $r(C) = |C|$

Each maximal independent set of a matroid $M = (E, I)$ is called a *basis* of the matroid. (by analogy to linear algebra). A rank of the matroid is defined as $r(E)$ (analogous to the dimension of a linear space).

Corollary

Each basis of a matroid has the same number of elements

Rado-Edmonds Theorem

Assume E is a finite set, $I \subseteq 2^E$ is a family of subsets of E, $w : E \to R^+$ is a weight function. For $A \subseteq E$ we define its weight as follows: $w(A) = \sum_{e \in A} w(e)$

Consider the following **greedy algorithm**:

```
sort E according to weights (non-increasingly), such that
$E = \{e_1, ..., e_n\}$ and $w(e_1) \geq w(e_2) \geq ... \geq w(e_n)$
$S = \emptyset$
for(i = 1; i <= n; i++) { if $S \cup \{e_i\} \in I$ then $S = S \cup \{e_i\}$ }
return S
```

Theorem (Rado, Edmonds)

If $M = (E, I)$ is matroid, then S found by the algorithm above is an independent set of maximum possible weight. If $M = (E, I)$ is not a matroid, then there exists a weight function w so that S is not an indpendent set of maximum possible weight

Proof ($\Rightarrow$)

Assume $M = (E, I)$ is a matroid and $S = \{s_1, ..., s_k\}$ is the set found by the algorithm, where $w(s_1) \geq w(s_2) \geq ... \geq w(s_k)$. Consider an independent set $T = \{t_1, ..., t_m\}$, where $w(t_1) \geq w(t_2) \geq ... \geq w(t_m)$. Notice that $m \leq k$, since S is a basis of the matroid: any element $e_i \in E$ "rejected" in some step of the algorithm must be dependent on the set of all the previously selected elements and thus on the whole S. We show that $w(T) \leq w(S)$, more precisely that $\forall_{i \leq m} w(t_i) \leq w(s_i)$. Assume the contradiction: $\exists_i w(t_i) > w(s_i)$ and consider two independent sets $A = \{s_1, ..., s_{i-1}\}$ and $B = \{t_1, ..., t_{i-1}, t_i\}$. According to the condition (2) (of the matroid definition), $\exists_{t_j}$, so that $j \leq i$ and $\{s_1, ..., s_{i-1}, t_j\}$ is independent. We have $w(t_j) \geq w(t_i) > w(s_i)$, what implies that $\exists_{p \leq i}$ such that $w(s_1) \geq ... \geq w(s_{p-1}) \geq w(t_j) > w(s_p)$ which contradicts the fact that s_p is the element of maximum weight such that its addition to $\{s_1, ..., s_{p-1}\}$ will not destroy its independence. Thus it must hold that $w(t_i) \leq w(s_i)$ for $1 \leq i \leq m$.

Proof ($\Leftarrow$)

Assume that $M = (E, I)$ is not a matroid. If condition (1) of the matroid definition is not satisfied there exist sets $A, B \subseteq E$ such that $A \subseteq B \in I$ and $A \notin I$ and lets define the weight function as follows $w(e) = \lfloor e \in A \rceil$. Notice that in this case A will be not contained in the selected set S but $w(S) < w(B) = w(A)$.

Now assume that (1) is satisfied, but (2) is not satisfied. Thus, there exist two independent sets A, B so that $|A| = k$ and $|B| = k+1$ and for each $e \in B \setminus A$ the set $A \cup \{e\}$ is dependent. Denote $p = |A \cap B|$ (notice: $p < k$) and let $0 < \epsilon < 1/(k-p)$.
Now, lets define the weight function as: $w(e) = 1 + \epsilon$ for $e \in A$, $w(e) = 1$ for $e \in B \setminus A$ and $w(e) = 0$ else. In this setting, the greedy algorithm will first select all the elements of A and then will reject all the elements $e \in B \setminus A$. But this implies that the selected set S has lower weight that B:

$$w(S) = w(A) = k(1+\epsilon) = (k-p)(1+\epsilon) + p(1+\epsilon) <$$

$$< (k-p)\frac{k+1-p}{k-p} + p(1+\epsilon) = (k+1-p) + p(1+\epsilon) = w(B)$$

so that S selected by the greedy algorithm is not optimal.

(proof after: Witold Lipski, "Combinatorics for Programmers", p.195, WNT 2004, Polish Edition)

Graph Matroids and the MST Example (again)

Consider an undirected graph $G = (V, E)$. Lets define $M(G) = (E, I)$, where $I = \{A \subseteq E : \textit{A is acyclic}\}$. Notice that $M(G)$ is a matroid, since any subset of acyclic set of edges must be acyclic (condition (1)) and any maximal acyclic subset of edges of a graph has the same cardinality: $|V| - c$, where c is the number of connected components of G (condition (3)). $M(G)$ is called the *graph matroid* of G.

Consider the MST problem again. By a simple trick we can make each maximal independent set in the graph matroid an MST. To achieve this, define the weights of elements as $W - w(e)$ where W is the maximum edge weight in the graph and $w(e)$ are weights on edges. Now, notice that a maximum independent set in the graph matroid is exactly a MST.

Matroid theory guarantees that a simple greedy algorithm presented in the Rado-Edmonds theorem, must always find the globally optimal independent set. Thus, it can optimally solve the MST problem.

Applications of Matroids and beyond...

Matroids provide a tool for mathematical verification whether some optimisation problems can be optimally solved by a greedy algorithm of some *particular* form (presented in the Rado-Edmonds theorem)

There are many other optimisation problems to which Rado-Edmonds theorem applies.

However, there are successful greedy algorithms of *different* forms, that optimally solve some important optimisation problems where there is no natural notion of a matroid for a given problem (example very soon).

There is also a generalisation of matroid, called *greedoid*, for which some more theoretical results are proved (not in this lecture)

Example: Huffman Coding

(we will present an optimal greedy algorithm for this problem without specifying the matroid)

Symbol-level, loseless binary compression: given a *fixed* sequence of symbols,

e.g.: “aabaaaccaabbcaaaabaaaacaabaa”

find a binary encoding of each symbol such that the encoded sequence is:

- decodeable in the unique way (feasibility of solution)
- the shortest possible (optimization criterion)

Simple solution:
fixed-length coding (simple, no problems with decoding)
but: is it optimal?

Idea: exploit the symbol frequencies

Idea: more frequent symbols can be represented by shorter codes.

The only problem: to make it decodeable in unique way

e.g.: a-0, b-1, c-00, d-01
but how to decode: "001"? As "aab" or "ad", or ...

Let's apply a **Prefix code**:
(a method of overcoming the decoding problem)

prefix code: no code is a prefix of another code.

Prefix code tree

A prefix code can be naturally represented as a binary tree, where

- the encoded symbols are in leaves (only)
- edges are labelled with '0' or '1': the path from root to a leaf constitutes the code (going "left" means '0' going "right" means '1')

Now, the optimisation problem is to construct the optimal coding tree (called Huffman tree).

Constructing Huffman tree

input: set of symbols with assigned frequencies

Initially, treat each symbol as a 1-node tree. It's weight is initially set to the symbol frequency.

```
while(more than one tree)
  join 2 trees with the smallest weights
  (make them children of a new node)
  the weight of the joined tree is the sum of the ``joined weights''
```

The least frequent symbols go to the deepest leaves.

Can you see why it is a greedy algorithm?

Standardisation Conventions

Lets assume the following arbitrary standardisation convention when building the Huffman tree (it assumes an order on the symbols). The convention does not affect the length of the code but may have effect on particular codes assigned to particular symbols:

1. the subtree with a smaller weight always goes to left
2. if the weights are equal, the subtree that contains the earliest symbol label always goes to left

Optimality of the greedy algorithm for Huffman code

It can be proved that the Huffman algorithm finds the optimum prefix code.

The average code length cannot be shorter than entropy of the distribution of the input symbols.

Constructing Huffman tree - implementation

input: set S of symbols with assigned frequencies

We will apply a (min-type) priority queue to implement the algorithm. Assume that weight of a node is used as the priority key.

```
PriorityQueue pq

for each s in S
    pq.insert(s)

while(pq.size > 1)
    s1 = pq.delMin()
    s2 = pq.delMin()
    s_new = join(s1, s2)
    pq.insert(s_new)

return pq.delMin()
```

Constructing Huffman tree - complexity analysis

data size: number of symbols (n)

dominating operation: comparison

initialization: n times insert (or faster: bin heap construct)

code tree construction: (n-1) times (2 delMin + 1 insert)

$W(n) = \Theta(nlogn)$

Summary

- Optimisation Problems
- Greedy Approach
- Matroids
    - Rado-Edmonds Theorem
- Huffman Code

Thank you for attention