

Discrete Optimisation Problems

Introduction

(c) Marcin Sydow

Topics covered by this lecture:

Discrete
Optimisation
Problems

(c) Marcin
Sydow

Introduction


- the role of optimisation in AI
- definition of optimisation problem
- example 1: knapsack
- example 2: TSP (travelling salesman problem)
- class of hard problems (NP-complete class)
- other examples of discrete optimisation problems
- Karp's list
- techniques for discrete optimisation
- bruteforce method

The role of optimisation in AI

AI has many subfields, for example:

- machine learning
- natural language processing
- robotics, etc.

Many practical problems in various AI subfields can be represented as **optimisation problems** i.e. problems for which we find a feasible solution that is **optimal**¹ in some precisely specified way.

¹The word “optimal” has etymology in Latin, from “optimus”, meaning “the best”. Warning: a solution can be “optimal”, but never say “more (?) optimal”, since it does not make sense (“more the best”?) 

The role of optimisation in AI, cont.

Optimisation has intersection with AI for at least two reasons:

- solving many problems in AI involve optimisation techniques (e.g. training a machine learning model or tuning hyperparameters of a neural network, etc.)
- many problems in AI are themselves specified as optimisation problems (e.g. optimally assign resources to tasks, find an optimal route of an autonomous vehicle on the map to get to the target, select best moves while playing a game, etc.)

Continuous vs discrete optimisation

According to one of many possible criteria, we can split the optimisation problems into 2 groups:

- continuous optimisation (the parameters to be optimised have continuous nature)
- discrete (or: combinatorial) optimisation: the parameters to be optimised are discrete (e.g. bits, natural numbers, sets, sequences, etc.)

Continuous vs discrete optimisation techniques

There are different methods for solving the two above groups of problems.

- in general, optimising continuous functions involves differential calculus (finding extrema of continuous functions)
- in discrete (combinatorial) case, the optimised function is not differentiable (even not continuous), so that there are needed special techniques (some of them regarded as belonging to AI) for finding optimal solutions

Furthermore, actually the discrete optimisation problems are typically computationally much harder than continuous ones.

Optimisation Problem

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

The task in *minimisation* is to find feasible solution $s \in F$ such that $o(s)$ is minimum possible in F . In such case o can be named “cost function” (to be minimised). s is called *globally optimal solution* to the instance (F, o) ²

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

²Another variant is a *maximisation* problem, when we look for a feasible solution f with maximum possible value of $o(s)$ – in this case we can call o a *profit* function

Specification of optimisation problem

Precise specification (definition) of an optimisation problem typically consists of:

- INPUT description
- OUTPUT description, that consists of the description of the solution including 2 parts:
 - feasibility (conditions for the feasibility of solution)
 - optimality (definition of objective function and conditions of the optimality in terms of minimising or maximising the objective function)

Working Examples

There are numerous known combinatorial optimisation problems

Examples:

- Knapsack (K)
- Travelling Salesman Problem (TSP)
- Minimum Spanning Tree (MST)
- Optimal Prefix Code (Huffman)
- Bin Packing (BP)
- Vertex Cover (VC)
- Satisfiability problem (SAT)

There are currently hundreds of known catalogued combinatorial optimisation problems.

Knapsack Problem: textual description

There is a finite set of items.

Each has its numeric **value** and **weight** (or size).

There is a knapsack of limited numeric **capacity**.

Goal: select the subset of items so that:

- total weight of selected items does not exceed capacity
(feasibility of solution)
- total value of selected items is maximum possible
(optimality of solution)

Knapsack Problem: more formal definition

INPUT: set $X = \{x_1, \dots, x_n\}$ of items, capacity $C \in \mathbb{R}^+$, value function $v : X \rightarrow \mathbb{R}^+$, size function $s : X \rightarrow \mathbb{R}^+$

OUTPUT: solution: a subset $S \subseteq X$ of items so that the following conditions are satisfied:

- (feasibility) $\sum_{x \in S} s(x) \leq C$
- (optimality) the solution S is such that: the objective function $o(S) = \sum_{x \in S} v(x)$ (total value of selected items) has maximum possible value over all possible feasible subsets of items

Travelling Salesman Problem (TSP): textual description

Discrete
Optimisation
Problems

(c) Marcin
Sydow

Introduction

There is a set of cities to be visited by a salesman and there is known distance between any two cities.

Goal: design a visiting order of all the cities so that the total distance is minimum possible³ (the salesman starts and ends in the same city)

³notice that it implies that each city must be visited exactly once

Travelling Salesman Problem (TSP): more formal definition

Discrete
Optimisation
Problems

(c) Marcin
Sydow

Introduction

INPUT: given a full graph $G(V, E)$ with weights $w : E \rightarrow \mathbb{R}^+$ on edges (where the vertices represent cities and the weights on edges represent the pairwise distances)

OUTPUT: find a cycle S in the graph $G(V, E)$ so that:

- (feasibility): S is a hamiltonian cycle (i.e. it contains each vertex exactly once)
- (optimality): the total sum of weights on the cycle S (the objective function) is minimum possible.

A few more examples of optimisation problems

More examples of discrete optimisation problems:

- 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

Most of the discrete optimisation problems of interest are so hard that there is no fast (i.e. upper bounded by a polynomial time complexity)⁴ algorithm known to solve it.

⁴if 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

Complexity theory, NP-hard problems

Discrete
Optimisation
Problems

(c) Marcin
Sydow

Introduction

Complexity theory studies hardness of computational problems.

It formally introduces concepts, definitions and theorems that make it possible to mathematically analyse problems to tell whether they are computationally hard (and to what “degree” they are hard)

In particular there is a **class P** (for “polynomial”) of computational problems that are considered as computationally easy (there exist fast algorithms to solve them).

There is also another class of **NP-hard** problems that represents problems for which there are not known fast algorithms (i.e. known algorithms have over-polynomial, e.g. exponential, time complexity)

(there are many more complexity classes than the two above)

The role of AI in solving hard problems: heuristics

Discrete
Optimisation
Problems

(c) Marcin
Sydow

Introduction

Most of the combinatorial optimisation problems of interest fall into NP-hard category (including Knapsack, TSP, etc.)

Thus, there are not known fast algorithms for solving them and there are designed special techniques (called **heuristics**), many of them belonging to AI, for approaching them.

A heuristic means an acceptably fast algorithm for solving a problem, without a 100% guarantee to find global optimum.

Techniques for solving discrete optimisation problems

There are several general approaches to solve combinatorial optimisation problems, especially computationally hard ones, e.g.:

- greedy algorithms (heuristics)
- local search heuristics
- evolutionary heuristics
- linear programming

We give a short overview of some of the existing techniques to solve hard discrete optimisation problems.

Technique 0: Brute Force Method (systematic search)

For any optimisation problem it is always possible to systematically generate all potential solutions, check them and identify the best one (for which the objective function is optimal).

Systematic search always guarantees finding an optimum in finite time (thus it is **not a heuristic**)

On the other hand, in practice, the solution space usually grows at least exponentially with data size, and exhaustive search is impractical except some small instances.

Systematically generating all feasible solutions may be more difficult for some problems than for others (e.g. TSP vs K).

Example: what is potential solution space size for knapsack with 30 items, TSP with 30 cities?

Brute Force, cont.

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

- 1 consider 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).

Technique 1: Greedy Approach

Greedy approach: (“never look back” approach)

- in greedy approach, the solution is constructed in iterations.
- in each iteration some decision is made.
- the decisions are irrevocable (never look back)

Examples: knapsack density-based approach, MST, Dijkstra.

Greedy Approach, cont.

Greedy algorithms are usually very fast (due to irrevocability of the decisions - the steps are done once in a given order and the algorithm usually stops after a small number of steps).

On the other hand, they are usually simple and often give sub-optimal solution (non-global optimum is found).

For some problems greedy solution is optimal (e.g. mst, Huffman, shortest paths). For other not (e.g. K - except the continuous variant, TSP, BP, VC, SAT, etc.)

Technique 2: Local Search (hill climb, simulated annealing, etc.)

Discrete
Optimisation
Problems

(c) Marcin
Sydow

Introduction

In this approach we start with a random solution and (sometimes randomly) apply small (local) changes in order to *improve* it until we find a local optimum.

Examples: hill climbing, simulated annealing + variants (k-bundle, tabu search, etc.)

Local search are usually a trade-off between perfect-slow and fast-poor approaches. It is usually possible to control the degree of this trade-off (e.g. the running time limit and other parameters)

Technique 3: Evolutionary Approach (genetic algorithms, etc.)

Discrete
Optimisation
Problems

(c) Marcin
Sydow

Introduction

In this approach, each potential solution is represented as an individual in a population of solutions.

Individual is represented by a sequence of **genes** that encode the solution.

The fitness of each individual is a function of the value of its objective function (e.g. the total value of items in knapsack)

Example of Evolutionary Approach: Genetic Algorithm (GA)

Discrete
Optimisation
Problems

(c) Marcin
Sydow

Introduction

The most known example is *Genetic Algorithm*:

- create random population
- select individuals according to fitness (selection)
- let the individuals to produce offsprings (reproduction)
- mutate each offspring with some (low) probability (mutation)
- replace the population with the new generation of offsprings
- in each generation record the best individual

Issues in GA

Important problems:

- how to represent the individual with genes?
- how to define selection, cross-over and mutation operators

The above depends on the problem and *strongly* influences the quality of solutions found.

Example:

Propose a genetic representation and operators for Knapsack, TSP.

Many possible improvements:

- eliticism (protect a few best individuals to survive to the next generation)
- diversity control of population (to avoid getting stuck in a local optimum)

Technique 4: Dynamic Programming

Dynamic programming is an algorithmic technique of designing effective and efficient algorithms for some optimisation problems (that deserves a separate full lecture, at least) that satisfy some specific properties:

- any optimal solution is composed of optimal solutions to subproblems.
- if a subproblem has many optimal solutions it does not matter which one is selected

Practically it means writing a recursive formula for the solution that bases the solution of the problem on the solutions to (smaller) subproblems. However, this is usually not equivalent to recursion, since each of the smaller subproblems is solved only once and re-used.

Example: analogy to computing the value of binomial coefficient (do not use full recursion)

Example: dynamic programming for Knapsack

Let assume n items, the w, v vectors for the weight and value, and the capacity of C .

Let $P(i, c)$ denote the maximum profit achievable when only the first i items are considered, and the capacity is c .

The goal is to compute $P(n, C)$ (and the corresponding solution).

Obviously: $P(0, c) = 0$ and $P(i, 0) = 0$ (the base of the recurrent equations)

Dynamic programming for Knapsack, cont.

Thus, for $i, c > 0$, the solution that maximises the profit we either

- not use the i -th item ($P(i-1, C)$)
- or use it ($P(i-1, c - w_i) + v_i$)

We obtain the following recursive formula for $P(i, c)$:

- $\max(P(i-1, C), P(i-1, c - w_i) + v_i)$ if $w_i \leq c$
- $P(i-1, C)$ if $w_i > c$

Now, it is enough to compute the value of $P(n, C)$ bottom-up, that is equivalent to filling a rectangular table $P(i, c)$ (rows: items, columns: capacities), row by row or column by column, starting from the $P(0, 0)$ corner. Total time complexity is nC .

Pseudo-polynomial algorithms: a computational complexity remark

Discrete
Optimisation
Problems

(c) Marcin
Sydow

Introduction

Warning: this does not mean that the algorithm is polynomial, since C is represented in binary form and implies *exponential* number of steps as the function of data size.

However, this algorithm is quite fast for small values of C .

Such algorithms are called **pseudo-polynomial** (depend polynomially on numeric values in the input)⁵

⁵and such problems as *weakly NP-hard*

Technique 5: Linear Programming (LP) and Integer Linear Programming (ILP)

Discrete
Optimisation
Problems

(c) Marcin
Sydow

Introduction

Linear Programming and Integer Linear Programming are large fields (deserving separate courses) and are two variants of a very generic approach for:

- formulating optimisation problems
- solving optimisation problems

In this approach, the problem is formulated in a very specific form (so called *linear program*) as a **set of linear** inequalities and or equations concerning some *real variables* that represent the feasibility of solution and the optimality constraint. The solution is simply the solution to this set of inequalities/equations. When the variables are constrained to *integers* it is called integer linear programming.

LP cont.

Many known problems can be encoded in such form, but some problems cannot be expressed in the form of linear programs (some need squares, other non-linear mathematical functions, etc.)

There exist highly effective ready-to-use black-box software solvers for linear programs.

Linear programming is polynomially solvable (when variables are not constrained to be integers). Interestingly: the integer linear programming is **not** computationally easier than LP (rather the other way round) - notice the analogy to the difference in hardness between discrete and continuous knapsack problem (continuous is easier to solve)

Example of a linear program

Consider the following linear program:

Given vectors of rational (non-negative) variables v and w find the solution vector of rational (non-negative) variables $x \leq 1$ that satisfies:

- $v^T \cdot x \leq C \in \mathbb{Q}^+$ (feasibility constraint)
- so that: $w^T \cdot x$ is maximum possible (optimality constraint, i.e. objective function)

Is this LP or ILP? Is this linear program equivalent to any of the discussed optimisation problems? Which one? What changes if x are constrained to be integers $\{0, 1\}$?

Thank you for attention