Algorithms

Lecture Notes — Local Search

Imdad ullah Khan

Contents

1 Strategies for Dealing with Hard Problems

Suppose you are tasked with solving some kind of problems in your company. If you are lucky, the problem is solvable using some design paradigm that we have studied so far in this course. In particular, dynamic programming and linear programming are able to take care of many problems. However, you may not be so lucky. Then, you would tell your boss one of the following three things:

- "I cannot solve the problem, because I am too dumb"
- "The problem is not solvable (in poly-time)". However, you would need a proof, which will actually amount to $P = NP$ (if your problem is in NP). In this case, you would no longer need the job. Simply collect your million dollars from the Clay Institute and enjoy your life.
- "I can not solve the problem but neither can all these extremely smart people." if you can prove that your problem is NP-complete. If you really worked hard to find a solution but your attempts were fruitless, a little more work may lead you to this proof.

The trouble with the above scenarios is that the problem at hand remains and this theoretical exercise does not help practically. Hence, we will now study what to do in such a case.

"NP-Completeness is not a death certificate, it is the beginning of a fascinating adventure"

When you prove a problem to be NP-Complete (or NP-hard), then, as per popular belief that $P \neq NP$, it essentially means that

1. There is no polynomial time
2. deterministic algorithm
3. to exactly solve this problem
4. for all possible input instances

The four keywords impose very strict requirements. Unless our goal is to prove $P = NP$ or $P \neq NP$, regarding which we already assume the latter, then, in practice we may relax one of these requirements to sustain our job. It turns out that relaxing any of these requirements does indeed help a lot practically and opens up huge avenues of possibilities.

So what are our options to deal with NP-Hard problems? Let's consider the following questions:

- Do we need to solve the problem for all valid input instances?
  - Sometimes, we just need to solve a restricted version of the problem that includes realistic instances (special cases)
- Is exponential-time algorithms OK for our instances?
  - The problem with exponential-time algorithms is not primarily that they are "slow" but rather that they don't scale well. So if our relevant instances are small, then exponential-time may be acceptable. Moreover, we can reduce the base or exponent in many practical cases. For example from 2^n to $2^{\sqrt{n}}$ or 1.5^n.
- Is non-optimality acceptable?
  - It is OK if our algorithm just outperforms other algorithms in some cases. To understand this idea better, consider the following scenario: A fit person and a non-fit person are being chased by a bear. The fit person says: "Spending so much time in the gym is worth it." The non-fit person says: "Why? You still won't outrun the bear." The fit person replies: "I don't need to outrun the bear. I just need to outrun you."

Therefore, we may sacrifice one of three desired features i.e. solve any arbitrary instance of the problem, optimally and in polynomial time by designing algorithms that,

respectively, solve special cases of the problem, or approximately solve problems, or may take exponential time. We summarize these strategies to cope with the hard requirements of NP-Complete problems by relaxing them in Table 1.

Poly-time	Deterministic	Exact/Optimal solution	All cases/ Parameters	Algorithmic Paradigm
✓	✓	✓	✗	Special Cases Algorithms Fixed Parameter Tractability
✓	✓	✗	✓	Approximation Algorithms Heuristic Algorithms
✓	✗	$\mathbb{E}$(✓)	✓	Mote Carlo Randomized Algorithm
$\mathbb{E}$(✓)	✗	✓	✓	Las Vegas Randomized Algorithm
✗	✓	✓	✓	Intelligent Exhaustive Search

Table 1: Coping with NP-hard Problems

We briefly describe each of the above listed approaches to tackle hard problems before diving deeper into each one individually.

1. Special cases can be based on the structure of input instances or depend on particular range of one or more parameters. The problem is easier for such special cases, for which exact results are attainable in polynomial time.

2. Approximation algorithms and heuristic algorithms provide nearly exact solutions, i.e. the output is 'close' to the optimal solution. While approximation algorithms output solutions of guaranteed quality in poly-time, heuristics algorithms do not have any guarantees on the solution which is hopefully good in poly-time.

3. Randomized algorithms use coin flips for making decisions. In addition to being used for approximation of hard problems, they are also used for easy problems (in P). Monte Carlo algorithms output solutions which may not be exact but always take polynomial time whereas Las Vegas algorithms always output the optimal solution but not necessarily in polynomial time.

4. Intelligent exhaustive search takes exponential time in the worst case but could be very efficient on typical more realistic instances where the base and/or exponent

are usually smaller. Techniques for this approach include Backtracking, Brand-and-Bound, and Local Search.

2 Local Search

2.1 TSP

2.2 Max-Cut

2.3 Local Optimum

2.4 Simulated Annealing

2.5 Gradient Descent