Nonliner Optimisation – Introduction Lecture 1

Justin Pearson

November 9, 2007

`http://user.it.uu.se/~justin/Teaching/Nonlinear/`

Course Content

- Lecture 1
  - Introduction, revision(?) of complexity theory, modelling with decision variables and summary of solution methods.
- Lecture 2
  - Introduction to Local Search, Neighbourhoods and meta-heuristics
- Lecture 3
  - More on local search, simulated annealing and Variable Nieghbourhood Design.
- Lecture 4
  - Complete Search, Sat
- Lecture 5
  - Complete Search, Constraint Satisfaction
- Lecture 6,
  - Constraint Satisfaction, Modelling and Symetry breaking.

Note that this might change (possibly can be changed by you)!

- Why study complexity theory?
  - Complexity theory can be a fascinating study of the fundamentals of computer science (algorithms).
  - Complexity studies how hard problems are and tries to put them into classes

$$P \subseteq NP \subseteq PSPACE \subseteq \cdots$$

  - The fundamental problem in complexity is deciding if the above inclusions are proper or not that is for example:

$$P \neq NP?$$

Background in Complexity Theory

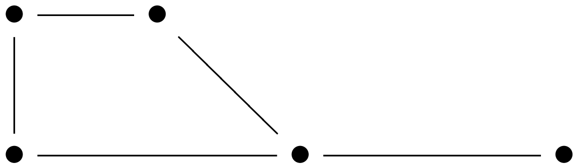
- What do we need to know from Complexity theory?
- We need to know how hard people think problems are.
- If we know that a problem is in NP even though it is not known if $P = NP$ or not we know that lots of clever people believe that the problem is hard.
- Which means that if we think we've found a polynomial time algorithm then either we are very clever or we've made a mistake.

- *NP* stands for non-deterministic polynomial time.
- The actual definition is quite complicated:
  - A decision problem is in the class *NP* if it can be solved in time which is a polynomial function of the input size by a non-deterministic Turing machine.
- A non-deterministic Turing machine is a computer with an unbounded amount of memory where certain instruction steps are non-deterministic and magically pick the right answer.

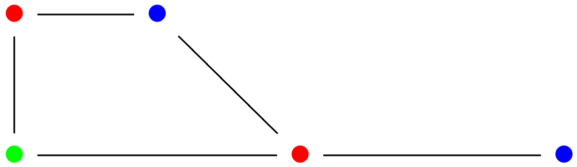
What is NP? — Better Definition

- A decision problem is in NP a solution can be verified to be a correct solution in polynomial time.
- This gives a non-deterministic solution method, guess a solution (and magically get the right answer) then test if it is a correct solution in polynomial time.

Given a graph



Is it possible to colour the nodes with *n*-colours such that if two nodes are directly connected with a node then they do not have the same colour?



- We don't have access to non-deterministic computers. Quantum computing is still a way away (and they are not really the same thing)
- Essentially the separation into guessing and testing gives a very simple and naive algorithm.
  - Enumerate all possible solutions, send them to the polynomial tester and stop when you've found a solution.
- So go through all the possible assignments of nodes to colours until you find a solution.
- The problem is that the solution space is usually exponential in size relative to the input.
- For example k colours, N nodes then we have k^N possible colourings.
- Obviously people keep looking for better ways.

NP completeness

- Completeness is an important concept in complexity theory.
- A problem is NP-complete if
  - it is in NP and
  - It is *NP-hard* that is every other problem in NP is reducible to it.
- A problem L is *reducible* to C is there is a polynomial time reduction from instances of L to instances of C such that yes answers to L correspond to yes answers of C under the reduction.

- Many problems are NP-complete, for example graph colouring.
- This means if we have a solution technique for one of the problems that runs in deterministic polynomial time then we can solve them all.
- A NP-complete problem is powerful enough to simulate any other NP-complete problem.
- NP-complete problems are all as hard as each other, in the worst case with any algorithm we expect exponential time. In the worst case we will just be enumerating the solutions.

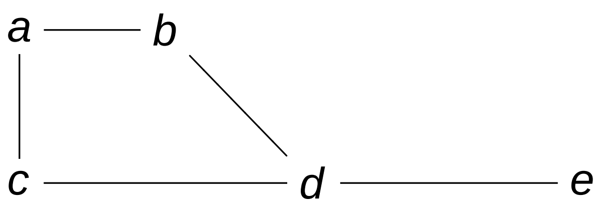
NP — What to do?

- What happens when we have an NP-complete problem to solve?
- Complexity theory tells us that the problem is hard, it tells us that lots of people think it is hard as well. But if we actually have a problem to solve then we can't just give up and go home.
- There are many possibilities:
    - Maybe there is some aspect of you particular instance that makes the problem easy to solve. For example if you are only interested if it is possible to colour the graph with only 2 colours then the problem has a polynomial time algorithm.
    - Particular heuristics can be applied that work well on your datasets. NP completeness only tells us about the worst case behaviour.

- Boolean formula satisfaction
- Timetabling
- Traveling Salesman problem
- Knapsack problem.

Modelling NP-complete problems

- The graph colouring example suggest a very simple way of *modelling* the problem.
- A graph is a pair (V, E) where E is a set of edges that is a set of pairs $\{a, b\}$ where $a \in V$ and $b \in V$.
- For example our graph form before:



- Would be represented as

$$(\{a,b,c,d,e\},\{\{a,b\},\{a,c\},\{b,d\},\{c,d\},\{d,e\}\})$$

- We could represent the problem of colouring with N colours as follows:
  - Given the variables

$$v_a, v_b, v_c, v_d, v_e$$

  - Each variable has the initial domain $D = \{1, \ldots N\}$
  - Subject to the constraints:

$$v_a \neq v_c, v_a \neq v_b, v_b \neq v_d, v_c \neq v_d, v_d \neq v_e$$

- A solution is a mapping

$$f : V \rightarrow D$$

such that all constraints are satisfied.

$$v_a \neq v_c, v_a \neq v_b, v_b \neq v_d, v_c \neq v_d, v_d \neq v_e$$

- For example given $D = \{1, 2, 3\}$

$$f(v_a) = 1, f(v_b) = 2, f(v_c) = 2, f(v_d) = 1, f(v_e) = 2$$

is a solution.

- While

$$f(v_a) = 1, f(v_b) = 1, f(v_c) = 2, f(v_d) = 1, f(v_e) = 2$$

is not a solution because is violates the constraint $v_a \neq v_b$ (amoungst others).

- Given conjunction of boolean formulas:

$$\phi = (x \vee y) \wedge (x \vee \overline{y} \vee z) \wedge (y \vee \overline{z})$$

The variables are v_x, v_y, v_z all with the domain $\{0, 1\}$ subject to the constraints the code up the meaning of the boolean formulas.

- So for example $f(v_x) = 1, f(v_y) = 1, f(v_z) = 0$ is a solution.
- While $g(v_x) = 0, g(v_y) = 0, g(v_z) = 0$ is not a solution because $(x \vee y)$ is violated.

Optimisation

- The general form a NP-complete problem is something like:
  - A set of variables:
    $$V = \{v_1, v_2, v_3, \ldots, v_n\}$$
  - Each taking values from some domain D
  - Subject to a set of constraints $C_1, \ldots, C_n$ that restrict the allowed values of subsets of the variables.
  - Note, how you represent constraints efficiently or how the problem is actually solved is a different matter.
- An optimisation problem includes an extra piece of information a cost function $c : V \rightarrow \mathbf{R}$ (or more often $\mathbf{N}$) Which for each solution assigns a cost.
- The problem then is to find a solution that minimizes or maximizes the cost.

- Complexity theory normally only deals with yes/no problems.
- Given a set X, let X^* be the set of all strings using symbols from X.
- For example $X = \{a, b\}$ then

$$X^* = \{\epsilon, a, b, aa, aaa, b, bb, ba, ab, aab, bbaa, \ldots\}$$

- X^* is always an infinite set.

Complexity Theory of optmisation problems

- A language L is simply a subset of X^*.
- In complexity theory we code problems as languages and study the complexity of machines needed to recognize the language.
- A machine M recognises L is for all $I \in L$, $M(I)$ outputs a yes and $I \notin L$ then $M(I)$ outputs a no.
- For example if we code all Sat instances somehow, let Sat be the set of all satisfiable sat instances in the coding.
- Then a Satisfaction tester will output yes if given a coding of satisfiable instance.

- Many problems have the property that if it possible to find a `yes,`no answer in polynomial time then it is possible to find an actual solution.
- Being told that you have a satisfying assignment to a problem is less useful then actually having the assigment.

Complexity Theory of optmisation problems

- Optimisation problems are at least as hard as decision problems. Given an optimization with cost function c, we can construct the descsion problem for a given bound

$$\exists s . M(s) \wedge c(s) \leq B$$

- Sometimes it is not important to have exactly the best solution,
- but if we can have something that is guaranteed to be with in a certain ratio of the best, we might be ok.
- Let $\mathcal{A}$ be an approximation algorithm for a given combinatorial optmization problem. Then R is a *performance bound* of $\mathcal{A}$ for a given problem instance I if

$$\frac{\mathcal{A}(I)}{OPT(I)} \leq R$$

Where OPT is the cost of the optimal solution (this assumes that the problem is a maximization problem).

- So lets say you had a an algorithm that for all instances of I which gave 90% of the optimal.
- Which would mean that

$$\mathcal{A}(I) = OPT(I) * 0.9$$

- So

$$\frac{\mathcal{A}(I)}{OPT(I)} = \frac{0.9 * OPT(I)}{OPT(I)} = 0.9$$

- So R is equal to 0.9.

- The class APX is the class of optimisation problems for which there exists an approximation algorithm with some finite constant performance ratio.
- Assuming that $P \neq NP$ then there problems for which there are not in APX. (For example winner determiniation in a combinatorial auction).

Knapsack Problem – Problème du sac à dos

- Suppose that you are going on a camping trip, you want to put everything in a knapsack.
- The problem is that you have too many things you can only carry 20Kg. You have to decide which subset of items you will carry.
- To make the problem harder you assign a value to each item, for example a bottle of wine would be valued more than a bottle of coke, while a bottle of wine weighs more than a bottle of coke.
- You want to maximize your value.

- Given n items which with a value $p_1, \ldots, p_i, \ldots, p_n$ each with a weight $w_1, \ldots, w_i, \ldots, w_n$.
- The Decision variables are a bit harder in this case.
- We have n-decision variables $x_1, \ldots, x_n$ all with the domain $\{0, 1\}$. 0 means we don't pack it, 1 means we take it with us.
- So we want to maximize

  $$\sum_{j=1}^{n} p_j x_j$$

  subject to the constraint

  $$\sum_{j=1}^{n} w_j x_j \leq c$$

This seemingly simple problem is NP-complete.

Problem Modelling

We have three things:

- The actual problem.
- Modelling the problem with variables, values and constraints.
- Using your model within some solution technique.

As we have seen with the knapsack it is not always obvious how to model things with variables and values.

- Further your model might not have such a good fit with your solution technique.
- For example if you are using a SAT-solver. That is a solver for boolean formulas. By NP-completeness you can solve any NP-complete problem using a suitable coding.
- But some codings are better than others.

Different solver have different quirks, often re-modeling the problem can result in very different performance. We will talk about modelling issues later.

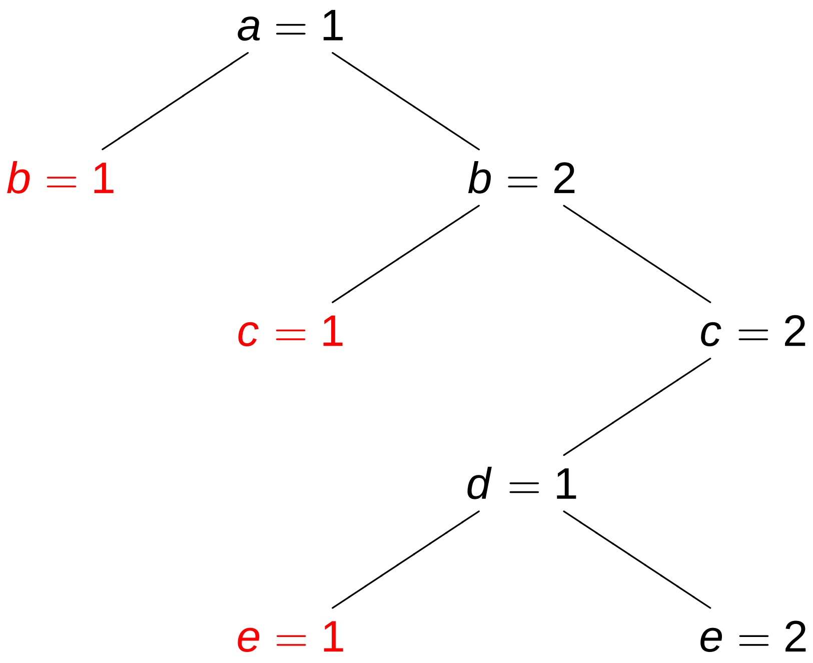
A very quick tour of solution methods

Solutions to NP-complete problems can be divided up into two classes:

- Complete Search (often trying to be more intelligent)
- Incomplete search via various heuristics.

A complete search is guaranteed to find the best solution. Complete search works by intelligently pruning parts of the search space. Incomplete search tries some heuristic that often converges to a solution very fast but does not guarantee that we have found the best solution.

Back to our graph-colouring example. Take the graph $G = (\{a, b, c, d, e\}, \{\{a, b\}, \{a, c\}, \{b, d\}, \{c, d\}, \{d, e\}\})$ If we search the variables in the order v_a, v_b, v_c, v_d, v_e and the domain in the order $\{1, 2, 3\}$ we get the following search tree:



Backtrack Search

- Complete search often proceeds by backtracking. A partial solution is built up at each stage it is extended, if the extension leads to a failure then try to extend it with a different value.
- Repeat until either the whole search space has been searched and hence there is no problem or stop when a solution is found.
- It is a little more complicated when you have cost functions, you have to use branch and bound (more later).

To optimise backtrack search, you want to fail as early as possible and learn from your failures.

- Assign an initial random solution.
- For a node in the graph, count the number of neighbours with the same colour. Call the conflict of a node. Define the conflict of the whole problem as the sum of all the conflicts of all the nodes.
- Pick a node with the maximum conflict (there might be more than one). Change the colour to reduce the total conflict.
- Repeat until a solution is found.

Local Search

A preview of the next lecture.

- A local search algorithm consists of the following:
  - A notion of a configuration, (variables and domains)
  - A neighbourhood function, that takes a configuration and returns a set of neighbours.
  - A notion of cost.
- The basic local search algorithm then
  - Starts with an initial random configuration
  - Picks a nieghbour from the nieghbourhood set that reduces the cost
  - Then continues looping until a solution is found.

There are many problems with local search:

- Finding good neighbourhoods, too big, takes to long to find a cost reducing move, too small takes to long to converge to a solution.
- Local minimum
- Cycles in the search space