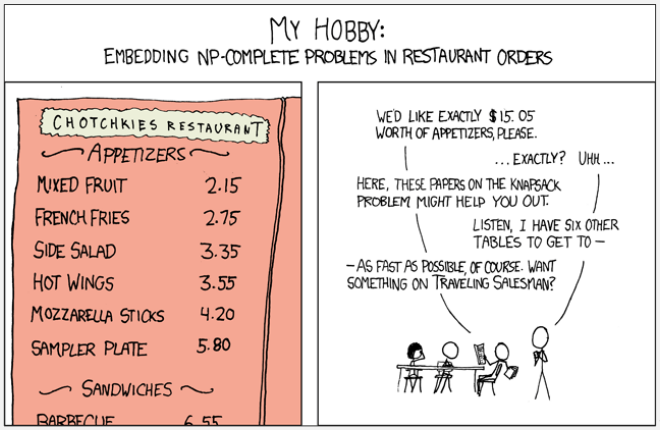
Many Interesting Problems are NP-Complete



http://xkcd.com/287/

Many Interesting Problems Are NP-Complete

More than 3000 NP-complete problems are known.

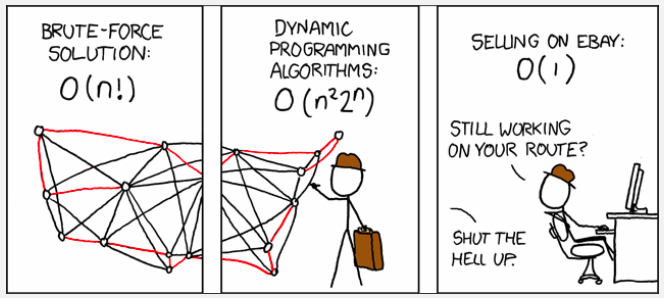
- http://en.wikipedia.org/wiki/List_of_NP-complete_problems

traveling salesman	min cost hamiltonian cycle
closest string	find a string with minimum distance to all other strings in a set
art gallery problem	minimum number of guards to cover all hallways
berth allocation problem	assignment of ships to berths in port to minimize service time, minimize delayed departures, optimize fuel consumption, etc
generalized assignment	assignment of agents to tasks so as to not exceed the agents' budget and the total profit of the assignment
maximum common subgraph isomorphism	applications in cheminformatics
multiprocessor scheduling	min time required to schedule jobs on multiple processors
vehicle routing	minimize cost of distribution of goods from central warehouses to customers
bin packing	pack objects into bins to minimize number of bins (e.g. loading trucks, backing up files onto removable media)
register allocation	assign variables to available registers

note: most of these descriptions are for the function version of the problem – technically not in NP, but the function version isn't any easier than the decision version

So...

...you need to solve an NP-complete (or -hard) problem. What do you do?



http://xkcd.com/399/

Tactics

- dodge the bullet
  - you may not need to solve for large inputs
  - you may only need to solve for a special class of inputs which have an efficient algorithm
- bite the bullet
  - recursive backtracking – clever pruning!
  - branch-and-bound – clever bound functions!
- settle for good enough
  - heuristics – clever tricks which seem to work well, but without guarantees on runtime or solution quality
  - approximation algorithms – with bounds on the quality of the approximation

Best-First Search

(*) ADM refers to best-first search as branch-and-bound but the order in which you address subproblems (ADM) is distinct from pruning away non-optimal solutions (our usage) – though the two can go nicely together

- recursive backtracking uses depth-first search
- if the goal is only a single solution (whether any or optimal), *best-first search* can speed up finding that solution
  - for optimization problems, combine with branch-and-bound
- best-first search explores the most promising subproblem first
  - standard best-first uses the cost of the partial solution
    - favors shorter partial solutions first
    - can end up exploring a lot if there's a late expensive choice
  - A* uses the cost of the partial solution + estimate of cost of the remaining solution
    - estimate must be safe (pessimistic)

Best-First Search

- a major drawback of best-first search is memory usage
  - DFS only stores the ancestors of the current subproblem
    - depends on the height of tree
  - best-first search stores up to the width of the tree
    - A* can make enough of an improvement to be usable

> To get an answer from a slow program you just have to be patient enough, but a program that crashes because of lack of memory will not give an answer no matter how long you wait.
>
> – ADM p 302

Heuristics – Local Search

One family of heuristics involves *local search*.

The idea.

- apply small changes to a solution, keeping the changes that result in improvements

An example.

- TSP – replace edges (u,v) and (u',v') in a solution with edges (u,v') and (u',v)

Heuristics – Local Search

Local search can be effective when –

- there is great *coherence* in the solution space
  - nearby solutions are a little better or a little worse
  - ideally only one hill
- *incremental evaluation* is much cheaper than global evaluation
  - e.g. updating cost of TSP solution when swapping two edges is O(1) vs O(n) to compute cost of a cycle in the first place

Hazards.

- local minima
  - solutions for which no small change results in improvement, but which are not optimal

Dealing with Local Minima

- randomization and restarts
  - choose a random starting solution
  - randomly select the local move from amongst the available choices
  - repeat and take the best result
    - quickly increases the probability of finding a good local optimum, but there may be many more bad local optima than good ones...
- simulated annealing
  - occasionally allow moves that make the current solution worse
    - increases the time needed to find a local optimum
  - decrease the probability of bad moves as time goes on
    - coming up with a good annealing schedule is not necessarily easy
- other tactics
  - e.g. genetic algorithms – maintain a population of solutions, allow crossover between parts of solutions

Random Sampling

- repeatedly choose a random solution, keeping the best one found so far

To be effective, requires

- a high proportion of legal solutions
  - so you can stumble on one
- no coherence in the solution space
  - coherence means there can be a notion of getting closer to a solution, which would be advantageous to exploit (local search)

Approximation Algorithms

Heuristics are hopefully fast and result in hopefully good (though not necessarily optimal) solutions, without guarantees.

Approximation algorithms yield a guaranteed “close enough” solution.

Approximation Algorithms

But even approximation can be hard.

- the good: polynomial-time approximation algorithms which get arbitrarily close to the optimal solution
  - e.g. 0-1 knapsack – based on the idea of scaling and rounding weights so W is polynomial in *n*, then using dynamic programming (approximation comes from roundoff)
- the OK: polynomial-time approximation algorithms, but with limits on how close the approximation can get to the optimal solution
  - e.g. MST approximation for TSP satisfying the triangle inequality (cost at most twice optimal)
- the bad: any polynomial-time approximation algorithm can be arbitrarily far from the optimal solution
  - e.g. general TSP

Developing Approximation Algorithms

- simple procedures are not necessarily useless
  - e.g. vertex cover – repeatedly pick an edge (u,v), add u and v to the cover, remove edges incident on u and v
    - at most twice as large as optimal cover (any cover must include at least one vertex per edge picked)
    - can do slightly better with a more complex algorithm, but can't get arbitrarily close
- greedy is not necessarily advantageous
  - e.g. vertex cover – picking highest degree vertex can lead to worse worst-case performance
- even heuristics that don't impact the worst case can still improve performance in practice
- can get the best of both worlds
  - use both the approximation algorithm and a heuristic, and take the best

Key Takeaways

- what tactics exist for making things manageable
  - realize that you have an easier special case
  - embrace exhaustive search and work on pruning
  - let go of optimality
- a brief introduction to those tactics so you have a starting point for brainstorming and for further study
- the importance of experience and good references
  - recognizing your problem as a known problem in disguise means you can leverage what is known about the other problem