Tackling **NP-Hard** Problems

Dr Kamal Bentahar

School of Engineering, Environment and Computing
Coventry University

05/12/2016

Last time...

NP-Hardness

A language is **NP-hard** if every problem in **NP** is (polytime) reducible to it.

NP-Completeness

A language is **NP-complete** if:

1. it is **NP-hard**
2. and it is itself in **NP**

Optimization Problems

Maximize or minimize a function of the input variables.

Tackling **NP-hard** problems

1. Exact methods
   - Exhaustive search.
   - Possibly better exponential time algorithms, e.g. Dynamic Programming.
   - Tractable special cases which can be solved quickly.
2. Approximation methods (Inexact methods)
   - e.g. **(meta-)heuristics** – fast, but not always correct.

Exact Methods: Exhaustive search

- General problem-solving method
- Always finds solution if it exists
- Usually expensive – tends to grow exponentially

Exhaustive search

1: **for all** possible candidates **do**
2:     **if** candidate satisfies the problem's conditions **then**
3:         **return** candidate
4:     **end if**
5: **end for**
6: **return** no solution

Exact Methods: Dynamic Programming

- Build solution by first solving smaller problem instances
- Suitable when the problem has:
  1. overlapping sub-problems
  2. and optimal sub-structure making global optima a function of local optima.

Dynamic Programming

1: Characterize structure of optimal solution.
2: Recursively define value of optimal solution.
3: Compute in a bottom-up manner – store intermediate results in a table.

Time-Space trade off

Dynamic Programming vs Exhaustive Search

- Exhaustive search tends to require less space but more time.
- Dynamic programming: space complexity can be big (table size).

Approximation algorithms: Meta-heuristics

Optimization problem

Optimize the value of an “objective function” f.

- Greedy search
- Multi-starts
  - GRASP
  - Tabu Search
- Iterative improvement (Local search)
- Simulated annealing (Probabilities for worsening moves)
- Tabu search (Adaptive memory)
- Genetic Algorithms

Approximation algorithms: Greedy

- Try to optimize f by choosing one component at a time.
- At each stage, a component that maximizes immediate gain is selected.
  (Decisions best in the short term without considering long term consequences)
- Can be quite efficient.

Local Search – Neighbourhoods & Optima

Trying to improve approximate solutions

Each possible solution can be thought of as having a neighbourhood of solutions which can be reached by making a small change.

However, local search may fail to reach a global optimum: it may get stuck in a local optimum which is not a global optimum.

Local Search – Strategies

Best fit: search the whole neighbourhood and then move to the best neighbour solution.

First fit: search the neighbourhood and move to the first improving solution found.

Random first fit: pick random solutions from the neighbourhood and move to the first one found.

Candidate list strategies: reduce the number of possible choices at each step: only search a subset of the neighbourhood solutions.

Multi starts: restart every time the algorithm gets stuck (with a random solution or some variation on the greedy solution (random changes, ruling out previous choices.

Local Search – Iterative Improvement

- Search a “neighbourhood” of a solution for an improvement.
- Move to improved solution and search its neighbourhood.
- Keep going until you find no more improvements.

We can use this with initial solutions from greedy algorithms or randomly generated ones:

determine initial candidate solution s $\triangleright$ e.g. through greedy search
while s is not a local optimum **do**
    choose a neighbour s' of s such that $f(s') < f(s)$
    $s \leftarrow s'$
end while
return s

Randomized Iterative Improvement

choose a probabilty threshold $w \in [0..1]$
determine initial candidate solution s
while termination condition is not satisfied **do**
    randomly generate a number $p \in [0..1]$
    if $p < w$ **then**
        choose a neighbour s' of s uniformly at random
    else
        choose a neighbour s' of s such that $f(s') < f(s)$
        or if no such s' exists then choose s' such that $f(s')$ is minimal
    end if
    $s \leftarrow s'$
end while
return s

Greedy Randomized Adaptive (GRASP)

Greedy Randomized Adaptive Search Procedure (GRASP)

while termination criterion is not satisfied **do**
    generate candidate solution s using subsidiary greedy randomized constructive search
    perform subsidiary local search on s
end while
return s

Approximation algorithms: Simulated Annealing

Effective approach modelled on the cooling of molten materials.
We have a variable called *temperature*, which decreases simulating cooling.
Probabilities are based on the Boltzmann distribution.

Simulated Annealing

determine initial candidate solution s
set initial temperature T according to annealing schedule
while termination condition not satisfied **do**
    probabilistically choose a neighbour s' of s
    if s' satisfies probabilistic acceptance criterion (depending on T) **then**
        $s \leftarrow s'$
    end if
    update T according to annealing schedule
end while
return s

Approximation algorithms: Tabu search – Adaptive memory

An alternative to the randomized approach is the memory-based approach

- Solutions consist of many components
- After removing a component from a solution, we mark it as tabu (forbidden) for some number of iterations
- The number of iterations is called the tabu tenure
- The neighbourhood is then restricted to use non-tabu components

Tabu Search

1: determine initial candidate solution s
2: **while** termination condition not satisfied **do**
3: determine set N of non-tabu neighbours of s
4: choose a best improving solution s' in N
5: update tabu attributes based on s' $s \leftarrow s'$
6: **end while**
7: **return** s

Genetic Algorithms

*So far we have looked at **trajectory approaches**, where we keep only one current solution and make progressive modifications to it*

Population based approaches use more than one solution at a time and make progressive changes to that population:

- Genetic/evolutionary algorithms
- Swarm intelligence (ant colony optimisation etc)

Genetic Algorithm

1: determine initial population p
2: **while** termination criterion not satisfied **do**
3: generate set pr of new candidate by recombination
4: generate set pm of new candidates from p and pr by mutation
5: select new population p from candidates in p, pr, pm
6: **end while**

When to use meta-heuristics?

Under what circumstances is it best to use heuristics to solve optimization problems?
When the problem is NP-Hard, otherwise solve exactly