Heuristic Optimisation

Revision Lecture

Sándor Zoltán Németh

http://web.mat.bham.ac.uk/S.Z.Nemeth

s.nemeth@bham.ac.uk

University of Birmingham

Topics

1. Basic difficulties in problem solving. The need for heuristic optimization
2. Basic concepts: representation, optima, neighbourhood
3. Exhaustive search and local search
4. Greedy algorithm
5. A* search
6. Simulated annealing
7. Tabu search
8. Evolutionary algorithms

What this course was about

- It was about traditional and modern heuristic optimization methods for problem solving, but not about telling what the ultimate method to use is

Some problems are difficult to solve because:

- The size of the search space is **huge**
- The problem is very complicated, the solutions to a simplified model are useless
- The evaluation function varies with time, so a set of solutions is required
- There are heavy constraints, it is hard to even construct a feasible solution

Finding the model

$Problem \Rightarrow Model \Rightarrow Solution$

Simplify the model and use a traditional optimizer:

$Problem \Rightarrow Model_{approx} \Rightarrow Solution_{prec}(Model_{approx})$

Keep the exact model and use a non-traditional optimizer to find a near-optimal solution:

$Problem \Rightarrow Model_{prec} \Rightarrow Solution_{approx}(Model_{prec})$

Heuristic optimization

The original Greek word means “I found it”

We use some information available about the problem

to reduce the region(s) of the search space to be checked

or to speed up the search

at the cost of not guaranteeing the optimum

Ingredients

Problem $\Rightarrow$ *Model* $\Rightarrow$ *Solution*

For any algorithmic approach:

- The **representation** - the encoding of alternative solutions
- The **objective** - the purpose (may or may not be minimising one function)
- The **evaluation function** - how good a solution (given the representation) is

Evaluation function

Generally it is not the same thing as the objective

The **evaluation function** is a mapping from the space of (feasible) solutions to a set of numbers (reals). The numeric value indicates quality of solution

A solution that **meets** the objective should also have the **best** evaluation

A solution that **fails** to meet the objective **cannot** be evaluated to be better than a solution that meets the objective

Sometimes it is sufficient to know whether one solution is better than the other, without knowing how much better

Defining the optimisation problem

Search problem = optimisation problem

Given a search space S together with its feasible part $F \subseteq S$, find $x \in F$ such that

$eval(x) \leq eval(y), \forall y \in F$ (minimisation)

x is a **global solution**

Reasons for using heuristics

- We can avoid combinatorial explosion
- We rarely need the optimal solution, good approximations are usually sufficient
- The approximations may not be very good in the worst case, but in reality worst cases occur very rarely
- Trying to understand why a heuristic works/does not work leads to a better understanding of the problem

Classification of optimisation algorithms

- Algorithms that only evaluate **complete solutions**:

  exhaustive search
  local search
  hill-climbing
  simulated annealing
  tabu search

- Algorithms that evaluate **partially constructed or approximate solutions**:

  greedy search
  divide & conquer
  A*

Algorithms on complete solutions

1. Initialise **best** solution.
2. Generate a solution x according to the specifics of the algorithm.
3. If x is better than **best**, replace **best** by x.
4. Repeat steps 2.-3.

Algorithms on partial solutions

- **Incomplete** solution to the **original** problem

  A subset of the original problem's search space with a particular property hopefully shared by the real solution

- **Complete** solution to a **reduced** problem
  1. We decompose the original problem into smaller & simpler problems
  2. We solve these problems
  3. We try to combine the partial solutions into a solution to the original problem

Exhaustive search

Looks at every possible solution in the search space until the global optimum is found

If the value of the global optimum is not known,
all points in the search space must be checked

The algorithm is **very simple**!

We have to generate every possible solution to the problem in a systematic way

How to actually generate the sequence of all possible solutions depends on the **representation**!

Local search

1. Pick a solution from the search space. Define this as the **current** solution.
2. Apply a transformation to the current solution to obtain a **new** solution.
3. If the new solution is better than the current one, replace current solution by the new solution.
4. Repeat steps 2.-3. until no improvement is possible.

Small neighbourhood - quick, but we might get stuck in local optima

Large neighbourhood -less chance for getting stuck, but takes longer

Greedy algorithm

A very simple algorithm that constructs the solution step by step

At each step the value for one decision variable is assigned by making the best available decision

A **heuristic** is needed for making the decision at each step: what is the best now?

The best 'profit' is chosen at every step - the algorithm is **greedy**!

We **cannot** expect the greedy algorithm to obtain the **overall** optimum

Divide and conquer

Divide&conquer(P)

1. Split problem P into subproblems $P_1, P_2, \ldots, P_k$.
2. For i taking all the values from 1 to k

   get the solution S_i to problem P_i
3. Combine $S_1, S_2, \ldots, S_k$ into the solution S for problem P.
4. Return the solution S.

The algorithm is cost effective **only if** its cost is **less than** the cost of solving the original problem

A*

$$f(n) = g(n) + h(n)$$

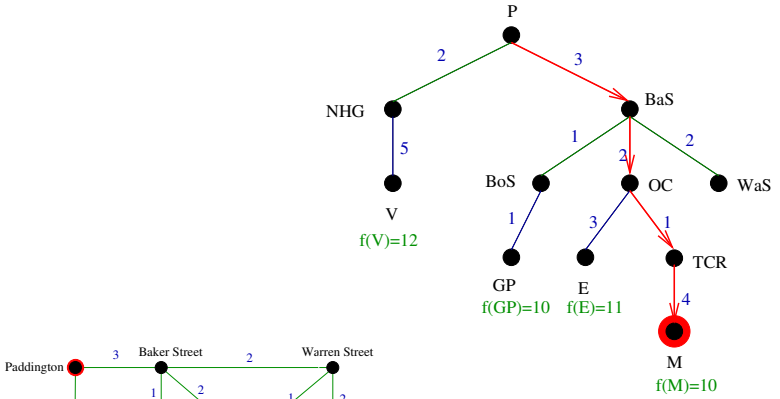
$g(n)$ is a **measure** of the cost of getting from the initial state to the current state n.

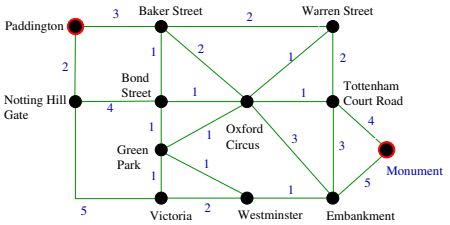
$h(n)$ is an **estimate** of the cost of getting from the current state n to the goal state.

The better h estimates the real distance, the closer A* is to the **"direct"** path

If h never overestimates the real distance, A* is **guaranteed** to find the optimal solution

A* example





Simulated annealing: analogy

Methods for simulation of the physical annealing process can be directly applied to solve optimisation problems.

Physical system	Optimisation problem
state	feasible solution
energy	evaluation function value
ground state	optimal solution
temperature	control parameter T
careful annealing	simulated annealing

Properties of SA

- In contrast to hill-climbing, simulated annealing accepts some **deterioration** in the quality of solutions. This helps avoiding local optima.
- Initially, at high temperatures, **large** deteriorations are accepted.
- As temperature decreases, only **smaller** deteriorations are accepted.
- As temperature approaches 0, SA behaves as **local optimisation**.
- Simulated annealing is a generalisation of local search.

Tabu search

- Accepts non-improving solutions **deterministically** in order to escape from local optima (where all the neighbouring solutions are non-improving) by guiding a hill-climbing algorithm
- Uses memory in two ways:

  to prevent the search from revisiting previously visited solutions

  to explore the unvisited areas of the solution space

Tabu search extensions

Aspiration criterion:

In specific circumstances an ”outstanding” tabu can be accepted as the next point to be visited

The memory discussed so far is **recency-based**

Frequency-based memory can be used to **diversify** the search:

$$H(i) = j$$

”during the last h iterations of the algorithm variable i was flipped j times”

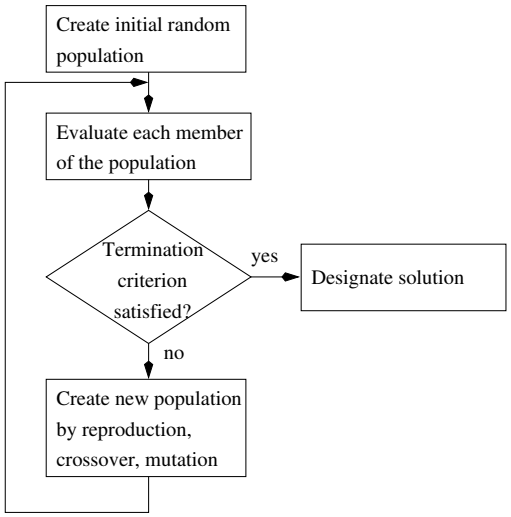
Tabu search vs simulated annealing

- Both were designed to escape local optima
- Both work on complete solutions
- Tabu search only selects worse moves if it is stuck, whereas simulated annealing can do that all the time
- Simulated annealing is stochastic
- Tabu search is deterministic
- The parameters must be carefully chosen for both

Evolutionary algorithms

Nature	Evolutionary algorithms
Individual	Solution to a problem
Population	Collection of solutions
Fitness	Quality of a solution
Chromosome	Representation of a solution
Gene	Part of representation of a solution
Crossover	Binary search operator
Mutation	Unary search operator
Reproduction	Reuse of solutions
Selection	Keeping good subsolutions

Genetic Algorithm



Previously evolved good parts of solutions (**schemata**) can be transferred to subsequent generations through crossover.

Genetic Algorithms for TSP

- **Adjacency representation**: City j is in position $i \Leftrightarrow$ the tour contains (i, j)
  - **Alternating edges crossover**
  - **Alternating random length subtours**
  - **Heuristic crossover**
- **Ordinal representation**: The ith element of the list is city j from the **remaining** cities, unvisited so far
  - **One-point-crossover**
- **Path representation**
  - With two cut points
    - **Partially-mapped crossover (PMX)**
    - **Order crossover (OX)**
  - **Cycle crossover (CX)**: Preserves the absolute position of elements in parent sequence
- **Edge recombination** based on the **precedence binary matrix** M ($m_{ij} = 1 \leftrightarrow$ city i before city j): **intersection and union operators**

Constraints in GAs

In the simplest case, constraints occur as well-defined intervals for design parameters.

Methods for handling constraints in GAs:

- Reject individuals that violate constraints (infeasible individuals).
- Repair infeasible individuals.
- Penalize infeasible individuals.
- Incorporate constraints in the representation.

Recommended reading

Z. Michalewicz & D.B. Fogel
How to Solve It: Modern Heuristics

Chapters 1-7, 9

S. Russell and P. Norvig: Artificial Intelligence, A Modern Approach

Section 4.1