Heuristic Optimisation

Part 3: Classification of algorithms. Exhaustive search

Sándor Zoltán Németh

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

s.nemeth@bham.ac.uk

University of Birmingham

Overview

- Classification of classic algorithms
- Algorithms on complete solutions
- Algorithms on partial solutions
- Exhaustive search
- Examples: SAT, TSP, NLP

Classification of classic algorithms

Classic optimisation methods can be very effective if appropriate to the task

- Algorithms that only evaluate complete solutions
- Algorithms that evaluate partially constructed or approximate solutions

Algorithms on complete solutions

All decision variables are specified

- SAT: a binary string of length n
- TSP: a permutation of n cities
- NLP: a vector of n real numbers

If the algorithm is stopped at any time, we do have a solution

Two complete solutions can be compared easily

Algorithms on complete solutions cont'd

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.

Examples:

- exhaustive search
- local search
- hill-climbing
- gradient-based optimisation methods

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

Examples: greedy search, divide and conquer, A* algorithm, dynamic programming, (branch and bound)

Incomplete solution to the original problem

- SAT: all binary strings of length n that start with 11
- TSP: all permutations that start with $7 - 5 - 11$
- NLP: all solutions having $x_2 = 3.4567$

Complete solution to a reduced problem

- SAT: if the formula is in conjunctive normal form, try the conjuncts
- TSP: consider only $k < n$ cities and try to find the shortest route $i \to j$ passing through all k cities
- NLP: limit the domain of some variables

Problems with partial solutions

A way for organising the subspaces of the search space to assure effective search has to be devised

Tree?

Depends on representation

A new evaluation function is needed that can assess the quality of partial solutions

Choosing the representation

SAT:
Vectors of real numbers in the range $[-1, 1]$

$x_i \geq 0 \Rightarrow \mathit{TRUE}$

$x_i < 0 \Rightarrow \mathit{FALSE}$

NLP:
Binary strings of length kn, where each variable $x_i \in [l_i, u_i]$ is encoded as a binary string of length k

$$(< b_{k-1} \ldots b_0 >)_2 = (\sum_{j=0}^{k-1} b_j \times 2^j)_{10} = x'$$

$$x_i = l_i + x' \times \frac{u_i - l_i}{2^k - 1}$$

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

Really exhausting

TSP for 50 cities has search space size 10^{62}

But the algorithm is very simple!

Exhaustive search cont’d

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

Backtracking reduces the amount of work:

If after assigning values to some variables it turns out that there is no solution that has those values, we do not continue assigning values to the remaining variables but we go back one step and try a different option

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

Enumerating the SAT

We have to generate all binary strings of length n, from $< 00\ldots0 >$ to $< 11\ldots1 >$

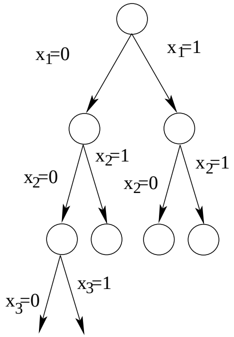
Easy: we just have to generate all the integers from 0 to $2^n - 1$ and convert them into the matching binary string

Evaluation function:

- 1 if the Boolean statement is satisfied
- 0 otherwise

Partitioning the search space for SAT

Into a tree:



Depth-first search:

1. Visit current node
2. For each child of current node perform depth-first search

Enumerating the TSP

How to generate all permutations of $1, 2, \ldots, n$?

1. Fix 1 in the first position
2. Generate all permutations of numbers $2, \ldots n$ for positions $2, \ldots, n$
3. Fix 1 in the second position
4. Now list the permutations found in step 2. in positions $1, 3, 4, \ldots, n$
5. Repeat with 1 fixed in all positions.

Enumerating the TSP cont'd

More suitable for the TSP:
Enumerate by successively exchanging adjacent elements

Then the cost of evaluating the next solution is reduced to calculating the effect of the change on the previous solution

But how can we deal with the situation when not all pairs of cities are connected?

Enumerating the NLP

There is an infinite number of alternatives

We can divide the continuous domain for each variable into a finite number of intervals

$\Rightarrow$ The search space becomes a set of cells

We can evaluate each cell as one point (a corner or the center)

With the best cell found, the whole procedure can be repeated

Problems with enumerating NLP

- Fine granularity – a large number of small cells

  Impractical

- Coarse granularity – fewer large cells

  The possibility of missing the best solution increases

Recommended reading

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

Chapter 3. Traditional methods - Part 1, pp. 54-64