Implementation Report: HyperOpt

Authors: E.K. Burke, P. Cowling and R. Keuthen
Contact:
Ralf Keuthen
University of Nottingham
School of Computer Science & IT
Jubilee Campus
Nottingham NG8 1BB, UK
email: rxk@cs.nott.ac.uk

This report describes the implementation of a heuristic approach, called **HyperOpt**, represents a local improvement algorithm inspired by a practical problem arising in electronics manufacture [?][?]. This approach has so far only been applied to this problem and proved to be able to deal with the complicated side constraints which arise. However, the basic idea behind this approach should be applicable to a wider range of sequencing and combinatorial optimization problems. The heuristic successively breaks the original problem into small subproblems of similar properties. These subproblems are then solved using exact algorithms and the optimal solution is re-embedded into the original problem.

Basic Concept

The basic concept of the h-HyperOpt approach for the TSP is as follows. Starting with an initial tour, first, two non-intersecting hyperedges are defined by the cities p_0 and q_0 and their h successors. The edges within each of these hyperedges are then removed from the tour while the starting and ending cities of each hyperedge stay attached to the tour. This leaves the inner cities of both hyperedges unconnected to the rest of the tour. These inner cities are then reconnected with the tour optimally. Obtaining the optimal reconnection involves finding an optimal tour of the small Travelling Salesman Problem induced by the cities $p_0, p_1, \ldots, p_h$ and $q_0, q_1, \ldots, q_h$ where the two edges between p_h, q_0 and q_h, p_0 have been fixed.

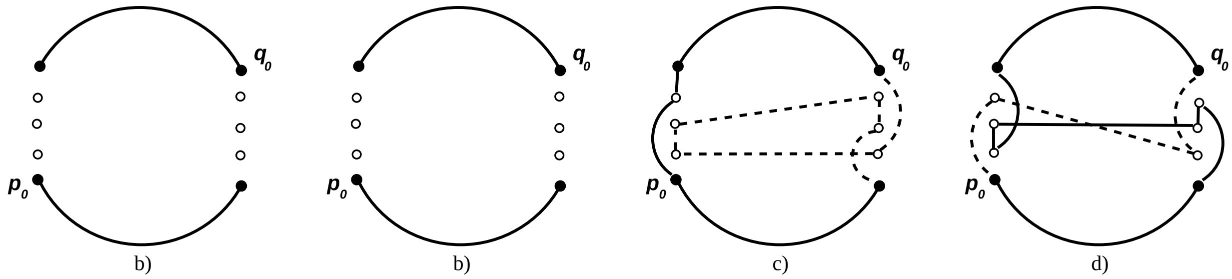

Figure 1: Schema of HyperOpt moves

This basic concept and two possible outcomes of a $4-$HyperOpt move are illustrated in fig. 1. The optimal reconnection process is solved by enumeration in the case of $h = 2$ and for $h \geq 3$ by dynamic programming. For the enumeration approach, the number of different tours which are to be evaluated for a single h-HyperOpt move is $2 \cdot (2h-1)!$. However, this complexity can be reduced substantially for larger h using dynamic programming, which evaluates a single move in $2^{2h-1}(2h-1)^2$ iterations. Computational results are included for the computationally still tractable cases of h in the range between 2 and 4. Further information concerning the heuristic can be found in the technical reports [?][?]. In common with 2-opt, there are $O(n^2)$ h-hyperopt moves of a TSP tour on n vertices, and each move evaluation takes an amount of time which does not depent upon n.

As initial tour to start HyperOpt from, we have implemented the *Multiple Fragment* (sometimes referred to as *Greedy*) algorithm as described in [?].

Implementation Details

Search strategy: One hyperedge is successively fixed and for this hyperedge a set of suitable candidates (see section on candidate sets below) is searched for an improving move. If more than one improving move is found, then the move gaining the maximum improvement is chosen. As in [?], hyperedges affected by a previous hypermove are stored on a stack. Only hyperedges which lie on the stack are considered for move evaluation in subsequent iterations. This approach is described in more detail in [?].

Rejection of unpromising moves: As move evaluation is expensive, requiring an enumerative approach, we have tested a number of rejection approaches based on the length of initial hyperedges. In this implementation we have chosen the Variant2 rejection described in [?][?]. Here a move between hyperedges H_1 and H_2 is rejected without further evaluation if the distance between the starting city of H_1 and the starting city of H_2 is greater than the length of fixed hyperedge H_1 multiplied by some constant α. This rejection approach proved to be very successful for the small value of parameter h we are considering here, although possible improving moves may not be evaluated. The parameter value of α was fixed at $\alpha = 1.20$. Experimental evidence to indicate that this is a good value is presented in [?].

Candidate Sets: The candidate set approach we have implemented works as follows. For each of the first h cities of the fixed hyperedge H_1, i.e. $(p_0, p_1, \ldots, p_{h-1})$, a set of 40 candidates will be searched and the predecessor in the current tour of each of this them considered as starting city q_0 of hyperedge H_2. This means that once hyperedge H_1 is fixed at most $40h$ hyperedges are considered as H_2. However, assuming we have a tour of reasonable quality, the overall set of neighbours should typically contain far less distinct hyperedges as a city of the nearest neighbour set of p_i is very likely to also appear in the nearest neighbour set of $(p_{i+1}, \ldots, p_h)$. The re-evaluation of a move which has already been considered is strictly avoided.

For each city a set of 40 nearest neighbour candidates is determined and stored in a list before the heuristic search takes place. For non-Euclidean instances the candidate set consists of the 40 nearest neighbours for each city. In the case of Euclidean instances we have chosen 40 quadrant nearest neighbours as candidates which are determined using k-dimensional search trees [?][?].

Possible improvements, which have not been implemented: For reasons of simplicity, the tour data structure has been implemented as an array. This slows down our implementation considerably for larger instances. Since many HyperOpt moves involve a 2-opt flip operation, which in the worst case is $O(N)$ using the array representation, this operation starts to dominate 2-HyperOpt for problems of more than 100000 cities. More sophisticated tour data structures like the *splay tree* or *two-level tree* representations, as described by Fredman, Johnson, McGeoch and Ostheimer in [?], could be used to overcome this problem.

A second approach to reduce computation time often mentioned in the literature [?] which has not been considered in our implementation is the cacheing of distances. Particularly 2-HyperOpt, our fastest approach, spends a large portion of its overall computing time on the evaluation of distances between cities. However, cacheing of distances is rather complex for HyperOpt, as each move evaluation involves $2h + 2$ cities.

Technical details:

The heuristics described above have all been coded in GNU *C++* using the `-O2` compiler option running on a Linux 2.2.13 kernel. The calculations were carried out on a Pentium II 400 MHz PC with 256 Mb memory. Computational results for the test battery of up to 3 million cities are included in the file `TestSet`. The code was written by Peter Cowling and Ralf Keuthen.

Computational Results:

Computational results for various HyperOpt approaches are included in the the following files:
2-**HyperOpt:** All Euclidean instances of up to 1 million cities.
File: `Results_BurkeCowlingKeuthen_HyperOpt2`.
3-**HyperOpt:** All Euclidean instances of up to 1 million cities.
File: `Results_BurkeCowlingKeuthen_HyperOpt3`.
4-**HyperOpt:** All Euclidean instances of up to 316,228 cities.
File: `Results_BurkeCowlingKeuthen_HyperOpt4`.