Improving the Held and Karp Bound with Constraint Programming

Pascal Benchimol[1], Jean-Charles Régin[2],
Louis-Martin Rousseau[1], Michel Rueher[2], Willem-Jan van Hoeve[3]

[1] CIRRELT,École Polytechnique de Montréal, Montréal,
C.P. 6128, succ. Centre-ville, Montreal, H3C 3J7,Canada
{pascal.benchimol,louis-martin.rousseau}@polymtl.ca
[2] Universit de Nice - Sophia Antipolis / CNRS
I3S, 2000, route des Lucioles - Les Algorithmes
BP 121 - 06903 Sophia Antipolis Cedex - France
jean-charles.regin@unice.fr,rueher@polytech.unice.fr
[3] Tepper School of Business, Carnegie Mellon University,
5000 Forbes Avenue, Pittsburgh, PA 15213,USA
vanhoeve@andrew.cmu.edu

Abstract. The Held and Karp bound is a classical relaxation of the well known travelling salesman problem. The general idea of the approach is to apply Lagrangian relaxation to the degree constraints of the TSP (i.e., allowing node degrees to be different than 2). Through the use of branch and bound approaches, this technique can solve different TSP instances to optimality. In this paper we demonstrate that the use of Constraint Programming techniques such as specialized filtering can significantly boost the performance of Held and Karp based Branch and Bound methods.

1 Introduction

Held and Karp have proposed, in the early 1970s, a relaxation for the Travelling Salesman Problem (TSP) as well as a branch-and-bound procedure that can solve small to modest-size instances to optimality [**?** **?**]. It has been shown that Held-Karp relaxation produces very tight bounds in practice [**?**], and this relaxation is therefore applied in TSP solvers such as Concorde [**?**]. In this short paper we show that the Held-Karp approach can benefit from well-known techniques in Constraint Programming (CP) such as domain filtering and constraint propagation. Namely, we show that filtering algorithms developed for the weighted spanning tree constraint [**?** **?**] can be adapted to the context of the Held and Karp procedure. In addition to the adaptation of existing algorithms, we introduce a special-purpose filtering algorithm based on the underlying mechanisms used in Prim's algorithm [**?**]. Finally, we explored two different branching schemes to close the integrality gap. Our initial experimental results indicate that the addition of the CP techniques to the Held-Karp method can be very effective.

The paper is organised as follows: section 2 describes the Held-Karp approach while section 3 gives some insights on the Constraint Programming techniques used, as well as the braching schemes. In section 4 we demonstrate, through preliminary experiments, the impact of using CP in combination with Held and Karp based branch-and-bound on small to modest-size instances from the TSPlib.

2 The Held-Karp Approach

Letting $G = (V, E)$ where $|V| = n$ and $|E| = m$ be the graph under consideration, where $c_e, e \in E$ is the cost of each edge, $\delta(i)$ contains all the edges adjacent to i, and $(S, \bar{S})$ contains all edges (i, j) such that $i \in S$ and $j \in \bar{S}$. The linear formulation of the TSP can written as:

$$\min \sum_{e \in E) c_e} x_e \tag{1}$$

$$s.t. \sum_{e \in \delta(i)} x_e = 2 \; \forall i \in V \tag{2}$$

$$\sum_{(i,j) \in (S, N \setminus S)} x_{(i,j)} \geq 1 \; \forall S \subset N \tag{3}$$

$$x_e \in \{0, 1\} \; \forall e \in E \tag{4}$$

The iterative approach proposed by **? ?**], uses Lagrangian relaxation to produce a sequence of connected graphs which increasingly resemble tours. To do so, lets relaxe constraints (2) into the objective to obtain the following model:

$$\min \sum_{e \in E) c_e} x_e + \pi_i (2 - \sum_{e \in \delta(i)} x_e) \tag{5}$$

$$s.t. \sum_{(i,j) \in (S, N \setminus S)} x_{(i,j)} \geq 1 \; \forall S \subset N \tag{6}$$

$$x_e \in \{0, 1\} \; \forall e \in E \tag{7}$$

Where constraints (6)-(7) define the structure of a covering tree. The original idea of Held and Karp was to use a structure defined as a minimum 1-tree, which is a minimum spanning tree (MST) on the vertices 2, 3,..., n together with the two lowest cost edges incident with city 1. In such case, if a solution vector x^* to the minimum 1-tree represents a tour, the penalty component of (5) is null. Given the optimal π^* vector the linear relaxation of (1)-(4) is equal to the linear relaxation of (5)-(7) and it is known as the Held-Karp bound (HK). To obtain HK, or at least approximated it, one thus need to maximize the value of (5) by searching for the best π vector. This search is normally performed through sub-gradient optimization by iteratively perturbing the set of weights π_i allocated to

the vertices of N and constructing a minimum 1-tree. These weights augment the cost of edges adjacent to nodes that have degrees higher than 2 and reduce of the costs of edges adjacent to leafs.

In the end, the integer model (5)-(7) is solved through branch-and-bound, a techniques that as been widely used on this problem (see [**?**] for a survey). A good upper bound, UB, can be computed easily for this problem with any of the popular heuristics that have been devised for this problem, e.g., [**?**].

3 Improving the Approach Using CP

In this section we describe the different refinements introduced to the original Held-Karp approach [**? ?**], which consist of two filtering procedures based on the weighted Minimum Spanning Tree and one based on the underlying structure of Prim's algorithm.

In the following procedures let T be a minimum 1-tree of G computed with by the Held and Karp relaxation described above. For a subset of edges $S \subseteq E$, we let $w(S)$ denote $\sum_{e \in S} c_e$ and $T(e)$ be the minimum 1-tree where e is forced into T.

3.1 Removing Edges Based on Marginal Costs

The *marginal cost* of an edge e in T is defined as $c'_e = w(T(e)) - w(T)$, that is, the marginal increase of the weight of the minimum 1-tree if e is forced in the 1-tree.

The following algorithm can compute, in $O(mn)$, the marginal costs for edges $e \notin T$. Each non-tree edge $e = (i, j)$ links two nodes i, j, and defines a unique i-j path, say P^e, in T. The replacement cost of (i, j) is defined by $c_e - max(c_a | a \in P^e)$, that is the cost of (i, j) minus the cost of largest edge on the path from i to j in the 1-tree T. Finding P^e can be achieved through DFS in $O(n)$ for all the $O(m)$ edges not in T. If $HK + c'_e$ ¿ UB, then e can be safely removed from E.

3.2 Forcing Edges Based on Replacement Costs

Conversely, it is possible to compute the *replacement cost* of an edge $e \in T$ as the increase the Held-Karp bound would incur if e would be remove from E, which we defined by $c^r_e = w(T \setminus e) - w(T)$.

This computation can be performed for all edges $e \in T$, with the following algorithm: a) set all $c'_e = \infty \; \forall e \in T$ b) for all $e = (i, j) \notin T$ identify the i-j path P^e in T which joins the end-point of e. Update all edges $a \in P^e$ such that $c'_a = min(c'_a, c_e)$. This computation can be performed in $O(mn)$, or, at no extra cost if performed together with the computation of marginal costs. If $HK + c'_e$ ¿ UB, then e is a mandatory edge in T.

3.3 Forcing Edges Based During MST Computation

Lets recall that Prim's algorithm computes the minimum spanning tree in G (which is easily transformed into a 1-tree) in the following manner. Starting from any node i, it first partitions the graph into disjoints subsets $S = \{i\}$ and $\bar{S} = V \setminus i$ and creates an empty tree T. Then it iteratively adds to T the minimum edge $(i, j) \in (S, \bar{S})$, defined as the set of edges where $i \in S$ and $j \in \bar{S}$, and moves j from $\bar{S}$ to S.

Since we are using MST computations as part of a Held-Karp relaxation to the TSP, we know that there should be at least 2 edges in each possible $(S, \bar{S})$ of V (this property defines one of well known subtour elimination constraints of the TSP). Therefore, whenever we encounter a set $(S, \bar{S})$ which has only two edges during the computation of the MST with Prim's algorithm, we can force these edges to be mandatory in T.

3.4 Tuning the Propagation Level

The proposed filtering procedures are quite expansive computationally, therefore it is interesting to investigates the amount of propagation that we wish to impose during the search. A first implementation consists in calling each filtering (defined in section 3.1, 3.2 and 3.3) only once before choosing a new branching variable. A second approach would be to repeat these rounds of propagation until none of these procedures is able to delete nor force any edge, that is reaching a fix point. Finally, if reaching a fix point allows to reduced the overall search effort, a more efficient propagation mechanism could be develop in order to speed up its computation.

3.5 Choosing the Branching Criterion

Once the initial Held-Karp bound as been computed and the filtering has been performed it is necessary to apply a branching procedure in order to identify the optimal TSP solution. We have investigated two opposite branching schemes, both based on the 1-tree associated to the best Held-Karp bound, say T. These strategies consist in selecting, at each branch-and-bound node, one edge e and splitting the search in two subproblems, one where e is forced in the solution and one where it is forbidden. In the strategy *out* we pick $e \in T$ and first branch on the subproblem where it is forbidden while in the strategy *in* we choose $e \notin T$ and first try to force it in the solution.

Since there are $O(n)$ edges in T and $O(n^2)$ edges not in T, the first strategy will tend to create search trees which are narrower but also deeper than the second one. However, since the quality of the HK improves rapidly as we go down the search tree, it is generally possible to cut uninteresting braches before we get to deep. Preliminary experiments, not reported here, have confirme that strategy *out* is generally more effective than strategy *in*.

4 Experimental Results

To evaluate the benefits of using CP within the Held-Karp branch-and-bound algorithm, we ran experiments on several small instances of the TSPlib. We report both the number of branching nodes and CPU time required solve each instance, with different propagation level. To eliminate the impact of the upper bound can have on search tree, we ran these experiments using the optimal value of each instance as its UB.

Table 1 clearly shows the impact filtering techniques can have on the original Held-Karp algorithm. In fact the reduction of the graph not only considerably reduces the search effort (BnB nodes) but also sufficiently accelerates the computation of 1-trees inside the Held-Karp relaxation to completely absorb the extra computations required by the filtering mechanisms. This can be seen as the proportional reduction in CPU times largely exceeds the reduction in search node.

Finally we cannot conclude that the extra effort required to reach the fix point is worthwhile, as it is sometimes better and sometimes worst than a single round of filtering. Results on these preliminary tests seem to show that more than one round of computation is most often useless as the first round of filtering was sufficient to reach the fix point about 99.5% of the search nodes. More tests are thus required before investigating more sophisticated propagation mechanisms.

5 Conclusion

We have shown in this paper that Constraint Programming can significantly boost the performance of a very well know OR approach to the TSP, the Held-Karp algorithm. The three filtering techniques proposed allow to considerably reduce the search effort both in terms of explored nodes and CPU times.

	OriginalHK		1-round		Fix point	
	time	B&B	time	B&B	time	B&B
burma14	0.1	28	0	0	0	0
ulysses16	0.16	32	0	0	0	0
gr17	0.14	34	0	0	0.01	0
gr21	0.16	42	0	0	0.01	0
ulysses22	0.19	0	0	0	0.01	0
gr24	0.23	44	0.01	0	0.03	0
fri26	0.36	48	0.01	2	0.01	2
bayg29	0.35	54	0.04	6	0.07	6
bays29	0.33	88	0.05	10	0.1	10
dantzig42	0.65	92	0.09	4	0.17	4
swiss42	0.79	112	0.09	8	0.09	8
att48	1.7	140	0.21	18	0.23	15
gr48	94	13554	5.18	2481	7.38	3661
hk48	1.37	94	0.17	4	0.16	4
eil51	15.9	2440	0.39	131	0.84	426
berlin52	0.63	80	0.02	0	0.02	0
brazil58	13	878	1.09	319	1.02	296
st70	236	13418	1.21	183	1.1	152
eil76	15	596	1.03	125	0.88	99
rat99	134	2510	5.44	592	4.88	502
kroD100	16500	206416	11	7236	50.83	4842
rd100	67	782	0.76	0	0.73	0
eil101	187	3692	8.17	1039	9.59	1236
lin105	31	204	1.81	4	1.85	4
pr107	41	442	4.65	45	4.49	48

Table 1. Results on TSPlib instances