Reformulation techniques for a class of permutation problems

Toni Mancini

Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Via Salaria 113, I-00198 Roma, Italy
tmancini@dis.uniroma1.it

Introduction

State-of-the-art systems and languages for constraint modelling and programming (e.g., AMPL [4] and OPL [6]) clearly separate the *specification* of a problem from its *instance*, some of them (e.g., AMPL) allowing the user to choose *a posteriori* one out of several solvers. Others (e.g., OPL) go one step further, by automatically choosing the most appropriate solver for a problem, thus offering a limited form of *reasoning* on the spec.

We aim to a more ambitious long-term goal, i.e., to *automatically reformulate* the spec, to improve the computation process efficiency. We get inspiration from the relational database technology, since it is well-known that reformulating queries –independently on the database– may result in greater efficiency. Reformulation is a difficult task in general: a spec is essentially a formula in second-order logic, and the equivalence problem is undecidable already in the first-order case [1]. For this reason, our current research focuses on restricted forms of reformulation, and, more specifically, on selecting constraints that can be safely "delayed", and solved afterwards. Intuitively, this offers several advantages, since the instantiation phase will be faster (delayed constraints are not taken into account), and solving the simplified problem might be easier; removing constraints makes the set of solutions larger, i.e., for each instance:

$$\{\text{sols of orig. problem}\} \subseteq \{\text{sols of reform. problem}\}.$$

Finally, ad hoc efficient methods for solving delayed constraints may exist.

The goal of our current research is to understand in which cases a constraint can be delayed. In previous work [2], we gave a characterization of such constraints wrt a syntactic criterion on the spec, obtaining a mechanism for the automated reformulation of a spec applicable to a great variety of problems, including *functional* ones.

In this paper we focus on *permutation problems*, and more particularly on the subclass characterized by constraints that bind an element of the permutation to the next or previous one. Problems in this class arise frequently both in theory and in practice: NP-complete *Hamiltonian circuit* (HC) and *Flow-shop scheduling* problems, are good examples.

In the following, we use *existential second-order logic* (ESO) with equality as modelling language, which allows to represent all search problems in the complexity class NP [3]. We believe that studying this formal but simplified scenario (as opposed to richer languages provided by state-of-the-art systems) is a mandatory starting point for more complex investigations, and that our results can serve as a basis for reformulating specs written in higher-level languages. Coherently with all state-of-the-art systems, we represent instances by means of *relational databases*, with *uninterpreted* constants.

The Hamiltonian circuit problem and its reformulations

A permutation of the tuples in a (monadic) relation R can be viewed as a new (binary) relation P with components in R defining a linear order among its tuples. To simplify formulae, we allow for a limited use of integers, i.e., pre-interpreted symbols belonging to an instance-dependent range. In this simplified scenario, a permutation of the n tuples in R is a mathematical relation π from the set of the n tuples to the integer range $[1, n]$, which has to satisfy the following constraints: *(i) Total*, i.e., it has to be defined for every tuple in R; *(ii) Surjective*, i.e., every integer in $[1, n]$ has to be in the image of π; *(iii) Monodrome*, i.e., every tuple in R is mapped into at most one value; *(iv) Injective*, i.e., every integer in $[1, n]$ is the image of at most one tuple in R. These constraints force π to be a *bijective function* from the set of n tuples to the range $[1, n]$. Moreover, under the hypotesis that $|dom(\pi)| = |codom(\pi)| = n$, every set of three of the above constraints entails the fourth one. In particular, Total, Monodrome and Injective entail Surjective. For this reason, in what follows, we do not further consider the Surjective constraint.

In the following, we will refer to the HC problem only, but reformulation techniques we propose can be applied to any problem in the class defined in the previous section. The HC problem can be stated in the following way:

> Given a graph as input, find a permutation of the nodes, s.t. every node is linked to its successor, and the last to the first one (wrt the order induced by the permutation)

and can be specified in ESO as follows:

$$\begin{aligned}
&\exists P\ \forall v \exists i\ P(v,i) \wedge && \text{(Total)}\\
&\quad \forall v, i, i'\ P(v,i) \wedge P(v,i') \rightarrow i = i' \wedge && \text{(Monodrome)}\\
&\quad \forall v, v', i\ P(v,i) \wedge P(v',i) \rightarrow v = v' \wedge && \text{(Injective)}\\
&\quad \forall v, v', i\ P(v,i) \wedge P(v',i+1) \rightarrow edge(v,v') \wedge && \text{(Good1)}\\
&\quad \forall v, i\ P(v,i) \wedge \left(\neg \exists v' P(v',i+1)\right) \rightarrow \forall v''\ P(v'',1) \rightarrow edge(v,v''). && \text{(Good2)}
\end{aligned}$$

(Good1) and (Good2) impose the existence of an edge from any node to its successor, and from the last to the first one. Fig. 1(a) shows an instance and a solution of the HC problem.

Suppose now to ignore the constraint (Injective). As a result, relation P can map several nodes into the same integer, thus inducing only a partial order

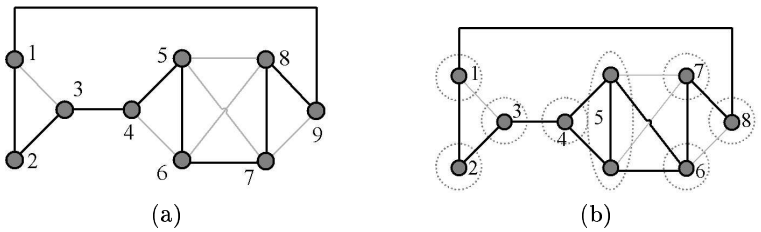

Fig. 1. (a) An instance and a solution for HC. (b) A solution of Reformulation 1 for the same instance, subsuming the one in (a).

on the graph nodes. We say that nodes have been divided into *clusters*. As an example, Fig. 1(b) shows a situation in which the two nodes in cluster 5 can be visited in any order, one of them leading to the solution in Fig. 1(a). More formally, the problem can be *abstracted* from the level of nodes to the level of clusters. (Good1) and (Good2) are still necessary, but have different meaning: the former says that every cluster (i.e., every node of every cluster) must be linked to the following one (i.e., to every of its nodes); the latter, instead, says that the last cluster must be linked (in the same manner) to the first. If we now add the following constraint:

$$\forall v, v', i \quad P(v,i) \wedge P(v',i) \rightarrow v = v' \vee edge(v,v') \qquad \text{(Cliques)}$$

imposing that clusters are cliques, we obtain a possible *reformulation* of the original problem, since a solution of the former can be translated (in $O(n)$ time) to a solution of the latter (via a post-processing stage), by choosing an arbitrary order of nodes in the same cluster. It has to be observed that (Cliques) can be viewed as an extension of (Good1) and (Good2) on a cluster; moreover, the post-processing stage of the computation is the same as enforcing the *delayed* constraint (Injective), i.e., forcing the partial order on nodes to be total.

The HC problem can be reformulated as follows (**Reformulation 1**):

$$\begin{aligned}
&\exists P^* \ \text{Total} \wedge \text{Monodrome} \wedge \text{Cliques} \wedge \\
&\quad \forall v, v', i \quad P^*(v,i) \wedge P^*(v',i+1) \rightarrow edge(v,v') \ \wedge && \text{(Good1)} \\
&\quad \forall v, i \quad P^*(v,i) \wedge \big(\neg \exists v' \ P^*(v',i+1)\big) \rightarrow \\
&\qquad \big(\forall v'' \ P^*(v'',1) \rightarrow edge(v,v'')\big) \ \wedge && \text{(Good2)} \\
&\quad \forall i \quad i > 1 \rightarrow \exists v \ P^*(v,i) \rightarrow \exists v' \ P^*(v',i-1). && \text{(Compactness)}
\end{aligned}$$

Constraints (Total) and (Monodrome) partition the graph nodes into clusters, opportunely linked by edges (Good1, Good2) to allow the traversal of the whole graph. Since clusters are cliques, paths linking all nodes in them surely exist. Of course, (Good1) and (Good2) should be slightly modified in that "$i+1$" and "$i-1$" should be read with modulo k semantics, where $k \leq n$ is the number of clusters. Moreover, they should be read as *the successor* and *the predecessor* cluster wrt i, since the sequence of cluster numbers can have "holes" (i.e., P can map nodes to, e.g., clusters 1 and 3, but not to 2). To deal with these issues in a

simple way, we added a new constraint (Compactness) saying that the sequence of cluster numbers does not have holes. It is worth noting that this constraint preserves satisfiability and can be removed by a more complex handling of cluster numbers. Fig. 1(b) shows a solution of the reformulated problem for the same instance in Fig. 1(a), which leads to a set of solutions of the original one.

A major drawback of this strategy can be easily observed: constraint (Cliques) allows for the abstraction from the level of nodes to the level of clusters, since every path linking nodes in a cluster is a good sub-path of an HC of the whole graph. But the price for this is very high, since (Good1) and (Good2) impose *every* node of a cluster to be linked to *every* node of the next.

A first optimization is to force every node of a cluster to be linked to *at least one* (and not to every) node of the next one. This yet guarantees that every permutation of nodes in a cluster is a good sub-path of the whole circuit, and leads to **Reformulation 2**, whose formal spec is omitted, due to lack of space (cf. Fig. 2(a)). We can make this intuition much more reasonable, observing that what we really want is to go from one cluster to the next without touching a node twice. So, if we start traversing cluster i from node v, we need an edge from a node $v' \neq v$ in cluster i to cluster $i+1$ (if present), or cluster 1. This allows us to choose a path for cluster i starting from v and ending in v'. Such a path surely exists, since cluster i is a clique (cf. Fig. 2(a)). This (simplified) constraint can be formalized in the following way: each cluster i needs to have at least two distinct nodes, v^i_{in} and v^i_{out}, such that: *(i)* An edge exists from $v^{(i-1)}_{out}$ to v^i_{in} (if $i = 1$, the edge has to start from v^k_{out}, where k is the cluster having maximal number); *(ii)* An edge exists from v^i_{out} to $v^{(i+1)}_{in}$ (if cluster $i+1$ does not exist, the edge has to end in v^1_{in}). The whole spec for **Reformulation 3** is the following:

$$
\begin{aligned}
\exists P^* \;\; & \text{Total} \wedge \text{Monodrome} \wedge \text{Cliques} \wedge \text{Compactness} \wedge \\
& \forall i \; \exists v^i_{in}, v^i_{out} \;\; P^*(v^i_{in}, i) \wedge P^*(v^i_{out}, i) \wedge v^i_{in} \neq v^i_{out} \;\; \wedge \\
& \quad (i{>}1) \rightarrow \exists v^{i-1}_{out} \;\; P^*(v^{i-1}_{out}, i-1) \wedge edge(v^{i-1}_{out}, v^i_{in}) \;\; \wedge \\
& \quad (i{=}1) \rightarrow \forall i' \; last(P^*, i') \rightarrow \exists v^{i'}_{out} P^*(v^{i'}_{out}, i') \wedge edge(v^{i'}_{out}, v^i_{in}) \;\; \wedge \qquad \text{(Good)}_3 \\
& \quad \neg last(P^*, i) \rightarrow \exists v^{i+1}_{in} \;\; P^*(v^{i+1}_{in}, i+1) \wedge edge(v^i_{out}, v^{i+1}_{in}) \wedge \\
& \quad last(P^*, i) \rightarrow \exists v^1_{in} \;\; P^*(v^1_{in}, 1) \wedge edge(v^i_{out}, v^1_{in})
\end{aligned}
$$

where $last(P^*, i) \doteq \exists v^i \;\; P^*(v, i) \;\; \wedge \neg \exists v^{i+1} \;\; P^*(v^{i+1}, i+1)$, i.e., cluster i is the last one wrt the order given by P^*.

We remark that, according to our goal, every solution of the reformulated problem corresponds to a *set* of solutions of the original one: in particular, the solution given in Fig. 1(a) is one of them. In general, for any instance, the following inclusion relationship holds:

$$\{\textit{Sols of orig. spec}\} \subseteq \{\textit{Sols of Ref. 1}\} \subseteq \{\textit{Sols of Ref. 2}\} \subseteq \{\textit{Sols of Ref. 3}\}.$$

Having a larger set of solutions, the reformulated spec is likely to be more efficiently solvable.

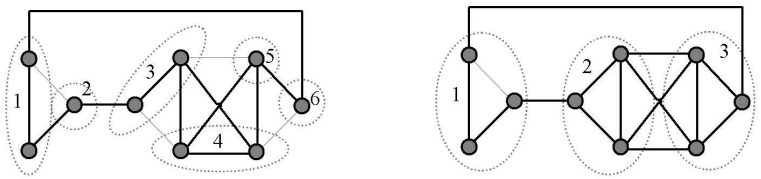

Fig. 2. A solution of Reformulation 2 (a) and 3 (b) for the instance of Fig. 1(b).

Experimentation. We made a preliminary experimentation of our reformulation techniques on HC (random instances between 16 and 24 nodes). Instantiaton was made using ad hoc programs, thus obtaining SAT instances. As for the solver, we used the DPLL-based SAT system SATZ [5]. Consistent time savings (from several minutes to few seconds) have been obtained for negative instances. On the other hand, time for solving positive instances was typically very low, both for the original and the reformulated specs, and so highly influenced by translation time. A more exaustive experimentation is planned as future work.

Conclusions. Few methodological comments are mandatory: it is well-known that HC instances can be efficiently solved either by means of ad hoc algorithms or by using CP languages in a sophisticated way. Our purpose here is not to propose an efficient method for problem solving, but rather to prove that we can achieve a consistent speed-up by means of a mere reformulation of a purely declarative spec. So, the experimentation should be considered only as a preliminary stage to test whether the reformulation approach we propose is promising or not. Finally, it is worth noting that, even if the above reformulations seem quite complex, they are absolutely *syntax-based*, and therefore can eventually be performed *automatically*, before any other optimization strategy.

Acknolegments. The author is grateful to his advisor, Prof. Marco Cadoli, for guiding, supporting and encouraging its research activity.

References

1. Egon Börger, Erich Gräedel, and Yuri Gurevich. *The Classical Decision Problem.* Perspectives in Mathematical Logic. Springer, 1997.
2. Marco Cadoli and Toni Mancini. Towards automated reformulation of specifications. Submitted, 2003.
3. R. Fagin. Generalized First-Order Spectra and Polynomial-Time Recognizable Sets. In R. M. Karp, editor, *Complexity of Computation*, pages 43–74. AMS, 1974.
4. Robert Fourer, David M. Gay, and Brian W. Kernigham. *AMPL: A Modeling Language for Mathematical Programming.* International Thomson Publishing, 1993.
5. C. M. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proc. of IJCAI'97*, pages 366–371, 1997.
6. Pascal Van Hentenryck. *The OPL Optimization Programming Language.* The MIT Press, 1999.