Declarative Local Search for Predicate Logic

Tu-San Pham[1(✉)], Jo Devriendt[2], and Patrick De Causmaecker[1]

[1] KU Leuven, Leuven, Belgium
san.pham@kuleuven.be
[2] KTH Royal Institute of Technology, Stockholm, Sweden

Abstract. In this paper we introduce a framework built on top of the Knowledge Base System IDP, which allows local search heuristics to be synthesized from their formal descriptions. It is introduced as a new inference to solve *optimization* problems in IDP. To model a local search heuristic, users need to specify its components, among which *neighbourhood moves* are the most important. Two types of neighbourhood moves, namely standard moves and Large Neighbourhood Search moves, are supported. A set of built-in local search heuristics are provided, allowing users to combine neighbourhoods in different ways. We demonstrate how the new local search inference can be used to complement the existing solving mechanisms for logic programming.

Keywords: Heuristics · Local search · Knowledge representation · Predicate logic

1 Introduction

IDP (*Imperative-Declarative Programming* [2]) is a Knowledge Base System (KBS) which consists of two main components: (i) a formal declarative language that allows describing domain knowledge (as a *knowledge base*); and (ii) a set of inference methods that allows solving a wide variety of tasks around a knowledge base. Its language FO(·) is based on classical first-order logic (FO), extended with inductive definitions, types, aggregates and arithmetics. In this paper, we focus on IDP's ability to solve combinatorial optimization problems, which is provided through the inference method *optimization* using MiniSAT(ID) [3] as the backend engine. As a CP-SAT-based solver, it shows limited performance on many optimization problems, such as the assignment problem [4], or real-world problems with large-sized instances. In the field of operational research, *local search heuristics* have shown their ability to solve such problems successfully.

In this work, we introduce *declarative local search*, a framework that allows specifying local search heuristics declaratively in IDP. Local search is provided as a new inference method, serving as an alternative to solve optimization problems. To use the inference, beside a problem's modelling, users need to specify necessary components of a local search heuristic, chief of which are the neighbourhood moves. The initial idea was reported in [12] where only the modelling of a single neighbourhood is supported. In this work, users can specify multiple neighbourhoods and combine them in different ways using a set of built-in heuristics and metaheuristics. Two types of moves are supported, namely standard moves and Large Neighbourhood Search moves. This work is similar in spirit to [1], where neighbourhoods are declaratively modelled in the constraint programming language MiniZinc [11]. The source code of the solver along with all the modellings in this paper and experimental results can be found at [10].

Section 2 shortly introduces IDP, while the modelling of local search heuristics is showcased in Sect. 3. In Sect. 4, we present how IDP is extended with a local search back-end to synthesize local search heuristics. Section 5 concludes the paper.

2 Modelling TSP in FO(·) with IDP

A thorough introduction to IDP and its language FO(·) can be found at dtai.cs.kuleuven.be/software/idp. An IDP specification (or modelling) consists of different components. The four most important components are: *vocabularies* specifying the symbols and types used; *theories* specifying problem constraints; *structures* representing both input data and feasible solutions; and *terms* for objective functions. These are combined and reused through imperative code written in the Lua scripting language [8]. As a running example we employ the Travelling Salesman Problem (TSP), which consists of finding the shortest Hamiltonian cycle of a given graph. A model for the TSP in IDP can be found at goo.gl/TTv85c.

Example 1 (TSP). The four components of the TSP modelling are as follows:

– The vocabulary V specifying the parameters ($Node$, $Distance$, $Depot$), whose values define a problem instance, and the decision variables ($Path$, $Reachable$), whose values define a solution of the problem.
– The theory T built over V, specifying the problem's constraints:

$$\begin{aligned}
&\forall x\colon \exists! y\colon Path(x,y).\\
&\forall x\colon \exists! y\colon Path(y,x).\\
&\{\quad Reachable(Depot).\\
&\quad Reachable(x) \leftarrow \exists y\colon Reachable(y) \land Path(y,x).\}\\
&\forall x\colon Reachable(x).
\end{aligned}$$

The first two lines represent the flow constraints. Line three and four feature an *inductive definition* which defines the *Reachable* predicate, starting from the depot, and inductively adding neighbouring nodes according to the links present in *Path*. The last line then states that all nodes must belong to *Reachable*, forming a subtour elimination constraint.

- The term $\Sigma_{(x,y)\in Path} Distance(x, y)$ represents the total travelling distance and serves as objective function *obj*.
- A (partial) structure S describing parameter values.

3 Modelling Local Search Heuristics

Local search is a heuristic which iteratively applies local changes – known as *(neighbourhood) moves* – on solutions to improve solution quality. In a simple *descent* search, a solution becomes the starting point of a new iteration if it improves the current solution. Descent search ususally ends up in a *local optimum*, which can be quite far away from optimality. *Metaheuristics* are local search-based heuristics, which use some *diversification* techniques to escape from local optima. In this paper, "local search heuristics" indicate both heuristics and metaheuristics.

To model a local search algorithm in our framework, users first need to extend the knowledge base with a moves modelling. These are then passed to IDP as inputs to synthesize the desired local search algorithm – IDP is extended with some "off-the-shelf" (meta) heuristic techniques to combine moves in various ways. The following local search techniques are implemented: *first improvement search*, *best improvement search*, *Tabu search* [7], *Large Neighbourhood Search* [13] and *Iterated Local Search* [9]. Two types of moves are supported: standard moves and Large Neighbourhood Search moves.

3.1 Standard Neighbourhood Moves

To model a standard move, the following information is crucial: (i) how to get valid moves given a solution; (ii) how to compute a neighbour solution given a move and a current solution; (iii) how to evaluate a move. To represent these pieces of information, a user should add the following components to the specification: (1) a vocabulary *Vmove* consisting of functions and predicates representing a move; (2) a query *getValidMoves* built over V, describing how to get valid moves from a given solution; (3) a theory *Tnext* built over *Vnext* containing a definition for a neigbhour solution given a solution and a move, with *Vnext* is the vocabulary representing the neighbour solution; and (4) a query *getDeltaObj* calculating the difference between the objective values of the current solution and the neighbour solution resulting from a move.

Example 2. To illustrate the standard move modelling, we hereby model the *2-opt* move for the TSP, where two edges are removed from the solution and replaced by two new edges (see Fig. 1). To model the 2-opt move, an auxiliary predicate *Before* represents the order of nodes appearing along the solution path.

$$\begin{aligned}
&\{\forall x : Before(x, x). \\
&\ \forall x, y : Before(x, y) && \leftarrow Path(x, y) \land y \neq Depot. \\
&\ \forall x, y : Before(x, y) && \leftarrow \exists z : Path(x, z) \land Before(z, y) \land z \neq Depot.\}
\end{aligned}$$

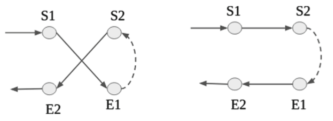

Fig. 1. 2-opt move of the TSP

Vmove consists of 4 constants $S1, E1, S2$ and $E2$ representing the four nodes involved in the 2-opt move. *Vnext* consists of the predicate $next\_Path(Node, Node)$, whose values define a neighbour solution. The 2-opt move replaces two edges $(S1, E1)$, $(S2, E2)$ by $(S1, S2)$, $(E1, E2)$, and reverses the segment from $E1$ to $S2$. This mapping is defined in the theory *Tnext* as below:

$$\begin{aligned}
&\{next\_Path(S1, S2).\\
&\;next\_Path(E1, E2).\\
&\;next\_Path(x, y) && \leftarrow Path(x, y) \wedge Before(y, S1) \wedge y \neq Depot.\\
&\;next\_Path(x, y) && \leftarrow Path(x, y) \wedge Before(E2, x) \wedge E2 \neq Depot.\\
&\;next\_Path(y, x) && \leftarrow Path(x, y) \wedge Before(E1, x) \wedge Before(y, S2) \wedge y \neq Depot.\}
\end{aligned}$$

To complete the modelling, two queries are specified. Query *getDeltaObj* evaluates a move by calculating the difference between the total travelling time of the current solution and its neighbour: $\Delta = d_{S1S2} + d_{E1E2} - (d_{S1E1} + d_{S2E2})$. Query $getValidMoves$ defines valid moves of a given solution, which are the tuples of edges $(S1, E1)$ and $(S2, E2)$ appearing in this order on the solution path.

3.2 LNS Moves

Large Neighbourhood Search (LNS) [13] allows exploring a large neighbourhood of a solution by alternating a *destroy* and a *recreate* phase to gradually improve the objective value. In the destroy phase, a part of the solution is destroyed, resulting in a partial solution which is then repaired in the recreate phase. Destroying the "bad quality" parts of a solution is more likely to lead to a better solution in the recreate phase. Therefore, we support users to model the destroy phase in LNS moves, while the recreate phase is handled by the solver.

To get an intuition on how the destroy phase should be modelled, let us consider an example of the nurse scheduling problem (NSP) where shifts are assigned to nurses, subject to more complex constraints. An example of a meaningful LNS move for the NSP is to destroy the schedules of two, often randomly selected, nurses whose preferences are violated, and then reschedule them in the recreate phase. To model this move, users should be allowed to specify which parts of the solution can be destroyed (e.g. shift assignments to nurses whose preferences are violated).

With that intuition in mind, an LNS move can be modelled in our framework by specifying: (i) *random variables* – symbols which the framework can randomly interpret with (tuples of) values (domain elements); (ii) the set of valid interpretations to the random variables; and (iii) which parts of the solution should be destroyed given the selected values for the random variables. The first piece of information (i) is encoded in a vocabulary *Vmove* while (ii) is given through a query *getRandomVars*. By solving the query, possible options for the values to the random variables are obtained, from which the solver selects randomly. Given the chosen values to the random variables, users then can specify parts of the solution to be destroyed through (iii) the query *getMoves*. Besides a user-defined LNS move, the framework also supports automatic LNS moves, where parts of the solution to be destroyed are chosen randomly. An example of an LNS move modelling of the running example of the TSP is presented below.

Example 3. Each solution of the TSP is a Hamiltonian path that visits all vertices of the graph. Let say users want to destroy a part of this path, from node S to node E, given that S appears before E in the path starting from the depot. Vocabulary *Vmove* then consists of two constants S and E. S and E are random factors, which are chosen at each iteration of the algorithm. Query *getRandomVars* specifies valid values of S and E:

$$\{s, e \mid \mathit{Before}(s, e) \wedge s \neq e \wedge e \neq \mathit{Depot}\}$$

Given the chosen values of S and E, the part to be removed from the solution in the destroy phase of the LNS is the path from S to E, which is encoded in query *getMove*:

$$\{x, y \mid \mathit{Before}(S, x) \wedge \mathit{Before}(y, E)\}$$

4 Metaheuristics Framework

In this section, we explain how the descent first improvement search (FI) with a single standard move is synthesized in IDP. The synthesis of other local search algorithms with standard moves is straightforward given the description of FI while the synthesis of the LNS is straightforward from the description of LNS moves in Sect. 3.2.

Let us first recall the components of the modelling. A problem's modelling consists of a vocabulary V, a theory T, a query *getObjVal* and a term *obj*. Each neighbourhood move modelling consists of a vocabulary *Vmove*, a query *getValidMoves*, a theory *Tnext*, and finally a query *getDeltaObj*. Given an input instance, we let IDP execute its *model expansion* inference to obtain the first feasible solution, which will serve as the initial point of our FI search. At each iteration, a set of valid neighbourhood moves Ω from the current solution s is achieved by solving the query *getValidMoves*, using IDP's *query solving* inference. Each move $\omega \in \Omega$ is then evaluated by solving the query *getDeltaObj* on s_ω, where s_ω is a joined structure between the current solution s and the move ω. If the obtained delta objective improves the solution, the corresponding neighbour solution is created by applying *model expansion* on s_ω over theory *Tnext*, which contains a definition applying the move to the current solution. This neighbouring solution is the starting point of the next iteration, until a stopping criterion is met and the best solution found is returned.

Given this declarative local search framework, a user can easily mix-and-match several modelled neighborhoods and (meta) heuristics. For example, we modelled a set of neighbourhood moves, including three standard moves and 2 LNS moves, for the running example TSP, from which we synthesized no less than 15 local search heuristics [10]: 9 local search configurations based on *first improvement*, *best improvement* and *tabu search* for each standard move; 2 LNS configurations corresponding to two *LNS* moves; and 4 ILS configurations with different combination of simple local search configurations. We also ran a preliminary experiment comparing our framework to two black box approaches: IDP's *minimization* inference and optimization with clingo [6], both in their default settings. These early results demonstrate the fast prototyping potential of the framework: most of our local search heuristics outperformed IDP and clingo, especially two among the four ILS configurations result in a less than 10% average deviation from optimality, which is 7 or 8 times better than the two logic solvers. These results indicate that declarative local search could be a good complement to existing logic solvers.

5 Conclusion

In this paper, we propose a local search framework that synthesizes local search heuristics from their formal, declarative descriptions in predicate logic. Local search is introduced as an alternative back-end for IDP to solve optimization problems. The framework is illustrated by the modelling of local search heuristics for the TSP and some preliminary experiments are conducted.

A thorough experimental analysis of our declarative approach is to be performed in the future. Further work also includes extending the framework to allow more flexibility in metaheuristics modelling. The combination between user-defined neighbourhoods and automatically generated neighbourhoods [5] is also interesting.

Acknowledgements. This research was supported by Swedish Research Council grant 2016-00782, FWO research grant G.0922.13, and KU Leuven project C24/17/012.

References

1. Björdal, G., Flener, P., Pearson, J., Stuckey, P.J., Tack, G.: Declarative local-search neighbourhoods in MiniZinc. In: Tsoukalas, L.H., Grégoire, É., Alamaniotis, M. (eds.) IEEE 30th International Conference on Tools with Artificial Intelligence, ICTAI 2018, 5–7 November 2018, Volos, Greece, pp. 98–105. IEEE (2018). https://doi.org/10.1109/ICTAI.2018.00025

2. De Cat, B., Bogaerts, B., Bruynooghe, M., Janssens, G., Denecker, M.: Predicate logic as a modeling language: the IDP system. In: Declarative Logic Programming, pp. 279–323. Association for Computing Machinery and Morgan & Claypool (2018)
3. De Cat, B., Bogaerts, B., Devriendt, J., Denecker, M.: Model expansion in the presence of function symbols using constraint programming. In: 25th International Conference on Tools with Artificial Intelligence, 4–6 November 2013, USA, pp. 1068–1075 (2013)
4. Devriendt, J.: Exploiting symmetry in model expansion for predicate and propositional logic. Ph.D. thesis, Informatics Section, Department of Computer Science, Faculty of Engineering Science, February 2017
5. Devriendt, J., De Causmaecker, P., Denecker, M.: Transforming constraint programs to input for local search. In: The Fourteenth International Workshop on Constraint Modelling and Reformulation, pp. 1–16 (2015)
6. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Clingo = ASP+ control: preliminary report. arXiv preprint arXiv:1405.3694 (2014)
7. Glover, F., Laguna, M.: Tabu search. In: Du, D.Z., Pardalos, P.M. (eds.) Handbook of Combinatorial Optimization, pp. 2093–2229. Springer, Boston (1998). https://doi.org/10.1007/978-1-4613-0303-9_33
8. Ierusalimschy, R., de Figueiredo, L.H., Celes, W.: Lua - an extensible extension language. Soft.: Pract. Exp. **26**(6), 635–652 (1996)
9. Lourenço, H.R., Martin, O.C., Stützle, T.: Iterated local search. In: Glover, F., Kochenberger, G.A. (eds.) Handbook of Metaheuristics, pp. 320–353. Springer, Boston (2003). https://doi.org/10.1007/0-306-48056-5_11
10. Modelling and instances (2019). https://github.com/tusanpham/DeclarativeLocalSearch
11. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: towards a standard CP modelling language. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 529–543. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74970-7_38
12. Pham, T.-S., Devriendt, J., De Causmaecker, P.: Modelling local search in a knowledge base system. In: Daniele, P., Scrimali, L. (eds.) New Trends in Emerging Complex Real Life Problems. ASS, vol. 1, pp. 415–423. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00473-6_44
13. Shaw, P.: Using constraint programming and local search methods to solve vehicle routing problems. In: Maher, M., Puget, J.-F. (eds.) CP 1998. LNCS, vol. 1520, pp. 417–431. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-49481-2_30