# A Case Study on TSP: What to Optimize and How?

Martin Gebser[0000−0002−8010−4752]

University of Klagenfurt, Universitätsstr. 65-67, 9020 Klagenfurt, Austria
martin.gebser@aau.at

**Abstract.** Since the high-level modeling language of Answer Set Programming (ASP) supports a compact, uniform representation of concepts like recursion, transitivity, reachability, etc., the well-known Traveling Salesperson Problem (TSP) can be conveniently expressed by just a few first-order rules. Such an encoding illustrates all building blocks of the Generate-and-Test modeling pattern with optimization, which qualifies TSP as a comprehensive introductory example to instruct and inspire ASP learners. When turning to the optimization performance of ASP systems like `clingo`, the question what is an elegant encoding appears in a new light, where TSP can give inspiration for ASP experts as well.

## 1 Encoding TSP in ASP

The following TSP instance is inspired by [6], also taking TSP as a modeling example:

```
place(b). link(b,h,2). link(b,l,1). link(b,p,1). % Berlin
place(d). link(d,b,2). link(d,l,2). link(d,p,4). % Dresden
place(h). link(h,b,2). link(h,l,2). link(h,w,3). % Hamburg
place(l). link(l,d,2). link(l,w,3).              % Leipzig
place(p). link(p,b,1). link(p,d,4). link(p,h,3). % Potsdam
place(w). link(w,d,2). link(w,h,3). link(w,l,3). % Wolfsburg
```

The TSP is about finding a round trip that visits each place exactly once, where the sum of connection costs is subject to minimization. For the given instance, there is a unique optimal round trip taking `b` as the starting place: $(b, p, h, l, w, d, b)$ with the sum $1 + 3 + 2 + 3 + 2 + 2 = 13$ of connection costs.

A uniform first-order encoding, structured according to the Generate-and-Test modeling pattern [7], can be written as follows in the language of the `clingo` system [3]:

```
1  % DOMAIN
2  start(X) :- X = #min{Y : place(Y)}.
3  % GENERATE
4  {travel(X,Y) : link(X,Y,C)} = 1 :- place(X).
5  {travel(X,Y) : link(X,Y,C)} = 1 :- place(Y).
6  % DEFINE
7  visit(X) :- start(X).
8  visit(Y) :- visit(X), travel(X,Y).
9  % TEST
10 :- place(X), not visit(X).
11 % OPTIMIZE
12 :~ link(X,Y,C), travel(X,Y). [C,X]
```

The encoding illustrates several crucial modeling features: (1) use of a `#min` aggregate in line 2 to determine a lexicographically smallest starting place among arbitrary place

identifiers, (2) choice rules with cardinality bounds in lines 4-5 that provide sets of connections with exactly one incoming and one outgoing connection per place as solution candidates, (3) positive recursion such that atoms derived by the rules in lines 7-8 yield precisely the places reached from the starting place via connections of a solution candidate, (4) an integrity constraint in line 10 to discard solution candidates for which default negation by `not` exhibits unreached places, and (5) a weak constraint stating in line 12 that the sum of costs over the connections of solutions is subject to minimization.

## 2 A Closer Look at Optimization

The tuple `[C,X]` of the weak constraint in line 12 associates each place `X` with the cost `C` for its outgoing connection. This representation exploits the cardinality bound of the choice rule in line 4, asserting that neither less nor more than one cost is incurred per place. However, the implied condition that $lb = \sum_{\texttt{place}(x)} \min\{c \mid \texttt{link}(x,y,c)\}$ constitutes a tighter lower bound than 0 on the sum of connection costs is not reflected: even if a solution whose sum of connection costs matches $lb$ could be found, an ASP system may need to continue search to eventually prove the solution's optimality. In general, (too) loose lower bounds may make proofs of optimality virtually infeasible.

A penalization scheme such that the difference to $\min\{c \mid \texttt{link}(x,y,c)\}$ is taken as cost for a place $x$, also introduced and empirically studied by [4], looks as follows:

```
12 sort(X,N,C)  :- link(X,Y,C), N = #count{D : link(X,Z,D), D <= C}.
13 gap(X,N,D-C) :- sort(X,N,C), sort(X,N+1,D), link(X,Y,D), travel(X,Y).
14 gap(X,N,D-C) :- sort(X,N,C), sort(X,N+1,D), gap(X,N+1,P).
15 :~ gap(X,N,P). [P,X,N]
```

In view of two gaps, amounting to a difference of 2 or 1, respectively, for the outgoing connections of $p$ to $h$ and $l$ to $w$, the sum 13 of connection costs for the optimal round trip $(b, p, h, l, w, d, b)$ is mapped to just 3, obtained by summing up the two gaps only.

While the reformulated **`OPTIMIZE`** part improves the optimization performance [4], it makes the encoding harder to read and unsuitable for an introductory example. Moreover, the approach to exploit a partition of the atoms occurring in weak constraints along with cardinality bounds on each part is of general relevance and also applied, e.g., by `aspcud` [5]. However, introducing a respective penalization scheme by hand on a per-problem basis is tedious and error-prone, and system support of such rewriting methods would be beneficial [1]. We expect that automated rewriting will be computationally costly and imperfect when applied uninformed at the ground level, e.g., instances like `{travel(`$b$`,`$h$`); travel(`$b$`,`$l$`); travel(`$b$`,`$p$`)} = 1.` and `{travel(`$d$`,`$b$`); travel(`$h$`,`$b$`); travel(`$p$`,`$b$`)} = 1.` of the rules in lines 4-5 refer to both incoming and outgoing connections, so that a partition of `travel`/2 atoms is difficult to reconstruct. First-order rewriting, as supplied by the `ngo` tool, is limited by imperfect information on instance properties, e.g., the unique cost per connection, as well as inherent cardinality bounds for derived predicates. To not give up rewriting ideas altogether, assertions similar to the **`#heuristic`** statements of `clingo` [3] may be the way to go for empowering the automatic rewriting of weak constraints to improve the performance. Such an approach could relieve the user of modeling burdens, comparable to the computational means supplied by ASP(Q) for replacing sophisticated saturation encodings [2].

## References


1. Alviano, M., Ianni, G., Pacenza, F., Zangari, J.: Rethinking Answer Set Programming Templates. In: Gebser, M., Sergey, I. (eds.) Proceedings of the Twenty-sixth International Symposium on Practical Aspects of Declarative Languages (PADL'24), pp. 82–99 (2024). https://doi.org/10.1007/978-3-031-52038-9_6
2. Faber, W., Mazzotta, G., Ricca, F.: An Efficient Solver for ASP(Q). Theory and Practice of Logic Programming **23**(4), 948–964 (2023). https://doi.org/10.1017/S1471068423000121
3. Gebser, M., Kaminski, R., Kaufmann, B., Lindauer, M., Ostrowski, M., Romero, J., Schaub, T., Thiele, S., Wanko, P.: Potassco User Guide (2019). http://potassco.org
4. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Morgan and Claypool Publishers (2012). https://doi.org/10.1007/978-3-031-01561-8
5. Gebser, M., Kaminski, R., Schaub, T.: aspcud: A Linux package configuration tool based on answer set programming. In: Drescher, C., Lynce, I., Treinen, R. (eds.) Proceedings of the Second International Workshop on Logics for Component Configuration (LoCoCo'11), pp. 12–25 (2011). https://doi.org/10.4204/eptcs.65.2
6. Gebser, M., Schaub, T.: Modeling and language extensions. AI Magazine **37**(3), 33–44 (2016). https://doi.org/10.1609/AIMAG.V37I3.2673
7. Lifschitz, V.: Answer set programming and plan generation. Artificial Intelligence **138**(1-2), 39–54 (2002). https://doi.org/10.1016/S0004-3702(02)00186-8