Bound Founded Answer Set Programming

Rehan Abdul Aziz and Geoffrey Chu and Peter James Stuckey
University of Melbourne and NICTA

Bound Founded Answer Set Programming (BFASP) [2] is a formal system that generalizes two well-known declarative programming paradigms Constraint Programming [7] and Answer Set Programming [6]. Two important features of combinatorial solvers are the ability to reason over finite integer domains and *foundedness* (which, we hope, will become clearer later in the article). CP systems support the first but lack the second, whereas while ASP systems do support foundedness over Boolean variables, they are inherently propositional (Boolean), and are therefore, handicapped in reasoning over integer variables efficiently. There have been attempts to combine the strengths of ASP and CP, but almost all of them merely put the two systems together, i.e., the combination supports integer variables and foundedness only for Boolean variables. BFASP is a more complete integration that actually *defines* what foundedness means for numeric quantities. Let us motivate BFASP with the help of an example from this domain.

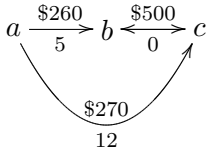

Figure 1: A simple potential road network with distances, and costs to build road segments.

The directed graph in the Figure 1 is a potential road network where the edges represent one-way roads. Moreover, each edge has a distance and a cost (in thousands of dollars) associated with it, which are written below and on top of the edge respectively. Our goal is to minimize the sum of shortest paths from a to b and a to c. [1] The sites b and c have a negligible

[1] Realistically speaking, there are not many one-way roads. We have simplified our example to keep the number of rules short. This simplification is also inconsequential

physical distance if we build the expensive two-way bridge between them (e.g. they are two sites on the opposite sides of a narrow river). We are also constrained by our available budget, which is \$550. In this simple example, the optimal solution is to build the roads ab and ac with the total cost of 530. If we build the bridge between b and c, then we have no money to build any other road and as a result, sites b and c become unreachable from a. Let us now model this problem in BFASP and show why it cannot be efficiently encoded in any competing formalism.

Let $build_{xy}$ represent the decision whether we build the road from x to y and let sp_{xy} represent the length of shortest path from x to y. For every edge (u, v) in the graph, and any node x, it is easy to see that if the edge is built, then sp_{xv} is at most sp_{xu} plus the distance along the edge. Furthermore, the shortest path from a node to itself is at most 0. Let *Nodes*, *Edges* be the set of nodes and edges, and let d_{uv} represent the distance for the edge (u, v). These above ideas can be used as *rules* to define shortest paths in BFASP as follows:

$$\begin{aligned} \forall x \in \mathit{Nodes} : \quad & sp_{xx} \leq 0 \\ \forall x \in \mathit{Nodes}, (u,v) \in \mathit{Edges} : \quad & sp_{xv} \leq sp_{xu} + d_{uv} \leftarrow build_{uv} \end{aligned}$$

We can instantiate above rules for our example to get the following ground rules:

$$\begin{array}{rrr} sp_{aa} \leq 0 & sp_{bb} \leq 0 & sp_{cc} \leq 0 \\ sp_{ab} \leq sp_{aa} + 5 \leftarrow build_{ab} & sp_{ac} \leq sp_{ab} \leftarrow build_{bc} & sp_{bc} \leq sp_{bb} \leftarrow build_{bc} \\ sp_{ac} \leq sp_{aa} + 12 \leftarrow build_{ac} & sp_{ab} \leq sp_{ac} \leftarrow build_{bc} & sp_{cb} \leq sp_{cc} \leftarrow build_{bc} \end{array}$$

We ignored some ground rules like $sp_{bb} \leq sp_{ba} + 5 \leftarrow build_{ab}$ since they are subsumed by $sp_{bb} \leq 0$ etc. We also ignored some rules like $sp_{cb} \leq sp_{ca} + 5 \leftarrow build_{ab}$ since c can never reach a etc. Our budget constraint can be formally written as:

$$260 \times build_{ab} + 270 \times build_{ac} + 500 \times build_{bc} \leq 550$$

To the best of our knowledge, shortest path definitions like the one above, cannot be feasibly modeled in ASP. One way to do so is to propositionalize or ground the shortest path upper bounds for each value in the integer domain and add further ASP rules to ensure that the upper bound variables

since we are only interested in the shortest paths from a, which means that we omit all arcs entering a.

collectively correctly encode the integer domain. This, unsurprisingly, becomes impractical for large distances. We have explained this in detail in [1, 2].[2]

Now, imagine if we pass the above model to a CP (or a CASP) solver; no matter what the values of *build* variables are, as long as the budget constraint is satisfied, all shortest path variables can be set to 0 and it is still a valid model. This is, of course, not correct. The reason is that a CP solver treats the shortest path variables as regular integer variables, whereas the semantics we wish to give these variables is different. In absence of any rule, we require a shortest path variable to be ∞. Furthermore, if there is any rule, then it can justifiably take on a smaller value as given by the LHS of that rule. In other words, its upper bound needs to be *founded* by some rule. For this reason, we refer to the type of variables like *sp* as *upper-bound founded* in BFASP. Similarly, there can be *lower-bound founded* variables which are by default equal to $-\infty$ and further rules justify a higher value. With this generalization, ASP variables are simply lower-bound founded Boolean variables. Another rich feature of BFASP is that as opposed to syntactically limited ASP rules which are simple logical implications (with certain limited extensions like weight rules etc.), rules in BFASP can be formed using a variety of primitive rules recursively. We have discussed the language aspects of BFASP along with how to ground first-order rules efficiently in [3].

We might think that it is possible to model the shortest path rules in CP using standard variables only and setting shortest path variables equal to the minimum value allowed by their rules as follows: (here M is a sufficiently large number, used to model ∞, and in slight abuse of notation, we access *build* variables as 0-1 integer variables):

$$\begin{aligned} &\forall x \in \mathit{Nodes}: && sp_{xx} = 0 \\ &\forall x \in \mathit{Nodes}: && sp_{xv} = \min\{M, (sp_{xu} + d_{uv}) * \mathit{build}_{uv} \\ & && \quad +M * (1 - \mathit{build}_{uv}) : (u, v) \in \mathit{Edges}\} \end{aligned}$$

Let us see what happens in our example if the CP solver decides to build only the bridge, and not any other road, i.e., $\mathit{build}_{ab} = \mathit{build}_{ac} = \mathit{false}$ and $\mathit{build}_{bc} = \mathit{true}$. We get the following simplified expressions:

[2] An attempt to model the problem above in ASP can be found at: `http://people.eng.unimelb.edu.au/pstuckey/bound_founded/encodings/road.gringo` and some instances can be downloaded from: `http://people.eng.unimelb.edu.au/pstuckey/bound_founded/road.zip`

$$sp_{aa} = 0 \quad sp_{bb} = 0 \quad sp_{cc} = 0$$
$$sp_{ab} = \min\{M, sp_{ac}\} \qquad sp_{ac} = \min\{M, sp_{ab}\}$$

which means any assignment that satisfies the constraint $sp_{ab} = sp_{ac}$ is a valid solution, which is wrong. This is because in BFASP semantics, which extend the stable model semantics used in ASP, two or more variables cannot simultaneously justify bounds on each other without having any independent justification. For example, if we have two ASP variables i and j, and only two rules $i \leftarrow j$, and $j \leftarrow i$, then they cannot be set to *true* since the only way to justify that value relies on circular reasoning. In fact, avoiding this kind of circular derivation of bounds is at the heart of BFASP implementation. The main idea of the algorithm, which generalizes a similar algorithm for founded Booleans in ASP, is to keep a *justification graph*, defined over bounds of founded variables, that records for each bound, other bounds that were used to derive it. The algorithm ensures that this justification graph is acyclic at all times, which guarantees that there is no circular derivation. Our experiments have shown that this algorithm for implementing BFASP allows us to solve a range of problems more efficiently than ASP, CP, or *Constraint ASP* (CASP) [5, 4] systems.

References

[1] Aziz, R. A. Bound founded answer set programming. In *Doctoral Consortium, International Conference on Logic Programming, Vienna, 2014*.

[2] Aziz, R. A., Chu, G., and Stuckey, P. J. Stable model semantics for founded bounds. *Theory and Practice of Logic Programming 13*, 4–5 (2013), 517–532. Proceedings of the 29th International Conference on Logic Programming.

[3] Aziz, R. A., Chu, G., and Stuckey, P. J. Grounding bound founded answer set programs. In *30th International Conference on Logic Programming* (2014).

[4] Drescher, C., and Walsh, T. Answer set solving with lazy nogood generation. In *Technical Communications of the 28th International Conference on Logic Programming* (September 2012), pp. 188–200.

[5] Gebser, M., Ostrowski, M., and Schaub, T. Constraint answer set solving. In *Proceedings of the 25th International Conference on Logic Programming* (July 2009), Springer, pp. 235–249.

[6] Gelfond, M., and Lifschitz, V. The stable model semantics for logic programming. In *ICLP/SLP* (1988), pp. 1070–1080.

[7] Rossi, F., Beek, P. v., and Walsh, T. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science, New York, NY, 2006.