BDD-based Parity Games Solver

Saurabh Sharma

March 24, 2020

1 Objectives

1. Implement a procedure for binary encoding of parity games into BDD-based symbolic representation
2. Implement BDD-based *Zielonka algorithm* (returning $W_0((G))$)

2 Method and code-design

2.1 Algorithm

The following design choices were made to make a BDD-based parity games solver based on the Zielonka algorithm[6]:

1. **Game graph representation**: The game graph, i.e the vertices V, the vertice colors, the vertice owners, and the vertice successors, were stored simply in Python lists in the order that they were parsed. A dictionary data-structure was used to keep track of the position of a vertex $v \in V$ in the lists, and another dictionary to store the set of vertices with the same color.
2. **Binary encoding of the game**: The function $f : V \mapsto \mathbb{B}^n$, where $n = \mathcal{O}(log(|V|)$, is implemented efficiently by using the first bit to encode the player, and the remaining bits to encode the node identifier v. The BDD for the game thus contains $2n$ variables, n to maintain the current state and n for the next state.
3. **Attractor computation** The attractor computation using BDDs to represent the sets V_0, V_1, $E(x, X)$, where x and X represent the current and next state of the game respectively, is based on the following pseudo-code to compute the Controlled Predecessor of a region R,

$$CPred_i(R) = (V_i \wedge \exists X(E(x, X) \wedge R) \vee (V_{1-i} \wedge \neg\exists X(E(x, X) \wedge \neg R) \quad (1)$$

This formulation is based on the paper by Sanchez et al. [5]. It is observed that operations on sets, i.e comparison, intersection, union and complement are easily expressed as BDD operations using equivalence, conjunction, disjunction and negation. The rest of the API is based on the starter code in Attractors.py for priming latches and finding a fixed-point iterate.

4. **Removing vertices from the graph**: This was implemented easily using the BDD operations as follows: to remove a set of vertices A from the graph, $V_0 \setminus A$ and $V_1 \setminus A$ were computed as $V_0 \wedge \neg A$ and $V_1 \wedge \neg A$. Moreover, a node mask was maintained to specify which nodes were in use in a function call.

5. **Implementing Zielonka's algorithm** Zielonka's algorithm, as described in the lecture, was implemented directly using the data-structures described above and APIs for attractor computation and removing a set of vertices from the graph.

2.2 Implementation details

The code was implemented in Python. The following packages/libraries were used to implement the above algorithm,

2.2.1 ANTLR

The Another Tool for Natural Language Recognition (ANTLR) parser [1] for Python was used to parse a parity game provided in a PGsolver [2] format specification, which is as follows:

- First line contains: type of game, size of V.
- Each following line contains:
  - node identifier v
  - color $\Omega(v)$
  - player (i with $v \in V_i$)
  - list of successors (all $w \in V$ with $(v, w) \in E$)
  - comment (no effect on game)

2.2.2 DD

The dd library[3] in Python with the dd.autoref.bdd interface was used to implement all BDDs [4] and BDD based operations for the project.

References

[1] https://github.com/antlr/antlr4

[2] https://github.com/tcsprojects/pgsolver

[3] https://github.com/tulip-control/dd

[4] Bryant, R.E.: Graph-based algorithms for boolean function manipulation. Computers, IEEE Transactions on **100**(8), 677–691 (1986)

[5] Sanchez, L., Wesselink, W., Willemse, T.A.: A comparison of bdd-based parity game solvers. arXiv preprint arXiv:1809.03097 (2018)

[6] Zielonka, W.: Infinite games on finitely coloured graphs with applications to automata on infinite trees. Theoretical Computer Science **200**(1-2), 135–183 (1998)