Int. J. Appl. Math. Comput. Sci., 2015, Vol. 25, No. 3, 577–596
DOI: 10.1515/amcs-2015-0043

A SYMBOLIC SHORTEST PATH ALGORITHM FOR COMPUTING SUBGAME–PERFECT NASH EQUILIBRIA

PEDRO A. GÓNGORA [a,*], DAVID A. ROSENBLUETH [a]

[a] Institute for Research in Applied Mathematics and Systems
National Autonomous University of Mexico, A.P. 20-126, C.P. 01000, Mexico D.F., Mexico
e-mail: pedro.gongora@gmail.com, drosenbl@unam.mx

Consider games where players wish to minimize the cost to reach some state. A subgame-perfect Nash equilibrium can be regarded as a collection of optimal paths on such games. Similarly, the well-known state-labeling algorithm used in model checking can be viewed as computing optimal paths on a Kripke structure, where each path has a minimum number of transitions. We exploit these similarities in a common generalization of extensive games and Kripke structures that we name "graph games". By extending the Bellman–Ford algorithm for computing shortest paths, we obtain a model-checking algorithm for graph games with respect to formulas in an appropriate logic. Hence, when given a certain formula, our model-checking algorithm computes the subgame-perfect Nash equilibrium (as opposed to simply determining whether or not a given collection of paths is a Nash equilibrium). Next, we develop a symbolic version of our model checker allowing us to handle larger graph games. We illustrate our formalism on the critical-path method as well as games with perfect information. Finally, we report on the execution time of benchmarks of an implementation of our algorithms.

Keywords: shortest path, Bellman–Ford, Nash equilibrium, BDD, model checking.

1. Introduction

We explore connections between (i) Nash equilibrium computation and (ii) reachability verification through model checking. In a tree representation of a game, a subgame-perfect Nash equilibrium can be regarded as a collection of optimal (i.e., having a minimum or maximum weight) paths, going from every internal node to some leaf. Similarly, in model checking, the behavior of non-deterministic systems is sometimes represented as a tree structure. When verifying a reachability property for these systems, a usual approach is to go backwards, from the known destination to every possible source in the branching-time structure. By going backwards, this computation always visits the closest system states first. In this manner, such a computation can also be regarded as the construction of optimal paths going from every system state to the appointed destination. We propose first using the benefits of symbolic model checking algorithms in Nash equilibria computation, and secondly extending the model checking applicability to game-like situations. More specifically, we approach these goals by (i) presenting a symbolic Bellman–Ford algorithm for games with many players and turns, and (ii) extending model checking procedures for verifying such games.

We can graphically represent finite, perfect information, non-cooperative games using trees, i.e., extensive form games. In this paper we work with a game tree direct generalization: graph games. The two main differences are that graph games (i) may contain cycles and (ii) have weights (or utility values) associated with every transition. We also define path optimality in these graph games as a generalization of subgame-perfect Nash equilibria. Following this definition, we show how to use an extended Bellman–Ford algorithm for computing such optimal paths, and thus computing subgame-perfect Nash equilibria.

It is important to mention that the method we present here may be adapted to game trees. By generalizing to graph games we, nonetheless, add little to none mathematical expense. On the contrary, we ease the connection with model checking and the applicability of the Bellman–Ford shortest path algorithm.

For model checking, we can represent reactive and non-deterministic systems as finite automata. We can

*Corresponding author

unfold the behavior of such automata as infinite trees. For describing some properties of these trees we can then use computation tree logic (CTL). A commonly used algorithm for CTL verification is the state-labeling one. As we mentioned above, when verifying reachability properties with this algorithm, we actually compute an optimal path from every state of the system to the selected destination states. As CTL models' transitions are not weighted, such optimal paths consider only the number of transitions. We propose using game-graph-based models in CTL model checking. We then develop a CTL extension for describing not only reachability properties, but also properties corresponding to the costs/utilities of the game's players.

One of the greatest strengths of model checking is the possibility of verifying large systems. This strength is mainly due to the use of symbolic algorithms. Symbolic algorithms use efficient data structures for representing and manipulating data, such as binary decision diagrams (BDDs). In particular, we use a generalization of BDDs known as multi-terminal BDDs (MTBDDs (Fujita *et al.*, 1997)).

We are interested in MTBDD matrix representation and multiplication. The reason is that the Bellman–Ford algorithm can be formulated as a succession of matrix multiplications. In this paper, we extend the symbolic version of the Bellman–Ford algorithm developed by Fujita *et al.* (1997). Their algorithm computes matrix multiplication and thus shortest paths for single weighted graphs (see also Bahar *et al.*, 1997). Our extension computes shortest paths for graph games, that is, weighted graphs having multiple cost functions and turns.

In the following sections, we will present a symbolic version of the Bellman–Ford algorithm for computing shortest paths in graph games. We will use this algorithm for extending the model checking state-labeling algorithm, and thus will verify CTL models based on graph games. We also present some experimental results comparing symbolic versus non-symbolic extended Bellman–Ford implementations.

Our results show that the advantages of using (MT)BDDs are less pronounced than in CTL model checking, for example. In a sense this is to be expected, as the presence of agents involves additional computation steps, absent in a state-labeling model checker for CTL. The use of MTBDDs, however, requires less memory than a nonsymbolic method and allows handling larger examples.

1.1. Related work.

Our graph games are similar to the dynamic networks of Lozovanu and Pickl (2009). In their games each player independently selects a vector control parameter. Each state of the network is then determined by the selection of all players. Lozovanu and Pickl (2009) define Nash-equilibrium based solutions as well as other game theoretic notions. Then the authors show a Dijkstra algorithm variant for finding such solutions.

In the scope of game and graph symbolic algorithms, we can mention the works by Bolus (2011) as well as Berghammer and Bolus (2012), who employ quasi-ordered BDDs for computing winning coalitions and solving other game theoretic problems for simple games—cooperative games having only two possible cases of coalitions. Our symbolic shortest path algorithm is an extension of the symbolic Bellman–Ford introduced by Fujita *et al.* (1997) and later optimized by Bahar *et al.* (1997). A different approach of symbolic shortest path computation is presented by Sawitzki (2004), who uses ordinary BDDs, as opposed to multi-valued, for codifying weighted graphs, and presents symbolic versions of the Dijkstra and Bellman–Ford algorithms. Both the Nash equilibrium and shortest path computation are particular cases of multi-criteria optimization. Although not symbolic, we refer the reader to Garroppo *et al.* (2010) for a summary of various multi-criteria path optimization approaches.

On the model checking side, Dasgupta *et al.*. (2001) introduce min-max CTL. They use their min-max CTL language for verifying and querying quantitative properties of timed systems. Compared with our approach, Dasgupta *et al.*. (2001) neither use a shortest path nor a symbolic algorithm for min-max CTL verification and querying. Another approach of symbolic model checking for multi-agent systems is that of Raimondi and Lomuscio (2007), who introduce a logical framework for describing temporal, epistemic, and deontic properties. The authors also provide BDD-based algorithms for automated verification.

Finally, assuming a more theoretical approach, there exist several modal logic characterizations of game solutions, such as the Nash equilibrium. Compared with this paper, those works only *characterize* Nash equilibria, whereas our method *computes* them. We find worth mentioning the seminal work by Bonanno (2001), who introduces the use of temporal logic for analyzing the logical structure of perfect information games. Another representative work in this line is that of Harrenstein *et al.* (2003), who also present a logical characterization of perfect information games. By using dynamic logic instead of temporal logic, the work of Harrenstein *et al.* (2003) adds a more operational flavor to Nash equilibria characterization.

2. Games and graph games

In this section, we first observe a correspondence between shortest paths in extensive games and subgame-perfect Nash equilibria. Next, we define both the graph game and the notion of a shortest path.

2.1. Motivation. Games, in a nutshell, are formal descriptions of rational agents' interactions. We may formalize such interactions either as collections of possible actions available to each player (strategic form), or as sets of sequences of events (extensive form). The nature of the game (e.g., of conflict or agreement) is sometimes described numerically, assigning utility units to each agent at every game outcome. In this paper, we deal only with finite, non-cooperative, complete and perfect information games in extensive form. We thus refer the reader to Osborne and Rubinstein (1994) for a more thorough introduction to game theory.

A key feature of the game-theoretic approach to decision making is that all players are assumed rational. Rationality dictates a player to always choose the actions maximizing the player's own utility. Consequently, a *game solution* is a method for choosing strategies maximizing utility for all players. A well-known solution concept is that of a *Nash equilibrium*. A strategy profile is a Nash equilibrium if every player cannot increase its utility by unilaterally deviating from the profile.

Consider the game depicted in Fig. 1. The game has two players, 1 and 2. The player's turns are depicted above the non-final nodes and the utility gains below the leaves. Player 1 moves first, then player 2 moves and the game ends. Suppose that 1 moves right and then 2 also moves right; then the players will gain 4 and 2 utility units, respectively. From the root node, this outcome is determined if all players follow the strategy profile emphasized with thick arrows in the figure.

For claiming this solution to be *subgame-perfect*, we require that all decisions taken at every state be optimal. Thus, player 2 must choose right at state s_1.

Suppose we add a fictitious final state s_{end}, such that every leaf z in Fig. 1 has a unique and costless/gainless output arc arriving at s_{end}. The solution we described above defines a unique path going from every state to s_{end}. Moreover, this unique path is optimal, as the best choice is taken at every state (assuming that all agents are rational). Also observe that, in the case of a single player, this problem reduces to finding the longest path from every state to the destination s_{end}, and thus can be solved using some shortest-path algorithm.

In the rest of this section, we will generalize these game tree notions to graphs. Later, we will show how it is possible to extend a shortest-path algorithm for finding shortest and longest paths in these more general games, and thus finding their subgame-perfect Nash equilibria.

2.2. Graph games. In this section, we extend the notion of game trees to directed graphs. Basically, we allow the game to take the form of a directed graph and associate cost or utility units with every edge of such a graph.

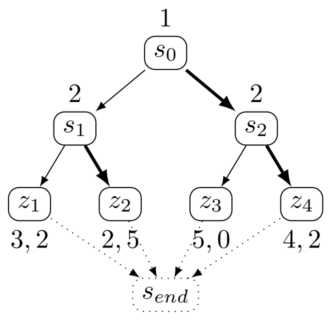


Fig. 1. Two-player game and its subgame-perfect Nash equilibrium.

We define a *graph game* as the quintuple $G = (V, E, N, P, C)$ such that

- V is a finite set of nodes,
- $E \subseteq V^2$ is the set of edges,
- $N = \{1, \ldots, n\}$ is a set of players,
- $P : V \to N$ is a player's turn function, and
- $C : E \to (\mathbb{R}_+)^n$.

Given two distinct nodes v and v' of a graph game G, we define a *path of length k from v to v'* as a sequence of distinct nodes $v_1 v_2 \cdots v_k$, such that $v = v_1$, $v' = v_k$, and $(v_i, v_{i+1}) \in E$ for $1 \le i < k$. We also define the *cumulative cost* of such a path as the pairwise summation $Cost(v_1 \cdots v_k) = \sum_1^{k-1} C(v_i, v_{i+1})$. If $Cost(v_1 \cdots v_k) = x$ and $p \in N$, we define $Cost_p(v_1 \cdots v_k) = x_p$, where x_p is the p-th component of x.

Note that standard game trees are a special case of graph games. For these trees, a subgame-perfect Nash equilibrium is a function defining a path from every node of the tree to some leaf. As we informally suggested in the previous subsection, in our graph games, Nash equilibria correspond to a generalization of the notion of shortest path in the multiple player case.

As graph games may have no leaves, for defining this generalization of the notion of a shortest path we must first choose a single destination node. In this manner, our solution must find a shortest path from every node of the graph to the selected destination. Secondly, in this generalization, we must take into account the player's turns. At every node of the graph, each player optimizes its utility, and thus our notion of shortest path uses the player's turns as constraints over the cumulative costs.

A path $v_1 \cdots v_k$ of a graph G is a *shortest path* from v_1 to v_k if for all $i \in \{1, \ldots, k-1\}$ and for all u such that $(v_i, u) \in E$, every path $uu' \cdots v_k$ (i.e., any path from u to v_k) satisfies $Cost_{P(v_i)}(v_i v_{i+1} \cdots v_k) \le Cost_{P(v_i)}(v_i u u' \cdots v_k)$.

3. Bellman–Ford algorithm with agents and turns

3.1. Bellman–Ford algorithm. Several efficient algorithms exist for finding shortest paths in weighted graphs. See, for example, the work of Cormen *et al.* (2009) for a detailed explanation of some of these algorithms. In this section we extend the Bellman–Ford algorithm for finding shortest paths in our graph games.

Usually the Bellman–Ford algorithm is presented as solving the problem of computing shortest paths from a given source to all vertices of a graph. In our case, however, we wish the shortest paths from all vertices to a given destination. The Bellman–Ford algorithm also solves this other problem by inverting the edges.

The Bellman–Ford algorithm (shown in Algorithm 1) finds the shortest path from every node to a single destination in $O(|V| \cdot |E|)$ time. Briefly, the algorithm consists in maintaining, for every vertex v, (i) an over-approximation $\mathit{dist}(v)$ to the cumulative cost of the shortest path from v to a given destination d and (ii) an approximate best successor $\mathit{succ}(v)$ of v in such a shortest path. These over-approximations are repeatedly updated until converging to the optimal values. The updates are made by refining the over-approximation of the first node of each edge in the graph. RELAX refines a single edge, and TRIANGLE refines all edges.

Algorithm 1. Bellman–Ford algorithm.

```
1: BELLMAN–FORD($G, d$):
2: for all $v \in V$ do
3:     $succ(v) \leftarrow \perp$
4:     $dist(v) \leftarrow \infty$
5: end for
6: $dist(d) \leftarrow 0$
7: for $x \leftarrow 1$ to $|V| - 1$ do
8:     TRIANGLE()
9: end for
10: TRIANGLE():
11: for all $(v, v') \in E$ do
12:     RELAX$(v, v')$
13: end for
14: RELAX$(v, v')$:
15: if $dist(v) > C(v, v') + dist(v')$ then
16:     $succ(v) \leftarrow v'$
17:     $dist(v) \leftarrow C(v, v') + dist(v')$
18: end if
```

Before presenting the extended version of the algorithm, we will recall two relaxation properties crucial to the Bellman–Ford algorithm's correctness:

R1 For every edge (v, v'), RELAX(v, v') cannot increase the over-approximation $\mathit{dist}(v)$.

R2 If $v_1 \cdots v_k$ is a shortest path from v_1 to v_k, then $\mathit{dist}(v_1)$ is optimal after relaxing all path edges in the order $(v_{k-1}, v_k), (v_{k-2}, v_{k-1}), \ldots, (v_1, v_2)$.

Property **R1** follows immediately from the definition of RELAX. For **R2**, observe that just after relaxing (v_{k-1}, v_k) the over-approximation $\mathit{dist}(v_{k-1})$ is optimal, as $\mathit{dist}(v_k) = 0$ is initialized to its optimal value. By following a simple inductive argument we can assert that after relaxing $(v_{k-(i+1)}, v_{k-i})$ the over-approximation $\mathit{dist}(v_{k-(i+1)})$ is optimal.

The correctness of the algorithm follows from these two properties. The algorithm executes the TRIANGLE() procedure $|V| - 1$ times. Thus, all edges are relaxed $|V| - 1$ times, the maximum possible length of a shortest path. At the i-th TRIANGLE() execution, the first edge (v, v') of an i-length shortest path is relaxed in the order required by **R2**, and $\mathit{dist}(v)$ will remain optimal from then on (because of **R1**).

3.2. Extending Bellman–Ford. In Algorithm 2 we present an extended version of the Bellman–Ford algorithm to graph games. In graph games, the player function application $P(v)$ represents a constraint to the node v. When visiting such v, the cumulative cost to be optimized must be that of player $P(v)$. Accordingly, in the relaxation procedure, the over-approximation to be refined is $\mathit{dist}_{P(v)}(v)$ (Algorithm 2, line 15). The decisions taken by player $P(v)$ affect the cumulative cost of all players. Thus, if an improvement is possible, the cumulative cost for *every* player must be updated (Algorithm 2, line 17).

For explaining the last lines of the algorithm, observe that a shortest path may not be unique. When searching for a shortest path from v to v', we optimize the cumulative cost function of player $P(v)$. The cumulative cost of players other than $P(v)$, however, may vary among those shortest paths. In some applications, like the one presented in Section 5, we are also interested in finding, among the shortest paths, the best path for some distinguished player i^*. For this reason, we check, without deteriorating the cumulative cost of player $P(v)$, if there is a better choice for player i^* (Algorithm 2, lines 18–23).

To state the correctness of this algorithm, we can rely on two properties analogous to **R1** and **R2**:

R1′ For every edge (v, v'), RELAX(v, v', i^*) cannot increase the over-approximation $\mathit{dist}_{P(v)}(v)$.

R2′ If $v_1 \cdots v_k$ is a shortest path from v_1 to v_k, then $\mathit{dist}_{P(v_1)}(v_1)$ is optimal after relaxing all path edges in the order $(v_{k-1}, v_k), (v_{k-2}, v_{k-1}), \ldots, (v_1, v_2)$.

Property **R1′** also follows from the definition of RELAX in Algorithm 2. For **R2′**, observe that, after relaxing (v_{k-1}, v_k), $\mathit{dist}_{P(v_{k-1})}(v_{k-1})$ is optimal, as $\mathit{dist}(v_k)$ is

Algorithm 2. Bellman–Ford with many players and turns.

```
1: BELLMAN–FORD($G, d, i^*$):
2: for all $v \in V$ do
3:     $succ(v) \leftarrow \bot$
4:     $dist(v) \leftarrow \overline{\infty}$
5: end for
6: $dist(d) \leftarrow \overline{0}$
7: for $x \leftarrow 1$ to $|V| - 1$ do
8:     TRIANGLE()
9: end for
10: TRIANGLE():
11: for all $(v, v') \in E$ do
12:     RELAX($v, v', i^*$)
13: end for
14: RELAX($v, v', i^*$):
15: if $dist_{P(v)}(v) > C_{P(v)}(v, v') + dist_{P(v)}(v')$ then
16:     $succ(v) \leftarrow v'$
17:     $dist(v) \leftarrow C(v, v') + dist(v')$
18: else if $dist_{P(v)}(v) = C_{P(v)}(v, v') + dist_{P(v)}(v')$ then
19:     if $dist_{i^*}(v) > C_{i^*}(v, v') + dist_{i^*}(v')$ then
20:         $succ(v) \leftarrow v'$
21:         $dist(v) \leftarrow C(v, v') + dist(v')$
22:     end if
23: end if
```

initialized to its optimal value $\overline{0}$. In general, at the i-th step, when relaxing (v_{k-i}, v_{k-i+1}), $dist_{k-i}(v_{k-i})$ will acquire its optimal value, as $dist_{k-i+1}(v_{k-i+1})$ is already optimal (i.e., the agent $P(v_{k-i+1})$ made its best choice, and thus the cumulative cost for the other players is part of this path optimality). Thus, we can use the same inductive argument as for the original Bellman–Ford to prove the correctness of the extended algorithm.

Finally, we will only mention that it is possible to use two properties similar to **R1**$'$ and **R2**$'$ for proving that, if there are several shortest paths from v to v', Algorithm 2 will find the shortest path with minimum $dist_{i^*}(v)$.

3.3. Maximizing utility functions.

Often, games are formulated as a utility maximizing problem, as opposed to a cost minimizing one. For solving such maximization problems it suffices to invert the sign of the utility functions. There is, however, a detail we must consider: the occurrence of negative cycles in the graph. The reason is that, by traversing a negative cycle, it is always possible to further minimize the resulting value.

In our graph games, negative weights only affect arcs (v, v') such that $C_{P(v)}(v, v') < 0$. Fortunately, the Bellman–Ford algorithm is capable of detecting such cycles.

After running $|V| - 1$ iterations of the TRIANGLE procedure, $dist$ necessarily converges to the optimal values. If we run again the algorithm (i.e., another $|V| - 1$ iterations of TRIANGLE), at some iteration, when relaxing arcs (v, v') in a negative-weighted cycle, the $dist(v)$ values will necessarily change. The reason is that player $P(v)$ can minimize even more its $dist_{P(v)}(v)$ value by subtracting $C_{P(v)}(v, v')$. Thus, after this extra execution, we can detect the nodes forming part of a negative cycle by comparing their over-approximations with the previous algorithm execution. If, for some v, $dist(v)$ changed, then we simply set $dist(v) = \overline{\infty}$. Finally, after finishing this negative cycle detection phase, we must restore the sign of the other $dist$ values to obtain the desired result.

4. Symbolic Bellman–Ford algorithm with many players and turns

In this section, we present a symbolic version of the extended Bellman–Ford algorithm. This algorithm is an adaptation of the matrix multiplication algorithm by Fujita *et al.* (1997). We first review the relation between matrix multiplication and the Bellman–Ford algorithm.

4.1. Matrix multiplication and Bellman–Ford.

Our algorithm is based on the fact that matrix multiplication and TRIANGLE are equivalent procedures (Bahar *et al.*, 1997). That is, matrix multiplication conforms to the semi-ring $(\mathbb{R}, +, \times, 0, 1)$ and TRIANGLE to the semi-ring $(\mathbb{R} \cup \{\infty\}, \min, +, \infty, 0)$. Therefore, it is possible to adapt a matrix multiplication implementation to compute shortest paths in a graph.

To exemplify the relation of these procedures observe the following matrix multiplication:

$$\begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix} \times \begin{pmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{pmatrix}$$

$$= \begin{pmatrix} a\cdot\alpha & + & b\cdot\beta & + & c\cdot\gamma & + & d\cdot\delta \\ e\cdot\alpha & + & f\cdot\beta & + & g\cdot\gamma & + & h\cdot\delta \\ i\cdot\alpha & + & j\cdot\beta & + & k\cdot\gamma & + & l\cdot\delta \\ m\cdot\alpha & + & n\cdot\beta & + & o\cdot\gamma & + & p\cdot\delta \end{pmatrix}.$$

In this product, we combine the rows of the left square matrix with each element of the right vector using the semi-ring operators $+$ and $\times$. If we follow the same procedure, but using $\min$ and $+$ as the semi-ring operators, we can compute TRIANGLE:

$$\begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix} \triangle \begin{pmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{pmatrix}$$

$$= \begin{pmatrix} \min\{a+\alpha, & b+\beta, & c+\gamma, & d+\delta\} \\ \min\{e+\alpha, & f+\beta, & g+\gamma, & h+\delta\} \\ \min\{i+\alpha, & j+\beta, & k+\gamma, & l+\delta\} \\ \min\{m+\alpha, & n+\beta, & o+\gamma, & p+\delta\} \end{pmatrix}.$$

Here, for the $\triangle$ operator, the left matrix is the adjacency matrix of the graph and the right vector is an over-approximation vector containing the values of $dist(v)$ for each $v \in V$.

Each element a_{ij} of the adjacency matrix of a weighted graph $G = (V, E, C)$ is defined as follows:

- $a_{ij} = C(v_i, v_j)$ if $(v_i, v_j) \in E$,
- $a_{ij} = 0$ if $i = j$ and $(v_i, v_j) \notin E$,
- $a_{ij} = \infty$ if $i \neq j$ and $(v_i, v_j) \notin E$.

4.2. Multi-terminal binary decision diagrams.

Multi-terminal decision diagrams (MTBDDs) (Clarke *et al.*, 1993; Fujita *et al.*, 1997) are an extension of the original reduced ordered binary decision diagrams (BDDs for short (see Bryant, 1986).

BDDs are a graphical representation of functional mappings $\mathbb{B}^n \to \mathbb{B}$. MTBDDs extend these mappings to the more general case $\mathbb{B}^n \to R$, where R is an arbitrary set, usually $R \subseteq \mathbb{R}$.

An MTBDD representing a function $f : \mathbb{B}^n \to R$ is a rooted, directed and acyclic graph. In such a graph, there are two kinds of nodes: the terminal and the non-terminal nodes. Every non-terminal node is associated with a unique Boolean input variable $x_i \in \{x_1, \dots, x_n\}$. Also, every non-terminal node has two children: a *lo* node and a *hi* node. A *lo* node represents the case when the variable of the parent node has the value 0, and the *hi* node represents the case when the variable has the value 1. Each terminal node is associated with a unique value in R, and has no descendants.

In a path from the root to a terminal, not all input variables need to occur. In such paths, however, all the occurring variables must be ordered. That is, a node associated with a variable x_i must be closer to the root than a node associated with a variable x_j iff $i < j$, and we say that $x_i < x_j$ in such a case. Also, an MTBDD is reduced (there are no redundant nodes or redundant arcs) and unique (i.e., canonical).

As an example, see the MTBDD in Fig. 2. This MTBDD represents a function with three input variables: x_1, x_2, and x_3. From non-terminal nodes, dotted arrows lead to *lo* nodes and continuous arrows to *hi* nodes. For evaluating the function, we can follow a path from the root to some terminal, choosing the desired Boolean value at each non-terminal node. For example, $f(0,0,0) = 1$, $f(0,1,1) = 3$, and so on. We can think of an MTBDD as obtained from a binary decision tree where redundant nodes have been removed. For example, below

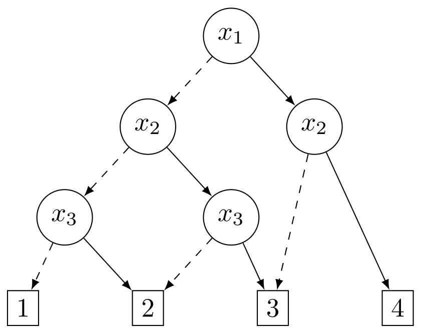


Fig. 2. MTBDD for a function $f : \mathbb{B}^3 \to \{1, 2, 3, 4\}$.

the rightmost node labeled x_2, there could be two child nodes labeled x_3, one pointing its child nodes to terminal '3', and the other pointing its child nodes to terminal '4'. These nodes, however, would be redundant, and thus we eliminate them. Also, note that there is, at most, one terminal 'r' for each $r \in R$.

Finally, before detailing our symbolic algorithm, we define some notation. Let A and B be two MTBDDs. Then

- we will refer as $top(A)$ to the variable labeling A's root node (assuming that such a node is not terminal);
- we define $top(A, B) = \min\{top(A), top(B)\}$;
- if $x = top(A)$, then $A|_{\neg x}$ and $A|_x$ refer to the A's *lo* and *hi* branches, respectively;
- let x be a variable occurring in neither A nor B, satisfying $x < x'$, for every variable x' occurring in either A or B; we will call $newNode(x, A, B)$ the MTBDD having (i) the root node labeled x, (ii) A as the *lo* branch, and (iii) B as the *hi* branch.

4.3. Symbolic Triangle.

The symbolic Triangle is based on the recursive definition of matrix multiplication. For this recursive definition, we divide the original matrices in four quadrants, or two halves in the case of vectors. For implementing the Bellman–Ford algorithm, we are only interested in the multiplication of a square matrix by a vertical vector. The algorithm, however, is easily extended to more general cases.

The matrix multiplication (by a vector) is recursively defined as follows:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \times \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = \begin{pmatrix} A_{11} \times B_1 + A_{12} \times B_2 \\ A_{21} \times B_1 + A_{22} \times B_2 \end{pmatrix},$$

where the square submatrices A_{ij} are the four exact quadrants of the left operand and the subvectors B_k are the two exact halves of the right operand.

Again, if we replace the matrix semi-ring operations by the triangle $\min$ and $+$ operations, we obtain a recursive $\triangle$ definition:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \triangle \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = \begin{pmatrix} \min\{A_{11} \triangle B_1, A_{12} \triangle B_2\} \\ \min\{A_{21} \triangle B_1, A_{22} \triangle B_2\} \end{pmatrix}.$$

In the symbolic Bellman–Ford, each matrix is represented by an MTBDD. We use MTBDD variables to codify the matrices' cells' rows and columns, and MTBDD terminals to store cell values. We codify rows and columns as binary numbers. We follow the common heuristic of alternating the MTBDD variables for minimizing the space required to store the diagram (cf. Enders *et al.*, 1992; Dsouza and Bloom, 1995; Hermanns *et al.*, 1999). In this manner, we use odd variables for codifying matrix rows and even variables for codifying matrix columns.

Following this alternating codification, in the bit sequence $x_1 \cdots x_n$, x_1 is the most significant bit codifying a matrix row, x_2 is the most significant bit codifying the matrix column, and so on.

For example, an MTBDD representing a 4×4 matrix, say A, would have, at most, four variables: two codifying the rows and two codifying the columns. Thus, the cell position at the fourth row (row 11) and the first column (column 00) is represented by the bit sequence 1010. See the next subsection for a complete example of this codification.

The symbolic TRIANGLE uses the standard MTBDD Apply algorithm for implementing $\min$ and addition. The Apply algorithm implements termwise binary operations. An operation $\Box$ is *termwise* if, for any two matrices A and B, $(A \Box B)_{ij} = A_{ij} \Box B_{ij}$.

When running the symbolic TRIANGLE, we traverse the MTBDDs from the root to the leaves. Each non-terminal node splits the matrix into two halves. The nodes with odd variables split the matrix horizontally and the nodes with even variables vertically. We then recursively apply TRIANGLE to both halves. On the one hand, we can see in the recursive $\triangle$ definition above that, when splitting the matrix horizontally, we just have to join the upper and lower halves into a single vector result. On the other hand, when splitting the matrix vertically, we must apply $\min$ (or $\min^i$ in the extended algorithm) for combining the left and right partial results.

It is important to mention that this algorithm requires that we can always recursively split the matrix into two exact halves. That is, the algorithm works only for $2^n \times 2^n$ matrices. Despite this, we can apply the algorithm to arbitrarily large matrices by attaching an identity submatrix (Fujita *et al.*, 1997):

$$\begin{pmatrix} A & 0 \\ 0 & 1 \end{pmatrix}$$

for adjusting the required matrices' size.

Observe, however, that we are working with a different instance of the product's semi-ring. Thus, an identity matrix is that with zeroes on the diagonal, with infinity values filling the rest of the matrix.

4.4. Symbolic extended Bellman–Ford. We will present a symbolic version of the extended Bellman–Ford algorithm. This algorithm finds shortest paths in a graph game $G = (V, E, N, P, C)$. The graph is represented by its adjacency matrix, and we use an extension of the symbolic matrix multiplication. An MTBDD representing a graph game is a map $\mathbb{B} \to (\mathbb{R}_+)^n$.

Let A be the adjacency matrix and D the first over-approximation vector (see Algorithm 2) of some graph game G:

$$A = \begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix}, \qquad D = \begin{pmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{pmatrix},$$

where each component a_{ij} of A is now a state-indexed vector defined as follows:

- $a_{ij} = C(v_i, v_j)$ if $(v_i, v_j) \in E$,
- $a_{ij} = \overline{0}$ if $i = j$ and $(v_i, v_j) \notin E$,
- $a_{ij} = \overline{\infty}$ if $i \neq j$ and $(v_i, v_j) \notin E$.

This graph game has four nodes, say s_0 to s_3. After running the TRIANGLE procedure, we have the following vector of state-indexed vectors:

$$\begin{pmatrix} \min^{P(s_0)}\{a+\alpha, & b+\beta, & c+\gamma, & d+\delta\} \\ \min^{P(s_1)}\{e+\alpha, & f+\beta, & g+\gamma, & h+\delta\} \\ \min^{P(s_2)}\{i+\alpha, & j+\beta, & k+\gamma, & l+\delta\} \\ \min^{P(s_3)}\{m+\alpha, & n+\beta, & o+\gamma, & p+\delta\} \end{pmatrix}.$$

Here, the $+$ operator is the vector pairwise addition. Note, however, that the $\min^i$ operator is relative to the players' turns.

If X is a set of vectors of length greater than or equal to i, then $\min^i X$ denotes an $x \in X$ with a minimal i-th component. For example, let $x = \min^i\{(5, 10), (8, 3), (20, 15)\}$. For $i = 1$ we have that $x = (5, 10)$, and for $i = 2$ we have that $x = (8, 3)$.

In the matrix resulting from the TRIANGLE application, the rows dictate the players' turns. Thus, $\min^{P(s)}$ corresponds to an optimal choice for the player $P(s)$.

The $\min^i$ operation is not termwise, and we cannot implement it using the standard *Apply* procedure. Note that our relative $\min^i$ operator depends on the position of the operands. Thus, we must define a way of implementing termwise position-sensitive binary operations.

An MTBDD binary operation is *termwise position-sensitive* if the resulting value depends at most on (i) the values of both operands and (ii) the trajectory leading from the root of the diagram to the terminal nodes.

The *Apply* algorithm traverses the MTBDDs until reaching the terminal nodes, and then applies the required operator to the node values. If the operation is termwise position-sensitive, then the resulting value may change if such terminal nodes are reached by following distinct trajectories. Equivalently, if the MTBDD represents a matrix, such a trajectory corresponds to the position of the cell in the matrix.

Our symbolic extended Bellman–Ford algorithm consists of two subroutines: the *RelTriangle* and *RelApply* procedures. These extended operations differ from the originals in that they are position-sensitive, and thus *RelTriangle* is able to deal with players' turns in graph games.

At first sight, it might seem that we can extend the algorithm of Bahar *et al.* (1997) to obtain *RelTriangle* and *RelApply*. In such extended operations, when traversing from top to bottom the two MTBDD operands, we would only consider the variables occurring in either path. However, it is important to note that such extended algorithms based on the one by Bahar *et al.* (1997) would be incorrect. The reason is that both these operations are position-sensitive (sensitive to the operand's position on the matrix or the MTBDD trajectories). By skipping variables, we would lose such information. This constraint forces us to extend the algorithm of Fujita *et al.* (1997) instead of the more efficient one by Bahar *et al.* (1997).

For the *RelTriangle* and *RelApply* procedures to be position-sensitive, we must record the values of the variables leading to a terminal node at the base of the recursion. We will use, in addition to the other procedure parameters, a bit-vector $\overline{b}$ for this purpose. The i-th bit of $\overline{b}$ stands for the value we assigned to the MTBDD variable x_i for reaching a terminal node. For both procedures, we set such values at each recursive call. As the procedures recur over every variable, when reaching the terminal node, every bit of $\overline{b}$ will have a previously assigned value.

***RelTriangle* procedure.** Given some graph game $G = (V, E, N, P, C)$, let (i) A be an MTBDD representing the adjacency matrix of G, (ii) D an MTBDD representing an approximate-costs vector, (iii) $\overline{b}$ a bit vector of length $2 \cdot |V|$, (iv) x_i the least variable associable with a non-terminal node, and (v) $i^* \in N$ a distinguished agent.

We define the $\mathit{RelTriangle}(A, D, \overline{b}, x_i, i^*)$ procedure as follows:

1. If x_i is odd (i.e., horizontally splits the matrix),
$$\begin{aligned}&\mathit{RelTriangle}(A, D, \overline{b}, x_i, i^*)\\ &\quad = \mathit{newNode}(x_i,\\ &\qquad \mathit{RelTriangle}(A|_{\neg x_i}, D|_{\neg x_i}, \overline{b}|_{\neg}, x_{i+1}, i^*),\\ &\qquad \mathit{RelTriangle}(A|_{x_i}, D|_{x_i}, \overline{b}|_{x_i}, x_{i+1}, i^*)).\end{aligned}$$

2. If x_i is even (i.e., vertically splits the matrix),
$$\begin{aligned}&\mathit{RelTriangle}(A, D, \overline{b}, x_i, i^*)\\ &\quad = \mathit{RelApply}(\\ &\qquad \mathit{RelTriangle}(A|_{\neg x_i}, D|_{\neg x_i}, \overline{b}|_{\neg x_i}, x_{i+1}, i^*),\\ &\qquad \mathit{RelTriangle}(A|_{x_i}, D|_{x_i}, \overline{b}|_{x_i}, x_{i+1}, i^*),\\ &\qquad \overline{b}, x_{i+1}, \mathit{min}^{i^*}).\end{aligned}$$

3. If A and D are terminal nodes,
$$\mathit{RelTriangle}(A, D, \overline{b}, x_i, i^*) = \mathit{Apply}(A, D, +),$$

where the bit vectors $\overline{b}|_{x_i}$ and $\overline{b}|_{\neg x_i}$ are obtained from the bit vector $\overline{b}$ by setting the i-th bit to 1 and 0, respectively.

***RelApply* procedure.** Let (i) A and B be two MTBDDs, (ii) $\overline{b}$ a bit vector of length $2 \cdot |V|$, (iii) x_i the least variable associable with a non-terminal node, and (iv) $\square$ a termwise position-sensitive binary operation. We define $\mathit{RelApply}(A, B, \overline{b}, x_i, \square)$ as follows:

1. If A and B are not both terminal nodes,
$$\begin{aligned}&\mathit{RelApply}(A, B, \overline{b}, x_i, \square)\\ &\quad = \mathit{newNode}(x_i,\\ &\qquad \mathit{RelApply}(A|_{\neg x_i}, B|_{\neg x_i}, \overline{b}|_{\neg x_i}, x_{i+1}, \square),\\ &\qquad \mathit{RelApply}(A|_{x_i}, B|_{x_i}, \overline{b}|_{x_i}, x_{i+1}, \square)).\end{aligned}$$

2. If A and B are terminal nodes,
$$\mathit{RelApply}(A, B, \overline{b}, x_i, \square) \quad = \quad A \ \square^{\overline{b}} \ B.$$

In addition to the alternating variable ordering, *RelTriangle* requires using the same variables for codifying both left matrix columns and right matrix rows. We can easily achieve this by transposing the right vector before applying the procedure. For computing the MTBDD representing the transpose of a vector, we can simply swap odd and even variables on the original MTBDD vector.

The first and second cases of $RelTriangle$ split the matrix and recursively apply the algorithm to both halves. The third case directly operates on terminal values: the cell values in the matrices. As the algorithm recursively descends through the MTBDD, the bit vector $\overline{b}$ carries the values of the variables leading to the terminal nodes.

Also, in the second case of $RelTriangle$, we use $RelApply$ for computing $\min^{i^*}$. This operation computes an MTBDD with terminals having minimum i^*-th component, among those already having a minimum $P(\overline{b})$-th component (see Algorithm 2).

In the first case of $RelApply$, we recursively descend through the diagram's *lo* and *hi* branches, recording the trajectory with the bit vector $\overline{b}$. When reaching the terminal nodes, in the second case, we simply apply the specified position-sensitive operator to the values.

Observe that in both procedures the x_i parameter iterates over all possible MTBDD variables, even when such variables do not occur in the given MTBDDs. These extra steps are present in the algorithm by Fujita *et al.* (1997), but not in the one by Bahar *et al.* (1997). In our case, however, these extra steps are needed for recording the exact position in the matrix when reaching the terminal nodes.

Computing a shortest-path matrix. So far, by using $RelTriangle$ we can compute the shortest path costs from every vertex to the selected destination (i.e., the minimum $dist(s)$). The last step of the process is to be able to also compute the shortest paths themselves. In our representation, we can perform this computation by simply manipulating the available matrices.

Our goal is to compute an adjacency matrix σ such that if there is a transition from vertex v to vertex v' in σ, then the transition $v \to v'$ is part of a shortest path in the given graph game. Also, in σ, the cost of transition $v \to v'$ is the cumulative cost of going from v to the given shortest-path destination.

Let A be the adjacency matrix of some graph game G. Let D be the optimal cost vector as computed by $RelTriangle$. The operation $A + D^\top$ is similar to running a partial relaxation. We add to every edge (in A) the computed approximation (in $D^\top$). The vertices increasing their over-approximation cannot be in a shortest path, as the over-approximation already converged to the optimal value. Thus, in this resulting matrix, we can prune the surpluses, setting the trimmed cells to $\overline{\infty}$. As a result, we end with an adjacency-like matrix σ having only shortest-path transitions: a shortest-path matrix.

We prune the matrix as described above with the termwise position-sensitive $BelowThreshold_i$ operation:

$$BelowThreshold_i(a, b) = \begin{cases} a & \text{if } a_i \le b_i, \\ \overline{\infty} & \text{otherwise,} \end{cases}$$

where i is the position parameter; in our case, the player's turn (given by the operands' row in the matrix).

We define the shortest-path matrix σ as follows:

$$\sigma = BelowThreshold(A + D^\top, D),$$

where the position-sensitive parameter is given by the players' turns.

Finally, we enumerate all the steps for the symbolic algorithm in Algorithm 3, where G is a graph game, d is the destination state, and i^* is a distinguished agent. Also, $initialAproximation(d)$ is the first over-approximation of the shortest-path cumulative costs for reaching d from each vertex of the graph, i.e., $initialAproximation(d)$ assigns $\overline{0}$ to the i-th vector component if vertex $v_i = d$; otherwise, it assigns $\overline{\infty}$.

Algorithm 3. Symbolic Bellman–Ford with many players and turns.

```
1: BELLMAN–FORD$(G, d, i^*)$:
2: $A \leftarrow adjacencyMatrix(G)$
3: $D \leftarrow initialAproximation(d)$

4: for $x \leftarrow 1$ to $|V| - 1$ do
5:     $D^\top \leftarrow transpose(D)$
6:     $D \leftarrow RelTriangle(A, D^\top, \overline{0}, i^*)$
7: end for

8: $D^\top \leftarrow transpose(D)$
9: $tmp \leftarrow Apply(A, D^\top, +)$
10: $\sigma \leftarrow RelApply(tmp, D, \overline{0}, BelowThreshold)$
```

4.5. Example. In this subsection we show some key steps of an extended Bellman–Ford sample execution.

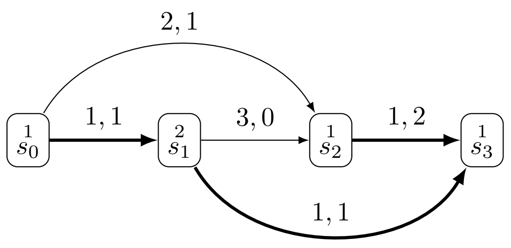
Consider the graph game G in Fig. 3(a). The adjacency matrix A and the optimal cost vector D for G are as follows:

$$A = \begin{pmatrix} 0,0 & 1,1 & 2,1 & - \\ - & 0,0 & 3,0 & 1,1 \\ - & - & 0,0 & 1,2 \\ - & - & - & 0,0 \end{pmatrix}, \quad D = \begin{pmatrix} 2,2 \\ 1,1 \\ 1,2 \\ 0,0 \end{pmatrix}.$$

For representing these matrices as MTBDDs, we must use an alternating binary codification. Recall that we use odd bits for codifying matrices' rows and even bits for codifying their columns.

In Fig. 3(b) we show A's direct binary codification, followed by A's alternating codification. Next, in Fig. 3(c) we show A's MTBDD representation. We add the optimal costs to every transition as follows:

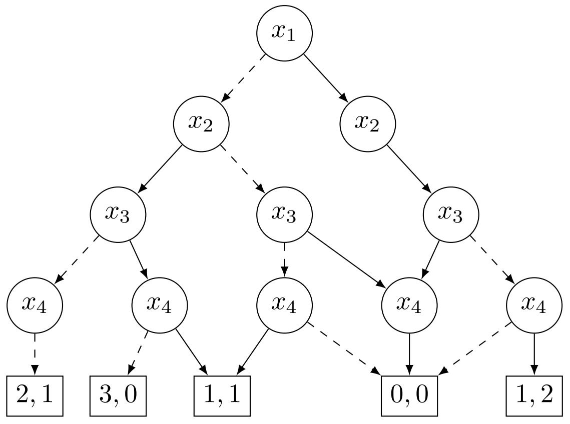
$$A + D^\top = \begin{pmatrix} 2,2 & 2,2 & 3,3 & - \\ - & 1,1 & 4,2 & 1,1 \\ - & - & 1,2 & 1,2 \\ - & - & - & 0,0 \end{pmatrix}.$$



(a)

s, s'	Binary codification	Alternating	Value
s_0, s_1	0001	0001	$1, 1$
s_0, s_2	0010	0100	$2, 1$
s_1, s_2	0110	0110	$3, 0$
s_1, s_3	0111	0111	$1, 1$
s_2, s_3	1011	1101	$1, 2$
s_0, s_0	0000	0000	$0, 0$
s_1, s_1	0101	0011	$0, 0$
s_2, s_2	1010	1100	$0, 0$
s_3, s_3	1111	1111	$0, 0$
⋮	⋮	⋮	∞, ∞

(b)



(c)

Fig. 3. Sample graph game G (a), G binary codification (b), G MTBDD representation (c).

We prune the graph game keeping only those transitions not exceeding the optimal costs:

$$\sigma = \mathit{BelowThreshold}(A + D^\top, D)$$

$$= \begin{pmatrix} 2,2 & \boxed{2,2} & - & - \\ - & 1,1 & - & \boxed{1,1} \\ - & - & 1,2 & \boxed{1,2} \\ - & - & - & 0,0 \end{pmatrix}.$$

In this example, the shortest path $s_0s_1s_3$ is marked with thick arrows in Fig. 3, and we enclose in a rectangle the cumulative costs of this path's transitions in the shortest-path matrix above. Also, note that, analogously to a subgame-perfect Nash equilibrium, we must find a shortest path from every state reaching the destination. For this reason, we also mark the transition $s_2 \to s_3$.

5. Model checking and shortest paths

One of the major applications of symbolic graph algorithms is in *model checking* (see Clarke and Emerson, 1982; Clarke *et al.*, 1986; Burch *et al.*, 1992). Briefly, model checking is a technique for automated verification of formal specifications. In a model checking approach, we use non-deterministic state-transition systems as *model* representations. In this approach, we sometimes refer to models as *Kripke structures*. Once having a Kripke model, we can use some *modal logic* for describing the model specifications. Among the plethora of modal logics there are a few common choices. Here we will focus on temporal logic, and more specifically on the *computation tree logic* (CTL). For a cogent introduction to model checking, the readers may consult the book by Baier and Katoen (2008).

In this section, we show how to use the extended Bellman–Ford algorithm in model checking. We first observe the close connection between model-checking *state-labeling* algorithm and shortest paths. Next, we show a CTL extension for describing models with weighted transitions. Finally, we show some potential applications of this approach.

5.1. CTL and state-labeling. CTL is a useful language for describing properties of discrete and branching-time, non-deterministic systems. Syntactically, CTL extends propositional logic with the following single-path temporal modalities:

- $\mathbf{X}\varphi$: the formula φ is satisfied at the next state;
- $\mathbf{F}\varphi$: the formula φ is satisfied now or at some future state;
- $\mathbf{G}\varphi$: the formula φ is globally satisfied (i.e., from now on);
- $\psi\ \mathbf{U}\ \varphi$: the formula φ is satisfied at some future state reachable by passing only through ψ-states (i.e., states satisfying ψ).

Additionally, in CTL these single-path operators must be preceded by an $\mathbf{A}$ or an $\mathbf{E}$ path quantifier. For example,

- $\mathbf{EF}\varphi$: *"there is a possible future state where the formula φ holds"*;

- $\mathbf{AG}\varphi$: *"the formula φ will always hold from now on"*.

Formally, given a set $\mathcal{P} \stackrel{\text{def}}{=} \{p, q, \ldots\}$ of atomic propositions, we construct CTL formulas, in existential normal form, according to the following grammar:

$$\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \sigma,$$
$$\sigma ::= \mathbf{EX}\varphi \mid \mathbf{EG}\varphi \mid \mathbf{E}\left[\varphi \,\mathbf{U}\, \varphi\right],$$

where $p \in \mathcal{P}$ and $\top \notin \mathcal{P}$.

We also define the other CTL operators as follows:

$$\begin{aligned}
\mathbf{EF}\varphi &\stackrel{\text{def}}{=} \mathbf{E}\left[\top \,\mathbf{U}\, \varphi\right],\\
\mathbf{AX}\varphi &\stackrel{\text{def}}{=} \neg\mathbf{EX}\neg\varphi,\\
\mathbf{AG}\varphi &\stackrel{\text{def}}{=} \neg\mathbf{EF}\neg\varphi,\\
\mathbf{AF}\varphi &\stackrel{\text{def}}{=} \neg\mathbf{EG}\neg\varphi,\\
\mathbf{A}\left[\psi \,\mathbf{U}\, \varphi\right] &\stackrel{\text{def}}{=} \neg\mathbf{E}\left[\neg\varphi \,\mathbf{U}\, (\neg\psi \wedge \neg\varphi)\right] \wedge \neg\mathbf{EG}\neg\varphi,
\end{aligned}$$

and we keep the usual definitions for the other Boolean connectives (e.g., by using the De Morgan laws).

We define a *Kripke model* as the following relational structure:

$$\mathfrak{M} \stackrel{\text{def}}{=} (S, R, \ell),$$

where

- $S \stackrel{\text{def}}{=} \{s_0, \ldots, s_m\}$ is a finite set of states;
- $R \subseteq S \times S$ is a serial accessibility relation (i.e., every state has at least one successor);
- $\ell : S \to 2^{\mathcal{P}}$ is a total function labeling the states in S with atomic propositions in $\mathcal{P}$.

The *satisfaction relation* $\models$ relates pairs $(\mathfrak{M}, s)$, $s \in S$, and formulas according to the following rules:

- $(\mathfrak{M}, s) \models \top$;
- $(\mathfrak{M}, s) \models p$ iff $p \in \ell(s)$;
- $(\mathfrak{M}, s) \models \neg\varphi$ iff $(\mathfrak{M}, s) \not\models \varphi$;
- $(\mathfrak{M}, s) \models \varphi \wedge \psi$ iff $(\mathfrak{M}, s) \models \varphi$ and $(\mathfrak{M}, s) \models \psi$;
- $(\mathfrak{M}, s) \models \mathbf{EX}\varphi$ iff there is an $s' \in S$ such that $(s, s') \in R$ and $(\mathfrak{M}, s') \models \varphi$;
- $(\mathfrak{M}, s) \models \mathbf{EG}\varphi$ iff there is an infinite path π such that $\pi[0] = s$ and $(\mathfrak{M}, \pi[i]) \models \varphi$ for all $i \geq 0$;
- $(\mathfrak{M}, s) \models \mathbf{E}\left[\psi \,\mathbf{U}\, \varphi\right]$ iff there is a path π and $i \geq 0$ such that $\pi[0] = s$, $(\mathfrak{M}, \pi[i]) \models \varphi$, and $(\mathfrak{M}, \pi[j]) \models \psi$ for all $0 \leq j < i$.

Here a path π is a sequence of states in S such that $\pi[i]$ denotes the i-th state on the sequence and, for all $i \geq 0$, $(\pi[i], \pi[i+1]) \in R$.

The model-checking *state-labeling algorithm* computes the set $Sat(\varphi)$ of all the states satisfying the formula φ. Starting with the atomic propositions and the labeling function ℓ, we induce the set $Sat(\varphi)$ in a bottom-up manner according to the following rules:

$$\begin{aligned}
Sat(p) &\stackrel{\text{def}}{=} \{s \mid p \in \ell(s)\},\\
Sat(\top) &\stackrel{\text{def}}{=} S,\\
Sat(\neg\varphi) &\stackrel{\text{def}}{=} S \setminus Sat(\varphi),\\
Sat(\varphi \wedge \psi) &\stackrel{\text{def}}{=} Sat(\varphi) \cap Sat(\psi),\\
Sat(\mathbf{EX}\varphi) &\stackrel{\text{def}}{=} Pre(Sat(\varphi)),\\
Sat(\mathbf{EG}\varphi) &\stackrel{\text{def}}{=} \text{the largest subset } S' \subseteq S \text{ such that}\\
&\qquad \text{(i) } S' \subseteq Sat(\varphi),\\
&\qquad \text{(ii) } s \in Pre(S') \text{ implies } s \in S',\\
Sat(\mathbf{E}[\psi \,\mathbf{U}\, \varphi]) &\stackrel{\text{def}}{=} \text{the smallest subset } S' \subseteq S \text{ such that}\\
&\qquad \text{(i) } Sat(\varphi) \subseteq S',\\
&\qquad \text{(ii) } s \in Sat(\psi) \text{ and } s \in Pre(S')\\
&\qquad\qquad \text{imply } s \in S',
\end{aligned}$$

where $Pre(X) = \{s \in S \mid (s, s') \in R \text{ for some } s' \in X\}$ is the preimage under R of some state set X.

We are mainly interested in the case $Sat(\mathbf{E}[\psi \,\mathbf{U}\, \varphi])$, representing reachability. For this case, we begin computing $Sat(\varphi)$. Then, we continue by iteratively accumulating the preimage until reaching a fixed point. At each iteration, we can also record the transitions made by each accumulated predecessor, thus defining a path from every state in S to a state in $Sat(\varphi)$. It is easy to prove that such paths have a minimum number of transitions.

Note that it is possible to define the preimage $Pre(X)$ in terms of the relational product of R by $\{(s, s) \mid s \in X\}$. In fact, this definition leads to an efficient symbolic implementation of Sat (see Clarke *et al.*, 1999, p. 77).

Based on the previous observations, we propose first using weighted transitions in the models, and then extending CTL for describing such models. Our CTL extension proposal employs formulas such as $\mathbf{min}[\alpha \,\mathbf{U}\, \varphi] < \mathbf{min}[\beta \,\mathbf{U}\, \varphi]$. We can use such formulas to, for example, compare the costs of α-routes against β-routes, for reaching φ-states.

In the next subsections, we pursue further these ideas. First, we consider models with weighted transitions. Next, we consider weighted models with multiple agents and turns, that is, game graphs.

5.2. CTL-with-costs. In this subsection we extend CTL with cost-comparison formulas. The purpose is to capture the behavior of models having weighted transitions. Also, for model checking, we show that it is possible to use a shortest-path algorithm as an extension to the state-labeling algorithm.

CTL-with-costs formulas are interpreted on relational structures based on weighted graphs. We define such structures as follows:

$$\mathfrak{M} \overset{\text{def}}{=} (S, R, \ell, C),$$

where

- we define S, R, and ℓ as in the previous subsection;
- the total function $C : R \to \mathbb{R}_+$ assigns costs to transitions, such that, for all $(s, s') \in R$, if $s = s'$, then $C(s, s') = 0$.

Given a set $\mathcal{P} \overset{\text{def}}{=} \{p, q, \ldots\}$ of atomic propositions, we construct CTL-with-costs formulas according to the following grammar:

$$\begin{aligned}
&\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \sigma,\\
&\sigma ::= \mathbf{EX}\varphi \mid \mathbf{EG}\varphi \mid \mathbf{E}\left[\varphi\ \mathbf{U}\ \varphi\right] \mid \zeta \bowtie \zeta,\\
&\zeta ::= c \mid \mathbf{min}\left[\varphi\ \mathbf{U}\ \varphi\right] \mid \mathbf{max}\left[\varphi\ \mathbf{U}\ \varphi\right],\\
&\bowtie ::= < \mid > \mid \leq \mid \geq \mid =,
\end{aligned}$$

where $p \in \mathcal{P}$, $\top \notin \mathcal{P}$, and $c \in \mathbb{R}_+$.

We also define the following derived cost-formulas:

$$\begin{aligned}
\mathbf{min}\left[\mathbf{F}\varphi\right] &\overset{\text{def}}{=} \mathbf{min}\left[\top\ \mathbf{U}\ \varphi\right],\\
\mathbf{max}\left[\mathbf{F}\varphi\right] &\overset{\text{def}}{=} \mathbf{max}\left[\top\ \mathbf{U}\ \varphi\right].
\end{aligned}$$

The $\mathbf{min}$ and $\mathbf{max}$ cost operators refer to the cumulative cost of the shortest and of the longest path, respectively, for reaching a state satisfying the right side of the $\mathbf{U}$ operator. For example,

- $\mathbf{max}[\mathbf{F}\varphi] < 10$: the longest (most expensive) path reaching a φ-state has a cumulative cost less than 10;
- $\mathbf{min}[\alpha\ \mathbf{U}\ \varphi] < \mathbf{min}[\beta\ \mathbf{U}\ \varphi]$: an α-based shortest path improves costs over a β-based shortest path for reaching a φ-state.

We define the *cumulative cost of a finite path* $\pi = s_0, \ldots, s_m$ as the sum

$$\mathit{Cost}(\pi) \overset{\text{def}}{=} \sum_{k=0}^{m-1} C(s_k, s_{k+1}).$$

The satisfaction relation for the CTL fragment is defined as in the previous subsection. For the new cost-comparison operators, we use the following semantics:

- $(\mathfrak{M}, s) \models \zeta \bowtie \xi$ iff $\mathit{Value}(s, \zeta) \bowtie \mathit{Value}(s, \xi)$.

  We define the Value function as follows:

- $\mathit{Value}(s, c) \overset{\text{def}}{=} c$ if $c \in \mathbb{R}_+$.
- $\mathit{Value}(s, \mathbf{min}[\psi\ \mathbf{U}\ \varphi]) \overset{\text{def}}{=} \min_\pi\{\mathit{Cost}(\pi)\}$ such that $\pi[0] = s$ and, for some $k \geq 0$, $\pi[k] \in \mathit{Sat}(\varphi)$ and $\pi[i] \in \mathit{Sat}(\psi)$ for all $0 \leq i < k$, if there exists such a π; otherwise, we define $\mathit{Value}(s, \mathbf{min}[\psi\ \mathbf{U}\ \varphi]) \overset{\text{def}}{=} \infty$.
- $\mathit{Value}(s, \mathbf{max}[\psi\ \mathbf{U}\ \varphi]) \overset{\text{def}}{=} \max_\pi\{\mathit{Cost}(\pi)\}$ such that $\pi[0] = s$ and, for some $k \geq 0$, $\pi[k] \in \mathit{Sat}(\varphi)$ and $\pi[i] \in \mathit{Sat}(\psi)$ for all $0 \leq i < k$, if there exists such a π; otherwise, we define $\mathit{Value}(s, \mathbf{max}[\psi\ \mathbf{U}\ \varphi]) \overset{\text{def}}{=} \infty$.

Note that, when maximizing a cumulative cost, if there is a path having cycles, then $\mathit{Value}(s, \mathbf{max}[\psi\ \mathbf{U}\ \varphi])$ will be ill-defined. In such cases, we assume $\mathit{Value}(s, \mathbf{max}[\psi\ \mathbf{U}\ \varphi]) = \infty$.

According to the previous definitions, extending the state-labeling algorithm requires computing the following set:

$$\mathit{Sat}(\zeta \bowtie \xi) \overset{\text{def}}{=} \{s \in S \mid \mathit{Value}(s, \zeta) \bowtie \mathit{Value}(s, \xi)\}.$$

For computing $\mathit{Sat}(\zeta \bowtie \xi)$ we must compute $\mathit{Value}(s, \zeta)$ for every s of the model. Thus, we must compute either the shortest or the longest path costs from every state to some destination.

We can compute the shortest paths by using either the Bellman–Ford algorithm or Dijkstra's algorithm. As we mentioned in Section 3.3, for computing longest paths we can simply invert the values of the cost functions. However, we must take extra care when maximizing, as Dijkstra's algorithm does not handle negative weight cycles. In such cases, we can use the Bellman–Ford algorithm as already described in Section 3.

In the following, we will assume the use of the Bellman–Ford algorithm. We will also take on the solution discussed before for maximization problems, and we will focus our presentation on the minimization problem.

The first step for using the Bellman–Ford algorithm for computing Value is to condense the destination set into a single fresh state. Given a model $\mathfrak{M} = (S, R, \ell, C)$ and two CTL-with-costs formulas ψ and φ, we define the following model:

$$\mathfrak{M}|_{\psi\mathbf{U}\varphi} \overset{\text{def}}{=} (S', R', \ell', C'),$$

where

- $S' \overset{\text{def}}{=} \{s_\varphi\} \cup ((S \setminus \mathit{Sat}(\neg\psi)) \setminus \mathit{Sat}(\varphi))$, with s_φ a fresh state not in S;

- $R' \overset{\text{def}}{=} \{(s,t) \mid (s,t) \in R$ and $s,t \in S'\} \cup \{(s, s_\varphi) \mid s \in S', t \in Sat(\varphi)$, and $(s,t) \in R\} \cup \{(s_\varphi, s_\varphi)\}$;
- $\ell'(s) \overset{\text{def}}{=} \ell(s)$ for all $s \in S'$, and $\ell'(s_\varphi) \overset{\text{def}}{=} \bigcup_{s \in Sat(\varphi)} \ell(s)$;
- $C'(s,t) \overset{\text{def}}{=} C(s,t)$ for all $s,t \in S'$, $C'(s, s_\varphi) \overset{\text{def}}{=} \min_{t \in Sat(\varphi)}\{C(s,t)\}$, and $C'(s_\varphi, s_\varphi) \overset{\text{def}}{=} 0$.

Note that we do not take into account the transitions leaving $Sat(\varphi)$, and that we always add the loop (s_φ, s_φ). The reason is that we only verify reachability properties for these sets.

The Bellman–Ford algorithm computes a map $dist(s)$ assigning to s the shortest-path cost of reaching the given destination.

By using $dist(s)$ we can compute $Value(s, \zeta)$:

- if $\zeta = c$, then $Value(s, \zeta) = c$;
- if $\zeta = \mathbf{min}[\psi \ \mathbf{U} \ \varphi]$, then $Value(s, \zeta) = dist(s)$, using the model $\mathfrak{M}|_{\psi \mathbf{U} \varphi}$ and the destination state s_φ;
- for all $s \in Sat(\varphi)$, we set $Value(s, \zeta) = 0$;
- for all s in $\mathfrak{M}$ but not in $\mathfrak{M}|_{\psi \mathbf{U} \varphi}$, we set $Value(s, \zeta) = \infty$.

5.3. Multi-agent case. We extend the approach of the previous subsection to multi-agent settings. We consider models involving a finite set of agents (or players), where each agent is associated with a cost function and is taking turns at each state. These models are basically a subclass of graph games augmented with a node labeling function.

The formulas of this new language are interpreted on relational structures:

$$\mathfrak{M} \overset{\text{def}}{=} (S, R, \ell, \{C_i\}_{i \in N}, N, P),$$

where

- we define S, R and ℓ as in the previous sections;
- for all $i \in N$, the total function $C_i : R \to \mathbb{R}_+$ assigns costs to transitions for the agent i, such that, for all $(s, s') \in R$, if $s = s'$, then $C_i(s, s') = 0$;
- the total function $P : S \to N$ assigns players' turns to states.

For the multi-agent case, we enrich the syntax of the cost-comparison operators from the previous subsection. In these enriched operators we must specify which agent cost we want to compare.

Let $\mathcal{P} = \{p, q, \ldots\}$ be a set of atomic propositions and $N = \{1, \ldots, n\}$ a finite set of agents. We specify the syntax of multi-agent CTL-with-cost in the following BNF grammar:

$$\begin{aligned}
&\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \sigma,\\
&\sigma ::= \mathbf{EX}\varphi \bigm| \mathbf{EG}\varphi \bigm| \mathbf{E}\,[\varphi \ \mathbf{U} \ \varphi] \bigm| \zeta \bowtie \zeta,\\
&\zeta ::= c \bigm| \mathbf{min}_i\,[\varphi \ \mathbf{U} \ \varphi] \bigm| \mathbf{max}_i\,[\varphi \ \mathbf{U} \ \varphi]\,,\\
&\bowtie ::= < \mid > \mid \leq \mid \geq \mid =,
\end{aligned}$$

where $p \in \mathcal{P}$, $i \in N$ and $c \in \mathbb{R}_+$.

Given a path $\pi = s_0, \ldots, s_m$ and an agent $i \in N$, we define the following cumulative cost:

$$Cost_i(\pi) \overset{\text{def}}{=} \sum_{k=0}^{m-1} C_i(s_k, s_{k+1}).$$

Note that a model $\mathfrak{M} = (S, R, \ell, \{C_i\}_{i \in N}, N, P)$ structure conforms a graph game. Consequently, we can use the same shortest path definitions from Section 2.

We define the satisfaction relation for this language similarly to the language without agents. The CTL fragment has the usual semantics, and for the cost-comparison operators we define the satisfaction relation as follows:

- $(\mathfrak{M}, s) \models \zeta \bowtie \xi$ iff $Value(s, \zeta) \bowtie Value(s, \xi)$,

where $Value$ is defined as

- $Value(s, c) \overset{\text{def}}{=} c$ if $c \in \mathbb{R}$.
- $Value(s, \mathbf{min}_i[\psi \ \mathbf{U} \ \varphi]) \overset{\text{def}}{=} \min_\pi\{Cost_i(\pi)\}$ such that for some $k \geq 0$, π is a shortest path from $\pi[0] = s$ to $\pi[k]$, $\pi[k] \in Sat(\varphi)$, and $\pi[j] \in Sat(\psi)$ for all $0 \leq j < k$, if such a π exists; otherwise, we define $Value(s, \mathbf{min}_i[\psi \ \mathbf{U} \ \varphi]) \overset{\text{def}}{=} \infty$.
- $Value(s, \mathbf{max}_i[\psi \ \mathbf{U} \ \varphi]) \overset{\text{def}}{=} \max_\pi\{Cost_i(\pi)\}$ such that, for some $k \geq 0$, π is a shortest path from $\pi[0] = s$ to $\pi[k]$, $\pi[k] \in Sat(\varphi)$, and $\pi[j] \in Sat(\psi)$ for all $0 \leq j < k$, if such a π exists; otherwise, we define $Value(s, \mathbf{min}_i[\psi \ \mathbf{U} \ \varphi]) \overset{\text{def}}{=} \infty$.

For computing $Value$, we may proceed as before by first condensing the destination set. Given some model $\mathfrak{M} = (S, R, \ell, \{C_i\}_{i \in N}, N, P)$ and a CTL-with-costs-and-agents formula φ, we define the following model:

$$\mathfrak{M}|_{\psi \mathbf{U} \varphi} \overset{\text{def}}{=} (S', R', \ell', \{C'_i\}_{i \in N}, N, P'),$$

where

- $S' \overset{\text{def}}{=} \{s_\varphi\} \cup ((S \setminus Sat(\neg\psi)) \setminus Sat(\varphi))$ with $s_\varphi \notin S$;
- $R' \overset{\text{def}}{=} \{(s,t) \mid (s,t) \in R$ and $s,t \in S'\} \cup \{(s, s_\varphi) \mid s \in S', t \in Sat(\varphi)$, and $(s,t) \in R\} \cup \{(s_\varphi, s_\varphi)\}$;

- $\ell'(s) \overset{\text{def}}{=} \ell(s)$ for all $s \in S'$ and $\ell'(s_\varphi) \overset{\text{def}}{=} \bigcup_{s \in Sat(\varphi)} \ell(s)$;
- the cost functions are as follows:
  - $C'_i(s,t) \overset{\text{def}}{=} C_i(s,t)$ for all $s,t \in S'$;
  - $C'_i(s_\varphi, s_\varphi) \overset{\text{def}}{=} 0$;
  - $C'_i(s, s_\varphi) \overset{\text{def}}{=} \min_{t \in Sat(\varphi)}\{C_i(s,t)\}$ if $i = P(s)$;
  - $C'_j(s, s_\varphi) \overset{\text{def}}{=} \min_{t \in Sat(\varphi)}\{C_j(s,t)\}$ for $i = P(s)$ and $j \neq i$;
- $P'(s) = P(s)$ if $s \in S'$, and $P(s_\varphi) = 1$.

For computing the shortest paths we use the modified Bellman–Ford algorithm shown in Algorithm 2. Here $dist_i(s)$ is the shortest-path cost from state s to destination d for the agent i, and i^* is a distinguished agent subject to the optimization.

As in the previous subsection, we define the computation of $Value(s, \zeta)$ as follows:

- if $\zeta = c$, then $Value(s,\zeta) = c$;
- if $\zeta = \mathbf{min}_i[\psi \ \mathbf{U} \ \varphi]$, then $Value(s,\zeta) = dist_i(s)$, using the model $\mathfrak{M}|_{\psi \mathbf{U} \varphi}$ and the destination state s_φ;
- for all $s \in Sat(\varphi)$, we set $Value(s,\zeta) = 0$;
- for all s in $\mathfrak{M}$ but not in $\mathfrak{M}|_{\psi \mathbf{U} \varphi}$, we set $Value(s,\zeta) = \infty$.

5.4. Examples.

5.4.1. Planning and scheduling. In this subsection, we will show how to use our single-agent CTL extension for the analysis of planning and scheduling problems. We focus on project analysis using the critical-path method (Kelley and Walker, 1959) in non-probabilistic settings (i.e., without duration uncertainties; see below).

A project consists of sequentially ordered jobs, with some of them possibly running in parallel. Each job has a duration, and some jobs may depend on the finishing of other jobs in order to begin. The critical-path method analyzes a project by finding sequences of jobs that, if not completed in time, may delay the whole project. Also, this method searches for jobs that may be delayed without affecting the total project completion.

A project may be graphically described using activity networks. These networks are acyclic weighted graphs also called *PERT graphs*. In a PERT graph, the nodes represent the completion of some jobs and are called *milestones*. The edges of the graph represent the project jobs, and we associate each job with a duration. The sequence and direction of the edges represent the dependencies between jobs. The paths that cannot be shortened without delaying the whole project are called *critical paths*. The extra time that some job may take without delaying the project is called *slack time*.

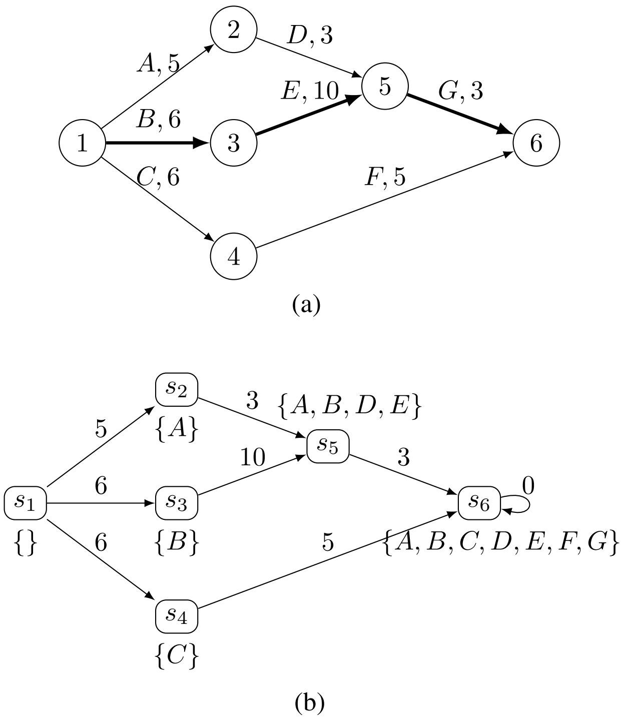


Fig. 4. Sample project's PERT graph (a) and Kripke model (b).

We can easily transform a PERT graph into a CTL-with-costs model. Figure 4(a) shows the PERT graph of a sample project. The project has seven jobs, A to G, and six milestones, 1 to 6. Each edge of the PERT graph is labeled with the job's name, and also with its duration. The critical path, $1 \to 3 \to 5 \to 6$, has a duration of 19 time units and is emphasized with thick arrows.

For converting the PERT graph in Fig. 4(a) into the CTL-with-costs model in Fig. 4(b), we can do the following steps:

- we create a state s_i for each milestone i;
- we add a transition (s_i, s_j) to R iff there is a job X taking the project from milestone i to milestone j (i.e., there is a transition $i \xrightarrow{X} j$ in the PERT graph), and we set $C(s_i, s_j) = c$ iff c is the duration of job X;
- for each job X, if there is a transition $i \xrightarrow{X} j$, then we add X to s_j labels, and also to the labels of all of the s_j descendants;
- we add the label ini to the first milestone (the beginning state), and the label end to the last milestone (the ending state);

- we set $C(s,s) = 0$ for all s, and we add (s_m, s_m) to R if m is the last milestone;
- finally, for identifying the initial and final state, we define $\mathit{ini} \overset{\text{def}}{=} \bigwedge_{X \in \mathit{Jobs}} \neg X$ and $\mathit{end} \overset{\text{def}}{=} \bigwedge_{X \in \mathit{Jobs}} X$, where Jobs is the set of all jobs.

For example, we can verify the following:

$$\begin{aligned} s_1 &\models \mathit{ini}, \\ s_6 &\models \mathit{end}. \end{aligned}$$

We can also verify the duration of the critical path:

$$s_1 \models \mathbf{max}\,[\mathbf{F}\mathit{end}] = 19$$

and of the one with the longest slack times:

$$s_1 \models \mathbf{min}\,[\mathbf{F}\mathit{end}] = 11.$$

We can also compare paths (possibly adding information to our model using atomic propositions):

$$\begin{aligned} s_1 \models \mathbf{max}\,&[(\mathit{ini} \vee \neg C)\ \mathbf{U}\ \mathit{end}] \\ &> \mathbf{max}\,[(\mathit{ini} \vee C)\ \mathbf{U}\ \mathit{end}]. \end{aligned}$$

Observe that by verifying a formula like

$$s_1 \models \mathbf{max}\,[\mathbf{F}\mathit{end}] > 0,$$

we can use the methods described in Section 4.4 for computing the critical path of the project.

For verifying the above formula, we must compute $\mathit{Value}(s_1, \mathbf{max}\,[\mathbf{F}\mathit{end}]) = \mathit{Value}(s_1, \mathbf{max}\,[\top\ \mathbf{U}\ \mathit{end}])$, as specified by the semantics of Section 5.2.

As this is a maximization problem, we invert the sign of the transition cost functions. We then use the Bellman–Ford algorithm as a matrix multiplication. The initial operands are the following adjacency matrix A and the initial over-approximation D_0:

$$A = \begin{pmatrix} 0 & -5 & -6 & -6 & \infty & \infty \\ \infty & 0 & \infty & \infty & -3 & \infty \\ \infty & \infty & 0 & \infty & -10 & \infty \\ \infty & \infty & \infty & 0 & \infty & -5 \\ \infty & \infty & \infty & \infty & 0 & -3 \\ \infty & \infty & \infty & \infty & \infty & 0 \end{pmatrix},$$

$$D_0 = \begin{pmatrix} \infty \\ \infty \\ \infty \\ \infty \\ \infty \\ 0 \end{pmatrix}.$$

We iteratively apply the matrix multiplication procedure, reaching the desired result $D = D_3$ after three iterations:

$$D_1 = \begin{pmatrix} \infty \\ \infty \\ \infty \\ -5 \\ -3 \\ 0 \end{pmatrix}, \quad D_2 = \begin{pmatrix} -11 \\ -6 \\ -13 \\ -5 \\ -3 \\ 0 \end{pmatrix},$$

$$D_3 = \begin{pmatrix} -19 \\ -6 \\ -13 \\ -5 \\ -3 \\ 0 \end{pmatrix}.$$

Before restoring the signs of the transitions, we compute the shortest-path matrix as described in Section 4.4:

$$A + D^\top = \begin{pmatrix} -19 & -11 & -19 & -11 & \infty & \infty \\ \infty & -6 & \infty & \infty & -6 & \infty \\ \infty & \infty & -13 & \infty & -13 & \infty \\ \infty & \infty & \infty & -5 & \infty & -5 \\ \infty & \infty & \infty & \infty & -3 & -3 \\ \infty & \infty & \infty & \infty & \infty & 0 \end{pmatrix}.$$

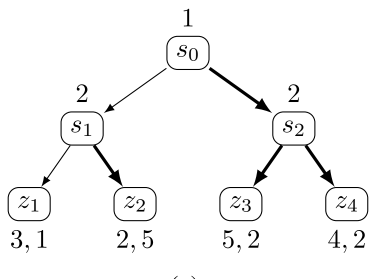
Then we apply the $\mathit{BelowThreshold}$ operation for obtaining the shortest-path matrix σ:

$$\begin{aligned} \sigma &= \mathit{BelowThreshold}(A + D^\top, D) \\ &= \begin{pmatrix} -19 & \infty & \boxed{-19} & \infty & \infty & \infty \\ \infty & -6 & \infty & \infty & \boxed{-6} & \infty \\ \infty & \infty & -13 & \infty & -13 & \infty \\ \infty & \infty & \infty & -5 & \infty & \boxed{-5} \\ \infty & \infty & \infty & \infty & -3 & \boxed{-3} \\ \infty & \infty & \infty & \infty & \infty & 0 \end{pmatrix}. \end{aligned}$$

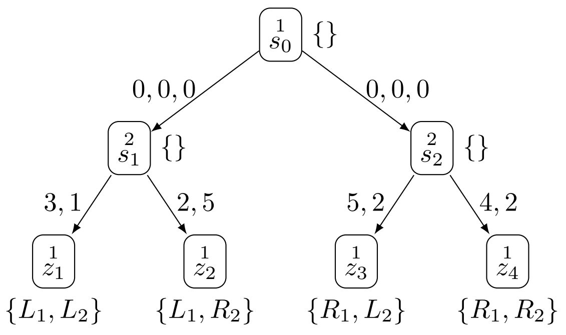
The transitions $1 \to 3 \to 5 \to 6$ of the critical path are marked with thick arrows in Fig. 4(a), and the remaining time from each milestone is framed in the shortest-path matrix above (we omit here the sign restoring step). By computing this matrix, we computed both the critical path timings and the critical path itself.

Finally, by verifying the formulas described in the present subsection with this method, we can also compute the timings of other paths with slack times, and use arithmetic subtraction to compute the slack time of particular paths or jobs.

5.4.2. Games with perfect information. Our graph games are motivated by extensive-form games with perfect information. In this section, we present a simple example showing how it is possible to use our language for computing and reasoning about backward induction solutions of such games (see, for example, the work of



(a)



(b)

Fig. 5. Three-player game (a) and its Kripke model (b).

Osborne and Rubinstein (1994) for a larger discussion about games).

Consider the extensive-form game in Fig. 5(a). For defining a model representing this game (Fig. 5(b)), we proceed as follows:

- there is a one-to-one correspondence between the tree nodes and the model states;
- we add a loop transition to every leaf z (we omit these loops in Fig. 5(b) for clarity);
- we define $C_i(s, s') = 0$ for every transition such that s' is not a leaf; and
- we define $C_i(s, z) = c$ if player i gains c utility units by reaching z.

We can further extend this model by codifying information about strategies using atomic propositions. For this game, we codify the strategies of player i as the atomic propositions L_i and R_i. By using these propositions, we characterize strategies and strategy profiles. For example,

$$\begin{aligned} z_1 &\models L_1 \wedge \neg R_1, \\ z_2 &\models L_1 \wedge \neg R_1, \\ z_4 &\models R_1 \wedge R_2. \end{aligned}$$

We can also verify the utility that the agents gain by following such strategies:

$$\begin{aligned} s_0 &\models \mathbf{max}_1[\mathbf{F}(R_1 \wedge R_2)] = 5, \\ s_0 &\models \mathbf{max}_1[\mathbf{F}(L_1)] = 2. \end{aligned}$$

Furthermore, as well as in the previous example, by model-checking a simple formula like

$$a \bowtie \mathbf{max}_1 \left[\mathbf{F}(L_1 \vee L_2 \vee L_3 \vee R_1 \vee R_2 \vee R_3)\right],$$

we can use the shortest-path matrix defined in Section 4.4 for computing the game's solution, rather than only giving a logical characterization (a and the comparator can be arbitrarily chosen, as in the previous example).

Note that the game has two subgame-perfect solutions (marked with thick arrows). One of these solutions is better for player 1. By using the operator $\mathbf{max}_1$ we can compute the better solution for agent 1.

Computing subgame-perfect solutions for perfect information games is usually done with the *backward induction algorithm* (see Osborne and Rubinstein, 1994). This computation is done in linear time in the size of the game tree, as it is equivalent to a depth-first search. Many real-life games, however, have a very large state space. Common examples of this are Chess and Go: the state space for this kind of games grows exponentially in the number of game moves (i.e., $O(2^m)$). For two-player, zero-sum games it is possible to optimize the computation with, for example, the alpha-beta pruning reducing the search space up to $O(2^{m/2})$ (see Russell and Norvig, 2003). For imperfect information games, the solution computation is even computationally harder. Gambit (McKelvey *et al.*, 2014) is a well known tool for the analysis of these games. McKelvey and McLennan (1996) review of the algorithms implemented by Gambit. There are efficient algorithms for solving games in normal (strategic) form, and it is possible to solve an extensive game by converting it to its normal form. This conversion has, however, an exponential time penalty. It is possible to solve games directly in their extensive form, but it is a computationally demanding problem (even some simple classes of games are NP-hard (see McKelvey and McLennan, 1996)).

Compared to the aforementioned methods, our approach proposes a memory-efficient game representation that can be useful for certain classes of games.

6. Experimental results

For illustrating our methods we implemented a C++ prototype of the *RelTriangle* procedure and of the shortest-path matrix computations. Our prototype implements a standard node table MTBDD representation (for some implementation details, see the work of Meinel and Theobald (1998)). We ran all tests on a machine with 4 GB of RAM.

The test cases consisted of graph games with 2^n states (i.e., having adjacency matrices of size $2^n \times 2^n$) for $n = 2, \ldots, 17$. For each n value we ran 20 different pseudorandomly generated tests.

For making a comparison, we ran the same tests for, the symbolic and the non-symbolic versions of the algorithm, both using matrix multiplication. In Fig. 6 we show the performance results of the non-symbolic algorithm and in Fig. 7 of the symbolic one.

The non-symbolic algorithm exhausted the computer's memory for all tests having $n > 13$, as most tests would require more than 4 GB of memory for storing their adjacency matrices. The biggest example we ran using the symbolic algorithm had an adjacency matrix of size $2^{17} \times 2^{17}$ and four agents. Considering that, for this example, each cell of the matrix has four floating point values, an explicit matrix representation would need 512 GB of memory (this is the current limit for a computer running Microsoft Windows 8 (see MSDN, 2013)).

The MTBDD matrix representation introduces additional computations, and the application of position-sensitive operations also requires recording a complete path to the terminal node, including the absent non-terminal nodes in a particular MTBDD (this is the reason why we extend the algorithm of Fujita *et al.* (1997) instead of the more efficient one by Bahar *et al.* (1997)). Because of this, the symbolic algorithm's memory efficiency has a time penalty. While the non-symbolic algorithm tests required only a few computational seconds, the largest tests took several hours for completion using the symbolic algorithm.

These tests are not conclusive on the efficiency of neither algorithm, and we highlight that this implementation is not intended to be industry strength. We can, however, use these results for illustrating the potential benefit of the symbolic algorithms. Although the explicit matrix representation is faster, we can see how using BDDs allows treating large problems, otherwise having expensive explicit representations.

Finally, we note that our algorithm and implementation use MTBDDs and for the general case when the cost functions have domains in $\mathbb{R}$ (or floating point numbers in their computer implementation). If an application only requires integer-valued functions, it may be possible to extend the algorithm of Sawitzki (2004) that uses ordinary BDDs with a bit vector representation of integer numbers. We leave this route for future research.

7. Conclusions

It is possible to use shortest-path algorithms for solving multi-objective discrete problems and for finding solutions of game-like network structures (see the work by Lozovanu and Pickl (2009), the surveys by Garroppo *et al.* (2010) and Tarapata (2007), and the references therein). An advantage of this approach is the possibility of using symbolic versions of these shortest-path algorithms. A symbolic algorithm is capable of treating large problems through the use of a memory-efficient data structure such as a BDD. Some shortest path-algorithms already have BDD-based symbolic implementations (for example, Bahar *et al.*, 1997; Fujita *et al.*, 1997; Sawitzki, 2004).

In this paper, we extended existing BDD procedures by Fujita *et al.* (1997) used in a symbolic Bellman–Ford implementation. Our extensions allow computing position-sensitive termwise operations. Such operations are sensitive to the position of a cell in a matrix, or, in other words, sensitive to the path leading to a terminal node in a BDD. We use these extended procedures for computing the shortest paths in graph games. Our graph games are generalizations of finite, non-cooperative games with perfect information. Thus, our symbolic algorithm is also applicable to such games, and amounts to finding all the subgame-perfect Nash equilibria. Furthermore, we also proposed using these algorithms (both the original and the extended versions, and both the symbolic and the non-symbolic variants) in CTL model checking. For this objective, we progressively presented two CTL extensions aiming at expressing graph game specifications. Finally, we reported some experiments with a prototype implementation of our extended algorithms.

The experiments we report compare the symbolic and non-symbolic versions of our algorithm. We do not provide comparisons with other methods, such as algorithms using sparse matrix multiplication, for example. The reason is that, as far as we know, such algorithms have not been extended as we have done here with BDD-based algorithms.

By using a symbolic algorithm for computing game solutions we expect to benefit both these related areas. For example, we can apply formal verification and symbolic algorithms to optimization and game-theoretic problems, or vice versa.

From these aims we can articulate some directions for further research. It would be interesting to find out whether it is possible to apply similar algorithms to other multi-objective or multi-constraint problems. We would also be interested in the application of symbolic algorithms and other formal verification techniques to other classes of games. Examples are games with imperfect information, infinitely repeated games, and cooperative games. Finally, we also suggest further

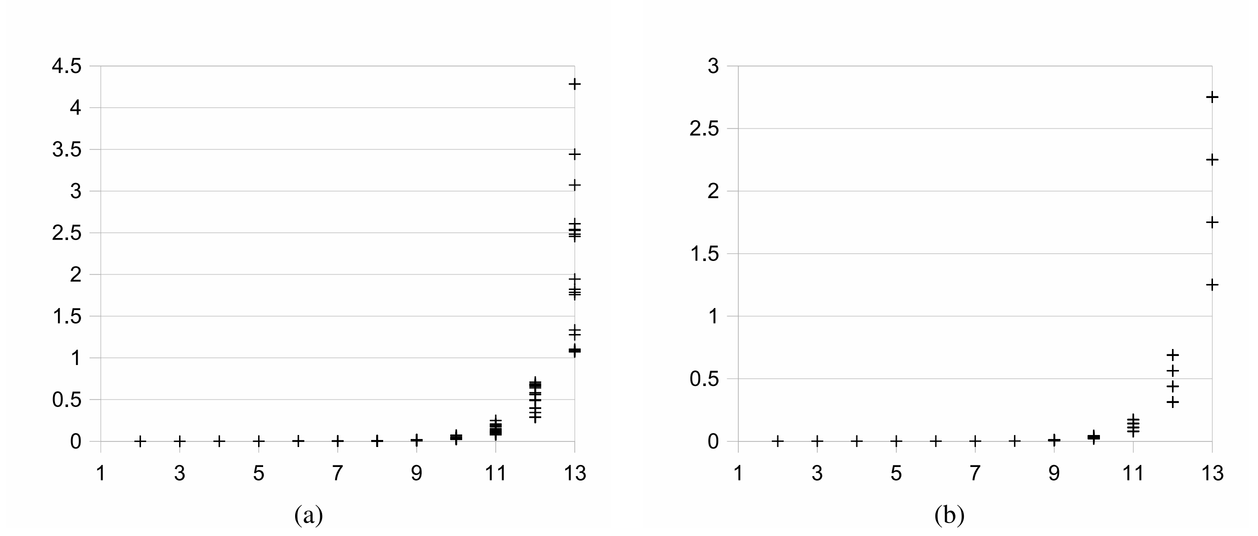


Fig. 6. Non-symbolic algorithm results: extended Bellman–Ford time in seconds (a), process resident size in GB (as measured by the Unix `time` command) (b).

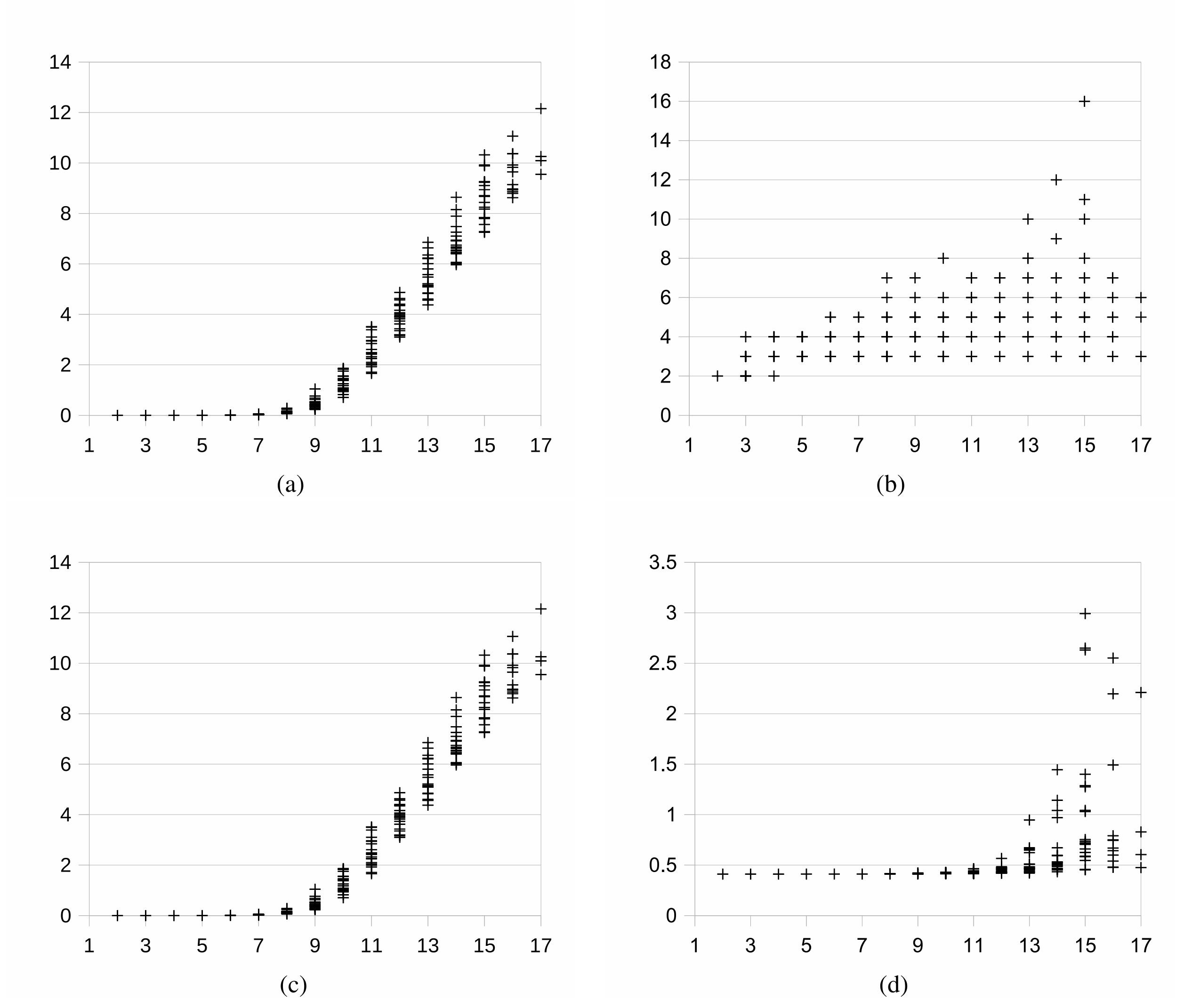


Fig. 7. Symbolic algorithm results: extended Bellman–Ford time in seconds (logarithmic scale) (a), maximum number of *RelTriangle* iterations before converging (b), extraction time, in seconds (logarithmic scale), of shortest paths from Bellman–Ford results (c), process resident size in GB (as measured by the Unix `time` command) (d).

investigating the possible benefits of using optimization algorithms and techniques in model checking and formal verification.

Acknowledgment

We would like to thank IIMAS, UNAM, for the facilities provided. Also, Pedro A. Góngora acknowledges the financial support from CONACyT and David A. Rosenblueth that from the grant PAPIIT IN113013 and Conacyt 221341.

References

Bahar, R., Frohm, E., Gaona, C., Hachtel, G., Macii, E., Pardo, A. and Somenzi, F. (1997). Algebraic decision diagrams and their applications, *Formal Methods in System Design* **10**(2–3): 171–206.

Baier, C. and Katoen, J.-P. (2008). *Principles of Model Checking*, MIT Press, New York, NY.

Berghammer, R. and Bolus, S. (2012). On the use of binary decision diagrams for solving problems on simple games, *European Journal of Operational Research* **222**(3): 529–541.

Bolus, S. (2011). Power indices of simple games and vector-weighted majority games by means of binary decision diagrams, *European Journal of Operational Research* **210**(2): 258–272.

Bonanno, G. (2001). Branching time, perfect information games, and backward induction, *Games and Economic Behavior* **36**(1): 57–73.

Bryant, R.E. (1986). Graph-based algorithms for Boolean function manipulation, *IEEE Transactions on Computers* **35**(8): 677–691.

Burch, J., Clarke, E., McMillan, K., Dill, D. and Hwang, L. (1992). Symbolic model checking: 10^{20} states and beyond, *Information and Computation* **98**(2): 142 – 170.

Clarke, E. and Emerson, E. (1982). Design and synthesis of synchronization skeletons using branching time temporal logic, *in* D. Kozen (Ed.), *Workshop on Logics of Programs*, Lecture Notes in Computer Science, Vol. 131, Springer, Berlin/Heidelberg, pp. 52–71.

Clarke, E.M., Emerson, E.A. and Sistla, A.P. (1986). Automatic verification of finite-state concurrent systems using temporal logic specifications, *ACM Transactions on Programming Languages and Systems* **8**(2): 244–263.

Clarke, E.M., Grumberg, O. and Peled, D.A. (1999). *Model Checking*, MIT Press, London.

Clarke, E., McMillan, K., Zhao, X., Fujita, M. and Yang, J. (1993). Spectral transforms for large Boolean functions with applications to technology mapping, *30th Conference on Design Automation, Dallas, TX, USA*, pp. 54–60.

Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C. (2009). *Introduction to Algorithms,* 3rd Edn., MIT Press, Cambridge, MA.

Dasgupta, P., Chakrabarti, P.P., Deka, J.K. and Sankaranarayanan, S. (2001). Min-max computation tree logic, *Artificial Intelligence* **127**(1): 137–162.

Dsouza, A. and Bloom, B. (1995). Generating BDD models for process algebra terms, *Proceedings of the 7th International Conference on Computer Aided Verification, Liège, Belgium*, pp. 16–30.

Enders, R., Filkorn, T. and Taubner, D. (1992). Generating BDDs for symbolic model checking in CCS, *Proceedings of the 3rd International Workshop on Computer Aided Verification, CAV'91, London, UK*, pp. 203–213.

Fujita, M., McGeer, P.C. and Yang, J.C.-Y. (1997). Multi-terminal binary decision diagrams: An efficient data structure for matrix representation, *Formal Methods in System Design* **10**(2–3): 149–169.

Garroppo, R.G., Giordano, S. and Tavanti, L. (2010). A survey on multi-constrained optimal path computation: Exact and approximate algorithms, *Computer Networks* **54**(17): 3081–3107.

Harrenstein, P., van der Hoek, W., Meyer, J.-J.C. and Witteveen, C. (2003). A modal characterization of Nash equilibrium, *Fundamenta Informaticae* **57**(2–4): 281–321.

Hermanns, H., Meyer-Kayser, J. and Siegle, M. (1999). Multi terminal binary decision diagrams to represent and analyse continuous time Markov chains, *in* B. Plateau, W.J. Stewart and M. Silva (Eds.), *3rd International Workshop on the Numerical Solution of Markov Chains, Zaragoza, Spain*, Prensas Universitarias de Zaragoza, Zaragoza, pp. 188–207.

Kelley, Jr, J.E. and Walker, M.R. (1959). Critical-path planning and scheduling, *Eastern Joint IRE-AIEE-ACM Computer Conference, Boston, MA, USA*, pp. 160–173.

Lozovanu, D. and Pickl, S. (2009). *Optimization and Multiobjective Control of Time-Discrete Systems*, Springer, Berlin/Heidelberg.

McKelvey, R.D. and McLennan, A. (1996). Computation of equilibria in finite games, *in* H.M. Amman, D.A. Kendrick and J. Rust (Eds.), *Handbook of Computational Economics*, Vol. 1, Elsevier, North Holland, Chapter 2, pp. 87–142.

McKelvey, R.D., McLennan, A.M. and Turocy, T.L. (2014). Gambit: Software tools for game theory, version 13.1.2, `http://www.gambit-project.org`.

Meinel, C. and Theobald, T. (1998). *Algorithms and Data Structures in VSLI Design: OBDD—Foundations and Applications*, Springer-Verlag, Berlin.

MSDN (2013). Memory limits for windows releases, *Microsoft Developer Network*, `http://msdn.microsoft.com/en-us/library/windows/desktop/aa366778(v=vs.85).aspx`.

Osborne, M.J. and Rubinstein, A. (1994). *A Course in Game Theory*, The MIT Press, Cambridge, MA.

Raimondi, F. and Lomuscio, A. (2007). Automatic verification of multi-agent systems by model checking via ordered binary decision diagrams, *Journal of Applied Logic* **5**(2): 235–251.

Russell, S. and Norvig, P. (2003). *Artificial Intelligence. A Modern Approach*, Prentince Hall, Englewood Cliffs, NJ.

Sawitzki, D. (2004). Experimental studies of symbolic shortest-path algorithms, *in* C.C. Ribeiro and S.L. Martins (Eds.), *Experimental and Efficient Algorithms*, Lecture Notes in Computer Science, Vol. 3059, Springer, Berlin/Heidelberg, pp. 482–497.

Tarapata, Z. (2007). Selected multicriteria shortest path problems: An analysis of complexity, models and adaptation of standard algorithms, *International Journal of Applied Mathematics and Computer Science* **17**(2): 269–287, DOI: 10.2478/v10006-007-0023-2.

Pedro A. Góngora received an M.Sc. degree in 2008 and a Ph.D. degree in 2014, both in computer science, at Universidad Nacional Autónoma de México, México. He has done research in applications of non-classical logics to computer science and biological systems. He has participated in the design and development of Antelope, an online model checker for gene regulatory networks. His current research activities include model checking applications to game theory.

David A. Rosenblueth graduated from the University of Victoria, B.C., Canada, in 1989. He has worked both in logic programming and in model checking. Within logic programming, he has contributions to connections between logic programs and context-free grammars, inductive logic programming, program transformation, and logic programming applied to genetic regulation. Within model checking, he has investigated the update problem, studied temporal logics applied to game theory, and applied model checking to genetic regulation. He has authored about fifty scientific publications in books, journals, and conference proceedings.

Received: 20 December 2013
Revised: 30 July 2014
Re-revised: 6 December 2014