Planning in RTS Games with Incomplete Action Definitions via Answer Set Programming

Marcello Balduccini and **Alberto Uriarte** and **Santiago Ontañón**
Computer Science Department
Drexel University
Philadelphia, PA, USA 19104
mbalduccini@drexel.edu, albertouri@cs.drexel.edu, santi@cs.drexel.edu

Abstract

Standard game tree search algorithms, such as *minimax* or *Monte Carlo Tree Search*, assume the existence of an accurate forward model that simulates the effects of actions in the game. Creating such model, however, is a challenge in itself. One cause of the complexity of the task is the gap in level of abstraction between the informal specification of the model and its implementation language. To overcome this issue, we propose a technique for the implementation of forward models that relies on the *Answer Set Programming* paradigm and on well-established knowledge representation techniques from defeasible reasoning and reasoning about actions and change. We evaluate our approach in the context of Real-Time Strategy games using a collection of StarCraft scenarios.

Introduction

Planning in Real-Time Strategy (RTS) games is a very challenging AI problem for three main reasons: (1) they have an enormous branching factor, (2) they are real-time (which means that players can perform durative actions simultaneously), and (3) complete forward models are usually not available. While there has been a recent push to address the first two problems (Chung, Buro, and Schaeffer 2005; Churchill, Saffidine, and Buro 2012; Ontañón 2013), little work exists on addressing the third, which we elaborate below and is the focus of this paper.

A *forward model* (also known as a *transition function* or as a *simulator*) is a function that, given the current game state and the actions that the players want to perform, generates the game state that results from executing those actions. Most game tree search algorithms such as *minimax* or *Monte Carlo Tree Search* (MCTS) inherently assume the existence of such forward model, without which a game tree simply cannot be generated. However, while public perfect forward models are available for classic games such as Chess or Go, they are not available for many other games, such as *StarCraft*. In those cases, a perfect forward model is available only to the game creator. Players rely on partial, often qualitative, models inferred by playing the game. Formalizing these models for automated reasoning is challenging, for example, *SparCraft* (Churchill 2013), a partial forward model for StarCraft, took several years to develop. This paper presents one step toward a practical effective formalization of forward models for complex games based on *Answer Set Programming* (ASP) (Gelfond and Lifschitz 1991; Niemelä and Simons 2000). ASP is a well-established knowledge representation framework from defeasible reasoning and reasoning about actions and change. We focus on (not uncommon) situations in which high-level qualitative information about the action effects is available. For example, we might be told by a fellow player that "if a unit A attacks another unit B, B will be killed", or that "in a combat, a larger force will defeat a smaller force." Similar statements also appear often in the game manuals. In either case, the exact details are not provided. These approaches have proved successful in a number of domains, but it is unclear whether they can be used to formalize scenarios of the complexity found in game playing and whether they yield fast enough computations. Our approach does not eliminate the challenge of having to provide a formal specification of the forward model, but aims at alleviating the complexities of the task by providing a higher-level language with good representation features.

From a technical perspective, we propose to encode the available domain knowledge using a combination of action languages (aimed at encoding the conditions and effects of actions in a compact and principled way, and capable of capturing indirect and non-deterministic effects) and defeasible statements. The defeasible statements are formalized as "defaults" (statements that are *normally* true) and their "exceptions", which allow one to define exceptions to the general rule. Defaults and exceptions provide a high-level and compact way of representing knowledge, since they are robust in the presence of exceptions and contradictions. For example, when learning to play StarCraft, the instructions might describe that the difference between units and structures is that structures do not move; however, the player will later discover that some *Terran* buildings can actually take off and move.

Although the forward models created by our approach can in principle be used with any reasoning algorithms, in this paper the planning algorithm is also formalized in ASP. This allows for a simple representation of a rather sophisticated reasoning component, which does not assume that the

action definitions are complete, and seamlessly handles defaults and exceptions – as well as indirect effects of actions.

To evaluate our approach, we use the domain of StarCraft. Specifically, we (1) created scenarios that require non-obvious planning to achieve a win, (2) compiled a natural language description of the relevant actions based on the statements used, in a preliminary experiment, by a human player to describe the scenarios to a fellow player, (3) encoded the description in ASP using an existing formalization methodology, and (4) used the formalization and a suitable planning module to solve the scenarios. The goal was to evaluate the ability of the corresponding agent to win in the scenarios in spite of the high-level knowledge provided. Although our agent is not yet incorporated into a StarCraft playing bot, and thus direct comparisons are not possible, we manually compared the strategies devised by our system with those employed by one of the bots that participated in the StarCraft AI competition.

Real-Time Strategy Games

RTS is a sub-genre of strategy games where players need to build an economy and military power in order to defeat their opponents. From an AI point of view, the main differences between RTS games and traditional board games such as Chess are: they have an enormous *branching factor*, they are *simultaneous move* games (more than one player can issue actions at the same time), they have *durative actions* (actions are not instantaneous), they are *real-time* (each player has a very small amount of time to decide the next move), they are *partially observable* (players can only see the part of the map that has been explored) and they are *non-deterministic*.

Classical game tree search algorithms have problems dealing with the large branching factors in RTS games. For example, the branching factor in StarCraft can reach numbers between 30^{50} and 30^{200} (Ontañón et al. 2013). To palliate this problem several approaches have been explored such as portfolio approaches (Chung, Buro, and Schaeffer 2005), abstracting the action space (Balla and Fern 2009), hierarchical search (Stanescu, Barriga, and Buro 2014), adversarial HTN planning (Ontañón and Buro 2015) or exploration strategies for combinatorial action spaces (Ontañón 2013). All of the previous approaches, however, share the fact that they assume that the system has access to either a forward model of the domain (in order to apply planning or game tree search), or that the system is allowed to use the actual game to run simulations (e.g., (Jaidee and Muñoz-Avila 2012)). The work presented in this paper differs in that we do not assume that the system has access to a completely defined forward model or simulator, but just to a rough definition of the effect of the actions in the game.

Answer Set Programming

Answer Set Programming enables an elegant and compact representation of key aspects of forward models, including defaults and exceptions, and what changes and does not change as an effect of the execution of actions, all of which are fundamental and non-trivial challenges in reasoning about actions and change.

ASP terms and atoms are formed as usual in first-order logic. A literal is an atom a or its strong negation $\neg a$. A *rule* is a statement of the form: $l_0 \leftarrow l_1, \ldots, l_m, \text{not } l_{m+1}, \ldots, \text{not } l_n$, where l_i's are literals. Its intuitive meaning is that a reasoner who believes $\{l_1, \ldots, l_m\}$ and has no reason to believe $\{l_{m+1}, \ldots, l_n\}$, must believe l_0. For instance, given a rule $\neg moves(X) \leftarrow is(X, structure), \text{not } is(X, terran)$ an agent will always conclude that a given structure s does not move, unless the agent has reason to believe that s is Terran[1]. A *program* is a set of rules. An *answer set* of a program Π can be informally defined as a set of non-contradictory literals that satisfies all the rules in Π (the formal semantics can be found in (Gelfond and Lifschitz 1991).

In certain situations, programs may have multiple answer sets, each encoding an *alternative, equally likely view of the world*. This feature of ASP is useful for formalizing high-level knowledge of the kind considered here.

For a convenient representation of choices, we make use of the syntax $m\{l_1, l_2, \ldots, l_k\}n$ (Niemelä and Simons 2000), where m, n are integers. The expression intuitively states that elements from $\{l_1, \ldots, l_k\}$ can be arbitrarily selected as long as their cardinality is between m and n.

ASP has been used before for other game AI tasks. For example, Thielscher showed how single-player games form the AAAI general game playing competition could be automatically translated to an ASP formalism for planning (Thielscher 2009). Smith and Mateas showed the usefulness of ASP for procedural content generation in games (Smith and Mateas 2011). Finally, Stanescu and Certicky used ASP in the context of RTS games to predict the opponent's army composition (Stanescu and Certicky 2014).

RTS Games: Representation Framework

In order to model states and action effects in RTS games, we employ *fluents*. In the context of reasoning about actions and change, fluents are first-order terms denoting the properties of interest of the domain, whose truth value typically depends upon time. For example, $in(u_1, t_2)$ may represent the fact that unit u_1 is (embarked) in transport t_2. A fluent literal is either a fluent f or its negation $\neg f$. Intuitively, fluents are *inertial*, meaning that their truth value persists over time.[2] Properties whose truth value does not depend on time are called *statics*. For example $connected(l_1, l_2)$ states that locations l_1, l_2 are connected. In this paper, we do not distinguish between statics and fluents unless necessary. The key properties used in our formalization are discussed next.

The physical layout of an RTS map is formalized as an undirected graph, whose nodes correspond to regions in the map and whose arcs define connectivity. The graph is represented by means of statics. For example, the following facts assert statics describing a layout consisting of two connected regions:

$$node(battlefield_1).\ node(bridge_1).$$
$$connected(battlefield_1, bridge_1).$$

[1] Strings with uppercase initials denote variables.

[2] More sophisticated types of fluents can be defined, but are not needed in the context of this paper.

Static is_a formalizes an ontology of node types and specifies the type of the nodes in a layout, e.g.:

$$is\_a(bridge_1, bridge).\quad is\_a(bridge, constrained\_site).$$

The latter statement formalizes the idea that a bridge is a "narrow" region. The properties of units are specified by suitable statics and fluents. The fact that a group of 8 enemy marines is on a bridge can be formalized by:

$$is\_a(enemy\_marines, marine).\quad count(enemy\_marines, 8).$$
$$alive(enemy\_marines).\quad at(enemy\_marines, bridge_1).$$
$$affiliation(enemy\_marines, enemy).$$

Elementary actions are first-order terms. In this work, we consider 4 elementary actions: $walk(u, l)$ (unit u walks to location l), $fly(t, l)$ (transport t flies to l), $load(u, t)$, and $unload(u, t)$ (respectively, for units loading on and unloading from transports). These actions are enough to capture the behavior of *marines* and *transports*, which are the only two StarCraft unit types considered in our experiments. A *compound action* is a set of elementary actions, denoting their concurrent execution.

The set of the possible evolutions of a domain is represented by a *transition diagram*, i.e., a directed graph whose nodes – each labeled by a complete and consistent set of fluent literals – represent the *states* of the domain, and whose arcs – labeled by sets of actions – describe *state transitions*. A state transition is identified by a triple, $\langle \sigma_0, a, \sigma_1 \rangle$, where σ_i's are states and a is a compound action. Finally, a sequence of transitions identifies a *trajectory* (or path) $\langle \sigma_0, a_0, \sigma_1, a_1, \ldots \rangle$ in the transition diagram.

While transition diagrams elegantly capture the evolution of domains over time, their size makes a direct representation unfeasible for anything but the simplest cases. However, transition diagrams can be compactly represented using an indirect encoding based on research on action languages (Gelfond and Lifschitz 1998). In this paper, we adopt the variant of writing such encoding in ASP – see, e.g., (Balduccini, Gelfond, and Nogueira 2000).

In the ASP encoding, the states in a trajectory are identified by integers (0 is the initial state). The fact that a fluent f speaker at a step i is represented by atom $h(f, i)$, where relation h stands for *holds*. If $\neg f$ holds, we write $\neg h(f, i)$. (For a static s, we write s and $\neg s$ respectively.) Occurrences of elementary actions are represented by an expression $o(a, i)$ (o stands for *occurs*). ASP rules (also called *laws* in this context) describe the effects of actions. An *action description* is a collection of such rules, together with rules formalizing the inertial behavior of fluents. Traditionally, laws are divided in *dynamic laws* (describing the direct effects of actions), *state constraints* (describing the indirect effects), and *executability conditions* (stating when the actions can be executed). Often, a fourth type of law – *triggers* – denotes actions whose execution is triggered directly by conditions on states. In our formalization of RTS games, we retain this classification, but to save space we do not give further details. Readers can refer to, e.g., (Gelfond 2002).

Action descriptions are the essential building block for defining the effect of the actions in a game. For example, the next three rules specify (1) a dynamic law saying that a direct effect of action $walk$ is that its actor changes location, (2) a state constraint stating that the unit loaded on a transport moves with the transport (an indirect effect of fly), and (3) an executability condition prescribing that units that are dead cannot move:

$$\begin{aligned} h(at(U, L2), S+1) \leftarrow\ & o(walk(U, L2), S), \\ & h(at(U, L1), S), \\ & connected(L1, L2). \end{aligned}$$
$$h(at(U, L), S) \leftarrow h(in(U, T), S),\ h(at(T, L), S).$$
$$\leftarrow o(walk(U, L), S),\ \neg h(alive(U), S).$$

Finally, the *inertia axiom* defines compactly and accurately the inertial behavior of fluents, which can be informally stated as "things tend to stay as they are":[3]

$$h(F, S+1) \leftarrow h(F, S), \text{not } \neg h(F, S+1).$$

An important advantage of this approach is that the task of planning can be reduced to finding answer sets of a program consisting of an action description together with a planning module. The planning module follows the Generate-Define-Test methodology from (Lifschitz 2002), which identifies, in a set of rules, a component that generates candidate solutions, one that defines what an acceptable solution is, and one that eliminates the non-solutions. In this paper, the generation of candidate plans is achieved by the single rule:

$$1\{o(A, S) : action(A)\}\mu \leftarrow step(S), \text{not } goal(S).$$

intuitively stating that between 1 and μ actions can be selected at each step, until the goal is reached. Parameter μ is specified as part of the problem. In the scenarios considered, the goal is achieved when the player-controlled marines reach the enemy base and the enemy marines are dead, formalized as:[4]

$$\leftarrow \text{not } goal.$$
$$goal \leftarrow reached(enemy\_base, S), \neg enemy\_alive(S).$$
$$\begin{aligned} reached(enemy\_base, S) \leftarrow\ & at(U, enemy\_base\_node, S),\ is\_a(U, marine), \\ & affiliation(U, player),\ alive(U, S). \end{aligned}$$

The first rule says that the goal must be achieved. The second states under which conditions the goal is achieved, and the third says that the enemy base must be reached by (at least) a unit of alive blue marines.

Formalization Methodology

In this paper we address the problem of formalizing qualitative, possibly incomplete domain knowledge in a way that allows an agent to effectively use it for solving game scenarios and to seamlessly expand it as new information becomes available, similarly to how a human might do. Our aim is to provide a demonstration that this can be accomplished by means of a combination of techniques from non-monotonic reasoning and from reasoning about actions and change. The challenge is that English descriptions are inherently vague and internally inconsistent. To capture them appropriately,

[3] The version for fluents that are false is omitted to save space.

[4] We omit the definition of $enemy\_alive$ to save space.

we rely on defeasible formalizations and on the solid theoretical foundations of the underlying formalisms.

Consider a representative subset of the statements used, in a preliminary experiment, by a human player to describe the scenarios used in our experiments to a fellow player:

1. If troops from different sides are in the same region or in two connected regions, they will fight each other until either group is killed.
2. Marines cannot fight while on a transport.
3. The larger group typically wins in a fight.
4. A group that is concentrated in a narrow region will lose even if it is larger than the enemy, since the marines can be easily picked out one by one.
5. The goal of the mission is for the marines to kill the enemy marines and reach the enemy base.

Notice that most of these statements are described in rather qualitative terms. Additionally, some of them appear to clash with each other (e.g., (1-2) and (3-4)). Yet, our choice of representation methodology allows for an effective representation of these challenging statements.

Let us consider (1-2). The first statement can be represented by an ASP rule:

$$\begin{aligned} o(fight(X,Y),S) \leftarrow\ & \\ & is\_a(X,marine), is\_a(Y,marine), \\ & affiliation(X,AX), affiliation(Y,AY), AX \neq AY, \\ & h(alive(X),S),\ h(alive(Y),S), \\ & near(X,Y,S),\ \text{not } ab(fight(X,Y),S). \end{aligned}$$

The rule relies on the definition of the notion of $near$, omitted to save space. Relation ab ("*abnormal*"), used at the end of the rule, ensures its defeasibility, intuitively stating that the conclusion will be drawn "unless there is reason to believe this to be an abnormal fight." Statement (2) can then be specified as an exception to a default:

$$\begin{aligned} ab(fight(Y,X),S) \leftarrow\ & is\_a(X,marine), is\_a(Y,marine), \\ & h(in(Y,T),S), is\_a(T,transport). \end{aligned}$$

In the presence of these two rules, a reasoning agent will tend to apply the first rule unless the second rule applies. Other special cases can be incrementally added by encoding further exceptions.

A generalization of this methodology is applied to statements (3-4). Given a suitable definition of a relation $total\_fighting\_against$, which counts the number of units fighting against an enemy from the same and connected locations, the first statement can be formalized by the rules:

$$\begin{aligned} & appl(d(1,o(kills(X,Y),S))) \leftarrow \\ & \quad o(fight(X,Y),S), \\ & \quad total\_fighting\_against(Y,T\_against\_Y,S), \\ & \quad total\_fighting\_against(X,T\_against\_X,S), \\ & \quad T\_against\_Y > T\_against\_X, \\ & \quad \text{not } ab(d(1,o(kills(X,Y),S))). \end{aligned}$$

$$o(kills(X,Y),S) \leftarrow appl(d(1,o(kills(X,Y),S))).$$

Relation $appl$ stands for "applied," i.e., the default is applied. Relation ab stands for "abnormal", i.e., the default is defeated/blocked. Intuitively, the first rule encodes (3) as a default, called $d(1,o(kills(X,Y),S))$, and states that the default is applied under the conditions stated in (3) unless the default is defeated. The second rule states that, if the default is applied, then the corresponding unit kills the other.

Statement (4) is encoded by a default named $d(2,o(kills(X,Y),S))$, shown below, and by another rule (omitted) triggering a $kills$ action:

$$\begin{aligned} & appl(d(2,o(kills(X,Y),S))) \leftarrow \\ & \quad o(fight(X,Y),S), \\ & \quad \text{not } spread\_out(Y,S), \\ & \quad h(at(X,LX),S), h(at(Y,LY),S), \\ & \quad is\_a(LY,constrained\_site), \\ & \quad \text{not } is\_a(LX,constrained\_site), \\ & \quad \text{not } ab(d(2,o(kills(X,Y),S))). \end{aligned}$$

The intuition that the second default takes precedence over the first is captured by a statement:

$$pref(d(2,o(kills(X,Y),S)),d(1,o(kills(Y,X),S))).$$

The formalization is completed by specifying that, if a more preferred default is applied, the least preferred is blocked:

$$ab(D2) \leftarrow pref(D1,D2), appl(D1).$$

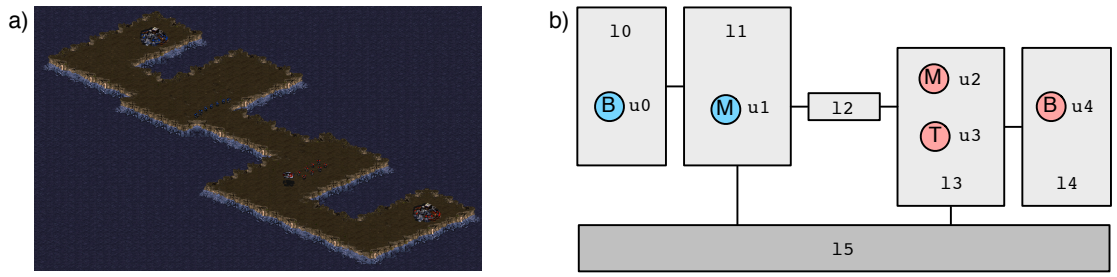
Experimental Evaluation

To evaluate our approach, we designed a collection of scenarios of increasing complexity, where one player needs to find a plan that can grant victory. The following two subsections describe first our experimental setup, including the scenarios used, and then the results obtained by our approach.

Experimental Setup

In order to assess the performance of our ASP system, we employed the following methodology:

- Each scenario consists of a StarCraft map m, where the *blue* player needs to devise a plan to destroy the base of the *red* player. Maps are set in a way that the red player has all of her defenses set up in advance, and will just wait for an attack from the blue player. The blue player is the one that will be controlled by our planner.
- We authored two versions of each scenario, one as a full-fledged StarCraft map, and one that is an automatic re-representation of such map using the ASP representation described above. Figure 1 shows an example map used in our evaluation, with its corresponding ASP version.
- Our system was then used to play each of the scenarios, and come up with a winning plan. Our planner is not yet connected to an actual StarCraft playing bot, and thus the resulting plans are not evaluated in the actual game. However, the feasibility of the generated plans was analyzed by hand by the authors. As part of our future work, we want to incorporate our ASP planner into a StarCraft bot.
- As a baseline, we employed NOVA (Uriarte and Ontañón 2012), one of the bots participating in the StarCraft AI competition (Ontañón et al. 2013). NOVA played using the actual StarCraft maps. Since comparison between NOVA and our planner cannot be done in a direct way (it would be unfair to penalize NOVA for losing a map



c)

```
node(l0).   connected(l0,l1).   h(at(u0,l0),0).  is_a(u0,base).       affiliation(u0,red).
node(l1).   connected(l1,l2).   h(at(u1,l1),0).  is_a(u1,marine).     affiliation(u1,red).
node(l2).   connected(l1,l5).   h(at(u2,l3),0).  is_a(u2,marine).     affiliation(u2,blue).
node(l3).   connected(l2,l3).   h(at(u3,l3),0).  is_a(u3,transport).  affiliation(u3,blue).
node(l4).   connected(l3,l5).   h(at(u4,l4),0).  is_a(u4,base).       affiliation(u4,blue).
node(l5).   connected(l3,l4).
                                h(count(u1,8),0).
is_a(l2,bridge).                h(count(u2,10),0).
impassable(l5).
```

Figure 1: An example map used in our evaluation: a) shows the StarCraft version, b) shows a visualization of the ASP representation, and c) shows the ASP representation itself.

just because NOVA has to deal with the actual low-level complexity of StarCraft), we analyzed the high-level plan that NOVA attempts to execute in order to win each scenario, and compare such plan against the plan generated by our planner.

In order to evaluate our approach, we used two sets of scenarios. Each set corresponds to a different type of map. For each set of scenario, we generated several specific scenarios of varying sizes (more or less units, smaller or larger maps).

- *scenario-A-k*: in these scenarios, the player and the enemy each occupy one side of a map that are connected by a narrow bridge. The enemy has one large group of marines blocking the bridge, and the player has k groups of marines spread over k regions. The player also has k transports available, each of which can carry a small group of marines directly to the enemy side, without going through the bridge (flying over an unwalkable area). Figure 1 shows an example of this scenarios (specifically, this is scenario-A-1), Figure 2.a) shows scenario-A-2 as an example. We evaluated scenarios with values of k ranging from 1 to 8. The key in these scenarios is that the number of enemy marines is slightly smaller than the number of friendly marines, but the friendly marines need to cross the narrow bridge in order to attack the enemy, and in the bridge, the enemy can pick the friendly marines one by one, thus neutralizing the numeric advantage. In order to win in this scenario, the player needs to load as many marines as possible into transports, and then drop them all at once on the left-hand side of the map, at the same time as the remaining marines attack via the bridge. In this way, the numeric advantage of the friendly marines is exerted, and the player can win the game.
- *scenario-B-k*: in these scenarios, the map consists of a series of islands connected by narrow bridges. In the k left-most islands, the enemy has groups of marines. The player marines are also distributed over a set of islands. The player has one more group of marines than the enemy, but again the bridges help the enemy cancel the numeric disadvantage. To win this scenario, the player must coordinate the attacks of its groups, and always attack the enemy from two bridges at a time. Figure 2.b) shows scenario-B-2, where we can see that the enemy has two groups of marines, and if the player wants to win, it must first attack $u2$ with $u5$ and $u4$ simultaneously, and then the surviving marines must attack $u1$ together with $u3$. Almost any other combination of attacks makes the player lose the game. We evaluated scenarios with values of k ranging from 1 to 8.

All the groups of friendly marines in all scenarios consisted of 8 marines. The key idea behind the design of these scenarios was to find maps in which there was only one or a small number of winning plans, and see whether the proposed planner was able to generate those plans, based on the encoded domain knowledge. Moreover, although we converted the scenarios to StarCraft by hand given the ASP representation, the opposite process (given a StarCraft map, generate the ASP representation) could in principle be automatically done with an algorithm to divide a map in regions, such as Perkin's (Perkins 2010), in order to incorporate our system in an actual StarCraft playing bot.

Experimental Results

The first experiment we performed is to determine whether the proposed ASP system could generate plans for the scenarios above, and what is the scalability of the system. Figure 3 shows the time in seconds the ASP system took to find winning plans for scenarios A and B for values of k ranging from 1 to 8. Notice that scenarios with $k = 8$ are rather large, since they involve 8 groups of marines with 8 marines

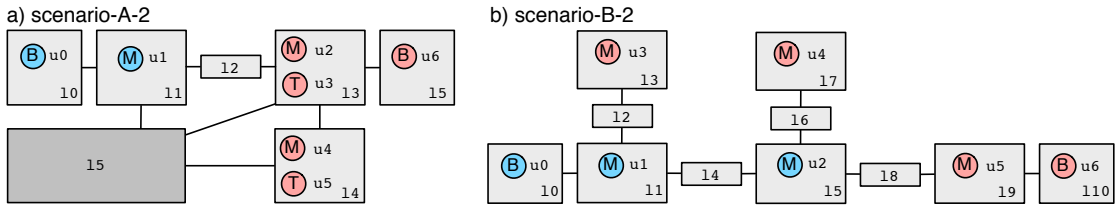


Figure 2: An example of each of the two types of scenarios used in our evaluation: a) scenario-A-2 (two regions, with two groups of marines and transports), and b) scenario-B-2 (two bridges).

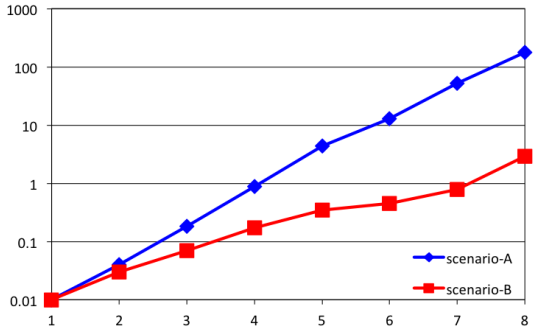


Figure 3: Time to find a winning plan in scenarios scenario-A-k, and scenario-B-k. The horizontal axis represents K, and the vertical axis the time in seconds (log scale).

each (64 friendly marines total). The resulting plans reflected the expected plan for each scenario, described above.

As the figure shows, the ASP system was able to solve all scenarios, but solution time grows exponentially with scenario size (as expected). Scenario-A-k maps take significantly more time than Scenario-B-k maps for the same value of k since the former involves transports in addition to marines, with the possibility of loading marines on to them, which increases the branching factor significantly. Considering the dynamics of a game like StarCraft, and the level of granularity at which our system is dealing with, we believe that solution times smaller than 1 second are acceptable for game play (the planner can run on the background until a plan is found). This means that in maps of the type of scenario A, our system would scale up to size $k = 4$, and on maps of the type of scenario B, we could go up to $k = 7$.

Moreover, in order to assess the performance of a StarCraft bot in these scenarios, we encoded scenario-A-1, scenario-A-2, scenario-B-1 and scenario-A-2 in StarCraft, and had the StarCraft bot NOVA play them with the following results:

- scenario-A-1 and scenario-A-2: NOVA blocks since the bridge is too narrow for the formation it desires its troops to have, and never attacks.
- scenario-B-1: NOVA wins this scenario using the right strategy by attacking the enemy marines simultaneously with two friendly marine groups. However, this strategy emerges by chance, since NOVA was in fact just trying to merge its two marine groups into one larger group, and the enemy happened to be in the middle.
- scenario-B-2 (Figure 2.b): NOVA loses this scenario, since when the combat between groups $u5$ and $u4$ versus $u2$ starts, it tries to bring $u3$ to help, by walking through $u1$, but without attacking. This results in heavy losses.

Although direct comparison between NOVA and our approach is not possible from these experiments, the interesting observation here is that, most bots participating in the StarCraft competition, like NOVA, excel in low-level combat situations, but have very limited high-level and long-term planning capabilities. As can be seen, the plans devised by NOVA to solve these scenarios are not adequate. Thus, the potential of our planning approach for increasing the planing strength of StarCraft bots is significant. Moreover, we'd like to recall that these plans were generated using only an approximate forward model.

Conclusions

This paper presented a technique for the implementation of forward models in RTS games that relies on the Answer Set Programming paradigm. Our approach aims at alleviating the complexities of defining forward models by providing a higher-level language with good representation features. We evaluated the resulting representation methodology on a collection of StarCraft scenarios, with promising results.

Our experimental results show that the resulting forward models can be used effectively to solve non-trivial scenarios. Furthermore, a simple planner built using ASP was able to satisfactorily find plans for a collection of increasingly complex StarCraft scenarios, where a state-of-the-art StarCraft bot could not. The scenarios were designed so that a plan would be hard to find and that a standard rush strategy would result in a loss.

The long term goal of our work is to design systems that can play a wide range of RTS games in the absence of a *detailed* model of the game dynamics, as humans seem to do. We plan to incorporate our planner into an actual StarCraft playing bot and fully assess its efficiency. We would also like to generalize our approach to other RTS games, and study how the amount of domain knowledge affects the quality of the plans. Additionally, our evaluation methodology did not consider an adversarial planner. Thus, in our future work, we want to experiment with the performance achievable with adversarial planners.

References

Balduccini, M.; Gelfond, M.; and Nogueira, M. 2000. A-Prolog as a tool for declarative programming. In *Proceedings of the 12th International Conference on Software Engineering and Knowledge Engineering (SEKE'2000)*, 63–72.

Balla, R.-K., and Fern, A. 2009. UCT for tactical assault planning in real-time strategy games. In *International Joint Conference of Artificial Intelligence, IJCAI*, 40–45. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Chung, M.; Buro, M.; and Schaeffer, J. 2005. Monte carlo planning in RTS games. In *IEEE Symposium on Computational Intelligence and Games (CIG)*.

Churchill, D.; Saffidine, A.; and Buro, M. 2012. Fast heuristic search for RTS game combat scenarios. In *Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*. The AAAI Press.

Churchill, D. 2013. Sparcraft: open source starcraft combat simulation.

Gelfond, M., and Lifschitz, V. 1991. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing* 9:365–385.

Gelfond, M., and Lifschitz, V. 1998. Action languages. *Electronic Transactions on AI* 3(16):193–210.

Gelfond, M. 2002. Representing Knowledge in A-Prolog. In Kakas, A. C., and Sadri, F., eds., *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*, volume 2408, 413–451. Springer Verlag, Berlin.

Jaidee, U., and Muñoz-Avila, H. 2012. CLASSQ-L: A q-learning algorithm for adversarial real-time strategy games. In *Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*.

Lifschitz, V. 2002. Answer set programming and plan generation. *Artificial Intelligence* 138:39–54.

Niemelä, I., and Simons, P. 2000. Extending the Smodels System with Cardinality and Weight Constraints. In Minker, J., ed., *Logic-Based Artificial Intelligence*. Kluwer Academic Publishers. 491–521.

Ontañón, S.; Synnaeve, G.; Uriarte, A.; Richoux, F.; Churchill, D.; and Preuss, M. 2013. A survey of real-time strategy game ai research and competition in StarCraft. *IEEE Transactions on Computational Intelligence and AI in Games (TCIAIG)* 5:1–19.

Ontañón, S. 2013. The combinatorial multi-armed bandit problem and its application to real-time strategy games. In *Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*, 58–64.

Ontañón, S., and Buro, M. 2015. Adversarial hierarchical-task network planning for complex real-time games. In *Proceedings of IJCAI 2015*, to appear.

Perkins, L. 2010. Terrain analysis in real-time strategy games: An integrated approach to choke point detection and region decomposition. In *Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*, 168–173.

Smith, A. M., and Mateas, M. 2011. Answer set programming for procedural content generation: A design space approach. *Computational Intelligence and AI in Games, IEEE Transactions on* 3(3):187–200.

Stanescu, M., and Certicky, M. 2014. Predicting opponents production in real-time strategy games with answer set programming. *IEEE Transactions on Computational Intelligence and AI in Games (TCIAIG)*.

Stanescu, M.; Barriga, N. A.; and Buro, M. 2014. Hierarchical adversarial search applied to real-time strategy games. In *Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*.

Thielscher, M. 2009. Answer set programming for single-player games in general game playing. In *Logic Programming*. Springer. 327–341.

Uriarte, A., and Ontañón, S. 2012. Kiting in RTS games using influence maps. In *Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*.