Counterplanning in Real-Time Strategy Games through Goal Recognition

Alberto Pozanco, Alejandro Blanco, Yolanda E-Martín, Susana Fernández, Daniel Borrajo
Departamento de Informática, Universidad Carlos III de Madrid
Avda. de la Universidad, 30. 28911 Leganés (Madrid). Spain

apozanco,alblanco@pa.uc3m.es, yescuder,sfarregu@inf.uc3m.es, dborrajo@ia.uc3m.es

Abstract

Real-Time Strategy (RTS) games have been widely used in AI research due to the many challenging subproblems they pose, such as goal reasoning, strategy construction, learning, etc. One of the main challenges in AI applied to RTS games is to autonomously synthesize plans that counter opponent's strategies given their observations. This task can be seen as a counterplanning problem. In this paper, we introduce an automated approach to counterplan opponents strategies in RTS. It combines; goal recognition to infer an opponent's goal; landmarks' computation to identify subgoals that can be used to block opponent's goals achievement; and classical automated planning to generate plans that prevent the opponent's goals achievement. Experimental results in StarCraft mini-games show the benefits of our novel approach.

Introduction

Games have always been interesting and great test beds for developing and trying new AI capabilities because of their well-defined set of rules, clear aims, and the possibility of evaluating results in an objective way. RTS games are a particularly difficult type of games with many analogies to real world problems. They differ from traditional board games in several aspects: they could be partially observable; their state space is usually enormous; all players make their moves simultaneously, which means that they have a small amount of time to decide the next action. These features present many challenging subproblems for AI research, such as decision making under uncertainty, collaboration, opponent modeling, or adversarial real-time planning [Buro, 2003].

Existing work on RTS games from an AI perspective can be classified according to several criteria, such as the problems it addresses, the AI techniques involved, or the level of abstraction employed. These levels of abstraction are usually divided into reactive control (micro game), and tactics and strategy (long-term or macro game). Focusing on the macro-based RTS games, players often resort to strategies, that is, sequences of actions that help them to achieve their goals. These strategies can be selected from a predefined pool or constructed from scratch.

Strategy selection is the most employed technique [Ontañón *et al.*, 2013] when deciding which strategy to follow. It is based on choosing a strategy from a set of predefined policies given the current game state. Multiple approaches for strategy selection exist, ranging from game-theoretic approaches [Sailer *et al.*, 2007; Tavares *et al.*, 2016], fuzzy rules [Preuss *et al.*, 2013], or case-based reasoning [Aha *et al.*, 2005; Jaidee *et al.*, 2011; Wender and Watson, 2014]. A common drawback of these approaches is that they must have access to a strategy set, which must be previously given by a domain expert or generated from games or simulations.

Recently, some works have explored strategy construction in RTS games [Churchill *et al.*, 2012; Uriarte and Ontañón, 2014; Stanescu *et al.*, 2014]. To deal with its huge search space, hierarchical and abstract game state representations are used along game-tree search algorithms such as *minimax* or *Monte Carlo Tree Search* to build the strategies. The drawback of these approaches is that they require heavy knowledge engineering processes to generate the abstraction levels and the goals involved in each of them. Moreover, these approaches do not take into account opponent's intentions.

The strategies can be selected or generated given the current game state. However, few works focus on predicting the opponent's movements or strategies to enhance agents' reasoning process. Weber and Mateas (2009) proposed a data mining approach to strategy prediction, Kabanza *et al.* (2010) recognize opponent's intentions using plan libraries, and Stanescu and Čertickỳ (2016) employ answer set programming to predict the units produced by the opponent. However, none of the aforementioned works are capable of observing the opponent and autonomously synthesize a good plan from scratch to counter the opponent strategy. This lack of adaptation has been identified as one of the major challenges in AI applied to RTS games [Ontañón *et al.*, 2013].

In this paper we focus on this challenge. Typically, a good strategy to win can be to prevent opponents from achieving their goals. This task has been previously named counterplanning [Carbonell, 1981], and we recently automated that pro-

cess and made it domain-independent [Pozanco *et al.*, 2018a]. This approach is based on: goal recognition, landmarks, and classical automated planning. Goal recognition aims to infer an agent's plan or goals from a set of observed actions. We use this technique to infer an opponent's goals. Fact landmarks are propositions that must be true in all valid solution plans [Hoffmann *et al.*, 2004]. We use landmarks to identify subgoals that can be used to block the opponent's goal achievement. Classical automated planning aims to generate a sequence of actions, namely a plan or strategy, which achieves some goals from an initial state. In this paper we apply this approach to RTS games. We use it to model the RTS game (actions and goals) and to generate plans that prevent the opponent's goal achievement. This approach presents some advantages over previous work: (1) it can detect opponent's intentions and strategies without resorting to plan libraries; (2) it uses this information to enhance the agent reasoning process, generating plans that counter opponents' goals achievement; and (3), it can be used in any RTS game and under any scenario, from micro (i.e., combat) to macro (i.e., building strategies) game scenarios. With respect to our previous counterplanning work, in this paper we modify the algorithm to generate counterplans in shorter time. We accomplish it by: (1) using cost estimation in the goal recognition phase; and (2) selecting any spot where the opponent can be blocked rather than reasoning about the best spot where to block it. In addition, we apply it to a set of scenarios within a RTS game.

The rest of the paper is organized as follows. In the next section we review the basic notions of classical planning, landmarks, and goal recognition. We next introduce our automated counter-planning process, followed by a description of our goal recognition approach to quickly infer opponent's goals. Then we give details about modelling a RTS game as planning. Finally we present an empirical study, discuss the results, and outline future work.

Background

Automated Planning

Automated Planning is the task of choosing and organizing a sequence of actions such that, when applied in a given initial state, it results in a goal state [Ghallab *et al.*, 2004]. Formally, a single-agent STRIPS planning task can be defined as a tuple $\Pi = \langle F, A, I, G \rangle$, where F is a set of propositions, A is a set of instantiated actions, $I \subseteq F$ is an initial state, and $G \subseteq F$ is a set of goals. Each action $a \in A$ is described by a set of preconditions (pre(a)), which represent literals that must be true in a state to execute an action, and a set of effects (eff(a)), which are the literals that are added (add(a) effects) or removed (del(a) effects) from the state after the action execution. The definition of each action includes a cost $c(a)$ (the default cost is one). The execution of an action a in a state s is defined by a function γ such that $\gamma(s, a) = (s \setminus \text{del}(a)) \cup \text{add}(a)$ if pre(a)$\subseteq s$, and s otherwise (it cannot be applied). The output of a planning task is a sequence of actions, called a plan, $\pi = (a_1, \ldots, a_n)$. The execution of a plan π in a state s can be defined as:

$$\Gamma(s, \pi) = \begin{cases} \Gamma(\gamma(s, a_1), (a_2, \ldots, a_n)) & \text{if } \pi \neq \emptyset \\ s & \text{if } \pi = \emptyset \end{cases}$$

A plan π is valid if $G \subseteq \Gamma(I, \pi)$. The plan cost is commonly defined as $c(\pi) = \sum_{a_i \in \pi} c(a_i)$. We will use the function PLANNER(Π) to refer to an algorithm that computes a plan π from a planning task Π.

Goal Recognition

Goal Recognition is the task of inferring another agent's goals through the observation of its interactions with the environment. The problem has captured the attention of several computer science communities [Geib and Goldman, 2009; Sukthankar *et al.*, 2014]. Among them, planning-based goal recognition approaches have been shown to be a valid domain-independent alternative to infer agents' goals [Ramírez and Geffner, 2009; 2010; E-Martín *et al.*, 2015; Vered and Kaminka, 2017]. Ramírez and Geffner (2010) developed an approach, which assumes that observations are actions, and formally defined a planning-based goal recognition problem as:

Definition 1 (Goal Recognition Problem) *A goal recognition problem is a tuple $T = \langle P, \mathcal{G}, O, Pr \rangle$ where $P = \langle F, A, I \rangle$ is a planning domain and initial conditions, $\mathcal{G}$ is the set of possible goals G, $G \subseteq F$, $O = (o_1, ..., o_m)$ is an observation sequence with each o_i being an action in A, and Pr is a prior probability distribution over the goals in $\mathcal{G}$.*

The solution to a goal recognition problem is a probability distribution over the set of goals $G \in \mathcal{G}$ giving the relative likelihood of each goal.

Landmarks

In Automated Planning, simple landmarks were initially defined as sets of propositions that have to be true at some time in every solution plan [Hoffmann *et al.*, 2004]. Formally:

Definition 2 (Fact Landmark) *Given a planning task $\Pi = \langle F, A, I, G \rangle$, a formula $L_\Pi \subset F$ is a fact landmark of Π iff L_Π is true in some state along all valid plans executions that achieve G from I.*

This definition was later extended to include action landmarks [Richter and Westphal, 2010]. We will use the function EXTRACTLANDMARKS(F, A, I, G) to refer to an algorithm that computes a set of landmarks $\mathcal{L}_\Pi$ from a planning task Π.

Counterplanning Process

We first formalize the two actors involved in a counterplanning problem as planning agents.

Definition 3 (Seeking agent) *A seeking agent ϕ is an agent that has an associated planning task $\Pi_\phi = \langle F_\phi, A_\phi, I_\phi, G_\phi \rangle$, and pursues its goal G_ϕ by following a plan π_ϕ computed from Π_ϕ.*

Definition 4 (Preventing agent) *A preventing agent α is an agent that has an associated planning task $\Pi_\alpha = \langle F_\alpha, A_\alpha, I_\alpha, G_\alpha \rangle$.*

G_α is initialized to $\emptyset$ and will be computed by Algorithm 1, which is described later. There can be varied relations between Π_ϕ and Π_α, and the information that one agent has from the other. For instance, the actions that both agents can perform could be the same $A_\phi = A_\alpha$, or totally different $A_\phi \cap A_\alpha = \emptyset$. They could also have different or equal observations of the world. We assume that: (1) the preventing agent knows the seeking agent's model of the world; (2) deterministic action outcomes and full observability of those actions by the preventing agent; (3) both agents stick to their plans, i.e., they do not replan or change their goals; and (4) the temporal duration of an action is determined by its cost. We use unit costs in this paper.

Since both agents operate in a common environment, the execution of their actions affects the shared environment. Therefore, we assume that any state of the environment s can be defined in terms of the set of propositions F_e ($s \subseteq F_e$), such that $F_\phi \cup F_\alpha \subseteq F_e$. That is, the state includes propositions in both F_ϕ and F_α. Those propositions in $F_\phi \cap F_\alpha$ will be observable and modifiable by both agents. In addition, the individual execution of actions by any of the two agents in F_e will be based on the respective action sets. Hence, the execution of an action a ($a \in A_\phi \cup A_\alpha$) in a state s is defined using the previous $\gamma(s, a)$. Furthermore, the joint execution of one action per agent in the same time step t can be defined as follows.

Definition 5 (Joint execution of two actions) *Given two actions $a_\phi \in A_\phi$ and $a_\alpha \in A_\alpha$ and an environment state $s \subseteq F_e$, the joint execution of both actions at a time step t results in a new state given by*

$$\gamma_{\phi,\alpha}(s, a_\phi, a_\alpha) = \begin{cases} \gamma(\gamma(s, a_\alpha), a_\phi) & \text{if } a_\phi \text{ not mutex with } a_\alpha \\ \gamma(s, a_\alpha) & \text{otherwise} \end{cases}$$

Similarly, the joint execution of two plans $\Gamma_{\phi,\alpha}(s, \pi_\phi, \pi_\alpha)$ can be defined by the iteration of the joint execution of actions of those plans using $\gamma_{\phi,\alpha}(s, a_\phi, a_\alpha)$. For simplicity, in this paper we assume that when two actions are mutex the preventing agent always executes its action first. Formally, we define two mutex actions as follows.

Definition 6 (Mutex actions) *Two actions a_x,a_y executed at a time step t are mutex if any literal in eff(a_x) deletes (adds) any literal in pre(a_y) or if any literal in eff(a_x) deletes (adds) a literal that is added (deleted) in eff(a_y).*

Using these definitions, we can now formally describe a counterplanning task.

Definition 7 (Counterplanning task) *A counterplanning task is defined by a tuple $CP = \langle \Pi_\phi, \Pi_\alpha, \mathcal{G}_\phi, O_\phi \rangle$ where Π_ϕ is the planning task of ϕ, Π_α is the planning task for the preventing agent, $\mathcal{G}_\phi$ is the set of sets of goals that ϕ can potentially pursue, and $O_\phi = (o_1, \ldots, o_m)$ is a set of observations by α of the execution of a plan $\pi_\phi = (o_1, \ldots, o_m, a_{m+1}, \ldots, a_k)$ that solves Π_ϕ.*[1]

We assume that ϕ generates a plan π_ϕ to solve its planning task Π_ϕ prior to counterplanning, and that such plan (as well as its corresponding goals) is unknown for α. Then, at some time step m of the execution of π_ϕ (where m can range from 1 to k, the length of π_ϕ), given all observed actions from the execution of π_ϕ, α has to infer the ϕ agent goals (from $\mathcal{G}_\phi$) and generate a solution to a counterplanning task, namely *counterplan*.

Definition 8 (Counterplan) *Given ϕ agent plan $\pi_\phi = (a_{m+1}, \ldots, a_k)$, a plan $\pi_\alpha = (a_1, \ldots, a_n)$ is a valid counterplan for $\pi_\phi = (a_{m+1}, \ldots, a_k)$ if the joint execution of π_α and π_ϕ does not allow ϕ to achieve the goals in G_ϕ; $G_\phi \not\subseteq \Gamma_{\phi,\alpha}(s, \pi_\phi, \pi_\alpha)$.*

Our approach to solve counterplanning tasks assumes that α can delete (or add in the case of negated literals) some proposition that ϕ *needs* in order to achieve its goals. There could be different definitions for *needed literals*. We use planning landmarks in this work. The seeking agent ϕ and the preventing agent α share some propositions, $F_\phi \cap F_\alpha \neq \emptyset$; at least one action a in α model, $a \in A_\alpha$, must delete (add) at least one of ϕ's plan landmarks. These requirements are fulfilled in RTS games, where agents typically share the same environment and have the same actions available to them.

Algorithm 1 shows the high-level algorithm used to solve a counterplanning task from the perspective of α. While no counterplan has been found, the algorithm asks for the last observed action, sending it to RECOGNIZEGOALS along with a planning domain, initial conditions, and a set of candidate goals $\mathcal{G}_\phi$. It returns T_ϕ, a set in the form of tuples $\langle$goal, number$\rangle$ and the updated initial state I_ϕ after observing o_ϕ. Next, we select the most likely goals' set $G\prime_\phi$ from T_ϕ. For each goal $g \in G\prime_\phi$, we extract the landmarks of the new ϕ planning task using EXTRACTLANDMARKS. This computation will return the set of common landmarks $\mathcal{L}_{\Pi_\phi}$ among all the most probable goals $G\prime_\phi$. If there are no common landmarks, the counterplanning task cannot be performed. Otherwise, the algorithm selects the set of counterplanning landmarks $\mathcal{L}_{\Pi_\phi,\Pi_\alpha}$ in EXTRACTCPLANDMARKS. The counterplanning landmarks are those landmarks that the preventing agent can delete (add) with its model of the world (domain). The counterplanning landmarks' set is a subset of the planning landmarks' set. As before, if there are not counterplanning landmarks, the counterplanning task cannot be performed. Otherwise, we iterate over the set of counterplanning landmarks $\mathcal{L}_{\Pi_\phi,\Pi_\alpha}$ and generate a plan to achieve that landmark for both agents (negated in the case of the preventing agent). If $c(\pi_\alpha) + t_r < c(\pi_\phi)$, it means that α can stop ϕ at that point and the plan π_α is returned. t_r stands for the needed reasoning time to generate the counterplan. Note that any of the landmarks in $\mathcal{L}_{\Pi_\phi,\Pi_\alpha}$ would be a valid goal for the preventing agent to block the seeking agent from achieving its goals. Although some reasoning processes could be applied at this point [Pozanco *et al.*, 2018a], we simply select one of them to speed-up the counterplanning process.

Cost Estimation Gradient Goal Recognition

In our previous work [Pozanco *et al.*, 2018a], we employed Ramírez and Geffner goal recognition approach [2010] to identify opponents' goals. However, this approach is often slow and therefore cannot be used to identify goals in RTS

[1] We have changed the notation o_i for a_j in the π_ϕ plan to differentiate between observations and future actions.

Algorithm 1 COUNTERPLANNING

Inputs: $\Pi_\phi, \Pi_\alpha, \mathcal{G}_\phi$
Outputs: π_α

$\pi_\alpha \leftarrow \emptyset$
while $\pi_\alpha = \emptyset$ **do**
  $o_\phi \leftarrow$GETOBSERVATION()
  $T_\phi, I_\phi \leftarrow$ RECOGNIZEGOALS$(F_\phi, A_\phi, I_\phi, \mathcal{G}_\phi, o_\phi)$
  $\mathcal{L}_{\Pi_\phi} \leftarrow F_\phi$
  $G\prime_\phi \leftarrow \text{goal}(\arg\min_{t \in T_\phi} \text{n}(t))$
  for $g \in G\prime_\phi$ **do**
    $\mathcal{L}_{\Pi_\phi} \leftarrow \mathcal{L}_{\Pi_\phi} \cap$ EXTRACTLANDMARKS$(F_\phi, A_\phi, I_\phi, g)$
  if $\mathcal{L}_{\Pi_\phi} \neq \emptyset$ **then**
    $\mathcal{L}_{\Pi_\phi,\Pi_\alpha} \leftarrow$EXTRACTCPLANDMARKS$(\Pi_\phi, \Pi_\alpha, \mathcal{L}_{\Pi_\phi})$
    if $\mathcal{L}_{\Pi_\phi,\Pi_\alpha} \neq \emptyset$ **then**
      $I_\alpha =$UPDATE$(I_\alpha, A_\alpha, o_\phi)$
      for l_i **in** $\mathcal{L}_{\Pi_\phi,\Pi_\alpha}$ **do**
        $\pi_\phi \leftarrow$ PLANNER$(\Pi_\phi = \langle F_\phi, A_\phi, I_\phi, l_i \rangle)$
        $\pi_\alpha \leftarrow$ PLANNER$(\Pi_\alpha = \langle F_\alpha, A_\alpha, I_\alpha, \neg l_i \rangle)$
        if $c(\pi_\phi) >= c(\pi_\alpha + t_r)$ **then**
          return π_α

games, where the actions are performed almost instantly. This is the reason why we propose a lighter planning-based goal recognition approach based on cost estimation gradient. It is described in Algorithm 2. The algorithm takes as inputs the current seeking agent's problem and domain (F, A, I), a set of hypothesis $_\mathcal{G}$, and the last observation (o). It estimates the cost of achieving each of the candidate goals, both from the previous input state I and the current state given the last observed action I'. This cost estimation can be performed using any heuristic or actually solving the problem and obtaining a plan. We compute this cost in parallel for each $g \in \mathcal{G}_\phi$ to better scale-up. The algorithm returns a list containing the cost estimation difference Δ_c between the new and the previous state for all the goals in $\mathcal{G}$.

Algorithm 2 COST ESTIMATION GRADIENT GR

Inputs: $F, A, I, \mathcal{G}, o$
Outputs: Δ_c, I'

$\Delta_c \leftarrow \emptyset$
$I' \leftarrow$UPDATE(F, A, I, o)
for $g_i \in \mathcal{G}$ **do**
  $c_{g_i} \leftarrow$ ESTIMATECOST(F, A, I, g_i)
  $c\prime_{g_i} \leftarrow$ ESTIMATECOST(F, A, I', g_i)
  INSERT$(c'_{g_i} - c_{g_i}, \Delta_c)$
return Δ_c, I'

An example of this process is shown in Figure 1. In this case, an agent (represented as a triangle) might want to achieve G1 or G2. From its initial state, the ESTIMATECOST() function returns $\langle 6, 6 \rangle$ for G1 and G2 respectively. After observing the first action of the agent, the function returns $\langle 5, 7 \rangle$, being $\langle -1, 1 \rangle$ the value of Δ_c returned by our goal recognition approach. We will refer to it as the function RECOGNIZEGOALS$(F, A, I, \mathcal{G}, o)$.

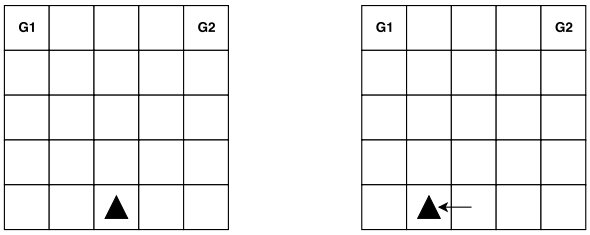

Figure 1: Goal recognition example. The agent is depicted by a triangle. Its goal can be either achieve G1 or G2.

Modelling StarCraft as Planning

StarCraft is a military science fiction real-time strategy video game developed and published by Blizzard Entertainment. The objective of a standard *StarCraft* match is to defeat your opponents by eliminating their units and buildings. Although there are different races featured in the game, for simplicity we will only take into account the *Terran* race. The game-play of *StarCraft* is based on resource management and base construction. These elements are the key to create an army that can overpower the enemy and, therefore, succeed in the game. A player relies on two kind of basic resources: minerals and vespene gas. The first one is necessary to build new structures and produce new units. The second one is needed for more advanced units and structures. Both of the resources can be collected by a worker unit from the nodes placed in the map and be brought to the main building. The resource is stored and becomes available for its use. However, the vespene gas can only be collected from the source (a vespene gas geyser) by building an extraction structure on top of it (for the Terran race, this structure is called a refinery). Different buildings or units require different amounts of minerals or vespene gas to be produced. When the sufficient amount has being collected, the player can initiate the creation process, which takes a specific time to be completed depending on the kind of structure or unit. Once the units with attack capability have been created, the player can attack the enemy. In the experiments, we consider that all the actions are executed with cost (time) equal to one.

In a standard *StarCraft* match, the most of the map is hidden to the player by the "fog of war". This feature does not let the player know the position of the enemy until that area has been explored. The areas where there is a player's building or unit are revealed only to this player. For purposes of this work, the fog of war has been disabled during the experiments, since the agent need total visibility of the actions of the enemy to produce a counter plan.

We have reduced the standard match to a set of mini-games in order to simplify the complexity of a whole *StarCraft* match. In these mini-games, both players will be represented only by one unit and the objective of each mini-game will be different. In order to use the counter-planning technique, we need to translate the features and actions of *StarCraft* to a planning domain and the state of the game to a planning problem.

For the planning domain, we need to replicate the actual game dynamics using a set of objects, predicates, and actions.

We use the objects *tile* and *building*. The object *tile* is used to identify the different positions in the actual map. Each tile corresponds to a *TilePosition*, as it is called in *StarCraft*. The object *building* is used to identify the different structures that can be constructed during the game. For that purpose, we define a different type of object *building* for every type of structure in the game. The defined predicates are shown below:

- `connected`: this predicate represents the connection between tiles in the map.
- `at`: this predicate indicates the position of a unit. We have two predicates of this kind to identify both units in the mini-game: "`at`" and "`at-enemy`".
- `empty`: indicates if a tile is empty.
- `at-vespene`: indicates the position of a vespene gas geyser.
- `at-mineral`: indicates the position of a mineral field.
- `at-`*`building`*: indicates the position of a structure. There is a predicate of this kind for each type of structure in the game and for both players. As an example, we define: "`at-`*`barracks`*", "`at-`*`barracks`*`-enemy`".
- `carrying-mineral`: indicates if a unit carries a chunk of mineral. We have two predicates of this kind to indentify both units in the mini-game: "`carrying-mineral`" and "`carrying-mineral-enemy`".
- `carrying-vespene`: indicates if a unit carries vespene gas. We have two predicates of this kind to indentify both units in the mini-game: "`carrying-vespene`" and "`carrying-vespene-enemy`".
- `have-mineral`: indicates that a unit has stored mineral. We have two predicates of this kind to indentify both units in the mini-game: "`have-mineral`" and "`have-mineral-enemy`".
- `have-vespene`: indicates that a unit has stored vespene gas. We have two predicates of this kind to indentify both units in the mini-game: "`have-vespene`" and "`have-vespene-enemy`".
- `alive`: indicates that a unit is alive. We have two predicates of this kind to identify both units in the mini-game: "`alive`" and "`alive-enemy`".
- *`building`*`-built`: indicates that a certain kind of structure has been built. There is a predicate of this kind for each type of structure in the game and for both players. As an example we define: "*`refinery`*`-built`", "*`refinery`*`-built-enemy`".
- *`unit`*`-trained`: indicates that a certain kind of unit has been created. There is a predicate of this kind for each type of unit in the game: "*`marine`*`-trained`", "*`firebat`*`-trained`", ...

Along with these predicates, a set of planning actions have been defined to represent all the possible actions that a unit can perform during the game. These actions are:

```
(:action move
 :parameters (?x1 - tile ?x2 - tile)
 :precondition (and (at ?x1)
                    (connected ?x1 ?x2)
                    (empty ?x2))
 :effect (and (not (at ?x1))
              (at ?x2)
              (not (empty ?x2))
              (empty ?x1)))
```

Figure 2: Example of the move action in the PDDL format. It moves the friendly unit from one tile to another.

- `move`: a unit moves between two tiles that are connected. A PDDL representation of this action is shown in Figure 2.
- `attack-unit`: a unit attacks an enemy's unit. We indicate a tile, and our unit attacks from that tile in its range of attack.
- `attack-structure`: a unit attacks a certain structure. There is an action of this kind for each type of structure in the game due to its different sizes.
- `gather-`*`resource`*: a unit gathers a resource from a source. It is represented through two actions: "`gather-`*`mineral`*" and "`gather-`*`vespene`*". This is due to the differences between the two actions. Minerals can be extracted directly from the source, but in order to gather vespene gas, a refinery must be built on top of the vespene geyser beforehand.
- `store-`*`resource`*: a unit, which is currently carrying a resource, stores it at the command center.
- `take`: a unit takes a chunk of resource present in the map. These types of objects have been placed on the map for some of the mini-games.
- `build-`*`structure`*: a unit builds a new structure in a place with enough space in the map. There is an action of this kind for each type of structure in the game. E.g. "`build-`*`refinery`*", "`build-`*`academy`*", ...
- `train-unit`: creating a new unit. There is an action of this kind for each type of unit in the game. E.g. "`train-`*`marine`*", "`train-`*`firebat`*", ...

In this paper we assume deterministic action outcomes. That is, the only changes in the environment state are the ones described by the effects of the actions in the planning domain model. For example, in the `attack-unit` action we assume that the other unit dies after the attack.

We need to know the state of the game and translate it to a *PDDL* file to define the planning problem. We get this information at each time step using the Brood War Application Programming Interface (BWAPI). BWAPI lets us read all the relevant information about the game state, like the organization of the map or the positions of our own units, the opponent units, and other units or structures that appear on the map. This information is saved into a *PDDL* file defining the problem and into two other files that contain the observations

about the opponent's units and the possible goals that the opponent could be trying to achieve in the future.

In order to create the problem, we start detecting all of the *TilePositions* that form our map for the mini-game. We use tiles that units cannot step in like water or high ground to model the shape of our custom maps for the mini-games. Every tile is named using its coordinates in the game. Then, we write the state of our players (`alive` by default), and the positions of each player unit and every building. A unit only occupies one tile, but a building needs more space, so it is represented by several `at-building` predicates. Then we proceed to specify the state of every tile, indicating as empty the tiles that are not occupied by unit or a building, and declaring the connections among them. The predicate `connected` is uni-directional, so we need two predicates `connected` for each pair of tiles to represent that they are fully connected in both directions. We only consider horizontal and vertical connections to simplify the complexity of the problem.

Finally, we identify the different goals that the enemy could be targeting, such as taking the chunks of mineral, attacking one of our buildings, or creating a certain type of structure or unit.

Evaluation

We have created five different StarCraft mini-games to test our approach. Each of the games is slightly more difficult than the previous one and introduces some variants, such as resources gathering, building construction, or troops confrontations. In all the mini-games there are two players, ϕ and α, which handle Terran units. The seeking agent has a precomputed plan π_ϕ that it will follow to achieve its actual goal G_ϕ. This goal is hidden for the preventing agent. The mini-games finish when the seeking player has achieved its initial goal, or if the preventing player stops it. At each mini-game we measure:

- $|\mathcal{G}_\phi|$: number of goals in the candidate goal's set.
- $|\mathcal{L}_{\Pi_\phi}|$: number of common landmarks for the seeking agent planning task.
- $|\mathcal{L}_{\Pi_\phi,\Pi_\alpha}|$: number of founded counterplanning landmarks.
- $\%Obs$: percentage of observed actions from the total ϕ's plan needed to perform the goal inference.
- Q: fraction of times that the actual goal G_ϕ was found to be the most likely goal $G\prime_\phi$. $Q = 1$ indicates that the opponent's goal was correctly inferenced.
- E: fraction of times that α, executing π_α, succeeds in stopping ϕ in achieving its goals. Ideally, $E = 1$.
- Pe: penalty value computed as the number of steps in π_ϕ that have been performed before its actual goal becomes unreachable, divided by the length of π_ϕ. It represents the cost paid by α at each time step that does not stop ϕ. Lower values of Pe indicate better performance, ideally $Pe = 0$.
- T_Q: time in seconds taken for solving the goal recognition problem. It is accumulated each iteration that a counterplan cannot be generated.

Figure 3: Take the gem mini-game. $g1$ and $g2$ indicate the two seeking's possible goals. The actual goal of the seeking agent is to take the gem at $g1$.

- $T_\mathcal{L}$: time in seconds taken for computing the landmarks.
- T_E: total time in seconds taken for producing a counterplan.
- $|\pi_\phi|$: number of actions in the seeking agent's plan.
- $|\pi_\alpha|$: number of actions in the generated counterplan.

Since *StarCraft* runs in Windows and the planners run in Ubuntu, we communicate both parts of the architecture via a TCP/IP connection. The StarCraft part of the architecture runs on a Windows 10 machine with Intel Core i5-4210U running at 1.7 GHz. The counterplanning algorithm runs on an Ubuntu machine with Intel Core 2 Quad Q8400 running at 2.66 GHz. We use the LMCUT heuristic [Pommerening and Helmert, 2013] for cost estimation and the PROBE planner [Lipovetzky *et al.*, 2014] for plans computation. The next subsections describe the mini-games and the results obtained.

Take the Gem

The first mini-game takes places in a 64x20 map. Each player only controls one unit. The seeking agent is located at the bottom of the map, while the preventing agent is located at the top side. There are two possible goals ($|\mathcal{G}| = 2$) that ϕ may want to achieve: the top-left gem g_1 or the top-right gem g_2. This initial situation is depicted in Figure 3.

The actual ϕ's goal is to achieve $g1$. After its first movement, α is able to correctly guess it and start to compute the involved landmarks. There are three landmarks: `at-enemy s31-18`, the initial location of the seeking agent, which is always a landmark; `at-enemy s3-3`, the position where the enemy needs to be in order to achieve g_1; and `at-target s3-3`, the position of the gem. From this set of landmarks, the only counterplanning landmark that the preventing agent can falsify is `at-target s3-3`, given that there is an action in its domain that deletes the `at-target` predicate. After the costs calculation to ensure that it can reach that landmark before the opponent does it, `not (at-target s3-3)` becomes preventing's goal, generating a counterplan π_α to achieve it. Figure 4 shows the moment when α takes the gem before its opponent. The results for this experiment as well as the others are shown in Table 1.

Gathering or Attacking?

The second mini-game takes place in a 64x64 map with Y shape as depicted in Figure 5. There are two mineral fields, one at the top-right and other at the top-left side of the map; the latter is the actual goal of ϕ. There are two command centers: one at the top (preventing agent) and another one at the bottom (seeking agent). The preventing agent thinks that the

Mini-game	$\|\mathcal{G}_\phi\|$	$\|\mathcal{L}_{\Pi_\phi}\|$	$\|\mathcal{L}_{\Pi_\phi,\Pi_\alpha}\|$	$\%Obs$	Q	E	Pe	T_Q	$T_\mathcal{L}$	T_E	$\|\pi_\phi\|$	$\|\pi_\alpha\|$
Take the gem	2	3	1	2.2	1	1	0.8	1.8	1.9	4.8	44	30
Gathering or attacking?	3	6	3	1.4	1	1	0.2	1.1	1.7	3.4	70	12
Gathering in the maze	3	3	1	79.5	1	0	∞	19.2	1.5	-	78	-
Base construction	2	11	3	1.9	1	1	0.6	12.7	2.7	24.4	51	9
Training army	2	6	2	2.6	1	1	0.5	3.1	2.2	4.3	75	30

Table 1: Table that summarizes the results obtained in the five mini-games.

Figure 4: Moment when the preventing agent takes the gem before its opponent.

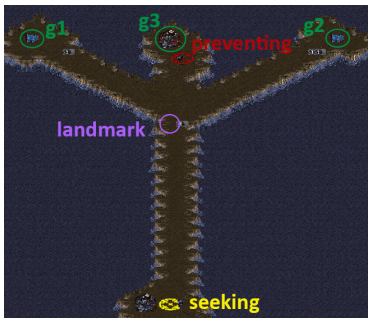

Figure 5: Resource gathering or attacking mini-game. The actual goal of the seeking agent is to gather the resources at $g1$. The landmark point indicates where the preventing's unit will stand to attack seeking's unit and block its goal achievement process.

opponent may want to gather resources from any of the mineral fields or attack its command center. The seeking agent running our counterplanning algorithm is able to find a common landmark that blocks the achievement of any of the candidate goals. So a counterplan is generated to send a unit to the bottle-neck spot placed in the middle of the Y.

Gathering in the Maze

The third mini-game takes place in a 64x64 maze-like map as depicted in Figure 6. In this case, there are three mineral fields placed in the map, one at the top-left, one on the right and one almost at the center of the map. As before, the seeking agent is placed at the bottom and the preventing agent at the top of the map. A set of bottleneck spots have been also placed around the map to allow blocking the path of the enemy. The preventing agent thinks that the goals that the seeking agent is pursuing are gathering mineral from one of the mineral fields.

Figure 6: Resource gathering in a maze mini-game. The actual goal of the seeking agent is to gather the minerals at $g2$.

In this experiment, the preventing agent is not able to find a common landmark to block the enemy's in advance, since the shape of the maze allows the seeking agent going through two different paths to any point of the map. If the goal is the mineral field at the center of the map, the preventing agent cannot infer this until the seeking agent has completed almost 80% of its whole plan. That is when the seeking agent has reached the spot at the middle part on the left of the map and begins to go right. At that time, the preventing agent cannot make it on time to block it.

Base Construction

The fourth mini-game takes place in a 64x26 map as depicted in Figure 7. There is one mineral field placed on the right and one vespene geyser placed at the top-left of the map. The positions of the two agents are the same as in the previous experiments. There are also a command center and a barrack for the seeking agent, which lets it build the structures that we are considering as possible goals for this experiment: building a supply depot or a factory. When the seeking agent starts to go to the left, the preventing agent considers that it is going to the vespene geyser placed on top and tries to counterplan by blocking its opponent way to the bottleneck point, which is a landmark. The preventing agent believes that the seeking agent's final goal is to build a factory. In order to achieve that, the seeking agent needs to build a refinery on top of the vespene geyser, extract the vespene, store it in the command center and, finally, build the factory. The preventing agent gets to the landmark point when the seeking agent is about to reach the same point, blocking its way. The seeking agent has only completed 1.9% of its plan.

Training Army

The fifth mini-game takes place in the exact same map that was used for the fourth experiment as depicted in Figure. 8.

Figure 7: Base construction mini-game. The actual goal is to build a factory. The landmark point indicates where the preventing agent is going to stand to block the way and attack.

Figure 8: Training army mini-game. The preventing agent detects a common landmark for both goals: destroying the barracks. The actual goal is to train a firebat.

For this experiment we have also placed an academy for the seeking agent, which allows it to train different kinds of units. This time, the possible goals that the seeking agent might be pursuing are training a marine or training a firebat. For the first goal, it needs to gather mineral, store it in the command center, and then train the marine using the barracks. For the second goal, the seeking agent needs also to gather and store vespene gas. Independently of the seeking agent's actions, the preventing agent will figure out that the common landmark that can invalidate both goals is to destroy the barracks where the units are trained. The preventing agent attacks the barracks when the seeking agent has only completed 2.6% of its plan.

Discussion and Future Work

In this paper we have presented a technique that addresses one of the major challenges on AI applied to RTS games: synthesize plans from scratch to oppose the opponent strategy. We have introduced two main modifications to our previous domain-independent counterplanning algorithm to properly work in a real time environment: (1) use cost estimation rather than plan computation in the goal recognition phase; and (2) select any spot where the opponent can be blocked rather than reasoning about the best stop where to block it. We have tested our approach in five different StarCraft mini-games. Some of the objectives of the mini-games include resource gathering, troop confrontations and building order, which are part of a real StarCraft game.

Experimental results show that our algorithm is able to produce counterplans in most of these situations. Moreover, our approach is not only able to guess the opponent goal soon (low $\%Obs$); it also reasons about intermediate spots (landmarks) where the opponent could be blocked. In fact, most of the times the preventing agent blocks its opponent when it has performed only a part of its plan ($Pe < 1$). It is also important to note that in our experiments, we always guessed the opponent's goal right ($Q = 1$) and most of the times we needed a low percentage of observations. These results show that our counterplanning algorithm can effectively generate plans to counter opponent's strategies. However, much work remains to be done in order to apply counterplanning in a complete StarCraft game:

- Our algorithm scales poorly with the size of the maps. All of our experiments were ran in 64x64 maps (or even subsets of them), while an average StarCraft map is twice the size. We will abstract the tiles generating meta-tiles in order to better scale-up to larger maps.
- Although the seeking agent's plans π_ϕ are quite large, they have a low number of landmarks. As the fourth experiment shows, the more landmarks it detects, the longer it will take it to extract them and generate a counterplan (larger T_E's value). When producing counterplans for long-time opponent's strategies, many landmarks can be found and our technique would delay too much. Some improvements on the landmark's extraction process should be applied to save up time.
- The presented mini-games are designed so that the preventing agent can stop its opponent most of the times ($E = 1$). α stands still observing ϕ's actions, so it is losing time while its opponent is actually moving. This fact, along with the average counterplanning time T_E of more than 4 seconds, make only possible to stop the opponent at some specific circumstances. The counterplanning results are good when: (1) the preventing agent can guess its opponent goal with a low percentage of observations; and (2) the preventing agent is closer to the goal or a landmark ($|\pi_\alpha|+T_E < |\pi_\phi|$). On the other hand we have the maze experiment, where α can only infer the goal (or find a common landmark) after the 80% of the opponent plan has been executed. In this case, it does not make it on time to prevent the seeker from achieving its goal. At this point some improvements can be done, as start moving to more promising spots while observing the opponent rather than standing still; or performing some kind of anticipatory planning [Pozanco *et al.*, 2018b; Fuentetaja *et al.*, 2018]. This will allow the preventing agent to stop its opponent more often.
- Finally, in the presented experiments we assume that the opponent neither replans nor changes its goals. This is a strong assumption that we will delete in the future by continuously monitoring and reasoning about the opponent intentions.

Acknowledgements

This work has been partially supported by MINECO projects TIN2014-55637-C2-1-R and TIN2017-88476-C2-2-R.

References

[Aha *et al.*, 2005] David W Aha, Matthew Molineaux, and Marc Ponsen. Learning to win: Case-based plan selection in a real-time strategy game. In *International Conference on Case-Based Reasoning*, pages 5–20. Springer, 2005.

[Buro, 2003] Michael Buro. Real-time strategy games: A new AI research challenge. In *IJCAI*, 2003.

[Carbonell, 1981] Jaime G Carbonell. Counterplanning: A strategy-based model of adversary planning in real-world situations. *Artificial Intelligence*, 16(3):295–329, 1981.

[Churchill *et al.*, 2012] David Churchill, Abdallah Saffidine, and Michael Buro. Fast heuristic search for rts game combat scenarios. In *AIIDE*, pages 112–117, 2012.

[E-Martín *et al.*, 2015] Yolanda E-Martín, Maria D R-Moreno, and David E Smith. A fast goal recognition technique based on Interaction estimates. In *IJCAI*, 2015.

[Fuentetaja *et al.*, 2018] Raquel Fuentetaja, Daniel Borrajo, and Tomás de la Rosa. Anticipation of goals in automated planning. *AI Communications*, pages 1–19, 2018.

[Geib and Goldman, 2009] Christopher W Geib and Robert P Goldman. A probabilistic plan recognition algorithm based on plan tree grammars. *Artificial Intelligence*, 173(11):1101–1132, 2009.

[Ghallab *et al.*, 2004] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice*. Elsevier, 2004.

[Hoffmann *et al.*, 2004] Jörg Hoffmann, Julie Porteous, and Laura Sebastia. Ordered landmarks in planning. *Journal of Artificial Intelligence Research*, 22:215–278, 2004.

[Jaidee *et al.*, 2011] Ulit Jaidee, Héctor Muñoz-Avila, and David W Aha. Case-based learning in goal-driven autonomy agents for real-time strategy combat tasks. In *ICCBR Workshop on Computer Games*, 2011.

[Kabanza *et al.*, 2010] Froduald Kabanza, Philipe Bellefeuille, Francis Bisson, Abder Rezak Benaskeur, and Hengameh Irandoust. Opponent behaviour recognition for real-time strategy games. *Plan, Activity, and Intent Recognition*, 10(05), 2010.

[Lipovetzky *et al.*, 2014] Nir Lipovetzky, Miquel Ramirez, Christian Muise, and Hector Geffner. Width and inference based planners: Siw, bfs (f), and probe. *International Planning Competition*, 2014.

[Ontañón *et al.*, 2013] Santiago Ontañón, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, and Mike Preuss. A survey of real-time strategy game ai research and competition in starcraft. *IEEE Transactions on Computational Intelligence and AI in games*, 2013.

[Pommerening and Helmert, 2013] Florian Pommerening and Malte Helmert. Incremental lm-cut. In *ICAPS*, 2013.

[Pozanco *et al.*, 2018a] Alberto Pozanco, Yolanda E-Martín, Susana Fernández, and Daniel Borrajo. Counterplanning using goal recognition and landmarks. In *IJCAI*, 2018.

[Pozanco *et al.*, 2018b] Alberto Pozanco, Susana Fernández, and Daniel Borrajo. Learning-driven goal generation. *AI Communications*, pages 1–14, 2018.

[Preuss *et al.*, 2013] Mike Preuss, Daniel Kozakowski, Johan Hagelbäck, and Heike Trautmann. Reactive strategy choice in starcraft by means of fuzzy control. In *CIG*. IEEE, 2013.

[Ramírez and Geffner, 2009] Miquel Ramírez and Hector Geffner. Plan recognition as planning. In *IJCAI*, pages 1778–1783, 2009.

[Ramírez and Geffner, 2010] Miquel Ramírez and Hector Geffner. Probabilistic plan recognition using off-the-shelf classical planners. In *AAAI*, 2010.

[Richter and Westphal, 2010] Silvia Richter and Matthias Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39:127–177, 2010.

[Sailer *et al.*, 2007] Frantisek Sailer, Michael Buro, and Marc Lanctot. Adversarial planning through strategy simulation. In *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on*, pages 80–87. IEEE, 2007.

[Stanescu and Čertickỳ, 2016] Marius Stanescu and Michal Čertickỳ. Predicting opponent's production in real-time strategy games with answer set programming. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(1):89–94, 2016.

[Stanescu *et al.*, 2014] Marius Stanescu, Nicolas A Barriga, and Michael Buro. Hierarchical adversarial search applied to real-time strategy games. In *AIIDE*, 2014.

[Sukthankar *et al.*, 2014] Gita Sukthankar, Christopher Geib, Hung Hai Bui, David Pynadath, and Robert P Goldman. *Plan, activity, and intent recognition: theory and practice*. Newnes, 2014.

[Tavares *et al.*, 2016] Anderson Tavares, Hector Azpurua, Amanda Santos, and Luiz Chaimowicz. Rock, paper, starcraft: Strategy selection in real-time strategy games. In *AIIDE*, 2016.

[Uriarte and Ontañón, 2014] Alberto Uriarte and Santiago Ontañón. Game-tree search over high-level game states in rts games. In *AIIDE*, 2014.

[Vered and Kaminka, 2017] Mor Vered and Gal A Kaminka. Heuristic online goal recognition in continuous domains. In *IJCAI*, 2017.

[Weber and Mateas, 2009] Ben G Weber and Michael Mateas. A data mining approach to strategy prediction. In *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, pages 140–147. IEEE, 2009.

[Wender and Watson, 2014] Stefan Wender and Ian Watson. Combining case-based reasoning and reinforcement learning for unit navigation in real-time strategy game AI. In *ICCBR*, 2014.