Adversarial Search with Procedural Knowledge Heuristic

Viliam Lisý, Branislav Bošanský, Michal Jakob and Michal Pěchouček
Agent Technology Center, Dept. of Cybernetics, FEE, Czech Technical University
Technická 2, 16627 Prague 6, Czech Republic
{lisy, bosansky, jakob, pechoucek}@agents.felk.cvut.cz

ABSTRACT

We introduce an adversarial planning algorithm based on game tree search, which is applicable in large-scale multi-player domains. In order to tackle the scalability issues of game tree search, the algorithm utilizes procedural knowledge capturing how individual players tend to achieve their goals in the domain; the information is used to limit the search only to the part of the game tree that is consistent with pursuing players' goals. We impose no specific requirements on the format of the procedural knowledge; any programming language or agent specification paradigm can be employed. We evaluate the algorithm both theoretically and empirically, confirming that the proposed approach can lead to a substantial search reduction with only a minor negative impact on the quality of produced solutions.

Categories and Subject Descriptors

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Plan execution, formation, and generation*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Heuristic methods*

General Terms

Algorithms

Keywords

Game Tree Search, Procedural Knowledge, Goals, Complex domain, Artificial Intelligence, Experimental (Systems / Architectures), Agents

1. INTRODUCTION

Recently, there has been a growing interest in studying complex systems, in which larger numbers of agents concurrently pursue their goals while engaging in complicated patterns of mutual interaction. Examples include real-world systems, such as various information and communication networks or social networking applications, as well as simulations, including models of societies, economies and/or warfare. Because in most such systems the agents are part of a single shared environment, situations arise in which their actions and strategies interact. Scenarios in which the outcome of agent's actions depends on actions chosen by others are often termed *games* and have been an interest of AI research from its very beginning. With the increasing complexity of environments in which the agents[1] interact, however, classical game playing algorithms, such as minimax search, become unusable due to the huge branching factor, size of the state space, continuous time and space, and other factors.

The solutions AI proposed to realize goal-oriented behaviour in such environments can generally be categorized as *declarative*, which utilize formalization of goals and use computationally expensive search-based planning, or *procedural*, where the explicit knowledge how to achieve goals is captured (typically as algorithms, plan templates, or predefined plans) and agents only decide which goals should be pursued (e.g. in most of BDI architectures).

We aim to combine both approaches within the context of game playing. Specifically, we take standard adversarial search (e.g. *minimax*, max^n) and extend it with the use of procedural knowledge as a heuristic. In standard adversarial search, all possible combinations of actions of all agents are evaluated in the system to a certain extent (called *look-ahead*), and the actions that lead to the best results for individual agents within the scope of the limited look-ahead are chosen to be executed. Conceptually, adversarial search performs two separate tasks in parallel:

- **single agent planning** - finding sequences of actions achieving some partial goals of individual agents
- **interactions consideration** - exploring how plans of individual agents interact with each other (in negative but possibly also positive way)

In our approach, we reduce the complexity of the first task by employing background knowledge in the form of procedural knowledge and leave only the second task to the search. In effect, only branches that support achieving some partial goal are evaluated. This significantly reduces the searched space yet keeps the interaction consideration at the detailed level. This is essential because of the strong impact the interaction can have on the achievability of player goals and the impossibility to model such impact sufficiently at a higher level of abstraction.

In this paper, we focus primarily on the specification of the employed procedural knowledge and its utilization in game-tree search. We do not address uncertainty in the

[1]Often termed *players* in this context.

Cite as: Adversarial Search with Procedural Knowledge Heuristic, Viliam Lisý, Branislav Bošanský, Michal Jakob, Michal Pěchouček, *Proc. of 8th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2009),* Decker, Sichman, Sierra and Castelfranchi (eds.), May, 10–15, 2009, Budapest, Hungary, pp. 899–906

game or models of opponents. The paper progress as follows. Section 2 describes the main components required for the application of our approach. The core of the paper – the algorithm for using procedural knowledge in game-tree search is explained in Section 3. Section 4 analyzes computational complexity of the algorithm. Search space reduction, precision loss and scalability of the algorithm are experimentally examined in Section 5. Section 6 reviews related work and the paper ends with conclusions and discussion of future research.

2. PROBLEM DEFINITION

Many game-like situations appearing in real-world domains (e.g. humanitarian relief operation) can be modelled as n-player non-zero-sum games. Such games often have characteristics that prohibit the application of standard game-tree search algorithms (such as *minimax*). The main problems in this regard are a huge branching factor, the need for long sequences of atomic actions to achieve significant effects in the game world, and simultaneous operation of all agents involved.

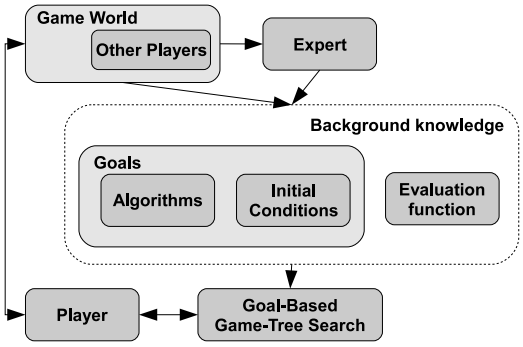

Figure 1: The conceptual scheme of a game-playing framework utilizing procedural background knowledge in game-tree search.

We address these problems by employing procedural knowledge. The resulting enhanced search technique can be applied as part of game-playing framework depicted in Figure 1. The player component represents the agent that observes the game world and performs actions to modify it. For choosing its actions, the player uses the algorithm described in Section 3. The background knowledge needed for the algorithm can be divided into three subparts:

- **Algorithms** that produce atomic actions leading to the fulfilment of their associated goal; such goals correspond to basic objectives in the game (e.g. transporting a single load of some commodity) and represent elementary building blocks of players' strategies.
- **Conditions** defining world states in which pursuing the goals is meaningful (optionally, representing conditions defining when individual players may choose to pursue the goals).
- **Evaluation function** assigning to each player and world state a numeric value representing desirability of the game state for the player (e.g. utility of the state for the player).

The overall background knowledge (i.e. each of its subpart) utilized in the search can also be split into a player-independent part (also termed *domain knowledge*) and a player-specific part (further termed *opponent models*). In this paper, we have all the background knowledge fully provided by an expert, but all parts of background knowledge can, in principle, be learned from behavior observations (see Section 6).

3. GOAL-BASED GAME-TREE SEARCH

In this section, we present the Goal-based Game-tree Search algorithm (denoted as GB-GTS) developed for game playing in large-scale multi-player scenarios. It is based on max^n [5] algorithm generalized to simultaneous moves and augmented with procedural knowledge heuristic.

3.1 Domain

The domains supported by the algorithm can be formalized as a tuple $(\mathcal{P}, \mathcal{U}, \mathcal{A}, \mathcal{W}, \mathcal{T})$, where $\mathcal{P}$ is the set of players, $\mathcal{U} = \bigcup_{p \in \mathcal{P}} \mathcal{U}_p$ is a set of units/resources capable of performing actions in the world, each belonging to one of the players. $\mathcal{A} = \mathsf{X}_{u \in \mathcal{U}} \mathcal{A}_u$ is a set of combinations of actions the units can perform, $\mathcal{W}$ is the set of possible world states and $\mathcal{T} : \mathcal{W} \times \mathcal{A} \to \mathcal{W}$ is the transition function realizing one move of the game where the game world is changed via actions of all units and world's own dynamics.

The game proceeds in moves. At the beginning of a move, each player assigns actions to all units it controls (forming the action of the player). Function $\mathcal{T}$ is subsequently invoked (taking the combination of assigned actions as an input) to modify the world state.

3.2 Simultaneous Moves

There are two ways simultaneous moves can be dealt with. The first one is to directly work with joint actions of all players in each move, compute their values and consider the game matrix (normal form game) they entail. The actions of individual players can then be chosen based on a game-theoretical equilibrium (e.g. Nash equilibrium in [7]). The second option is to fix the order of the players and let them choose their actions separately in the same way as in max^n, but using the unchanged world state from the end of the previous move for all of them and with the action execution delayed until all players have chosen their actions. This method is called *delayed execution* in [4]. In our experiments, we have used the approach with fixed player order, because of its easier implementation, allowing us to focus on core issue of utilizing the background knowledge.

3.3 Goals

For our algorithm, we define a goal as a pair (I_g, A_g), where $I_g(\mathcal{W},\mathcal{U})$ is the initiation condition of the goal and A_g is an algorithm that, depending on its internal state and the current state of the world, *deterministically* outputs the next action that leads to fulfilling the goal.

A goal can be assigned to one unit and it is then pursued until it is successfully reached or dropped because its pursuit is no longer practical. Note that we do not specify any dropping or succeeding condition, as they are implicitly captured in the A_g algorithm. We allow the goal to be abandoned only if A_g is finished; furthermore, each unit can pursue only one goal at a time. There are no restrictions on the form of algorithm A_g, so it can represent any type of

```
Input: W ∈ 𝒲: current world state, d: search depth,
       G[𝒰]: map from units to goals they pursue
Output: an array of values of the world state (one
        value for each player)
1  curW = W
2  while all units have goals in G do
3  |  Actions = ∅
4  |  foreach goal g in G do
5  |  |  Actions = Actions ∪ NextAction(A_g)
6  |  |  if A_g is finished then
7  |  |  |  remove g from G
8  |  |  end
9  |  end
10 |  curW = 𝒯(curW,Actions)
11 |  d = d − 1
12 |  if d=0 then
13 |  |  return Evaluate(curW)
14 |  end
15 end
16 u = GetFirstUnitWithoutGoal(G)
17 foreach goal g with satisfied I_g(curW, u) do
18 |  G[u] = g
19 |  V[g] = GBSearch(curW, d,Copy(G))
20 end
21 g = arg max_g V[g][Owner(u)]
22 return V[g]
```

Figure 2: GBSearch(W, d, G) – the main procedure of GB-GTS algorithm.

goal (e.g. maintain, achieve) and any kind of architecture (e.g. BDI, HTN) can be used to describe it.

The goals in GB-GTS serve as building blocks for more complex strategies that are created by combining different goals for different units and then explored via search. This contrasts with HTN-based approaches used for guiding the game tree search (see Section 6), where the whole strategies are encoded using decompositions from the highest levels of abstraction to the lower ones.

3.4 Algorithm Description

The main procedure of the algorithm (outlined in Figure 2 as procedure *GBSearch()*) recursively computes the value of a state for each of the players assuming that all the units will rationally optimize the utility of the players controlling them. The inputs to the procedure are the world state for which the value is to be computed, the depth to which to search from the world state and the goals the units are currently pursuing. The last parameter is empty when the function is called for the first time.

The algorithm is composed of two parts. The first is the simulation of the world changes based on the world dynamics and the goals that are assigned and pursued by the units, and the second is branching on all possible goals that a unit can pursue after it is finished with its previous goal.

The first part – *simulation* – consists of lines 1 to 15. If all units have a goal they actively pursue, the activity in the world is simulated without any need for branching. The simulation runs in moves and lines 3-10 describe the simulation of a single move. At first, for each unit, an action is generated based on the goal g assigned to this unit (line 5). If the goal-related algorithm A_g has finished, the goal is removed from the map of goals (lines 6-8) and the unit that was previously assigned to this goal becomes idle. The generated actions are then executed and the conflicting changes of the world are resolved in accordance with the game rules (line 10). After this step, one move of the simulation is finished. If the simulation has reached the required depth of search, the resulting state of the world is evaluated using the evaluation functions of all players (line 13).

The second part of the algorithm – *branching* – starts when the simulation reaches the point where at least one unit has finished pursuing its goal (lines 16-22). In order to ensure fixed order of players (see Section 3.2), the next processed unit is chosen from the idle units based on the ordering of the players that control the units (line 16). In the run of the algorithm, all idle units of one player are considered before moving to the units of the next one. The rest of the procedure deals with the selected unit. For this unit, the algorithm sequentially assigns each of the goals that are applicable for the unit in the current situation. The applicability is given by the goal's I_g condition. For each applicable goal, the algorithm assigns the goal to the unit and evaluates the value of the assignment by recursively calling the whole *GBSearch()* procedure (line 19). The current goals of the units are cloned because the state of the already started algorithms generating actions from goals for the rest of the units (A_g) must be the same for all the considered goals of the selected unit. After computing the value of each goal assignment, the one that maximizes the utility of the owner of the unit is chosen (line 21) and the values of this decision for all players are returned by the procedure.

3.5 Game Playing

The pseudo-code on Figure 2 shows only the computation of the values of the decisions; it does not deal with how the algorithm can be employed by a player to determine its next actions in the game. In order to do so, the player needs to extract a set of goals for its units from the searched game tree. Each node in the search procedure execution tree is associated with a unit – the unit for which the goals are tried out. During the run of the algorithm, we store the maximizing goal choices from the top of the search tree representing the first move of the game. The stored goals for each idle unit of the searching player are the main output of the search.

In general, there are two ways the proposed goal-based search algorithm can be used in game-playing.

In the first approach, the algorithm in started in each move and with all units in the simulation set to idle. The resulting goals are extracted and the first actions generated for each of the goals are played in the game. Such an *eager* approach is better for coping with unexpected events should also be more robust in case the background knowledge does not exactly describe the activities in the game.

In the second approach the player using the algorithm maintains a list of current goals for all units it controls. If none of its units is idle (i.e. has no goal assigned), the player uses the goals to generate next actions for its units. Otherwise, the search algorithm is started with goals for the player's non-idle units pre-set and all the other units idle. The goals generated for the previously idle are assigned and pursued in following moves. This *lazy* approach is significantly less computationally intensive.

3.6 Opponent Models

In Section 2 we introduced a player-specific part of background knowledge, termed opponent models. There are two types of opponent models in our approach. We now describe how they can be utilized in the algorithm. The first type, the evaluation function capturing preferences of each player is already an essential part of the max^n algorithm.

The other type of the opponent model can be used to reduce the set of all applicable goals (iterated in Figure 2 on lines 18-21) to the goals a particular player is *likely* to pursue. This can be done by adding player-specific constrains to conditions I_g defining when the respective goal is applicable. These constrains can be hand-coded by an expert or learned from experience; we call them *goal-restricting opponent models*.

The role of goal-restricting opponent model can be illustrated on a simple example of the goal representing loading a commodity to a truck. The domain restriction I_g could be that the truck must not be full. The additional player-specific constraint could be that the commodity must be produced locally at the location, because the particular opponent never uses temporary storage locations for the commodity and always transports it from the place where it is produced to the place where it is consumed.

Using a suitable goal-restricting opponent model can further reduce the size of the space that needs to be searched by the algorithm. A similar way of pruning is possible also in adversarial search without goals. We believe, however, that determining which goal a player will pursue in a given situation is more intuitive and easier to learn than to determine which low-level atomic action (e.g. going right or left on a crossroad) a player will execute.

4. TREE SIZE ESTIMATION

Before embarking on empirical evaluation of the proposed algorithm, we provide several elementary estimates of the algorithm's time complexity. Because the amount of computation time required is linearly proportional to the size of the search tree, we want to estimate the number of inner nodes evaluated during the search.

In estimating the size of the search tree, we need to take several factors into consideration:

- branching factor – the number of goals a unit can start pursuing at a time
- length of goal execution – the number of atomic actions that are executed to complete a specific goal
- look-ahead – the length of atomic action sequences considered in the search (i.e. the depth of the tree)

In actual execution of the GB-GTS algorithm, the above factors vary during the search (e.g. the branching factor is not constant and the number of current goals for a unit can change in time – e.g. for a transport unit, there can be only one city for loading a commodity, but several for its unloading). Such variability complicates the calculation of the size of the tree; in our approximation, we therefore make the following simplifying assumptions:

- the number of goals a particular unit can pursue is the same at all decision points (it can differ for different units)
- all goals of a particular unit take the same number of atomic actions to complete (again, this value can differ for different units)

Fixing the branching factor and goal execution length constant makes the execution of a unit's plan *independent* from actions performed by other units – the number of atomic actions for a goal is fixed and the unit will always select its goal from the set of the same size. Each different branch in the tree can thus be represented as a list of different choices made by all units until the given number of atomic actions (look-ahead) is reached. We can then estimate the number of different branches, i.e., the number of the leaves of the tree that is constructed under these assumptions.

Let d be the look-ahead, let N be the number of units, and $i = 1, \ldots, N$ be the index of the unit. Let b_i be the branching factor for the unit, let l_i be the number of atomic actions needed to accomplish each goal of unit i. We can calculate the number of possible different traces as a product of the number of different choices how to reach the look-ahead for each unit:

$$\#leaves = \prod_{i=1}^{N} \left((b_i)^{\lceil \frac{d}{l_i} \rceil} \right) \tag{1}$$

To approximate the size of the game tree, however, the number of the leaves is not sufficient (e.g. there can be units capable of pursuing a single goal only; such units do not produce any branching inside the tree). In order to obtain an approximation of the number of all the nodes, we further assume that all units have the same branching factor equal to a weighted average of branching factors of all units, where the weights are the inverted average lengths of each unit's goals:

$$\beta = \frac{\sum_{i=1}^{N} \frac{b_i}{l_i}}{\sum_{i=1}^{N} \frac{1}{l_i}} \tag{2}$$

For the number of nodes we now obtain

$$\#nodes = \sum_{i=0}^{\lceil log_\beta \#leaves \rceil} \beta^i \tag{3}$$

In the next section, we will show that despite the simplifying assumptions, the estimate obtained is in good agreement with empirical measurements. Besides allowing assessing the applicability of GB-GTS in specific domains, the analytic estimate can be used by the game-playing algorithm itself to dynamically adjust the look-ahead so that time constraints on finding a solution are met.

5. EXPERIMENTS

In order to practically examine the proposed goal-based (GB) adversarial search algorithm, we performed several experiments. Firstly, we compare it to the exhaustive search of the complete game tree performed by the simultaneous-move modification of max^n search (further called action-based (AB) search) in order to assess the ability to reduce the volume of search on one hand and to maintain the quality of resulting strategies on the other. Afterwards, we analyze the scalability of the GB algorithm in more complex scenarios.

Note that we use the eager, computationally more intensive version of the game playing algorithm in the experi-

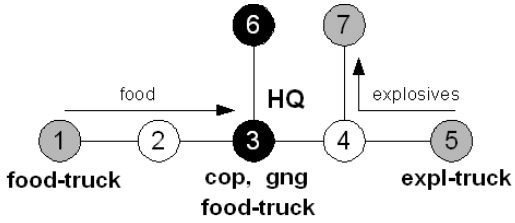

Figure 3: A schema of the simple scenario. Black vertices represent cities that can be controlled by players, grey vertices represent cities that cannot be controlled, and white vertices do not contain cities.

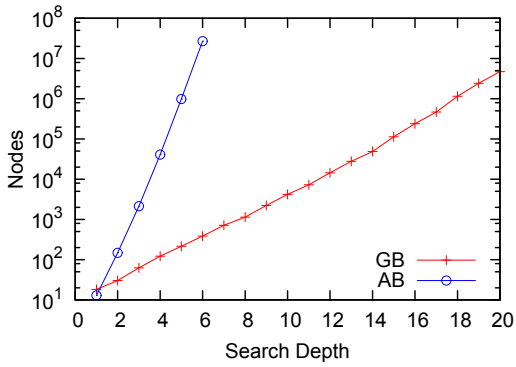

Figure 4: The search space reduction of the GB algorithm compared to the AB algorithm. An average number of the search tree nodes explored depending on the search depth is shown for both algorithms in a logarithmic scale.

ments (see Section 3.5), in which all units are choosing their goals at the same moment in each move.

5.1 Example Game

The game we use as a test case is modelled after a humanitarian relief operation in an unstable environment, with three players - government, humanitarian organization, and separatists. Each of the players controls a number of units with different capabilities that are placed in the game world represented by a graph. Any number of units can be located in each vertex of the graph and change its position to an adjacent vertex in one game move. Some of the vertices of the graph contain cities, which can take in commodities the players use to construct buildings and produce other commodities.

The utilities (evaluation functions) representing the main objectives of the players are expressed as weighted sums of components, such as the number of cities with sufficient food supply, or the number of cities under the control of the government. The government control is derived from the state of the infrastructure, the difference between the number of units of individual players in the city and the state of the control of the city in the previous move. Detailed description of the game can be found in [8].

Simple Scenario.

In order to run the standard AB algorithm on a game of this complexity, the scenario has to be scaled down to a quite simple problem. We have created a simplified scenario as a subset of our game with the following main characteristics (see also the scheme in Figure 3):

- only two cities can be controlled (Vertices 3 and 6)
- a government's HQ is built in Vertex 3
- two "main" units - police (cop) and gangster (gng) are placed in Vertex 3
- a truck is transporting explosives from Vertex 5 to Vertex 7
- another two trucks are transporting food from Vertex 1 to the city with food shortage Vertex 3

There are several possible runs of this scenario. The police unit has to protect several possible threats. In order to make government to lose control in Vertex 3, the gangster can either destroy food from a truck to cause starving resulting in lowering the well-being and consequent destroying of the HQ (by riots), or it can steal explosives and build a suicide bomber that will destroy the HQ without reducing wellbeing in the city. Finally, it can also try to gain control in city in Vertex 6 just by outnumbering the police there. In order to explore all these options, the search depth necessary is six moves.

Even such a small scenario creates a game tree too big for the AB algorithm. Five units with around four applicable actions each (depending on the state of the world) considered in six consequent moves results in $(4^5)^6 \approx 10^{18}$ world states to examine. Hence, we further simplify it for the AB algorithm. Only the actions of two units (cop and gng) are actually explored in the AB search and the actions of the trucks are considered to be a part of the environment (i.e. the trucks are scripted to act rationally in this scenario). Note that the GB algorithm does not need this simplification and actions of all units are explored in the GB algorithm.

The goals, used in the algorithm, are generated by instantiation of fifteen goal types. Each goal type is represented as a Java class. Only four of the fifteen classes are unique and the rest nine classes are derived from four generic classes in a very simple way. The actions leading towards achieving a goal consist typically of pathfinding to a specific vertex, waiting for a condition to hold, performing a specific action (e.g. loading/unloading commodities), or their concatenation.

5.2 Search Reduction

Using this simplified scenario, we first analyze how the main objective of the algorithm – *search space reduction* – is satisfied.

We run the GB and AB algorithms on a fixed set of 450 problems – world states samples extracted from 30 different traces of the game. On each configuration, we experiment with different values of the look-ahead parameter (1-6 for the AB algorithm and 1-19 for the GB algorithm). As we can see in Figure 4, the experimental results fulfilled our aim of substantial reduction of the search space. The number of nodes explored increases exponentially with the depth of the

search. However, the base of the exponential is much lower for the GB algorithm. The size of the AB tree for six moves look-ahead is over 27 million, while GB search with the same look-ahead explores only 385 nodes and even for the look-ahead of nineteen, the size of the tree was in average less than 5×10^6. These numbers indicate that using heuristic background knowledge can reduce the time needed to choose an action in the game from tens of minutes to a fraction of a second.

Our implementations of each of the algorithms processed approximately twenty five thousands nodes per second on our test hardware without any optimization techniques. According to [1], however, game trees with million nodes can be searched in real-time (about one second) when such optimization is applied and when efficient data structures are used.

5.3 Loss of Accuracy

With such substantial reduction of the set of possible courses of action explored in the game, some loss of quality of game-playing can be expected. Using the simplified scenario, we compared the actions resulting from the AB and the first action generated by the goal resulting from the GB algorithm. The action differed in 47% of cases. However, a different action does not necessarily mean that the GB search has found a sub-optimal move. Two different actions often have the same value in AB search. Because of the possibly different order in which actions are considered, the GB algorithm can output an action which is different from the AB output yet still has the same optimal value. The values of actions referred to in the next paragraph all come from the AB algorithm.

The value of the action resulting from the GB algorithm was in **88.1%** of cases exactly the same as the "optimal" value resulting from AB algorithm. If the action chosen by GB algorithm was different, it was still often close to the optimal value. We were measuring the difference between the values of GB and optimal actions, relative to the difference between the maximal and the minimal value resulting from the searching player's decisions in the first move in the AB search. The mean relative loss of the GB algorithm was 9.4% of the range. In some cases, the GB algorithm has chosen the action with minimal value, but it was only in situations, where the absolute difference between the utilities of the options was small.

5.4 Comparison with Theoretical Estimates

We now compare the theoretical estimates obtained in Section 4 with the real values. We analyzed and calculated the average branching factors and the average lengths of plans of all units in the simple scenario in order to feed them in the formula (3). We then compared the obtained estimation of the number of nodes of the evaluated trees with the average number of nodes of the tree from several runs of GB-GTS search for different look-ahead. The results are shown in Figure 5. We can see that even with the simplifications made we obtain fairly accurate estimates of the number of the nodes of the tree.

5.5 Scalability

Previous sections show that the GB algorithm can be much faster than and almost as accurate as the AB algorithm with suitable goals. We continue with assessing the limits on the complexity of the scenario where GB algorithm is still usable. There are several possible expansions of the simple scenario. We explore the most relevant factor – number of units – separately and then we apply the GB algorithm on a bigger scenario. In all experiments, we ran the GB algorithm in the initial position of the extended simple scenario and we measured the size of the searched part of the game tree.

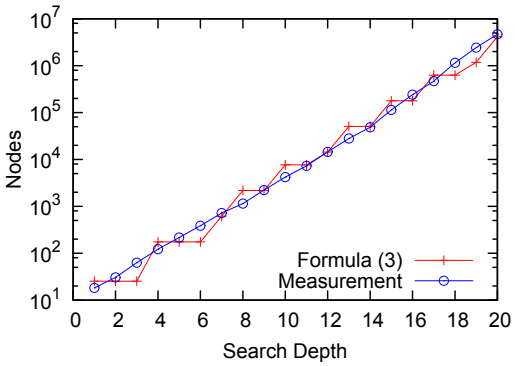

Figure 5: The estimation of the number of evaluated nodes during the search compared to experimental measurments

5.5.1 *Adding Units*

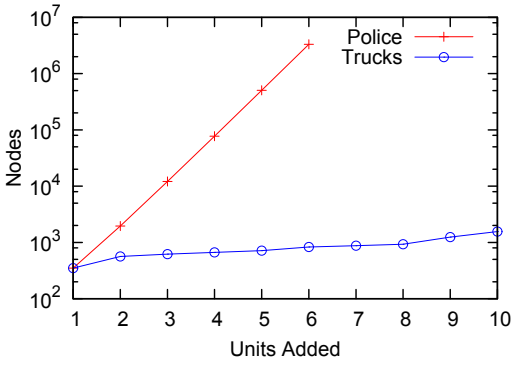

Figure 6: Increase of the size of the searched tree when adding one to ten police units and explosives trucks to the simple scenario with a 6 move look-ahead.

The increase of the size of the searched tree naturally depends on the average number of goals applicable for a unit when it becomes idle and the lengths of the plans that lead to their fulfilment. The explosives truck has usually only a couple of applicable goals. If it is empty, the goal is to load in one of the few cities where explosives are produced and if it is full, the goal is to unload somewhere where explosives can be consumed. On the other hand, a police unit has many possible goals. It can protect any transport from being robbed or it can try to outnumber the separatists in any city. We were adding these two unit types to the simple scenario and computed the size of the search tree with fixed

six moves look-ahead.

When adding one to ten explosives trucks to the simple scenario, each of them has always only one goal to pursue at any moment. Due to our GB algorithm definition, where goals for each unit are evaluated in different search tree node, even adding a unit with only one possible goal increases the number of evaluated nodes slightly. In Figure 6 are the results for this experiment depicted as circles. The number of the evaluated nodes increases only linearly with increasing the number of the trucks.

Adding further police units with four goals each to the simple scenario increased the tree size exponentially. The results for this experiment are shown in Figure 6 as pluses.

5.5.2 *Complex Scenario*

In order to test the usability of the GB search in a more realistic setting, we implemented a larger scenario within our test domain. We used a graph with 2574 vertices and two sets of units. The first unit set was composed of nine units, including two police units with up to four possible goals in one moment, two gangster units with up to four possible goals, an engineer with three goals, stone truck with up to two goals and three trucks with only one commodity source and one meaningful destination resulting to one goal at any moment. The second unit set included seven units – one police, one gangster unit and the same amount of units of the other types. The lengths of the plans to reach these goals is approximately seven atomic actions. There are five cities, where the game is played.

A major difference of this scenario to the simple scenario is, besides the added units, a much bigger game location graph and hence higher length of the routes between cities. As a result, all plans of the units that need to arrive to a city and perform some actions there are proportionally prolonged. This is not a problem for the GB algorithm, because the move actions along the route are only simulated in the simulation phase and do not cause any branching.

In a simple experiment to prove this, we changed the time scale of the simulation, so that all the actions were split to two sub-actions collectively effecting the same change of the game world. After this modification, the GB algorithm explored exactly the same number of nodes and the time needed for the computation increased linearly, corresponding to more simulation steps needed.

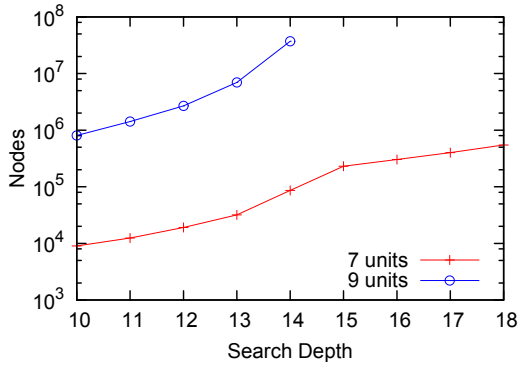

Figure 7: Size of the trees searched by the GB algorithm in the complex scenario.

If we assume that an optimized version of the algorithm can compute one million nodes in a reasonable time, then the look-ahead we can use in the complex scenario is 10 in the nine-unit case and 18 in the seven-unit case. Both values are higher than the average length of unit's typical plan, allowing it to find meaningful plans. If we wanted to apply the AB algorithm to the seven-unit case with the look-ahead of eighteen, considering only four possible move directions and waiting for each unit, it would mean searching through approximately $4^{7^{18}} \approx 10^{75}$ nodes of the game tree, clearly an impossible task.

6. RELATED WORK

In this section, we review work on two related areas. The first concerns other ways of using background knowledge heuristic in game-tree search; the other concerns obtaining the background knowledge required by our algorithm.

Knowledge Heuristic in Adversarial Search.

Existing work on using knowledge heuristic in adversarial search is surprisingly scarce. The most closely related work is probably the chess-playing system Paradise [11]. The system uses a learned set of rules that can match various situations on the chessboard. The satisfied rules generate only a small number of actions that need to be explored in each situation. Our approach is more general – it can use arbitrary algorithms for the description of goals, more complex game environments; the progress in hardware also allowed experiments of different magnitudes.

More recent work in this area is [9]. The authors used HTN formalism to define the set of runs of the game, which are consistent with some predefined hand-coded strategies. However, the approach is designed for the Bridge domain, which allows searching through the complete game tree defined by the strategies; the approach uses domain-specific abstractions of the information about the opponents instead of searching the game action-by-action. Searching trough whole strategies for more complex domains would hardly be tractable and describing them would be much harder then describing just the subgoals needed in our approach.

A plan library represented as HTN is used to play GO in [12]. The searching player simulates HTN planning for both the players, without considering what the other one is trying to achieve. If one player achieves its goal, the opponent backtracks (the shared state of the world is returned to the previous state) and tries a different decomposition. A notable property of this approach is that it does not use an explicit evaluation function and hence it is usable only in zero-sum games.

Another approach for reducing the portion of the tree that is searched for scenarios with multiple units is introduced in [6]. The authors show successful experiments with searching for one unit at a time only, while simulating the movement of the other units using a rule-based heuristic.

Obtaining Background Knowledge.

The background knowledge is often difficult and expensive to obtain. It can generally be hand-coded by an expert, learned, or something in between. We present a short review of this options that support usability of the proposed algorithm.

The *evaluation functions* represent the basic desires of the

players and they can often be easily expressed by an expert. However, in case that the opponent is completely unknown, methods form classical game tree search, such as [2], should be considered. The paper presents learning opponent evaluation function together with the look-ahead that explains the opponent moves best. It assumes that the opponent is using minimax algorithm and that its evaluation function is a weighted sum of world state characteristics known in advance and it performs hill-climbing in the space of the weights and depths to find the combination that explains the training set of opponent decisions best. A similar approach could be usable also in non-zero-sum case of multiple players, but the number of parameters in the search is multiplied by the number of players. Presence of the a priory known procedural knowledge heuristic would not change anything in the approach.

The *procedural knowledge* in form of general algorithms that produce actions that lead to fulfilment of subgoals are the most complex part of the background knowledge needed for our algorithm. It can be learned by completely unsupervised methods, or use some information from an expert and derive the rest form observation [10]. Learning agent behaviour from observations is most relevant in context of our work especially for capturing the information about the opponents. In [3], the authors present a system, that use ILP to clone behaviour of an expert. A part of their system is learning the algorithm for generating actions leading to fulfilling of subgoals as well as the initiation conditions. They prove its efficiency in complex domains with other agents involved so it is likely that the approach would be usable in our setting.

7. CONCLUSIONS

We proposed a novel algorithm that extends a multi-player simultaneous-move game-tree search with a heuristic based on procedural knowledge. As indicated by both the theoretical and empirical evaluation of the algorithm's complexity, the approach is particularly useful in domains where *long sequences* of actions lead to significant changes in the world state, each of the units (or other resources) can only pursue a *few goals* at any time, and the decomposition of a each goal to low level actions is *uniquely defined* (e.g. using the shortest path to move between locations).

In experiments, we have compared the performance of the algorithm to a slightly modified exhaustive max^n search, showing that despite examining only a small fraction of the game tree (less than 0.002% for the look-ahead of six game moves), the goal-based search is still able to find an optimal solution in 88.1% cases; furthermore, even the suboptimal solutions produced are often close to the optimum. This results have been obtained with the background knowledge designed *before* implementing and evaluating the algorithm and without further optimization to prevent over-fitting.

Furthermore, we have tested the scalability of the algorithm to larger scenarios where the modified max^n search cannot be applied at all. We have confirmed that although the algorithm's time complexity cannot escape exponential growth, this growth can be controlled by reducing the number of different goals considered for each unit and by making the action sequences generated by goals longer. Simulations on a real-world scenario modelled as a multi-player asymmetric game proved the approach viable, though further optimizations would be necessary for the algorithm to discover more complex strategies.

Besides the significant space reduction, another important advantage of the proposed approach is that no restrictions for algorithms representing goals in the background knowledge are needed. This fact opens a possibility to reuse knowledge already captured in other formalisms (HTN, BDI, etc.) also in the GB-GTS algorithm.

8. ACKNOWLEDGMENTS

Effort sponsored by the Air Force Office of Scientific Research, USAF, under grant number FA8655-07-1-3083 and by the Research Programme No.MSM6840770038 by the Ministry of Education of the Czech Republic. The U.S. Government is authorized to reproduce and distribute reprints for Government purpose notwithstanding any copyright notation thereon.

9. REFERENCES

[1] D. Billings, A. Davidson, T. Schauenberg, N. Burch, M. Bowling, R. C. Holte, J. Schaeffer, and D. Szafron. Game-tree search with adaptation in stochastic imperfect-information games. In *Computers and Games*, volume 3846 of *LNCS*, pages 21–34. Springer, 2004.

[2] D. Carmel and S. Markovitch. Learning and using opponent models in adversary search. Technical Report CIS9609, Technion, 1996.

[3] T. Könik and J. E. Laird. Learning goal hierarchies from structured observations and expert annotations. *Mach. Learn.*, 64(1-3):263–287, 2006.

[4] A. Kovarsky and M. Buro. Heuristic search applied to abstract combat games. In *Canadian Conference on AI*, pages 66–78, 2005.

[5] C. Luckhardt and K.B.Irani. An algorithmic solution of n-person games. In *Proc. of the National Conference on Artificial Intelligence (AAAI-86), Philadelphia, Pa., August*, pages 158–162, 1986.

[6] K. J. Mock. Hierarchical heuristic search techniques for empire-based games. In *Proceedings of the International Conference on Artificial Intelligence (IC-AI)*, pages 643–648, 2002.

[7] F. Sailer, M. Buro, and M. Lanctot. Adversarial planning through strategy simulation. In *IEEE Symposium on Computational Intelligence and Games (CIG)*, pages 80–87, 2007.

[8] E. Semsch, V. Lisý, B. Bošanský, M. Jakob, D. Pavlíček, J. Doubek, and M. Pechoucek. Adversarial behavior testbed. Technical Report GLR 90/09, CTU, FEE, Gerstner Lab., 2009. `http://agents.felk.cvut.cz/publications`.

[9] S. J. J. Smith, D. S. Nau, and T. A. Throop. Computer bridge - a big win for AI planning. *AI Magazine*, 19(2):93–106, 1998.

[10] M. van Lent and J. E. Laird. Learning procedural knowledge through observation. In *K-CAP '01: Proceedings of the 1st international conference on Knowledge capture*, pages 179–186. ACM, 2001.

[11] D. E. Wilkins. Using patterns and plans in chess. *Artificial Intelligence*, 14(2):165–203, 1980.

[12] S. Willmott, J. Richardson, A. Bundy, and J. Levine. Applying adversarial planning techniques to Go. *Theoretical Computer Science*, 252(1–2):45–82, 2001.