

Learning from Human Demonstrations for Real-Time Case-Based Planning

Santiago Ontañón and Kane Bonnette and Prafulla Mahindrakar and Marco A. Gómez-Martín and Katie Long and Jainarayan Radhakrishnan and Rushabh Shah and Ashwin Ram

CCL, Cognitive Computing Lab, Georgia Institute of Technology
Atlanta, GA 30332/0280

Facultad de Informática, Universidad Complutense de Madrid. 28040, Madrid, Spain
{santi, kane, ashwin}@cc.gatech.edu, {praful, jai.rad, rushabh}@gatech.edu
marcoa@fdi.ucm.es, gtg324x@mail.gatech.edu

Abstract

One of the main bottlenecks in deploying case-based planning systems is authoring the case-base of plans. In this paper we will present a collection of algorithms that can be used to automatically learn plans from human demonstrations. Our algorithms are based on the basic idea of a *plan dependency graph*, which is a graph that captures the dependencies among actions in a plan. Such algorithms are implemented in a system called *Darmok 2* (D2), a case-based planning system capable of general game playing with a focus on real-time strategy (RTS) games. We evaluate D2 with a collection of three different games with promising results.

1 Introduction

Real-time strategy games are complex domains in which artificial intelligence (AI) contenders still lag behind human players. RTS games are complex because they have huge decision spaces and state spaces, and they are non-deterministic, non fully-observable, and real time [Ontañón *et al.*, to appear; Aha *et al.*, 2005]. Most previous approaches to develop AI for RTS games have been based on incorporating domain knowledge into a system to make the problem tractable. This domain knowledge can take several forms: some approaches [Aha *et al.*, 2005; Sharma *et al.*, 2007] define abstracted state or action spaces that reduce the complexity of learning or planning; while some other approaches [McCoy and Mateas, 2008] encode domain knowledge as hand-coded procedures or rules that the system can execute. A few approaches [Ontañón *et al.*, to appear; Könik and Laird, 2006] have proposed using a different source of domain knowledge: observing humans play. In this paper we will follow the latter and present a domain-independent case-based planning system capable of learning how to play RTS games by observing human *demonstrations*. Human demonstrations have the advantage that anyone with knowledge about the domain can provide them. We will show that learning from demonstration can solve one of the bottlenecks of case-based planning systems, namely populating the case-base with plans.

In this paper we will present a system called *Darmok 2* (D2), an evolution of *Darmok* [Ontañón *et al.*, to appear].

D2 is a real-time case-based planning (CBP) [Spalazzi, 2001] system capable of exploiting the information contained in human demonstrations of an arbitrary adversarial game to play that game at a proficient level. However, due to the planning nature of the system, games with a focus on strategy and planning, such as RTS games, are handled better than games that focus on reactive actions. Specifically, we will focus on the set of algorithms that enable D2 to learn hierarchical plans from human demonstrations. The main contributions of D2 are a novel way to learn plans from demonstrations for case-based planning systems, efficient and domain-independent plan adaptation techniques, and a novel adversarial case-based planning algorithm.

The rest of this paper is organized as follows: Section 2 presents an small overview of previous work in this area. Section 3 presents our D2 system, and Section 4 presents an initial empirical evaluation of D2 using three different games. The paper closes with conclusions and future work.

2 Background

Case-based planning involves reusing previous plans and adapting them to suit new situations instead of planning from scratch like in STRIPS [Fikes and Nilsson, 1971] or HTN planning [Nau *et al.*, 2005]. Case-based planning is the application of CBR [Aamodt and Plaza, 1994] principles to planning, and as such, decomposes the process of planning into four stages: *retrieval, reuse, revise and retain*. Given a new problem description, a case-based planner first retrieves one or more relevant plans from a plan library and then it tries to reuse them (by adapting them if needed) to solve the new problem (during this process, more cases might be retrieved if the plan is decomposed hierarchically). After that, the case-based planner can already propose a solution to the problem. Additionally, the resulting plan can be revised (by executing it for example), and then the system will consider whether the new plan (or parts of it) is interesting enough to be retained into the plan base.

There are several motivations for case-based planning techniques [Spalazzi, 2001] but the main one is that they have the potential to increase the *efficiency* with respect to generative planners. Although, under certain conditions [Muñoz-Avila and Cox, 2007], reusing plans has the same or even higher worst-case complexity than planning from scratch [Nebel and Koehler, 1992], case-based planning can exploit regularities

in the problems being solved, and thus potentially greatly increase the efficiency. The main issues that case-based planners have to deal with are plan retrieval, plan adaptation, and how to learn from experience.

Learning from Demonstration, called sometimes “programming by demonstration” or “imitation learning”, has been widely studied in robotics [Bakker and Kuniyoshi, 1996], and offers an alternative to manual programming. Human demonstrations have also received some attention to speed-up reinforcement learning [Schaal, 1996], and as a way of automatically acquiring planning knowledge [Hogg *et al.*, 2008], among other uses.

Recently, Hogg *et al.* [Hogg *et al.*, 2008] present the HTN-Maker algorithm, that given a set of solved planning problems and a set of tasks is able to generate HTN methods for them. The HTN-Maker shows that learning from demonstration is an effective way to obtain planning knowledge.

König and Laird present a *Relational Learning from Observation* technique [König and Laird, 2006] able to learn how to decompose a goal into subgoals based on observing annotated expert traces. König and Laird’s technique uses relational machine learning techniques to learn how to decompose goals, and the output is a collection of rules, thus showing another approach to learning planning knowledge from demonstrations.

Most of the work on case-based planning does not focus on how to learn the cases in the case base. In this paper, we will present a case-based planning system that learns cases by observing human demonstrations.

3 Darmok 2

Darmok 2 (D2) is a real-time case-based planning system designed to play RTS games. D2 implements the *on-line case-based planning* cycle (OLCBP) as introduced in [Ontañón *et al.*, to appear]. The OLCBP cycle attempts to provide a high-level framework to develop case-based planning systems that operate on-line, i.e. that interleave planning and execution in real-time domains. The OLCBP cycle extends the traditional CBR cycle by adding two additional processes, namely *plan expansion* and *plan execution*. The main focus of D2 is to explore learning from unannotated human demonstrations, and the use of adversarial planning techniques. The most important characteristics of D2 are:

- It acquires cases by analyzing human demonstrations.
- It interleaves planning and execution.
- It uses an efficient transformational plan adaptation algorithm for allowing real-time plan adaptation.
- It makes use of a *simulator* in order to perform adversarial case-based planning.

The following sections explain the different components of D2 in more detail, emphasizing the learning from demonstration aspect of the system.

3.1 Representing Demonstrations, Plans and Cases

Demonstrations in D2 are represented as a list of triples $\{\langle t_1, G_1, A_1 \rangle, \dots, \langle t_n, G_n, A_n \rangle\}$, where each triple contains a

time stamp t_i game state G_i and a set of actions A_i (that can be empty). The set of triples represent the evolution of the game and the actions executed by each of the players at different time intervals. The set of actions A_i represent actions that were issued at t_i by any of the players in the game. The game state is stored using an object oriented representation that captures all the information in the state: map, players and other entities (entities include all the units a player controls in an RTS game: e.g. tanks).

Unlike in traditional STRIPS planning, actions in RTS games may not always succeed, they may have non-deterministic effects, and they might not have an immediate effect, but be durative. Thus, a typical representation of preconditions and postconditions is not enough. An action a is defined in D2 as a tuple containing 7 elements:

- Action name.
- *Parameters*, a list of named parameters with associated types. It is important to know the type of a parameter for plan adaptation purposes. Valid types are: integer, string, coordinates, entity identifier, or entity type (these determine the range of valid values). Additional constraints on the range of values can be specified.
- *Preconditions*, which must be satisfied for an action to start execution.
- *Success conditions*, which cause the action to succeed. Note that success conditions are not the same as postconditions.
- *Failure conditions*, which cause the action to fail.
- *Pre-failure conditions*, which if satisfied before the preconditions, indicate that the preconditions will never become true, so it’s useless to keep waiting. Notice that in RTS games, actions take time. It might be the case that if preconditions are not satisfied, they may become satisfied by merely waiting some time. Pre-failure conditions are useful for D2 to know when to stop waiting for the preconditions of an action.
- *Postconditions*, which are a superset of the success conditions, and include conditions that might happen as a result of an action, but that are not necessary for considering an action succeeded. For example, the *attack* action will have a success condition that a particular entity will be attacking another entity. Destruction of the target entity would be specified in the postconditions as a possible outcome.

Plans in D2 are represented as *hierarchical petri nets*. Petri nets [Murata, 1989] offer an expressive formalism for representing plans that include conditionals, loops or parallel sequences of actions. In short, a petri net is a graph consisting of two types of nodes: *transitions* and *states*. Transitions contain conditions, and link states to each other. Each state might contain *tokens*, which are required to fire transitions. The flow of tokens in a petri net represents it’s status. In D2, the plans that will be learned by observing demonstrations consist of hierarchical petri nets, where some states will be associated with sub plans, which can be primitive actions or sub-goals. The left hand side of Figure 1 shows an example

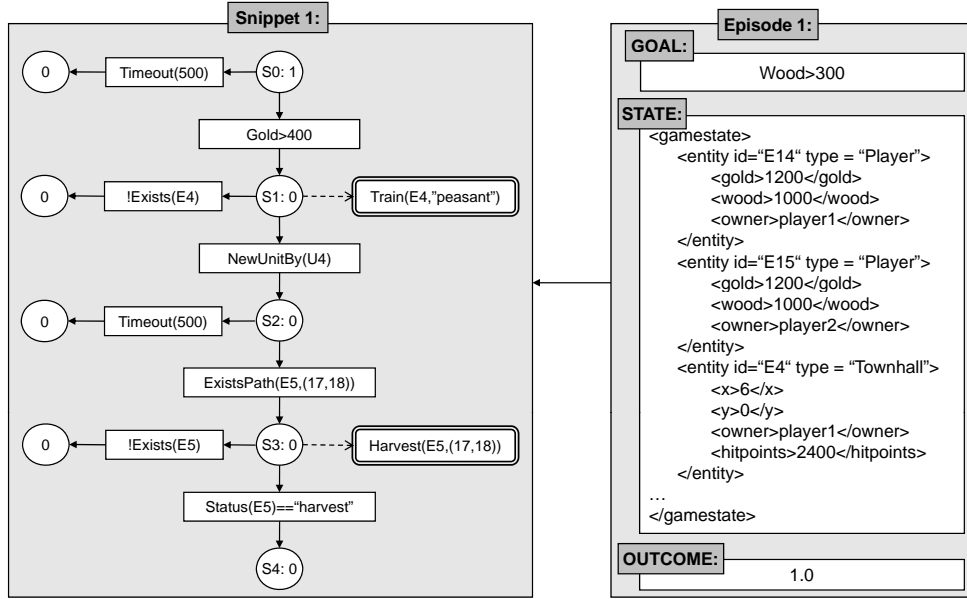


Figure 1: A case in D2 consisting of a snippet and an episode. The snippet contains two actions, and the episode says that this snippet succeeded in achieving the goal $Wood > 300$ in the specified game state. The game state representation is not fully included due to space limitations.

of a petri net representing a plan consisting of two actions to be executed in sequence: $Train(E4, "peasant")$ and $Harvest(E5, (17,18))$. Notice that the handling of preconditions, postconditions, etc. is handled by the petri net, making the execution module of D2 a simple petri net simulation component.

When D2 learns plans from demonstrations, such plans are stored as *cases*. Cases in D2 are represented like cases in the Darmok system [Ontañón *et al.*, to appear], consisting of a collection of plan *snippets* with *episodes* associated to them. As shown in Figure 1, a snippet is simply a plan, and an episode is a structure storing the outcome obtained when a particular snippet was executed in a particular game state intending to achieve a particular goal. The outcome is simply a real number in the interval $[0, 1]$ representing how well the goal was achieved: 0 represents total failure, and 1 total success. To measure success, all conditions in D2, instead of being boolean, are defined as returning a number in the interval $[0, 1]$ so that the execution and planning modules of D2 have a better perception of condition satisfaction progress.

3.2 Learning Plans and Cases from Demonstration

D2’s case base is populated by learning both snippets and episodes from human demonstrations. The input to the learning algorithm is one demonstration D (of length n), a player p (D2 will learn only from the actions of player p in the demonstration D), and a set of goals G for which to look for plans. The output is a collection of snippets and episodes. The set of goals G can be fixed beforehand for every particular domain, and is equivalent to the list of *tasks* in the HTN planning framework (thus, the inputs are the same as for the HTN-Maker algorithm). The learning process of D2 can be

Demonstration	g_1	g_2	g_3	g_4	g_5
$\langle t_1, G_1, A_1 \rangle$					
$\langle t_2, G_2, A_2 \rangle$					
$\langle t_3, G_3, A_3 \rangle$					
$\langle t_4, G_4, A_4 \rangle$					
$\langle t_5, G_5, A_5 \rangle$					
$\langle t_6, G_6, A_6 \rangle$				✓	
$\langle t_7, G_7, A_7 \rangle$			✓	✓	
$\langle t_8, G_8, A_8 \rangle$		✓	✓	✓	
$\langle t_9, G_9, A_9 \rangle$		✓	✓	✓	✓
$\langle t_{10}, G_{10}, A_{10} \rangle$		✓	✓	✓	✓
$\langle t_{11}, G_{11}, A_{11} \rangle$		✓	✓	✓	✓
$\langle t_{12}, G_{12}, A_{12} \rangle$	✓	✓	✓	✓	✓

Table 1: Goal matrix for a set of five goals $\{g_1, g_2, g_3, g_4, g_5\}$ and for a small trace consisting of only 12 entries (corresponding to the actions shown in Figure 2, $A_{12} = \emptyset$).

divided in four main stages: *goal matrix generation*, *dependency graph generation*, and *hierarchical composition*. Let us present each one of them in detail.

Goal Matrix Generation

The first step to learn plans from a demonstration is to generate the *goal matrix*. The goal matrix M is a boolean matrix, where each row represents a triplet in the demonstration D , and each column represents one of the goals in G . $M_{i,j}$ is true if the goal g_j is satisfied at time t_i in the demonstration. An example goal matrix can be seen in Table 1.

Once the goal matrix is constructed, a set of *raw plans* P are extracted from it in the following way:

1. For each goal $g_j \in G$ do

Plan
1.- Harvest(U2,(0,16))
2.- Train(U4,"peasant")
3.- Harvest(U3,(17,23))
4.- Train(U4,"peasant")
5.- Build(U5,"LumberMill",(4,23))
6.- Build(U5,"Barracks",(8,22))
7.- Train(U6,"archer")
8.- Build(U5,"tower")
9.- Train(U6,"archer")
10.- Attack(U7,EU1)
11.- Attack(U8,EU2)

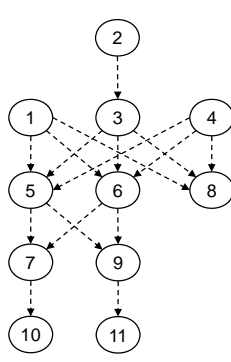


Figure 2: An example dependency graph constructed from a plan consisting of 11 actions in an RTS game.

- (a) For each $0 < i \leq n$ such that $M_{i,j} \wedge \neg M_{i-1,j}$ do
 - i. Find the largest $0 < l < i$ such that $\neg M_{l,j} \wedge (l = 1 \vee M_{l-1,j})$
 - ii. Generate a raw plan from the actions executed by player p in the set $A_l \cup A_{l+1} \cup \dots \cup A_{i-1}$ and add it to P

For example, five plans could be generated from the goal matrix in Table 1. One for g_1 with actions $A_l \cup \dots \cup A_{12}$, one for g_2 with actions $A_l \cup \dots \cup A_8$, one for g_3 with actions $A_l \cup \dots \cup A_7$, one for g_4 with actions $A_l \cup \dots \cup A_6$, and one for g_5 with actions $A_l \cup \dots \cup A_9$. Notice that the intuition behind this process is just to look at sequences of actions that happened before a particular goal was satisfied, since those actions are a plan to reach that goal. Many more plans could be generated by selecting subsets of those plans, but since D2 works under tight real-time constraints, currently it learns only a small subset of plans from each demonstration. Learning the maximum number of plans possible from a demonstration while maintaining D2's real-time performance is part of our future work.

Notice that this process is enough to learn a set of raw plans for the goals in G . The snippets will be constructed from the aforementioned sets of actions, and the episode will be generated by taking the game state in which the earliest action in a particular plan was executed. Notice that all plans extracted using this method are plans that succeeded, thus all episodes have outcome equal to 1. However, these raw plans might contain unnecessary actions and would be monolithic, i.e. they will not be decomposable hierarchically into sub-goals. Dependency graph generation and hierarchical composition are used to solve both problems.

Dependency Graph Generation

Given a plan consisting of a partially ordered collection of actions, a *dependency graph* [Sugandh *et al.*, 2008] is a directed graph where each node represents one action in the plan, and edges represent dependencies among actions. Such a graph will be used by D2 to remove unnecessary actions from the learned plans.

Such a graph is easily constructed by checking each pair of actions a_i and a_j in the plan, and checking first of all, if there

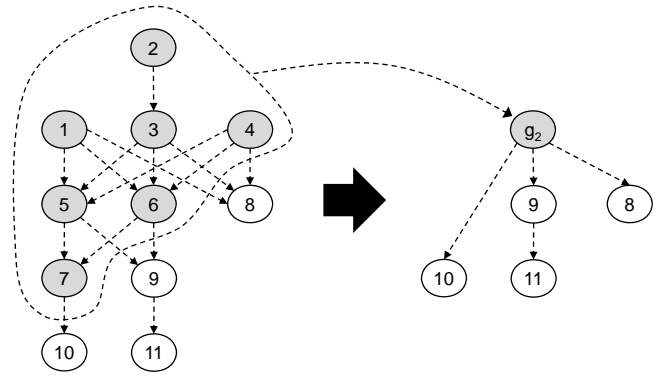


Figure 3: The nodes greyed out in the left dependency graph correspond to the actions in the plan learned from a goal g_2 , after substituting those actions by a single subgoal, the resulting plan graph looks like the one on the right.

is any order restriction between a_i and a_j . Only those pairs for which a_i can happen before a_j will be considered. Next, if one of the postconditions of a_i matches any precondition of a_j , and there is no action a_k that has to happen after a_i that also matches with that precondition, then an edge is drawn from a_i to a_j in the dependency graph, annotating it with which is the pair of postcondition/precondition that matched. Figure 2 shows an example dependency graph (where the labels in the edges have been omitted for clarity). The plan shown in the figure shows how each action is dependent on each other, and it is useful to determine which actions contribute to the achievement of particular goals.

D2 constructs a dependency graph of the plan resulting from using the complete set of actions that a player p executed in a demonstration D . This dependency graph will be used to remove unnecessary actions from the smaller raw plans learned from the goal matrix in the following way:

1. For each plan $p \in P$ do
 - (a) Extract the subgraph of the dependency graph containing only the actions in p .
 - (b) Detect which is the subset of actions A from the actions in p such that their postconditions match with the goal of plan p .
 - (c) Remove from p all actions that, according to the subgraph do not contribute directly or indirectly to any of the actions in A .

Moreover, the plan graph provides additional internal structure to the plan, indicating which actions can be executed in parallel, and which ones have to be executed in a sequence. All this information is exploited when generating the petri net corresponding to the plan.

Hierarchical Composition

Finally, D2 analyzes the set of plans P resulting from the previous step using the dependency graph to see if any of those plans are a sub-plan of another plan. Given two plans $p_i, p_j \in P$, if the set of actions in p_i is a subset of the set of actions in p_j , D2 assumes that p_i is a sub-plan of p_j , and all the actions in p_i also contained in p_j are substituted by a

single sub-goal in p_j . Converting flat plans into hierarchical ones is important in D2, since it allows D2 to combine plans learned from one demonstration with plans learned from another at run time, increasing its flexibility.

Figure 3 shows an example of this process taking the plan graph of the plan learned for goal g_1 in Table 1, and substituting some of its actions by a single subgoal g_2 . The actions marked in grey in the left hand side of Figure 3 correspond to the actions in the plan learned for g_2 .

Notice that the order in which we attempt to substitute actions by subgoals in plans will result in different final plans. Currently, D2 uses the heuristic of attempting first to substitute larger plans first. However, this issue is a subject of our ongoing research effort.

3.3 Hierarchical-Adversarial-Case-Based Planning

Since it is outside the scope of this paper to explain the planning algorithm of D2, we will just briefly outline the main ideas behind its design. ACBP (Adversarial Case-Based Planner) is an algorithm that combines ideas from case-based planning, HTN planning and game tree search in order to construct an efficient adversarial planner that can be used in real-time domains. Let us present some of the main ideas of the algorithm:

- *ACBP uses plans instead of actions:* traditionally, game tree search algorithms search in the space of possible sequences of primitive actions. That leads to a space that is too large to make the problem tractable. In ACBP we propose to search in the space of plans (from the set of plans in the case-base) instead of in the space of actions. This allows goals to be completed with one plan instead of a series of actions, shrinking the search space and making the problem tractable.
- *ACBP is not turn based:* game tree search algorithms assume turn based games, however, most modern games are not turn based. Instead, ACBP uses the idea of “commitment to plans”. A player using ACBP decides a combination of plans to apply to achieve some goals and commits to it. Unless the plans fail, or more plans are needed (when a new goal arises), no more decisions have to be taken. The search tree open by ACBP contains only decision nodes when players have to take decisions, which can happen any time in the game, not necessarily following turns.
- *ACBP is simulation-based:* during the search process, when each player is committed to a particular plan, ACBP uses a simulator to simulate the execution of the game to fast-forward till the next expected point when a player has to take a new decision.
- *ACBP does not open the full search tree:* ACBP is a case-based planning system, and as such, it uses a retrieval mechanism to select a small subset of plans that are adequate for the current situation. Moreover, the small set of retrieved plans are not used directly, but are adapted to fit the current situation in order to ensure that they are applicable.

3.4 Plan Adaptation

When a plan is retrieved from the case base by the planner, it will have to be adapted to fit the situation at hand. Plan adaptation [Muñoz-Avila and Cox, 2007] is one of the most complex processes in case-based planning. In order to maintain real-time performance, D2 implements a series of simplification assumptions to the general problem of plan adaptation. D2 uses a slightly improved version of the plan adaptation technique presented in [Sugandh *et al.*, 2008]. Let us just briefly mention the main characteristics of it:

- D2 uses a transformational adaptation process that divides adaptation in two independent processes: parameter adaptation, and structure adaptation. Assuming that those two processes can be performed independently increases the efficiency of D2’s plan adaptation.
- D2 uses a domain independent method for parameter adaptation based on potential fields.
- D2 uses a structural adaptation algorithm based also in plan dependency graphs [Sugandh *et al.*, 2008], which makes the assumption that different plans will not undo what other plans do (i.e. it assumes no retraction) in order to make adaptation tractable (in fact, polynomial). This algorithm can handle full petri nets, and therefore it can adapt plans consisting of loops, conditionals, sequences and parallel constructs.

4 Initial Experimental Evaluation

We have evaluated D2 with a collection of three games, shown in Figure 4: S2, Towers, and BattleCity. S2 is an RTS in the style of “Warcraft II”, with some simplifications (like the lack of naval or aerial units). Towers is a multiplayer version of the well known game “Tower Defense”, where players build towers in order to stop enemy forces attacking the player’s base. Finally, BattleCity is an action game in which the player controls a tank that has to destroy all the enemy tanks or the enemy bases. D2 plays the complete games without any abstraction or simplification.

The three games were implemented explicitly to evaluate D2, and they were selected since each game requires different skills: BattleCity requires fast reflexes and a low level reactive behavior; Towers requires geometrical skills in order to find optimal placement of towers; and S2 requires long term planning and strategic reasoning in order to manage resources and units. For each game, a set of subgoals was defined, so that the learning component of D2 can learn hierarchical plans. For instance, in BattleCity, goals such as: “get in line with the enemy base” are defined.

A thorough evaluation is part of our future work, but initial evaluation shows that D2 can play at human level (and super-human in some scenarios) in Towers, it can play slightly under-human level in S2, and performs poorly in BattleCity, where it lacks fast reactivity (BattleCity is an action game rather than an RTS game, which was the kind of game for which D2 was designed). Moreover, early evaluation suggests the performance of D2 depends on the quality of the demonstrations, and also on the quality of the set of goals available to D2 for decomposing plans. D2 was tested both

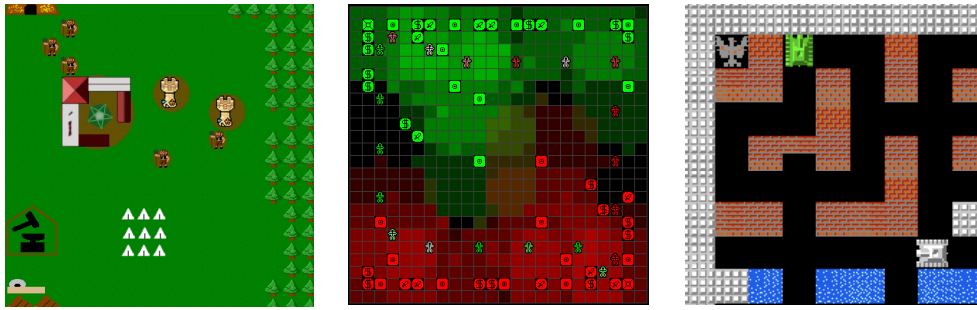


Figure 4: Screenshots of the three games used in our initial evaluation. From left to right: S2, Towers, and BattleCity.

against default AIs, created by members of our group for each of the games, as well as against human players.

In early evaluations we have only provided D2 one or two demonstrations in each experiment. Analyzing the behavior of D2 with an increasing number of demonstrations is also part of our future work.

5 Conclusions and Future Work

In this paper we have presented D2, a case-based planning system that can play RTS games. We have introduced a set of algorithms that can be used to learn plans, represented as petri-nets, from one or more human demonstrations. Finally, we have shown through an initial evaluation of our system that plans learned through these algorithms allow D2 to play a variety of RTS games.

As part of our future work, we plan to perform a thorough evaluation of D2, connecting it to standard domains used by other researchers for comparison purposes. We also plan to explore different plan learning algorithms. In particular, we want to explore the impact of adding loop and conditional learning algorithms to the performance of D2.

References

- [Aamodt and Plaza, 1994] Agnar Aamodt and Enric Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *Artificial Intelligence Communications*, 7(1):39–59, 1994.
- [Aha *et al.*, 2005] David Aha, Matthew Molineaux, and Marc Ponsen. Learning to win: Case-based plan selection in a real-time strategy game. In *ICCBR'2005*, number 3620 in LNCS, pages 5–20. Springer-Verlag, 2005.
- [Bakker and Kuniyoshi, 1996] P. Bakker and Y. Kuniyoshi. Robot see, robot do: An overview of robot imitation. In *AISB96 Workshop on Learning in Robots and Animals*, pages 3–11. 1996.
- [Fikes and Nilsson, 1971] Richard Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3/4):189–208, 1971.
- [Hogg *et al.*, 2008] Char M. Hogg, Héctor Muñoz-Avila, and Ugur Kuter. Htn-maker: Learning htms with minimal additional knowledge engineering required. In *AAAI-2008*, pages 950–956, 2008.
- [Könik and Laird, 2006] Tolga Könik and John E. Laird. Learning goal hierarchies from structured observations and expert annotations. *Mach. Learn.*, 64(1-3):263–287, 2006.
- [McCoy and Mateas, 2008] Josh McCoy and Michael Mateas. An integrated agent for playing real-time strategy games. In *AAAI 2008*, pages 1313–1318, 2008.
- [Muñoz-Avila and Cox, 2007] Héctor Muñoz-Avila and Michael Cox. Case-based plan adaptation: An analysis and review. *IEEE Intelligent Systems*, 2007.
- [Murata, 1989] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.
- [Nau *et al.*, 2005] D. Nau, T.C. Au, O. Ilghami, U. Kuter, D. Wu, F. Yaman, H. Muñoz-Avila, and J.W. Murdock. Applications of shop and shop2. *Intelligent Systems*, 20(2):34–41, 2005.
- [Nebel and Koehler, 1992] Bernhard Nebel and Jana Koehler. Plan modifications versus plan generation: A complexity-theoretic perspective. Technical Report RR-92-48, 1992.
- [Ontañón *et al.*, to appear] Santiago Ontañón, Kinshuk Mishra, Neha Sugandh, and Ashwin Ram. On-line case-based planning. *Computational Intelligence*, to appear.
- [Schaal, 1996] S. Schaal. Learning from demonstration. pages 1040–1046, 1996.
- [Sharma *et al.*, 2007] Manu Sharma, Michael Homes, Juan Santamaria, Arya Irani, Charles Isbell, and Ashwin Ram. Transfer learning in real time strategy games using hybrid CBR/RL. In *IJCAI'2007*, pages 1041–1046. Morgan Kaufmann, 2007.
- [Spalazzi, 2001] L. Spalazzi. A survey on case-based planning. *Artificial Intelligence Review*, 16(1):3–36, 2001.
- [Sugandh *et al.*, 2008] Neha Sugandh, Santi Ontañón, and Ashwin Ram. On-line case-based plan adaptation for real-time strategy games. In *AAAI 2008*, pages 702–707, 2008.