Transfer Learning through Analogy in Games: Capabilities of a HTN-based Approach

22. Juni 2012
Christoph Stocker
Matrikelnummer 1669352
Theodor-von-Zahn-Str. 19
91052 Erlangen
c.stocker.mail@googlemail.com
1 Introduction

Machine learning algorithms made it possible to achieve very efficient results in solving non-trivial problems. For instance, neural network systems are able to recognize facial expressions of people or Bayesian learning systems are able to judge whether a person is likely to get cancer or not.

Nevertheless, these approaches are quite inflexible when it comes to the task of solving 'similar, but not exactly' the same problems. A human who is able to say if a person shows pain in his facial expression is also likely able to say whether this person is sad or happy. A learning system which was trained to recognize pain cannot. Though the task is very similar, it would not reach very good results and if the task would be to recognize whether a dog shows pain in his face, the results would be even worse.

The reason for this is that humans are able to transfer knowledge which they have gained in one learning process and use this transferred knowledge to draw analogies or 'best practices' to enhance learning the solutions for new problems.

Currently a lot of research is done on how transfer learning can be done computationally to improve machine learning algorithms in solving tasks of different domains.

This coursework provides a short overview about what transfer learning is and what it implies. As an example for which efficiency can be already achieved in transfer learning, an approach by Hinrichs and Forbus will be explained, which focuses on solving computer games with learning algorithms and using learned strategies to solve similar games.

In the first part of this coursework, a short overview about transfer learning in general will be provided.

In the second part, the general game-playing (GGP) framework will be presented, which is used by Hinrichs and Forbus as the basic framework to model game logic.

In the third part, hierarchical task networks (HTNs) will be explained, which are the main data structure used by Hinrichs and Forbus to represent transferable knowledge.

The fourth part will then focus on the question how transfer learning can be done using HTNs in the context of this approach.

Finally, a short conclusion is drawn, which will illustrate the benefits of this approach.

2 What is transfer learning?

In general, transfer learning means that knowledge acquired in a source domain is used to guide a learning process in a target domain. In many approaches, like in the one of Hinrichs and Forbus, transfer learning is seen as an analogy problem. There is a problem in the source domain, where a solution is already found. This solution is then applied to a problem of the target domain which requires some kind of analogous mapping to translate the solution from the 'language' of the source domain in the one of the target domain.

Hinrichs and Forbus provide an even more detailed distinction between near transfer and far transfer. In near transfer, the "source and target domains are very similar and solutions can be transferred almost verbatim" ([HF09, p. 70]). In the domain of games,
a near transfer problem would be to carry over a solution to win the game to another instance of the same game, where only some small conditions changed. Hinrichs and Forbus illustrate this on the game Freeciv, an open-source strategy game, were the goal is to allocate resources, build a city and finally conquer all other players cities. The subtask of allocating resources is the same throughout all the games, namely decisions must be made on which parts of the city should be used for cultivation, for mining or fishing. The structure of this subtask never changes, but the starting conditions might change from one game instance to another, e.g. fish might not be available in every game instance, since not every city is located on a coast. Therefore the transfer from one game instance to another requires a respectively simple analogous mapping between the source and target game instance, by taking the different starting conditions into account and adopting the solution of the source game instance to the new task. As this can be done pretty much "out of the box" with analogical mapping and retrieval methods, this is a typical near transfer problem (for further details, see [HF09, p. 71f]).

In far transfer, the source and the target domain share only slight similarities, and a solution must first be generalized to be applicable to the target domain. For instance, Hinrichs and Forbus did some experiments with games called Escape, Wargame and Micro-Rogue. In Escape, the player has to find tools and resources like nails, a hammer and planks and combine them, e.g. to build a bridge and go to the exit on the other side of a river. In Wargame, the player has the role of a soldier, which has to find weapons and shoot enemies before he can exit. In Micro-Rogue, the player has to find treasures and other artifacts like keys, scrolls, potions or magical amulets, which must then be used to solve riddles or kill enemies like snakes. Though these are very different games with very different goals a human player who played Escape might also achieve good results in Micro-Rogue. The reason is that, despite having different domains, the strategies for winning are very similar. It is always necessary to find some kind of resources, combine them and apply them to overcome some kind of threat, and finally find the exit. Therefore there exists an analogy between the games, which a human player can learn and apply to refine his learning process. To do this computationally, a player system must be able to generalize over the solutions and find similarities within the structure of the solutions. Since this is not as easily done as just adopting solutions to different situations, this is a far transfer problem. Far transfer requires some conditions for the games to be transferred, which are discussed in the next chapter of this coursework.

3 The domain of games

Games have always been an exiting way to evaluate the efficiency of AI algorithms. The main reason for this is the competitive character of playing games against other players. If an AI system is able to beat another over several matches, it must be more efficient (at least in playing this on particular game). Some AI systems, like IBMs 'Deep Blue' are so efficient that they can even beat human professional players.
Nevertheless, efficiency is not the only property to judge how 'intelligent' an AI system is. ‘Deep Blue’, for instance, is an AI system which was designed to play chess. Therefore it might be able to beat even human championship players of chess, but it has no idea how to play such simple games like ‘tic-tac-toe’. The reason for this is, that systems like ‘Deep Blue’ are expert systems. Such systems use evaluation functions, which can tell for each move in the game, whether it is a good or a bad move. These evaluation functions are just a representation of human intelligence.

They might be really clever, but they are nothing else but human decisions encoded in the program. Therefore systems like ‘Deep Blue’ might be very cleverly programmed, but they are not acting cleverly by themselves. Therefore, if ‘Deep Blue’ were to play in a competition against other systems it will horribly fail when the game to be played is not chess.

A more sophisticated approach would be to build a system that is able to reach efficient results in many different games. An optimal system shall be able to learn even new games by itself, just like humans do.

Nevertheless a human game player would not learn a game entirely from scratch. People are able to find analogies between similar problems that appear in similar games, and are able to adopt already learned concepts to new but similar situations. Therefore an intelligent system should be able to do that transfer as well.

Unfortunately there are several problems arising with that task.

The first one is that an artificial player must know how to play the game, i.e what the rules of the game are, which actions are legal in which situation, and which are the goals to win the game. To acquire all this knowledge, the game is required to be encoded in such a way that a player can comprehend this. This requires some kind of generalization across all playable games. However the logic of the game might be encoded, winning conditions, legal actions and the current state of the game must at any time understandable to the player. For this matter, the games must be encoded in the same framework and use the same syntax\(^1\).

The second problem is that simply knowing what the goals of a game are and how the game works, does not mean that a player is good in playing the game. A player must be able to choose its actions wisely, which requires some kind of judgement of what actions are good ones and which are not. Unfortunately, the sheer representation of the games rules, actions and goals do not provide this knowledge. The game definitions are purely syntactically and provide no semantic information like which actions contribute to a win and which not.

Therefore a player system must use some kind of planning procedure to find elegant and efficient ways to achieve the goals.

The third problem is that if finally a good and reasonable plan how to win is found, it is not always the case that this plan can be used in playing another game. Some plans might be useful in several games, others might need slight adoptions, and other may be not useful at all. Therefore the player also needs some kind of learning facility, which

\(^1\)Or at least a syntax that can easily be transformed without loss of information.
is able to acquire and store learned knowledge about playing the games, carry it over to other games and adopt this knowledge if necessary.

There might be other problems as well, but this coursework focuses on this three major problems. The first and second can be adressed with quite traditional planning approaches and are therefore covered directly in this chapter.

The last problem opens a far more complex field, as it is a problem of knowledge transfer and transfer learning. Since this is one of the major topics of this coursework, it is discussed in chapter 5.

3.1 The GGP framework

One method to resolve the problems of generalizing game definitions is to use the General Game Playing Framework (GGP). The GGP framework was designed especially to address the problems mentioned in the previous chapter and to make general game playing possible.

To achieve this, the GGP uses a special kind of language, namely the Game Definition Language (GDL). Basically, the GDL is a restricted form of PROLOG. As in other approaches to game playing, the game description is encoded using first order logic statements. According to the GGP framework, a game description contains the following components (see [GLP05, p. 64]):

$S$, a set of game states
$r_1, ..., r_n$, the $n$ roles of a $n$-player game
$I_1, ..., I_n$, $n$ sets of actions, one set for each role
$l_1, ..., l_n$, where each $l_i \subseteq I_i \times S$. These are the legal actions in a state
$n$, a update function mapping $I_1 \times ... \times I_n \times S \rightarrow S$
$s_1$, the initial game state, an element of $S$
$g_1, ..., g_n$, where each $g_i \subseteq S \times [0...100]$
t, a subset of $S$ corresponding to the terminal states of the game

Figure 1: GGP Game Components

A state of a game is a set of first order logic predicates describing the properties of a game in a certain state. The roles correspond to the players, but with respect to the fact that not every player might have the same role, e.g. in games like chess, the two players have the same actions, but one controls the white figures, the other one the black ones.

In other games, the roles of the players might be even totally different, e.g. in a soccer game, one player might be a striker and another the keeper. This is important because the role determines which actions a player can do.

Additionally, not every actions is legal in every state, therefore the GDL description must define some rules to determine when an action is legal or not.

For the game to be carried out, a starting state and one or more final states must also be defined, along with a transition function which defines how actions of players traduce the game from one state to the other. A terminal state is a state in which no actions
can be taken anymore, and the game is declared over. This must not necessarily mean that one of the players has won the game, in some games (e.g. 'tic-tac-toe') a draw is also possible.

To make it possible for the players to determine which terminal states result in a loss and which in a win, there must also be a goal rewarding function defined, which assigns values from 0 to 100 to the terminal states, showing which states are worth to be reached. All of the above game definitions must be expressed using first order logic statements to be comprehensible and executable for the players.

Because these statements are purely syntactical, the GDL defines some special predicates by convention to make sure that players can recognize which statements are legal actions, which are states, etc. For this purpose, the GGP uses a restricted vocabulary of keywords to build special rules defining the statements (see [GLP05, p. 66]):

\[
\begin{align*}
\text{role}(< r >) & \quad \text{means that} \quad < r > \quad \text{is a role (player) in the game} \\
\text{init}(< p >) & \quad \text{means that the datum} \quad < p > \quad \text{is true in the initial state} \\
\text{true}(< p >) & \quad \text{means that the datum} \quad < p > \quad \text{is true in the current state} \\
\text{does}(< r >, < a >) & \quad \text{means that player} \quad < r > \quad \text{performs action} \quad < a > \quad \text{in the current state} \\
\text{next}(< p >) & \quad \text{means that the datum} \quad < p > \quad \text{is true in the next state} \\
\text{legal}(< r >, < a >) & \quad \text{means that it is legal for} \quad < r > \quad \text{to play} \quad < a > \quad \text{in the current state} \\
\text{goal}(< r >, < v >) & \quad \text{means that player} \quad < r > \quad \text{would receive the goal value} \quad < v > \quad \text{in the current state} \\
\text{terminal} & \quad \text{means that the current state is a terminal state}
\end{align*}
\]

*Figure 2: GDL Keywords*

With the keywords from figure 2 provided, a game player can easily perform a valid action just by evaluating all legal statements and choosing one of the legal actions for its next action. The actual game procedure is then done by all players simultaneously applying their chosen action. The state of the game is then updated according to the effects of the players actions.

This works very similar to traditional planning systems. The set of literals describing the game state is changed by adding positive literals resulting from the players actions and deleting all negative ones. In the GDL context, this is done by evaluating the next-statements, which contain conjunctions of true and does-statements. Therefore they define which literals are true in the game state, when a player does a certain action. As the closed world assumptions holds true for the GGP framework, everything that is not explicitly declared as being true is automatically false.

It must be mentioned that the GGP framework has some limitations. Since all players must perform their actions simultaneously (though some players actions might just have no effects) and since the game must have at least one terminal state, the GGP framework can only model games which can be expressed as an finite, synchronous state machine.
This is no problem in the context of this coursework, but generally speaking not every game could be modelled using the GGP. For instance, the number of players is always fixed throughout the game. But in some games, e.g. online games, the number might change if one of the players decides to leave the game.

3.2 Planning in games

In game playing, planning means finding a sequence of actions which leads to winning the game. Since this task depends on some factors, there are some special requirements to the planning algorithms to achieve good results. Genesereth et al. make a distinction between strongly and weakly winnable games.

Strongly winnable games are those in which "for some player, there is a sequence of individual moves of that player that leads to a terminal state of the game where that player’s goal value is maximal" ([GLP05, p. 68]). I.e. those games have a sequence of actions which is always linear and leads always to a win. In general, single player games like Sokoban are strongly winnable. if a sequence of actions can be found to push all crates at the goal positions, the same game can always be won with that sequence. Therefore, for finding a solution for a game, the player can simply apply some kind of forward or backward search throughout all possible sequences of actions. There exist several algorithms to achieve this, like for example the STRIPS algorithm.

In contrast, a game is considered weakly winnable, "if and only if, for every player, there is a sequence of joint moves of all players that leads to a terminal state where that player’s goal is maximal" ([GLP05, p. 68]). In those games, the sequence of actions which a player must take depends on the other players actions. Therefore the sequence of actions which leads to a win is not fixed. There must always exists at least one for every player, but it might change from one game instance to another. Consequently a simple forward or backward search for a possible sequence cannot be applied.

Despite this fact, some other efficient strategies are still available. Since GGP games can be represented as a finite, synchronous state machine a more promising approach would be to find the most beneficial action for every state in the game, or more precisely, for each possible state-to-state transition from this state.

To illustrate this, figure 3 shows the state machine representation of a game. There are two players, which have two possible actions, x or y (although not every action can be taken in every state). The grey states represent goal states, i.e. states with high goal values for the players. If the game is is in state f, the first player can achieve a high goal value by doing action y, independent from the other players action. Therefore this action should be preferred. If the game is in state a, the action y is still preferable, since it leads to a state with a high goal value, although in this state the final outcome depends on the other players action.

---

2The STRIPS algorithm in its basic form cannot be applied to game encoded in GDL, since the GDL does not explicitly define preconditions and effects of operations directly. Nevertheless this knowledge is encoded in the next-statements, since these manipulate the game state. Therefore before applying a STRIPS-like algorithm to a GDL game, the player must find out the preconditions and effects of operations by experimentation.
Since the result of this approach is not a sequence of actions, but a strategy of which actions to choose in which situations, a strategy planning algorithm must be used. Since the other players actions are still involved in the game, even an optimal strategy cannot guarantee a win. From the algorithmic point of view, this is a fact that cannot be avoided. More than doing the best action in each situation is not possible. If there is almost no chance of winning despite using the best strategy, the game can simply be called unfair. In most games however, finding the optimal strategy might surely lead to high success rates.

Nevertheless, finding such an optimal strategy is not always that trivial, since it involves guessing the other players actions. This might be quite an easy task, in cases when the goal value function of the other players are known. Then the most probable action of the other players are those who lead to high goal values. The planning algorithm can then try to avoid states which lead to high goal values for other players and prefer those which lead to high goal values for itself. But in other game settings, the goal value function of the other players might not be provided, and the actions have to be guessed entirely.

Then a learning approach like Q-learning can be applied. In Q-learning, a strategy consists of a table which is used to assign a value to each possible state transition. This table is initialized with the goal values, i.e. in figure 3, the transitions which have the states $c$ and $g$ as a target (or $h$ and $k$ from the point of view of the other player, respectively) will be initialized with the corresponding goal value, all others are initialized with zero. The values are then updated after each action, taking future transitions into account.
account. To not overrate the goal value of a transition, the future state transitions are discounted. This algorithm takes the other players’ action into account by assigning low values to states which benefit to the other players, and assign high values to states which contribute to a win.

![Figure 4: Two very similar Sokoban settings](image)

Although there exist efficient algorithms to plan sequences or strategies, there are further problems when it comes to transferring this solutions to other game instances with different initial settings. To illustrate this, two Sokoban game instances are depicted in Figure 4. The blue circle represents the player, the yellow diamonds are the goals, and the orange squares are the boxes. A possible winning sequence for setting #1 would be 'move-up, move-right, push-up, move-down, move-left, move-left, push-up'. Applying this sequence to setting #2 does not lead to a success. Nevertheless the correct solution is almost the same, the setting requires just one additional 'move-up' before the second box can be pushed to the goal. Therefore transferring knowledge from the solution of setting #1 looks quite promising, but it must be adapted to the new situation.

In the approach done by Hinrichs and Forbus, analogical mapping is used, which means that the structure of the source solution must be raised to a higher level of abstraction, until it matches a structure which can be instantiated to a solution of the target. In the Sokoban setting shown above, this means the subtasks for winning the game must be found, which can then be splitted up into more detailed tasks. Possible subtasks for the Sokoban setting shown above would be 'put-box-on-goal1, put-box-on-goal2'. In

---

3For a more detailed explanation of the Q-learning algorithm, see [Mit97, chap. 13)].
setting #1 the solution for reaching subtask ‘put-box-on-goal1’ would be be ‘move-up, move-right, push-up’. In setting #2, the solution for this subtask is the same, therefore the knowledge transfer was successful. In subtask ‘put-box-on-goal1’ an algorithm must be used to deploy a strategy of how to adopt the source subtask to reach the goals of the target subtask, i.e. in this Sokoban setting, to find out where the missing ‘move-up’ must be inserted.
This will be addressed in chapter 5. Moreover, it becomes clear that a more elaborated data structure is used to model this method of generalization. Hinrichs and Forbus use HTN networks to achieve this, which will be explained in the next chapter.

4 Hierarchical task networks

As it was stated in the last chapter, the data structure for applying transfer of plans with analogical mapping must be able to split up goals into several subtasks. For this purpose, hierarchical task networks (HTNs) are a widely used data structure. A definition of what HTNs are can be found in [Gha04, chap. 11.1, p. 229];

Hierarchical task network (HTN) planning is like classical planning in that each state of the world is represented by a set of atoms, and each action corresponds to a deterministic state transition. However, HTN planners differ from classical planners in what they plan for and how they plan for it.

In an HTN planner, the objective is not to achieve a set of goals but instead to perform some set of tasks. The input to the planning system includes a set of operators similar to those of classical planning and also a set of methods, each of which is a prescription for how to decompose nonprimitive tasks recursively into smaller and smaller subtasks, until primitive task are reached that can be performed directly using planning operators.

The objective of HTN planners match those that were stated in the previous chapter. A plan for winning a game should consist of finding all subtasks which are necessary and structure them hierarchically in the right order. The main difference is that the method descriptions must be provided to construct an HTN. Therefore the notion of a method must be defined.

According to [Gha04, chap. 11.5.2, p. 245], an HTN method is a 4-tuple

\[ m = (\text{name}(m), \text{task}(m), \text{subtasks}(m), \text{constr}(m)) \]

in which the elements are described as follows:

- \text{name}(m) is an expression of the form \( n(x_1, ..., x_k) \) where \( n \) is a unique method symbol (i.e. no two methods in the planning domain can have the same method symbol), and \( x_1, ..., x_k \) are all of the variable symbols that occur anywhere in \( m \).
- \text{task}(m) is a nonprimitive task [i.e. no operator].
A task network is defined as a pair \( w = (U, C) \), where \( U \) is a set of task nodes and \( C \) is a set of contraints[...]. These contraints are in general any combination of boolean expressions that must hold true under certain conditions, but in [Gha04, chap. 11.5.1, p. 245] a more elaborated definition of what a constraint is can be found:

- A precedence constraint is an expression of the form \( u \prec v \) where \( u \) and \( v \) are task nodes.[...] It says that in every solution \( \pi \) for \( P \), the action \( last(\{u\}, \pi) \) must precede the action \( first(\{v\}, \pi) \).[...]

- A before-constraint [...] is a constraint of the form \( before(U', l) \), where \( U' \subseteq U \) is a set of task nodes and \( l \) is a literal. It says that in any solution \( \pi \) for \( P \), the literal \( l \) must be true in the state that occurs just before \( first(U', \pi).[...] \)

- An after-constraint has the form \( after(U', l) \). It is like the before-constraint except that it says \( l \) must be true in the state that occurs just after \( last(U', \pi) \).

- A between-constraint has the form \( between(U', U'', l) \). It says that the literal \( l \) must be true in the state just after \( last(U', \pi) \), the state just before \( first(U'', \pi) \), and all of the states in between.

```plaintext
win()
task: win()
subtasks: \( u_1 = \) put-box-on-goal1()
\( u_2 = \) put-box-on-goal2()
constr: \( u_1 \prec u_2 \)

put-box-on-goal1()
task: put-box-on-goal1()
subtasks: \( u_1 = \) move-up, \( u_2 = \) move-right, \( u_3 = \) push-up
constr: \( u_1 \prec u_2 \prec u_3 \prec u_4 \)

put-box-on-goal2()
task: put-box-on-goal2()
subtasks: \( u_1 = \) move-down, \( u_2 = \) move-left, \( u_3 = \) move-left, \( u_4 = \) push-up
constr: \( u_1 \prec u_2 \prec u_3 \prec u_4 \)
```

Figure 5: HTN methods for Sokoban setting #1

To illustrate this, figure 5 shows the HTN methods required to represent the solution for the Sokoban setting #1 provided in the previous chapter. There exists a method \( win \) which is splitted up into the subtask of achieving goal 1 and goal 2, which is moving
the boxes on the corresponding places on the board. The two submethods consist of sequences of basic operations, which cannot be splitted up further. The order of the sequences is given in the respective contraints.

Since this is a very simple example, many strengths of this data structure are not shown here, but two points that are important for transfer learning can already be stated here. At first, ”HTNs are hierarchical plans that reduce complex task to sequences of simples subtasks, terminating at ground primitive actions to be performed by the agent.” ([HF11, p. 73]) Therefore HTN support generalization, which is an important part of analogical transfer. In the example above, this is done by recognising that put-box-on-goal1 and put-box-on-goal2 are two independet subtasks for winning the game, independently from which actions are necessary to achieve them.

Secondly, HTNs ”are an explicit symbolic representation of plans that can be composed or concatenated, and their hierarchical nature makes it possible to transfer partial knowledge. Missing or incorrectly transferred subtasks can be relearned without rejecting the entire network” ([HF11, p. 73]). This becomes important when the transferred plan is adjusted to cope with the target problem.

The HTN representation can be interpreted like a tree structure where each node can be seen as a subtask to achieve the the task of the parent node, and each leaf node is a basic operator. Therefore tree operations like pruning or refactoring can be applied. By doing this, transferred knowledge which is still viable in the target domain can be kept, while unimportant or even wrong parts of the transferred knowledge can be discarded by removing the respective subtasks and inserting newly learned knowledge, which is specific to the target problem.

In adjusting the HTN plan for setting #1 of the Sokoban example to setting #2, this could be done by removing and relearning the subtask put-box-on-goal2.

5 Learning game strategies by experimentation

Since the aim of Hinrichs and Forbus approach is to build a general game player, there are three problems that are to be solved when using HTNs to plan GGP games.

First of all, HTNs need operator descriptions (like traditional planners) and method descriptions to build HTNs, which the GGP framework does not provide. The input of a player contains only the game definition encoded in GDL. Therefore the operations and methods must be extracted from this description computationally.

Secondly, the actual planning of the game must be carried out, which involves finding out preconditions and effects for operations, finding out subgoals and finding out sequences of actions which contribute to the subgoals and which should be avoided.

Thirdly, since the game player should also be able to draw analogies from other, previously played games, the far transfer must be incorporated when learning to play the

---

4 Although a clever algorithm might recognize that the second subtask can be adjusted by inserting an additional leaf node for the missing 'move-up' operation

5 Since the player should be general, i.e. does not know the games in advance, these cannot be part of the input.
game. The first and second problem can be solved in one step, since both contribute to learning how to play the game, i.e. in creating an HTN winning strategy. Since the third problem does not appear in learning the source game, it must only be considered in the target domain. For this purpose, Hinrichs and Forbus approach can be split into two parts, namely source domain learning and target domain learning. Both learning types follow the same procedure, which is depicted in figure 6 (see [HF11, p.74]).

![Figure 6: Roadmap of Hinrichs and Forbus game learning approach](image-url)

The only difference between the both types is that the analogical transfer step is only done in target domain learning.

### 5.1 Source domain learning

For both types, the input is the game description encoded in GDL. The first step is to carry out a static analysis. In this step, the GDL description of the game is translated into a more elaborated form. Hinrichs and Forbus claim (see [HF11, p.74f]):

This [the static analysis] has two purposes. First, the experimentation strategies rely on background knowledge about games in general, for example, path planning and quantity planning. Applying this knowledge requires the static analysis to extract the representations of coordinates, movement operators, directions, quantities, and potential influences on quantities. The second purpose is that these more abstract representations provide a more effective language for analogical mapping and for experimentation.

The second point is very important for planning the game. Like it was stated in the previous chapter, planning algorithms like Q-learning can be used to distinguish between good states or bad states, with respect to winning the game. If this is carried out on final state machine level, it might be computationally intractable. Properties like coordinates, quantities etc. are semantic, i.e. these cannot be drawn directly from the GDL. Instead, Hinrichs and Forbus use and heuristic approach to find out about these properties. This is done the following way ([HF09, chap 3.1]):

---

6A game like chess has approximately $10^{30}$ states and thousands of possible actions (see [GLP05, p. 65])
By looking at the collection of rules, domain analysis [i.e. static analysis] first establishes some of the simple algebraic properties of predicates, such as whether or not they are functional, take numeric or symbolic values, are transitive, and/or cyclical.

Then, given the assumption of a spatial game, it identifies the most likely candidate predicates for a coordinate system and determines whether the game is a piece-moving or marking-type game and what the tokens or pieces are. Given the defined GDL vocabulary of \texttt{init}, \texttt{legal}, \texttt{next}, \texttt{terminal}, and \texttt{goal}, it extracts any special quantity thresholds that may win or lose the game. Sometimes this takes the form of a deadline (e.g. terminate the game after 50 turns) and sometimes there may be a 'health-like' quantity that must not go to zero.

In order to facilitate generalization and lifting, it computes equivalence classes of game pieces, which are those pieces of the same type that behave the same way and whose only distinguishing property is their location.

![Example of the elaborated representation for the Rogue game](image)

The outcome of this transformation is shown in figure 7. There are game-specific entities defined such as weapons, health, locations, etc..

On basis of these elaborated statements, the next step in source domain learning is the game learning phase, consisting of repeatedly playing the game, running learning
experiments and do dynamic analysis to evaluate the outcome of these experiments. Playing the game corresponds to performing actions depending on what the target of the current learning experiment is. The type of learning experiment determines which things are to be learned. First of all, basic actions are learned, like "the affordances of different entities in the game, the effects of an action, the applicability conditions of an action, and to learn how to decompose a goal into subgoals" ([HF11, p. 75]). Then, when all these basic action are known to the player, high-level actions, "such as going to an entity to find out what happens, trying a primitive action to see what it does, or achieving conjunctive subgoals in different orders" ([HF11, p. 75]) are learned.

The dynamic analysis phase is then judging good from bad actions which were taken in the experimentation phase. Hinrichs and Forbus do this by using preference heuristics, e.g. it "seeks explanations in terms of simple spatial configurations such as colocation [...]") and "[...] some entities may be classified as, for example, obstacles, threats, or hazards" ([HF11, p. 76]). The result of this can be represented by an HTN, which provides the currently known plan to win the game. This HTN can then be refined in each trial of playing the game.

After the learning phase is done, the dynamic analysis phase is applied once again, where the strategy is simplified "by removing tasks whose outcome is never used, such as achieving preconditions of actions that are never taken" ([HF09, chap. 3.3]).

5.2 Target domain learning

In target domain learning, the same steps like in source game learning are performed to build a winning strategy, but before the playing phase is started, the knowledge gained from the source game knowledge is taken into account. This happens directly after the static analysis.

An outline of the far transfer algorithm is depicted in figure 9 ([HF11, p. 79]). Base and target are the source and target game representations, which are then elaborated with static analysis in Step 1. Pr contains the HTN strategies of the source and additional information like information about certain types and relationships, like which predicates correspond to threads or obstacles, etc.

Before this information can be applied in the target domain it must be aligned to match the target domain. This is done in step 2 by applying the two techniques minimal ascension and metamapping. Minimal ascension is using hierarchical dependencies between predicates to match source and target predicates by generalization. Two structures $S_1$ and $S_2$ should be aligned if they are corresponding arguments in some larger mappable structure. "Further let $P_1$ be the predicate of $S_1$ and $P_2$ the predicate of $S_2$. Minimal ascension allows aligning $S_1$ and $S_2$ if $P_1$ and $P_2$ have a close common superordinate." ([HF11, p. 78])

An example for minimal ascension is provided in figure 9 ([HF11, p. 78]). The source game has a scroll, that the hero must use to achieve a goal, like reading it for magically restoring his health. In the target game, the same effect might be reached by drinking a magic potion. Because in the entity hierarchies constructed in the static analysis both are considered to be some kind of type, i.e. something the hero can interact with,
these to predicates can be aligned. While minimal ascension uses hierarchical features to create a mapping, metamapping uses structural properties, like "which relationships hold between them and other predicates" ([HF11, p. 78]). These relationships are also part of the result of the static analysis.

Figure 9 shows an example for metamapping ([HF11, p. 78]). Like in the example for minimal ascension, both in the source and target domain, there is some kind of predicate which is used for restoring the health of the hero. These is obvious from the structure, because both domains have some kind of action \((\text{actionPrimitive})\), which affects the health of the hero \((\text{HealthFn})\) by doing something \((\text{does hero})\). Therefore the actions read and quaff can be aligned.

In step 3 of the algorithm, the parts of the mapping that are not required in the target domain are discarded.

In step 4, the learned and transferred knowledge is translated to match the target vocabulary.

Finally in step 5, skolems are resolved by replacing "variables with variables, and integer constants with the same integer constants. Global constants [...] resolve to themselves. Finally, skolems of locations are resolved to entities in those locations and the locations of their corresponding entities in the target" ([HF11, p. 79f]).

The transfer algorithm is not guaranteed to provide useful knowledge to the target domain. "Minimal ascension and metamapping are both defeasible heuristics that depend on relational structure" ([HF11, p. 79]), i.e. these heuristics can be wrong in some situations. To prevent wrongly learned knowledge from being incorporated into the game.

---

**Figure 8: Outline of the far transfer algorithm**

Input: \textit{base}, a representation of the source game  
\textit{target}, a representation of the target game  
\textit{Pr}, a set of predicates relevant for transfer  

Output: \(CI\), a set of translated \textit{candidate inferences}

1: Elaborate base and target with structural information from static domain analysis  
2: \(M ← \text{best mapping of SME applied to base and target, using minimal ascension}\)  
3: \(CI ← \{c ∈ \text{CandidateInferences}(M) \mid \text{predicate}(c) ∈ \text{Pr}\}\)  
4: Translate non-identical predicates in \(CI\) into target vocabulary, preferring predicate alignments from minimal ascension over metamapping where available.  
5: Resolve skolems in \(CI\)
5.3 Evaluation

In general, the approach of Hinrichs and Forbus uses many heuristics and is therefore defeasible in nature (see [HF11, p. 82]). Nevertheless under most conditions the approach by Hinrichs and Forbus achieves good results.

Hinrichs and Forbus did some experiments to evaluate the outcomes of the transfer algorithm. The measure of success for transfer learning was the difference between the gamescore that was achieved with and without knowledge transfer. For this purpose, games were played both with and without transfer, until both variants achieved the maximal gamescore. The difference can then be calculated between the area under the curve of both outcomes. These is called the *regret*. In figure 11 (see [HF11, p. 81]), the regret scores for different transfer types was measured.
Type C required composing elements of two source domains. In type A, the games had common abstractions. Type V used games with different vocabularies of predicate and entity names. Transfer condition D drew source and target games from entirely different game domains (see [HF11, p. 81]).

For the types C, A, N and V, the results reached up to 60% improvement rates. In type D, there was almost no improvement, which is probably due to the heuristic aspect of the approach. E.g. if a game is not a spatial game, the heuristics fail since this is one of the main assumptions of the approach.

6 Conclusion

The aim of this coursework was to present one approach to construct a general game player, which is able to play arbitrary games and transfer knowledge from previously played games to unknown games. It was stated that there are three major problems occurring.

Firstly, a common framework must be used to standardize games and make each game comprehensible to the player, regardless of what the game is actually about. The solution to this issue is to use a framework like the GGP framework, which uses the GDL as a specific syntax which standardizes properties necessary to play all games, like states, actions and goals.

Secondly, the data structure to transfer knowledge must support generalization as a basis for aligning source and target domain representations as part of analogical mapping.

Thirdly, an algorithm must be provided to facilitate the actual knowledge transfer and incorporate this into learning the game. This is done by applying an approach by
Hinrichs and Forbus, which relies on analogical mapping of similar structures by applying heuristics which are specific to the domain of games, like declaring objects as threats, hazards, obstacles, etc.

This approach reaches good results, as long as the heuristics can be applied.
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>GGP Game Components</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>GDL Keywords</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>A game in its final state machine representation</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>Two very similar Sokoban settings</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>HTN methods for Sokoban setting #1</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>Roadmap of Hinrichs and Forbus game learning approach</td>
<td>13</td>
</tr>
<tr>
<td>7</td>
<td>Example of the elaborated representation for the Rogue game</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>Outline of the far transfer algorithm</td>
<td>16</td>
</tr>
<tr>
<td>9</td>
<td>Example for minimal ascension</td>
<td>17</td>
</tr>
<tr>
<td>10</td>
<td>Example for metamapping</td>
<td>17</td>
</tr>
<tr>
<td>11</td>
<td>Results for different transfer types</td>
<td>18</td>
</tr>
</tbody>
</table>
References


