2007:16 HIP

BACHELOR'S THESIS

Multi-agent planning using HTN and GOAP

Glenn Wissing

Luleå University of Technology

BSc Programmes in Engineering
BSc programme in Computer Engineering
Department of Skellefteå Campus
Division of Leisure and Entertainment

2007:16 HIP - ISSN: 1404-5494 - ISRN: LTU-HIP-EX--07/16--SE

Abstract

This thesis examines how Hierarchical Task Networks (HTNs) can be used to plan for overall strategies in a game environment where the agents are controlled by GOAP (Goal Oriented Action Planning) at an individual level. To solve this, a layer between a HTN planner and a game engine has been implemented, the *Planner Manager*. A simple HTN planner has also been modified to be able to handle parallel tasks. The results suggest that HTNs can be used, it will decrease the work load for the programmers but increase it for the designers during the production process. This is good because the designers should be the ones that are in control during the production phase.

Sammanfattning

Detta examensarbete undersöker hur HTNs (Hierarchical Task Network) kan användas för att planera övergripande strategier i spelmiljöer där agenter är kontrollerade med GOAP (Goal Oriented Action Planning) på en individuell nivå. För att lösa det implementerades ett lager mellan HTN-planeraren och spelmotorn, *Planner Manager*. En enkel HTN-planerare har också modifierats för att kunna hantera parallella uppgifter. Detta examensarbete visar att HTNs kan användas och att det kommer minska arbetsbördan för programmerare men öka arbetsbördan för designers under produktionsfasen. Om man tittar på hela produktionsfasen är detta positivt då detta bör ligga i designernas händer.

Preface

This thesis was written at Avalanche Studios with the purpose of improving their team AI behavior and the production process for these behaviors. I was interested in exploring how Hierarchical Task Networks integrated with Goal Oriented Action Planning could improve such areas.

I want to thank Avalanche Studious and especially my mentor Johan Fläckman.

Glenn Wissing (glewis-4@student.ltu.se), Stockholm, 1. Jun. 2007.

Table of Content

Introduction

Background

Games have focused more on Artificial Intelligence (AI) recently. One of the areas in game AI that has been improved much is planning. Instead of letting Finite State Machines (FSMs) choose deterministic behaviors they can be planned for, resulting in a less deterministic outcome.
Goal Oriented Action Planning (GOAP) are a well known planning technique that has often replaced FSMs in games.

Goal Oriented Action Planning

Goal Oriented Action Planning (GOAP) is a planning technique that deals with only actions [1]. It works by picking a series of actions that would take the agent, or whomever that are to execute the actions, to the goal state.
An action in GOAP is a set of *preconditions* and a set of *effects*. Preconditions and effects are applied on states. A state could be an arbitrary value but in the examples in this section it will be represented by a simple boolean value.

If I for example would have a GOAP goal that is `Kill_Enemy` and its goal state is that `enemy_dead` would be true. For this to be achieved the algorithm would search the space for any action that would satisfy the goal state, i.e. an action that would have the effect that `enemy_dead` would be true. So to achieve `enemy_dead` action `Attack` would be chosen. By applying this action the goal state, a dead enemy, would be achieved. However the action could enforce new goal states, for example to attack an enemy the agent must have his weapon drawn. So by applying the new action `Attack` a new goal state that has not been achieved is enforced. In this manner the planner would perform its regressive search until every goal state is satisfied.

An example like the one above could for example generate a series of action like. `DrawWeapon` followed by `Attack`. Another possibility could be `DrawMeleeWeapon` followed by `MoveToMelee` followed by `MeleeAttack`.

If I look at the latter example with the melee attack the action `MeleeAttack` would have two preconditions, that `in_melee_position` must be true and that `melee_weapon_drawn` must be true. To achieve this action `DrawMeleeWeapon` would be used, followed by the action `MoveToMelee`. But the series of action could in this case just as well have been `MoveToMelee` followed by `DrawMeleeWeapon` followed by `MeleeAttack`. This series of action would not be preferable, it would look stupid if the enemy would rush against the player screaming his lungs out and when he would finally reach his enemy he would stop in front of him to draw his weapon. In this case it might not have been such a big deal, but if I would scale the scenario up it could have resulted in catastrophic behaviors. This is one of GOAPs disadvantages, it does not have the control some scenarios would require to have to be solved in a good manner.

Hierarchical Task Network background

Another planning technique that on the other hand not has been used much in games is Hierarchical Task Networks (HTNs). This planning technique has been around for quite a while but has mostly been used in practical applications such as production line scheduling.

Planning in games

Behaviors in games are often separated in to different levels. For example, having one low level and one high level, where the low level could handle the individual behaviors of each agent and the high level could handle the behavior of the agents together. To explain it further the low level could involve path finding and the high level could deal with team strategies (team strategies could for example be a behavior that sends some agents to flank an enemy while others would suppress him).

If I divide planning into the same levels, the low level would fit GOAP and the high level would fit HTN planning best. In this manner HTNs has been used earlier in the few game cases that are documented [2].

When referring to these different levels a *high abstraction-level* of planning will be used when it concerns team strategies.

Issues

GOAP tend to be complicated in certain areas, meaning it would require much coding to make specific cases work which would slow down a production process. One of these areas are multi-agent planning. Since I had previous experience with Hierarchical Task Network (HTN) planning. I thought that this technique might offer a good solution for multi-agent scenarios since HTNs have been used in similar areas before [2].
This is why I in this thesis have examined how Hierarchical Task Networks could be used with Goal Oriented Action Planning to coordinate multiple agents at a high abstraction-level of planning. A high abstraction-level of planning in a game scenario could for example be the strategy that describes how a group of agents should behave to defend a fort.

This would require me to investigate what type functionality is required in a HTN planner to solve these game scenarios. I had a simple partially-ordered HTN planner implementation which I could extend with the functionality needed.
I would also have to investigate how HTNs could be integrated with GOAP, where and how the transition should occur and at what abstraction-level.
There is also the investigation how to actually use HTN to plan for a multi-agent system.

Thesis Outline

The next section is a theory section where HTN planning and multi-agent planning is described. After this section, I describe and discuss my implementation and the different approaches I chose between. This section is followed by a result section that summarizes how my solutions solve a scenario. It ends with a discussion where I conclude advantages and drawbacks, this section also includes suggestions for future work.

Theory

Hierarchical Task Network

Hierarchical Task Network (HTN) planning is much like classical planning (classical planning refers to planning for restricted state-transition systems [3], could also be known as STRIPS planning) in the aspect that it perform deterministic state transitions between atomic states. The difference between HTN planning and classical planning is what it plans for and how it plans for it [4].
HTN planning does not plan to reach a certain state but rather to perform a set of tasks.

The input to a HTN planner is, like in classical planning, a set of *operators*, there is also a set of *methods*. There is also an initial state and a goal to achieve, the goal is not a set of states, it is in the form of one or more tasks. I will refer to this as the *goal task*.

The methods are what make HTN special. A method is used on a *non-primitive task* (non-primitive tasks can also be known as *compound tasks*), a non-primitive task is an abstract task, a task that the entity that will perform the task do not know how to execute or what results it will produce. Methods decompose the non-primitive task into a set of subtasks. These subtasks are a set of less abstract tasks, they can take the form of non-primitive tasks or *primitive tasks* (primitive tasks are the concrete tasks, can also be known as *actions*).
A method can be used on a non-primitive task if it is assigned to solve the particular non-primitive task and as long as the methods precondition is valid.
A method can also include different types of constraints. These constraints imply some kind of relationship for the subtasks the method contains. A constraint could for example be an *ordering constraint*, a constraint that specifies that one subtask is to be performed before another subtask.

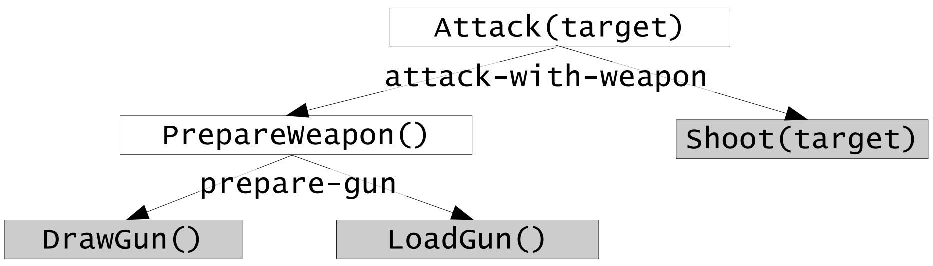


Illustration 1: Task Network example.

This is a task network. The first task `Attack(target)` is a non-primitive task that are decomposed into `PrepareWeapon()` and `Shoot(target)` by the method `attack-with-weapon`. `Shoot(target)` is a primitive task (gray background).

When a primitive task is encountered an operator is applied. The operator performs a deterministic state transition. The operators precondition must also be valid for the operator to be used.

The planning procedure works in this way, when a non-primitive task is encountered a method is used to decompose it. This is repeated until a primitive task is encountered, then an operator is used that changes the current state and the primitive task can be added as a task to perform in the plan list. In this manner the procedure proceeds until there are only primitive tasks left in the decomposition tree and an operator has been applied to every primitive task.

The methods and operators are saved in a so called d*omain*. This domain becomes the information on how a game *scenario* (*HTN problem*) can be solved.
The initial state and the goal task are saved in a so called *problem*.

Using a domain on a specific problem is all the input required to generate a plan, a series of actions that will solve the scenario.

The simplest way of HTN planning uses totally-ordered decomposition. This implies that the ordering for methods subtasks is always fixed, often in the order which they are defined. This result in a limitation in the expressiveness, the ability to express problems.
A more expressive form of decomposition is the partially-ordered decomposition. With this the ordering of tasks is not fixed, i.e. the ordering can remain unspecified.
If I have a set of three tasks, there can be an ordering constraint that says that the first task must be preceded by the second and the third task, the ordering between the second and the third task remains unspecified. This will result in two possible orderings of the task, one where the first task will be followed by the second and then the third, the other where the first task will be followed by the third and then the second.
This indeed increase the expressiveness, but it also increases search space so if you do not really need to have partial-ordering usage has shown that it is good not to overuse it.

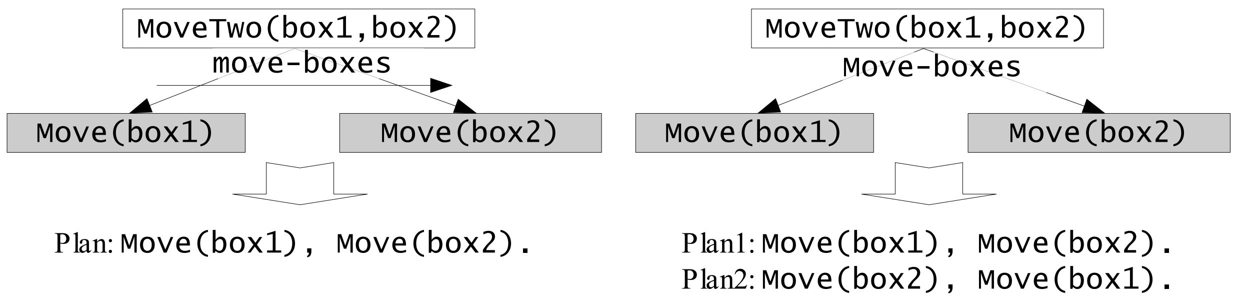


Illustration 2: Total-ordering versus partial-ordering.

The left example is a totally-ordered example, the orderings of the subtasks are fixed. The right example is a partially-ordered example where the ordering is unspecified resulting in two possible solutions.

Multi-agent planning

Multi-agent planning differs from normal planning. In normal planning there usually is one source that will complete a plan by doing a series of actions. Since there is only one source no sharing of resources can conflict, therefore this does not have to be taken in to consideration in the general case. However if multi-agent planning is to be performed this must be taken into consideration.
This can be bypassed however, without any changes in the planner. I do not really need such an advanced planner on this aspect because I am dealing with a game scenario, using functionality that already exists is enough. It will not really solve the issue but it is possible to design solutions to

scenarios anyway. The designer will have to be aware of this because the issue will be solved by him or her at the HTN problem design level, or rather bypassed. The expressiveness for the scenarios that will be solved is enough, if the scenarios became more advanced however this problem might have to be solved.

The problem with resource sharing occurs because of parallel tasks. Parallelism can not be done effectively without changes and since this is a very important part of the planner, which really is needed in a game environment, something has to be done in this area.

To solve problems like these, scheduling is often integrated with planning. Much work in this area has been performed but I can find no data on performance at very low times which leads me to believe that this can be hard to perform in a game environment where every millisecond counts. Therefore, and because of my limited time, I did not put focus on how to integrate scheduling in my planner but rather look into how parallelism can be solved without drastic changes to a simple partially-ordered planner.

Implementation

Earlier multi-agent planning with HTNs has been done by planning a scenario and then starts every task in the plan at the same time. When an entire plan is started in the same time the tasks can not be seen as a series of tasks but rather a group of tasks. This means that I can not plan for a scenario that involves tasks that are meant to be performed in a series. For example if two agents shall break in to a mansion, to be able to enter one agent have to disable the alarm before the other one can enter and eliminate the guards. This limits the ability to express scenarios too much and therefore I chose to not create an implementation that works in this manner.

I earlier stated that the resource sharing will be handled by using the already existing functionality, the only problem in the planner I have to solve is the parallelization.

At first I made an implementation similar to the technique involved in the paper written by Craig A. Knoblock [5], using a post-process step on a partially-ordered planner output to generate a parallel plan. Since a task in a normal plan is executed instantaneously it has no ends or starts. The algorithm I wrote worked by checking every task in every plan for when it at its earliest point can be started and at its latest point can be finished and with this information I created a start and end for every task. By then putting these starts and ends together I get the resulting parallel plan, what I call a *parallel stack*. In a parallel stack the first layer is a group of starts for some tasks and the second layer is a group of ends for some tasks that had already started. The parallel stack follows this pattern until every task is started and ended.

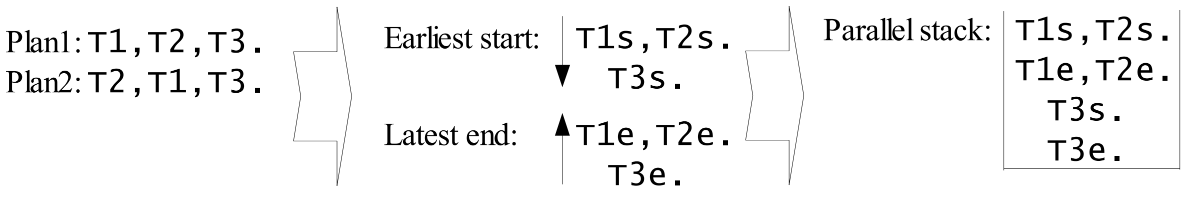

Plan1: T1,T2,T3.
Plan2: T2,T1,T3.
Earliest start: T1s,T2s. T3s.
Latest end: T1e,T2e. T3e.
Parallel stack: T1s,T2s. T1e,T2e. T3s. T3e.

Illustration 3: Parallel stack creation.

With partially-ordered plans the parallel stack could be created by this algorithm. `T1` and `T2` in the first and second plan are interpreted to be parallel.

This algorithm works, but there are some restrictions.

The first is a problem that occurred because of the nature of the partially-ordered planner, the second has to do with the definition of how a task is parallel.
In this case the definition for two parallel tasks is: *"parallel tasks must be started at the same time and all the parallel tasks in that start must be parallel*". This is not favorable because it can lock tasks to be started at undesired occasions when there really is no need to do that, this makes it impossible to create solutions like the one in *Illustration 4*.

The second problem occurs when I want to generate all possible plans, with a partially-ordered planner, for a scenario that includes five tasks that all can be parallel. The number of plans this will generate is 120 plans, O(n!). This is clearly not acceptable, it will increase the planning time and at the same time increase the post-process time. This is the problem that made me decide not to use this algorithm.

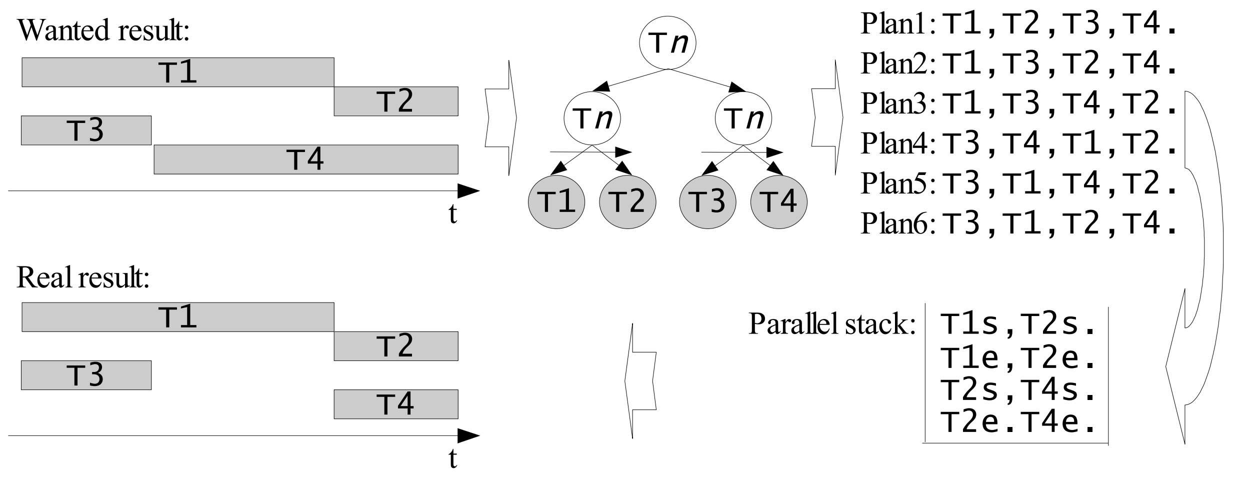


Illustration 4: Parallel stack algorithm problem.

To get the wanted result I will create a task network like the one in this example. That task network will result in six plans. By applying the parallel stack algorithm a parallel stack is created. The parallel stack however does not match the wanted result. The problem with the large amount of plans also appears in this simple example.

The second solution to the parallel planning is a much better solution and it is what I have used in my implementation. The definition for parallel is changed to make it much more flexible, the new definition is: *“parallel tasks can be started and ended independent of each other”*.
To avoid the search-space problem that a normal partially-ordered planner gives when planning for parallel tasks like in the first solution changes to the actual planner has been done. The changes involve adding a new type of task constraint. This constraint is inspired by [6] and I call it a *parallel constraint*. This constraint defines if tasks can be done in parallel, note that they *can* and not *have to* be done in parallel.
However since I do not have any special functionality that are handling resource sharing there is no logical checks between parallel tasks that will deal with conflicts. The planer will assume the user that wrote the HTN problem do not mark tasks as parallel if they can not in every case be parallel. This is one of the weaknesses, it will make it possible to create scenarios that can be solved but might conflict in the actual plan execution. If this where to happen it would result in a loop where the planner will output the same impossible plan every time followed by GOAP failing the task every time.

For example if there are two agents that are to drive a car to destination A and the designers defines that they can both drive at the same time (the designer thinks that there are two cars) but there is only one car. Then they will both try to get into the car, one of them will fail and the plan will be aborted. The state has not changed and the planner will again output a plan that tells them to drive to destination A at the same time which will again result in failure.
The burden for this issue will be placed on the HTN problem designer.

If two tasks are marked as parallel tasks by a parallel constraint there will occur no branching during the planning procedure as it would if they will remain unspecified, unordered. This is one of the greatest performance gains this change has given, from O(n!) to O(n) in a parallel case.
The plan outputted by the planning procedure is saved in a *plan container*. This container also

works as an interface between the user and the plan outputted by the planning procedure. Without this interface the plan outputted by the planning procedure can not be read correctly. The plan contains information about the parallelism which makes it possible to use the plan container during execution. During execution the plan container can give you the tasks that can be started and it will keep track of which is started. When a task has been completed this must be registered in the plan container so that it can be updated, allowing the succeeding tasks to be started.
Because of the plan container a task that is to succeed some other task can not be started until the preceding task has been completed. This is correct in the general case, however the special case that occur when I deal with tasks that are supposed to be preformed until some other task is completed can not work in this implementation.
To solve this there are two possible approaches.
The first is to solve it in the engine, i.e. let the agents that are performing the tasks communicate and tell each other of the progress so that when this special case is encountered the agent will be notified by other agents when he is supposed to complete his task. This approach needs some changes in the planner or some kind of information tag that lets the agents know when a task of this type is present.
The other solution does not require any changes in the planner. It requires a little change in the handling of tasks by the agents but its a minor problem, the change imply that when a agent is working on one task and gets a new task the new task must override the old task and the old task must be reported completed.
I chose to implement the second solution because it contained less overhead.

HTN Planner

As I mentioned earlier I already have a HTN planner implementation that I developed in the Specialization Project course I studied before this thesis work. It is a partially-ordered HTN planner that can only handle constants as a literal.

The change that has to been done is to make it able to handle parallel constraints.
A lot of work has been done on the parser which has to be able to parse the extra information, such information as parallel constraints and operator masking (more about the masking in the Planning Manager section).

Planning Manager

To handle the communication between the HTN planner and the game engine an interface is needed. I implemented such an interface and I call it the *Planning Manager*. This interface will use a HTN planner to solve a scenario when a HTN problem is inputted and then work as the layer between the plan and the agents that will perform the actions in the plan.
The goal is to lay as little burden on the designers and the programmers as possible so that the production of HTN problems will be fast and easy.

When I designed this manager I tried to make it as much stand alone as possible from the engine but also from the HTN planner.

The Planning Manager can be divided into two parts. The part that handle a preprocess step and the part that handle a runtime step.

The preprocess step will create a new file called the *processed file*. This file will contain the original HTN problem and the result of the preprocess.
The preprocess is designed to be able to work offline, this to lower the load in the runtime step and so that this step easily can be done in a tool.
Since I want to do as much as possible offline, I put everything that is suited for an offline process in this preprocess step.
The runtime step will then use the data created in the preprocess to be able to make the actual planning in-game. I considered to make the planning offline to and then letting the runtime step do a lookup in some database over plans matching HTN problems and in-game states, I abandoned the idea because the time it actually takes to do the planning is not that very high compared to the entire process. I have not done much testing in this area but if the planning time will become an issue doing the planning offline can be a possibility to consider.

The three main issues for the manager to solve are the conversion of HTN tasks to GOAP goals, the evaluation of the initial state and the part of the program that hand out the tasks to the specific agents. The solutions for these parts are pretty much integrated but I will separate them and point out the issues and solutions to them.

Conversion of HTN tasks to GOAP goals

The conversion of HTN tasks to GOAP goals have to be done, because the agents do not understand what a HTN task is and what to do with it. This gets especially hard when it is up to the person that designs the HTN problem to create the primitive tasks.
One possible solution for this can be to predefine every primitive task, i.e. predefine every operator. With this solution no extra burden will be placed on the designer, which is good. However it will also remove some of the ability to express scenarios. It will be hard to define every operator and at the same time it will probably not be appreciated by the designer if he or she did not have any control of how the operators work.
Therefore I went with my other solution, to let the designer map the connections of HTN tasks and GOAP goals. The downside with this is the extra burden it puts on the designer, it can probably be hard to come up with every possible HTN task combination to map to GOAP goals. The advantage instead will be that this will not restrict the designer and I think this is very important, it also does not require a programmer letting the designer work without having to go to a programmer for this. Therefore I chose to implement this solution.
The actual mapping of a HTN task to a GOAP goal is done based on the HTN tasks task name and its parameters and the GOAP goals name. The GOAP goal is just a simple name, a string. The HTN task on the other hand contains a name and a set of parameters, where the name and the parameters are strings. It will be possible to simply map the HTN tasks name to the GOAP goals name but it will not be preferable, it is an unnecessary limitation. It will be much better to also use the parameters. If there are two HTN tasks, `Attack(GUARD1,AGGRESSIVELY)` and `Attack(GUARD1,CAREFULLY)` the second parameter in the two example tasks can serve as a flag that will influence the mapping. However, this also adds a problem. There can very well be parameters that you do not want influencing the binding, for example the first parameter in the tasks above. You will probably not want different GOAP goals depending on which agent that are supposed to carry out the task (in my implementation the first parameter in a HTN task is always the object that are to perform the GOAP goal the HTN task is connected to).

To solve this, a new feature called operator masking is implemented. Operator masking works in the manner that it allows the designer to define what parameters that shall be ignored in a primitive task, the definition is defined and saved in the operators which will then later transfer their knowledge about the mask to the tasks. By using this, an identifying string can be extracted from a task that the designer can then map to a GOAP goal.
If I take the same tasks mentioned above I can for example define in the operator that works on `Attack(target,how)` that the first parameter will be ignored. This will enable me to map `Attack+AGGRESSIVELY` to the GOAP goal `Attack_Aggressively` and `Attack+Carefully` to the GOAP goal `Attack_Carefully` without having to consider who will perform the task.

The information for the task mapping is saved in the processed HTN problem file.

State evaluation

The evaluation of the initial state also have to be done. If just planning is performed the initial state is often defined in the problem definition, this is also the case for my HTN planner implementation. If I want to use planning in an interactive environment the initial state has to correspond to the state of the world. I wanted to do as few changes to the HTN planner as possible and I chose the approach to evaluate the initial state before the planning is started. I want to come up with a way to declare every possible state that can exist and then evaluate them.
Since I felt that all the information required to come up with every possible state should somehow exist in the domain I tried to extract the information from it. This turned out to be harder then I thought. You can find information such as how many parameters a predicate can describe but try to find the information which specific constants that every predicate use is not effectively possible. This is because there is no information that make it possible to set apart constants from each other except their names. Even if the constants do not have any type technically speaking the designer will think of it as a type. For example the predicate `at`, that predicate can be used to describe a unit's position, `at(AGENT2,LOCATION5)`. If I look at the example predicate it can technically also be `at(LOCATION1, LOCATION2)` since constants is typeless, to say that a location is at another location will not be logical. However when the designer created the predicate he or she assume that the predicate will always have parameters where the first constant will always be a unit and the second will always be a location.
The only one with this knowledge is the designer but I do not want to put this issue on the designer, because the solution if the designer will solve this will require the designer to declare every possible state that can be true. This will not be good because to foretell every possible state will not be easy and if the designer will miss any state this will not result in any technical issues showing that a state was missing when a scenario that is be possible to solve suddenly is not.
To solve this issue without the designer's interference and the information available in the domain the only possibility is to take every predicate and then match it with every constant, this will result in a gigantic amount of states which make this solution practically impossible. Evaluating this state will take too much time.
So the solution I have left is to simply give constants and predicate parameter types. This will allow a process to only find the combinations that will be logical. This solution will require every constant and predicate to be predefined, this will also make the actual implementation much easier. By using predefined constants and predicates the initial state where every possible state exists can easily be done. This is also something that can be done in the preprocess step so the time it would take to

calculate this initial state is negligible. Using this solution will also ease the designers work since he or she does not have to think of the initial state at all. The downside though is that if the designers want a new type of predicate or constant a programmer has to define it in code, this will demand that a programmer is accessible during the production of scenarios. This is the solution I have implemented because I think it is the best solution.

Now I have a method to create the initial state where everything is true. It is this initial state I will evaluate in runtime before the planning is done. There is still an issue left, how to get the world state. As I mentioned earlier I want to keep my implementation as stand alone as possible, to do this I create an interface that works between the engine objects and the Planning Manager.

Engine interface

The engine interface will be associated to the constants types, a constant named `AGENT1` for example can refer to a type named `agent` and be represented by the engine interface that is associated with the agent object in the engine. The interface is required to be inherited by every engine object that a constant can refer to.

With this I have the possibility to connect engine objects to constants which is the next think that must be done. Since a constant actually represents an engine object during planning it must of course represent the correct engine object, the specific in-game agent for example. This connection is made in the design step by the designer. The designer can pair engine object identifiers with constants. This is not saved in the processed file outputted by the preprocess step, it is saved in a separate file called the *constant connection file*. This is because the processed file is still generic for the same type of scenarios but when the constant connection is added to the scenario it will only work for a specific case (unless the objects do not have the same engine object identifiers).

When this connection is done, i.e. I know what each constant is representing, the initial state evaluation is finally possible. Each predicate has its own evaluate function that can get the information need through the engine interface to do the evaluation for the specific state. If the specific state is false it will be removed from the problem definition in the specific in-game case, letting the planner plan with only the state that are true.

Task distribution

When the planning process is done a plan is outputted if there is any plan that can solve the specific scenario under the specific world state. This is where the third main issue has to be solved, how the plan will be distributed among the agents.
The solution for this issue is integrated with the engine interface since every object that are supposed to carry out a GOAP goal has a engine interface attached because it will serve as a constant (the first parameter in every primitive HTN task). To the interface three functions is added, one that allows the manager to give the engine object a GOAP goal, one that allow the engine object to report back to the manager when it has completed or failed its GOAP goal and the last one that allows the manager to abort the GOAP goal the engine object is performing if the plan for example will fail.
This enforces only one engine object to carry out a GOAP goal, if I have a team of agents there must be a team object in the engine and that object will have to deal with the GOAP goal among its

agents internally.

Now all the functionality to prepare and execute a HTN problem exists.

Re-planning

If a scenario fails at the GOAP level it will result in the entire HTN plan failing. When this happens the manager will re-plan for the scenario in such a way that it will start the entire planning process all over again. It will evaluate the state and then do the planning, when a new plan has been planned it will proceed with the plan. This will happen automatically until the scenario can no longer be solved by the HTN planner.
This can cause one big issue. If for example all different version of a HTN problem result in plans that starts by moving to a specific area this will result in the agents trying to get there again. Issues like this will have to be solved in the HTN problem. At first this was solved in the manager but this was not a good solution because it can not, at that point, be decided if a task is of the type that it shall be done again or not.
Take for example a puzzle game. In a chamber I will guess a color, if it is the right color I win, otherwise I will start from the beginning again. To enter the next chamber a button has to be pushed outside the chamber. A plan that will solve this can be `Push_Button` followed by `Enter_Chamber` followed by `Guess_Color`. If the GOAP goal `Guess_Color` fails the Planning Manager will re-plan the scenario and the correct output for this scenario will be the same plan again.
However, if the rules for this puzzle game is the same except that I do not have to start from the beginning if I guess wrong, the correct plan I want when re-planning is only `Guess_Color`, since I are already standing in the room.
The difference between these two problems is only known by the designer, the Planning Manager can not distinguish the difference. Therefore this has to be solved in the HTN problem.

Optimization possibilities

It is also possible to do some optimizations. One optimization can be to remove states that the designer knows will not happen from the processed file. Another possibility will be for someone who is more familiar with the HTN planner to do optimizations in the domain, to write a faster domain simply.

Extra documentation

To get a more detailed understanding about my implementation have a look at Appendix A, B and C.

Appendix A contains a very simple class diagram that shows the relation between the interfaces. How new predicates inherit from `CPredicateInterface` and are added to the `CPlanningManager`. How new constant types are derived by the game engine and how they inherit from `CEngineInterface` and are added to the `CPlanningManager`.

Appendix B contains example files of a HTN problem file and a constant connection file.

Results

By using the Planning Manager it is possible to solve a game scenario.

First a HTN problem would have to be designed, a description on how a scenario could be solved. An example of a HTN problem like this can be read in Appendix C.
When designing a HTN problem the designer must have a set of constants and predicates, these must have been predefined by a programmer because it was the best solution to create types for constants and predicates.

During the last phase of the HTN problem design the HTN problem would be preprocessed by the Planning Manager, the Planning Manager is the interface that handles communication between a HTN planner and a game engine. This preprocess step would put together the information required by the planner to solve HTN problems in the runtime step, information such as the state that are supposed to be evaluated and how to convert HTN tasks to GOAP goals.
The state that are supposed to be evaluated was created by putting together every logical combination of predicates and constants. It was in the creation for this state the types of constants and predicates was needed. If there was no types defined for constants and predicates it was not possible for the state to be automatically generated, there would be knowledge of what a logical combination would be. Instead this would have been required to be done by the designers and that would have put an unnecessary burden on them.
The conversion of HTN tasks to GOAP goals was saved as simple maps in the processed file. The HTN task is mapped on its task name and specified task parameters. The GOAP goals was simply mapped on their name. To be able to choose which parameters to ignore, a mask was added in the operators.

The processed file, together with a constant connection file, could then be used by the Planning Manager during runtime to solve the scenario. The constant connection file would contain the identifiers for the specific engine objects that would be represented by the constants in every specific HTN problem. The reason to save the constant connection in a separate file was to keep the HTN problems generic.
The Planning Manager would be triggered by the game engine when a scenario should be solved, letting the Planning Manager know what processed file and constant connection file it should use.

The Planning Manager would begin with evaluation the state so that it would concur with the in-game state. Then it would perform the HTN planning, resulting in a plan. To be able to create HTN problems that can represent multi-agent scenarios a new constraint was added, the parallel constraint. With this constraint it was possible to describe parallel tasks. However, there are no checks between parallel tasks for conflicts that can occur when resources are being shared, the designer will have to think of this and that is the downside of this simple change.
The manager would then distribute the actions in the plan until the plan would have been completed. The distribution was done through the interface in the manager that was connected to the game engine, the same interface that would allow the manager to get data from the engine. The first parameter on a HTN task is the object that are supposed to carry out the GOAP goal the HTN task was mapped to.
When the Planning Manager has been triggered to solve a scenario it would continue until the scenario are solved or could no longer be solved.

Discussion

Conclusion

HTNs together with an interface could make group behavior possible. It could improve the group behavior if designed and used in a correct way.
It would still require a programmer to be present during the production process but he or she would be present much less then compared to solving the same scenario with GOAP.
I thought the solution I chose, to predefine predicates and constants, was superior to the alternative solutions I considered, the ones mentioned in the implementation section. I did not think the fact that a programmer would be requires are much of a problem.

Using my implementation would unfortunately increase the performance load, it is a bit calculation heavy. The load is not even, there are many big peeks instead. However, the peeks would not be hard to load balance. The only calculation heavy part that can be questioned if it is load balance friendly would be the state evaluation. I probably would want to do the entire evaluation in one frame, otherwise there could be a risk that the state would change during the evaluation and this could cause difficulties.

HTN planning would also put some burden on the designer, it would be a new technique to learn and it would be a bit more complicated than GOAP. The burden would probably be greater in the beginning of a project but when that threshold would be overcome the rest would probably be faster, i.e. looking at the entire production process, using HTNs would probably speed it up.

To reach a conclusion that would evaluate how good it would work to use HTNs with my Planning Manager a lot of personnel input would have to be done. Designers would have to test the HTN technique and programmers that have worked with the game engine would have to look into how good the Planning Manager is integrated with the engine.
First when this input would be known a real evaluation could be done.

The question of where the transition between HTN and GOAP should occur, where the abstraction-levels should meet, was also hard to answer without input from users. It is both a question of performance and scenario design. Since I am dealing with an abstract level of planning, the line where the transition should be made is therefore probably also abstract. The answer could not be answered in this thesis.

Future Work

A lot work could be done to improve the HTN planner. There are two areas I would especially want to look into, negative preconditions and the design for how preconditions are evaluated. The later is not really relevant but the negated preconditions would affect a lot.
If there would be no negated preconditions in the planner, solving problems where those are needed would require an extra predicate that describes the negation, for example `not-have`. This will require an implementation of the predicate in the Planning Manger, so if every predicate would have a negation possibility the programmer would have to do twice as many predicate implementations. Also a lot of unnecessary state evaluations would occur which would decrease

performance.
There is also some functionality like weights and external function calls that would be good to have.

A tool would make the creation of HTN problems much easier for a designer. Since HTNs looks like a tree-graph a tool could be solved in a manner where it lets the user connect nodes.
A tool would also help with the other things a designer have to do, such things as to map HTN tasks to GOAP goals and connect constants to engine identifiers.
The tool would of course also include the preprocessing step, i.e. the outputted file would be finished to run.

There could of course be improvements in the Planning Manager. Functionality one could want is the ability to handle more than one plan at once, which was not possible in my implementation.
Another example of functionality could be to make predicates handle more than one set of parameter definitions, a form of polymorphisms.

References

[1] Jeff Orkin. (2004) “Applying Goal-Oriented Action Planning to Games” In: *AI Game Programming Wisdom 2*. 2004. Pp. 217 – 227. Charles River Media. ISBN 1-58450-289-4

[2] H. Hoang, S. Lee-Urban, H. Muñoz-Avila. 2005. *Hierarchical Plan Representations for Encoding Strategic Game AI*. Proceedings of Artificial Intelligence and Interactive Digital Entertainment. AAAI Press.

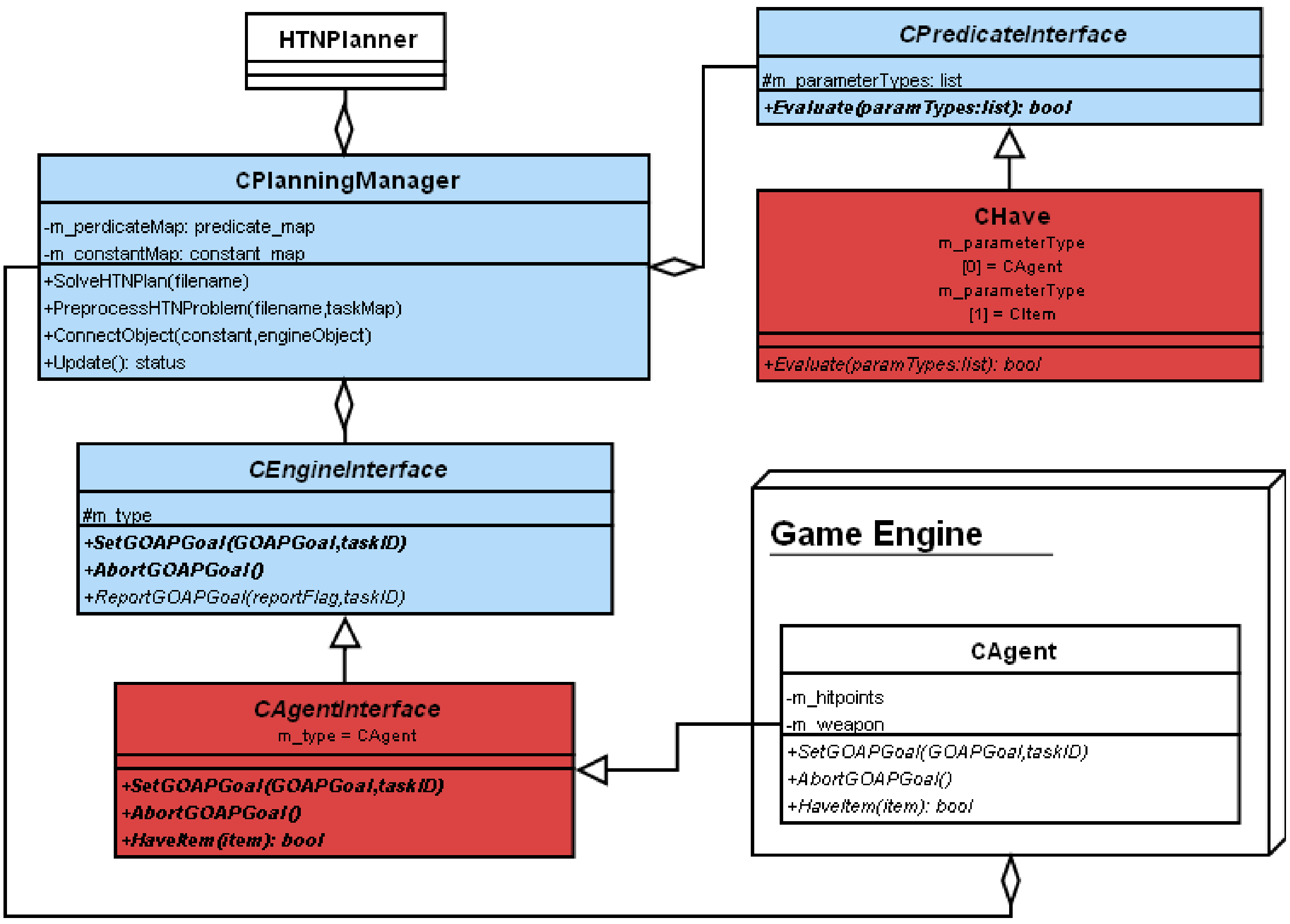
[3] M. Ghallab, D. Nau, P. Traverso. (2004) “Classical Planning” In: *Automated Planning: Theory and Practice.* 2004. Pp. 17 – 18. Morgan Kaufmann. ISBN 1-55860-856-7

[4] M. Ghallab, D. Nau, P. Traverso. (2004) “Hierarchical Task Network Planning” In: *Automated Planning: Theory and Practice.* 2004. Pp. 229 – 261. Morgan Kaufmann. ISBN 1-55860-856-7

[5] Craig Knoblock. (1994) Generating parallel execution plans with a partial-order planner. In: *Proc. 2nd Intl. Conf. on A.I. Planning Systems.* 1994. Pp. 98- 103. Morgan Kaufmann.

[6] L. Castillo, J. Fdez-Olivares, Ó. Garcia-Pérez, and F. Palao. 2005. *Temporal enhancements of an HTN planner*. Spanish Conference on Artificial Intelligence, CAEPIA 2005.

Appendix A


HTNPlanner
CPredicateInterface
#m_parameterTypes: list
+Evaluate(paramTypes:list): bool
CPlanningManager
-m_perdicateMap: predicate_map
-m_constantMap: constant_map
+SolveHTNPlan(filename)
+PreprocessHTNProblem(filename,taskMap)
+ConnectObject(constant,engineObject)
+Update(): status
CHave
m_parameterType [0] = CAgent
m_parameterType [1] = CItem
+Evaluate(paramTypes:list): bool
CEngineInterface
#m_type
+SetGOAPGoal(GOAPGoal,taskID)
+AbortGOAPGoal()
+ReportGOAPGoal(reportFlag,taskID)
Game Engine
CAgent
-m_hitpoints
-m_weapon
+SetGOAPGoal(GOAPGoal,taskID)
+AbortGOAPGoal()
+HaveItem(item): bool
CAgentInterface
m_type = CAgent
+SetGOAPGoal(GOAPGoal,taskID)
+AbortGOAPGoal()
+HaveItem(item): bool

This is a simple design overview of my implementation.
The blue classes are the classes that are the Planning Manager.
The red classes are example classes that have to be created for every new predicate or type of constant.

`CPlanningManager` is the core class. This is the class that the engine should work with to operate the Planning Manager. This class contains the predicate and constant definitions. It also contains the actual engine objects during a planning procedure, these are connected to the class through the function `ConnectObject`.

`CEngineInterface` is the interface that all new constant types are to inherit. The engine classes that will represent engine object during a planning procedure are in turn supposed to inherit the new constant type.
`CAgentInterface` is an example of a constant type, this predicates are of the type `CAgent`.

`CPredicateInterface` is the interface that all new predicates are to inherit.
`CHave` is an example of a predicate which has defined parameter one as the type `CAgent` and parameter two as `CItem`.

Appendix B

Example file *Example HTN-problem.txt*:

```
#<- Is used to write comments, affects the whole row.
#This is an example for how to write an HTN-problem file.
#White spaces and new lines have no affect on the parsing of the file.
#Identifiers: ':' - Primitive tasks is started with this identifier.
#            '!' - Negated predicates is started with this identifier.

#Constant declaration
constants( CONSTANT-1, CONSTANT-2, LOCATION )

#Predicate declaration
predicates( predicate1, predicate2, at )

#Variable declaration
variables( var1, var2, var3 )

#Domain declaration, methods and operators. The order which methods and operators is written does not matter.
domain(

  #A method declaration
  #This method decomposes the task "Goal-Task-1( var1, var2 )" into the tasks "TaskNonPrim1( var2 )",
  #"TaskNonPrim2( var1 )" and ":MoveTo( var2, var3 )". Where the tasks "TaskNonPrim1( var2 )" and
  #"TaskNonPrim2( var1 )" can be performed in parallel and both of the must be performed before the task
  #":MoveTo( var2, var3 )". For the planner to be able to use this method the precondition must be valid.
  Method-name(
  ( var1, var2, var3 ) #Method head, include all variables used in the method here.
  ( Goal-Task-1( var1, var2 ) ) #Task this method can be used in.
  ( predicate1( var1 ), predicate2( var2 ), at( var2, var3 ) ) #Precondition for this method.
  ( TaskNonPrim1( var2 ), TaskNonPrim2( var1 ), :MoveTo( var2, var3 ) ) #Decomposition tasks. The tasks that the
                                                                        #task this method is used on is supposed
                                                                        #to be decomposed into.
  ( (1<3) (2<3) [1,2] )) #Ordering and Parallelism declaration. '(' & ')' is used for ordering-constraints,
                         #'[' & ']' is used for parallel-constraints. The number represents the order which the
                         #decomposition tasks is written. In this example 1 is "TaskNonPrim1( var2 )".

  Method-1( ( var1 )
  ( TaskNonPrim1( var1 ) )
  (  )
  ( :TaskPrimitive1( var1 ) )
  ( ))

  Method-2( ( var1 )
  ( TaskNonPrim2( var1 ) )
  (  )
  ( :TaskPrimitive1( var1 ), :TaskPrimitive2( var1 ) )
  ( (1<2) ))

  #An operator declaration
  #This operator can be performed on the primitive task ":MoveTo( var1, var2 )" and it will change the state by
  #changing "predicate2( var2 )" with "predicate1( var2 )" if the precondition is valid.
  :MoveTo(( var1, var2 ) #Operator head, include all variables used in the method here. An operator is used on a
                     #primitive task that has the same name and parameters as the operator name and its head.
                     #The operator must be ground to work.
  ( 1 ) #Ignore mask. This lets you specify if you want to ignore any parameters for the task mapping. Normally
       #you would want to ignore the first parameter which is always the object that are supposed to perform the
       #task. An comma is used to separate more than one param for example ( 1, 2 ). The number represents the
       #order which the params is written. In this example 1 is "var2".
  ( predicate2( var1 ) ) #Precondition for this operator.
  ( predicate1( var1 ), !predicate2( var1 ) )) #Effect of this operator. This changes the current state. If an
                                        #predicate is written without the negation it will be added to the
                                        #current state. If it is negated that predicate will be removed
                                        #from the current state, for an predicate to be removed from the
                                        #current state it must exist.

  :TaskPrimitive1(( var1)
  ( 1 )
  ( )
  ( ))

  :TaskPrimitive2(( var1 )
  ( 1 )
  ( )
  ( ))
)

#Problem declaration, initial state and goal task. The initial state does not have to be declared because it will
#be created automatically during the preprocess step. However, to make it possible to run this example an initial
#state will be declared.
problem(

  #initial state
  init-state(
  at( CONSTANT-1, LOCATION ),
  at( CONSTANT-2, LOCATION ),
  predicate1( CONSTANT-1 ),
  predicate2( CONSTANT-2 )
  )

  #goal task declaration. The goal task must be ground. Can only add one goal task.
  goal-task(
  Goal-Task-1( CONSTANT-1, CONSTANT-2 )
  )
)
```

Example file: *Example HTN Constant connection.txt*:

```
constant-to-object-connection(
( CONSTANT-1, engine_object_ID_01 )
( CONSTANT-2, engine_object_ID_02 )
( LOCATION, engine_object_ID_03 )
)
```