Hierarchical Task Network Plan Reuse for Video Games

Dennis J. N. J. Soemers and Mark H. M. Winands
Department of Data Science and Knowledge Engineering, Maastricht University
d.soemers@gmail.com, m.winands@maastrichtuniversity.nl

Abstract—**Hierarchical Task Network Planning is an Automated Planning technique. It is, among other domains, used in Artificial Intelligence for video games. Generated plans cannot always be fully executed, for example due to nondeterminism or imperfect information. In such cases, it is often desirable to re-plan. This is typically done completely from scratch, or done using techniques that require conditions and effects of tasks to be defined in a specific format (typically based on First-Order Logic). In this paper, an approach for Plan Reuse is proposed that manipulates the order in which the search tree is traversed by using a similarity function. It is tested in the *SimpleFPS* domain, which simulates a First-Person Shooter game, and shown to be capable of finding (optimal) plans with a decreased amount of search effort on average when re-planning for variations of previously solved problems.**

I. Introduction

The problem of deciding what tasks should be executed by an agent in a real-time video game can be addressed in a number of different ways. Commonly used techniques [1], [2] are Finite State Machines, Behavior Trees, Utility-based decision making systems, and Automated Planning. One of the first well-known applications of Automated Planning in video games is in *F.E.A.R.* [3].

Hierarchical Task Network (HTN) Planning [4], [5] is an automated planning technique that has seen some use in video games. HTN Planning has been used to control a team of bots in the game *Unreal Tournament 2004* [6], to generate scripts offline for the game *The Elder Scrolls IV: Oblivion* [7], in serious gaming [8], [9], and in the adversarial real-time strategy game μRTS [10]. Examples of commercial video games that are known to use HTN Planners are *Killzone 3* and *Transformers 3: Fall of Cybertron* [11].

Generated plans cannot always be successfully executed, because HTN Planners are not able to predict the actions of other agents or deal with imperfect information and nondeterminism in the environment without incorporating extensions [12], [13]. Existing systems using HTN Planning in video games typically construct new plans from scratch whenever a plan fails during execution [8]. Planning systems outside the context of video games do the same in some cases [14], but there is also work describing more sophisticated approaches [15]–[20]. These approaches require conditions and effects of tasks on the environment to be explicitly defined in a predefined format (typically based on First-Order Logic).

This paper proposes an approach for reusing old plans of an HTN Planner that does not require conditions and effects to be explicitly defined, but allows for them to be defined in functions that are essentially black boxes from the point of view of the Planner. The main purpose of reusing plans is to speed up the process of finding a new plan. Another motivation for reusing plans is to increase the likelihood of finding a plan that is similar to a partially executed previous plan. In video games this can reduce the number of times that an agent abruptly changes behavior, and therefore increase the believability of the agent's behavior. It is tested on problems from the *SimpleFPS* [21] domain, using the *Unreal Engine 4 (UE4)* game engine as test environment. *UE4* is the latest version of a commercial and widely-used game engine. Experiments have been carried out to measure the effect of Plan Reuse on the search effort required to find optimal plans. The effect of Plan Reuse on the quality of plans returned when terminating a search process early has also been measured.

The remainder of the paper is structured as follows. Section II provides background information on HTN Planning. In Section III, the implementation of the HTN Planning algorithm is described. The approach used for reusing old plans is described in Section IV. A description of the experiments can be found in Section V. Finally, Section VI concludes the paper and provides ideas for future research.

II. Hierarchical Task Network Planning

A Hierarchical Task Network (HTN) Planning Problem [4], [5], [22] can be defined as a tuple $\mathcal{P} = (S, T, \mathcal{O}, \mathcal{M})$, where S is the current *World State*, T is the current *Task Network*, $\mathcal{O}$ is the set of *Operators*, and $\mathcal{M}$ is the set of *Methods*. More specifically, S is a description of the environment in which the agent is located, and should contain all the information that is relevant for the planning process. T is a collection of *tasks* that need to be accomplished by the agent, where tasks can be constrained to require accomplishment before certain other tasks in the network. A task can be either *primitive*, meaning that it directly corresponds to an action that the agent can execute, or *compound*, meaning that it represents a higher-level plan that needs to be decomposed into a Task Network. S and T change during the planning process.

Every Operator $o \in \mathcal{O}$ represents the execution of a single primitive task t_p. Given S, o defines conditions that must hold in S for t_p to be applicable in S, defines how S changes if t_p is applied (executed by the agent) in S, and defines a nonnegative cost for applying t_p. Similarly, every Method $m \in \mathcal{M}$ represents the execution of a single compound task t_c.

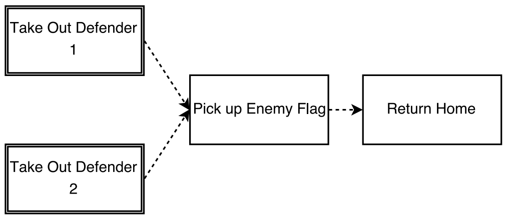

Fig. 1. Example Task Network. An arrow pointing from one task to another task means that the first task is constrained to require execution before the second task (the first task is a predecessor of the second task). Boxes with a double line are compound tasks, boxes with a single line are primitive tasks.

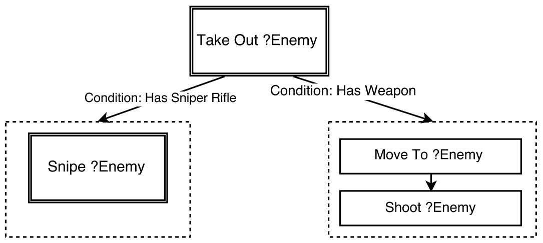

Fig. 2. Example Method. “Take Out ?Enemy” is the compound task for which this method is defined, where “?Enemy” is a variable. The boxes with dashed lines denote task networks. The two arrows pointing away from this compound task point to task networks that the original compound task can be decomposed into, under certain conditions. The left decomposition consists of only a single compound task, whereas the right decomposition consists of an ordered sequence of two primitive tasks.

Given S, m defines all the different Task Networks that t_c can be decomposed into. $\mathcal{O}$ and $\mathcal{M}$ remain constant during the planning process.

An HTN Planning system is expected to take a Problem $\mathcal{P}$ as input, and produce a valid plan Π as output. A plan Π is a valid plan for $\mathcal{P}$ if it is an ordered list of primitive tasks that can be obtained by consecutively applying Methods and Operators from $\mathcal{M}$ and $\mathcal{O}$ to the tasks in T in an order that satisfies the constraints of T, until T is empty. Applying an Operator o removes the corresponding primitive task t_p from T, appends it to Π, and changes S as defined by o. Applying a Method m replaces the corresponding compound task t_c in T with a subnetwork that is a valid *decomposition* according to m in S (m can have more than one valid decomposition in any given state S). A decomposition is a Task Network that must be fully executed for the original compound task to be considered executed, and can be viewed as a lower-level description of the more abstract compound task.

An example of a Task Network is depicted in Figure 1. An agent can execute this Task Network by first taking out two defenders, then picking up an enemy flag, and then returning home (to his own base). It does not matter in which order the two enemy defenders are taken out, but they both need to be taken out before the agent can pick up the enemy flag. The two tasks to take out defenders are compound tasks, meaning that they cannot be executed directly but need to be refined further. An example method to do so is depicted in Figure 2. This method defines that the compound task to take out an enemy can be decomposed into a single compound task to snipe the enemy under the condition that the agent has a Sniper Rifle, or it can be decomposed into an ordered sequence of two primitive tasks under the condition that the agent has a weapon. If neither condition is satisfied, the compound task cannot be decomposed and therefore cannot be executed.

In this description of the HTN Planning formalism, it has only been specified what information needs to be defined by the various structures (such as Operators and Methods), and not how this should be specified. Many existing planners, such as SHOP2, define world states, conditions and effects in (a subset of) First-Order Logic. The framework described in this paper makes no assumptions about the form in which this information is defined, and allows for it to be implemented directly in C++ functions and variables, as described in more detail in Section III. It means that there are few restrictions on what can be specified in an HTN Planning problem. It is possible to define planning problems where the tasks are not totally ordered, and variables are allowed. Such problems can become undecidable [23].

III. HTN Planner

This section describes the implementation of the HTN Planner plugin that has been developed for *Unreal Engine 4*.

A. Planning Problem Definition

Many HTN Planners, like *SHOP2* [22], define the structures of an HTN Planning problem, such as Operators, using formalisms based on logical expressions. The HTN Planner described in this paper does not use such a logic-based formalism, but instead uses a similar approach as *SHPE* [24]. Instead of defining all the relevant information of a planning problem using logical expressions, it is defined directly using C++ functions and variables. The main motivation for this approach is that it does not require an inference engine, and is expected to require less processing time [24].

For primitive and compound tasks, two base classes are provided with a number of virtual functions that can be implemented in subclasses to define domain-specific tasks. A primitive task t_p has the following functions that can be overridden; *ApplyTo(S)*, which should be implemented to apply any effects of t_p to a world state S; *ExecuteTask()*, which should be implemented to execute t_p during real gameplay (as opposed to during the planning process); *GetCost(S)*, which can be implemented to return the cost of applying t_p in the world state S; and *IsApplicable(S)*, which should be implemented to return a boolean value indicating whether or not t_p is applicable in a world state S. A compound task t_c only has a single function to override; *FindDecompositions(S)*, which should be implemented to return a list $\mathcal{T}$, where every $T \in \mathcal{T}$ is a *Task Network* that is a valid decomposition of t_c in the world state S. Note that, in this implementation, the concepts of *Operators* and *Methods* as mentioned in Section II are no longer used, and any information that these structures contained is instead located directly in the corresponding primitive and compound tasks.

Algorithm 1 The HTN Planning algorithm

```
1:  function INITIALIZE($T_0, S_0$)
2:      $Fringe \leftarrow [(T_0, S_0, \varnothing, 0)]$
3:      $BestCost \leftarrow \infty$

4:  function FINDPLAN
5:      while $Fringe \neq \varnothing$ and TIMEAVAILABLE() do
6:          $(T_i, S_i, \Pi_i, C_i) \leftarrow Fringe$.NEXT()
7:          if $T_i$ is empty then
8:              $BestPlan \leftarrow \Pi_i$
9:              $BestCost \leftarrow C_i$
10:             continue
11:         for all $t \in T_i$ without predecessors do
12:             if $t$ is primitive then
13:                 if $t$.ISAPPLICABLE($S_i$) then
14:                     $T' \leftarrow T_i$.REMOVE($t$)
15:                     $S' \leftarrow t$.APPLYTO($S_i$)
16:                     $\Pi' \leftarrow [\Pi_i, t]$
17:                     $C' \leftarrow C_i + t$.GETCOST($S_i$)
18:                     $Fringe$.ADD($(T', S', \Pi', C')$)
19:             else
20:                 $\mathcal{D} \leftarrow t$.FINDDECOMPOSITIONS($S_i$)
21:                 for all $D \in \mathcal{D}$ do
22:                     $T' \leftarrow T_i$.REPLACE($t, D$)
23:                     $Fringe$.ADD($(T', S_i, \Pi_i, C_i)$)
```

B. Finding a Plan

The HTN Planner is expected to take a *Task Network* T_0 and an initial *World State* S_0 as input, and produce a valid plan Π as output, as described in Section II. The algorithm that has been implemented to do this is similar to the algorithm used by *SHPE* [24] and *SHOP2* [22]. It performs a search through the space of all (partially) decomposed networks, starting from T_0. The intuition behind it is that T_0 is a highly abstract Task Network, containing a relatively large number of compound tasks, and it is gradually simplified by decomposing compound tasks and moving primitive tasks from the network into the plan. Primitive actions are inserted into the plan in the same order in which they are intended to be executed after planning.

Pseudocode for this algorithm can be found in Algorithm 1. The algorithm is initialized with a single tuple $(T_0, S_0, \varnothing, 0)$ in the *Fringe*. The *Fringe* is the collection of nodes in the search tree that have not been processed yet. Every element in this collection contains the Task Network of tasks that have not yet been completed, the current world state, the (partial) plan constructed so far, and the execution cost so far.

In each iteration, one node (T_i, S_i, Π_i, C_i) is removed from the *Fringe*, and every task $t \in T_i$ that does not have any predecessors is processed. A predecessor of t is a task that is constrained by T_i to require processing before t. All tasks that are allowed to be executed directly according to T_i are processed, and all other tasks are not yet processed.

If t is a primitive task that is applicable in S_i, t is applied to S_i, appended to Π_i, and removed from T_i. This results in a single new tuple that is placed in the *Fringe*. If t is a compound task, one new tuple is placed in the *Fringe* for every valid decomposition D of t in S_i. In this case, T_i is modified in every new tuple by replacing t in T_i with D.

In the planner described in this paper, the *Fringe* has been implemented as a stack. This means that the algorithm acts as a depth-first search. Many other planners, such as *SHOP2* and *SHPE*, are also implemented in this way.

C. Branch-and-bound and Heuristic Cost Estimation

A branch-and-bound optimization can speed up the search algorithm described above when searching for an optimal solution. Immediately after taking a tuple (T_i, S_i, Π_i, C_i) from the *Fringe* in line 6 of Algorithm 1, the cost C_i for executing the partial solution Π_i is compared to the cost of the best solution found so far (*BestCost*). If at this stage $C_i \geq BestCost$, Π_i cannot lead to an improvement on the best solution found so far, and the algorithm can immediately continue with the next element of the *Fringe*. This is the same branch-and-bound optimization as described in [22], [24].

An admissible heuristic function $h(T_i, S_i)$ that estimates the future cost of executing the remaining Task Network T_i given a current world state S_i can be used to improve the branch-and-bound optimization. Given such a function, the algorithm can prune partial solutions where $C_i + h(T_i, S_i) \geq BestCost$.

Such a heuristic function can incorporate domain-specific knowledge, but if two extra restrictions are placed on the problem definition it is also possible to define a domain-independent heuristic function. The first of these restrictions is that the cost of executing a primitive task cannot depend on the world state in which it is executed. The second restriction is that compound tasks cannot be defined in such a way that they can result in an infinitely long sequence of decompositions (which is possible when using recursion in the definition of compound tasks). Under these restrictions, the following domain-independent heuristic function $h(T_i)$ is well-defined:

$$h(T_i) = \sum_{t \in T_i} h(t) \tag{1}$$

$$h(t_p) = Cost(t_p) \tag{2}$$

$$h(t_c) = \min_{D \in \mathcal{D}} h(D) \tag{3}$$

In Equation 1, the world state S_i has been omitted as an argument because it cannot provide any information for a domain-independent heuristic. The heuristic function $h(t)$ for a single task t used in Equation 1 is defined by Equation 2 for the case where t is primitive, or Equation 3 for the case where t is compound. In Equation 3, $\mathcal{D}$ denotes the set of all possible decompositions of t_c in any possible world state.

IV. Plan Reuse

In this section, an approach for reusing old plans to more efficiently find new plans for similar problems is described. This approach does not require effects and conditions of tasks to be explicitly defined in a predefined format.

When an HTN Planner has previously found a plan for some planning problem, and is later required to find a plan for a new planning problem that is similar to the previous problem, it is expected to be possible to make use of the old solution to speed up the planning process. Existing approaches for reusing or repairing plans in an HTN Planner [15]–[20] require effects and conditions of tasks to be defined in a predefined format (typically using First-Order Logic). In most cases, this is because they analyze dependencies between the conditions and effects of tasks and store this information in graphs or other structures. These approaches are not compatible with the implementation of the planner as described in Section III, where the conditions of tasks and the effects of tasks on the world state are implemented in functions that are black boxes from the point of view of the Planner. The following approach does not have this problem.

Let $\mathcal{P}^{old} = (S^{old}, T^{old}, \mathcal{O}, \mathcal{M})$ be an old planning problem for which an optimal plan Π^{old} was generated using the HTN Planning algorithm as described in the previous section. Let $\mathcal{P}^{new} = (S^{new}, T^{new}, \mathcal{O}, \mathcal{M})$ be a new planning problem for which the HTN Planner needs to find a solution Π^{new}. $\mathcal{O}$ and $\mathcal{M}$ are equivalent for the two problems, so the same sets of primitive and compound tasks are defined. The assumption is made that S^{old} and S^{new} are in some sense similar, and that T^{old} and T^{new} are also in some sense similar. Finally, the assumption is made that an optimal solution Π^{new} will, due to the previous assumptions, also be similar to Π^{old}.

The intuition behind the approach is that it is likely to find higher quality solutions first if branches of the search tree that led to Π^{old} are prioritized when traversing the new search tree to look for Π^{new}. Similar ideas have also previously been used in game-tree search algorithms for abstract games [25], [26]. There are two benefits to finding high quality solutions as soon as possible. The first benefit is that, if in a real-time setting such as a video game the planning process is terminated early, a higher quality plan will be available. The second benefit is that the upper bound on the cost of the optimal plan is lowered more quickly, and therefore the branch-and-bound optimization can prune larger parts of the search space.

A. Search Tree Structure

Because the approach for plan reuse relies on manipulating the order in which the planning algorithm traverses branches of the search tree, it is useful to first take a closer look at the structure of this search tree.

An example of a search tree for a simple planning problem, with only a single compound task in the initial Task Network, is depicted in Figure 3. A node N_i in the search tree encapsulates a tuple (T_i, S_i, Π_i, C_i) as found in Algorithm 1. When N_i is visited (returned by *Fringe.Next()* and processed as seen in the pseudocode), a set of successor nodes $Successors(N_i)$ can be generated. For example, in Figure 3, $Successors(\mathbf{A}) = \{\mathbf{B}, \mathbf{E}\}$. $Successors(N_i)$ is empty if N_i gets pruned by the branch-and-bound optimization, or if N_i is a leaf node. A leaf node is either a *solution* node if T_i is empty, or a *failed* node if T_i is non-empty and does not contain any tasks without predecessors that can be executed in S_i. In the figure, **D** and **G** are solution nodes.

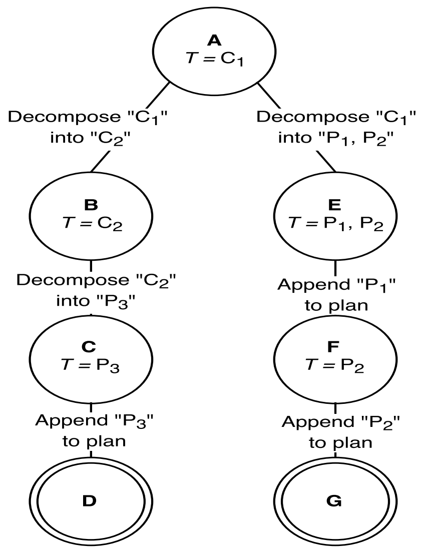

Fig. 3. Example search tree. Every circle represents a node in the search tree. The boldface letters are used to refer to specific nodes in the text. The Task Network of tasks that still need to be solved is shown under this identifier for every node. C_i are compound tasks, and P_i are primitive tasks. The text on every branch describes the change that is applied on that transition between two nodes. Nodes **D** and **G** have empty Task Networks.

If $Successors(N_i)$ is non-empty, N_i has at least one successor node. If T_i is *totally ordered*, meaning that there is *exactly one* task $t \in T_i$ that does not have any predecessors, there will be exactly one successor node if t is primitive, and there can be more than one successor node if t is compound. Traversing a branch in this situation can be viewed as committing to solve t in a certain, concrete way. If T_i is not totally ordered, there is one set of branches for every task $t \in T_i$ that does not have any predecessors, and within these sets it is again true that there is exactly one branch if t is primitive, and there can be more branches if t is compound. In this case, traversing a branch corresponds to selecting a task t and then choosing a concrete way to solve that task t. All Task Networks that appear in Figure 3 are totally ordered.

B. Similarity-Based Branch Reordering

The proposed approach for Plan Reuse requires a similarity function $Sim(\Pi^{old}, N_i)$, which computes a measure of similarity between Π^{old} and a current node N_i in the search tree. The information available in N_i is the tuple (T_i, S_i, Π_i, C_i). The approach described in this paper only makes use of Π_i, which simplifies the similarity function to $Sim(\Pi^{old}, \Pi_i)$.

The example search tree in Figure 3 depicts how the Task Network T changes in every node, and a description of the processing that is done to generate successor nodes is placed on every branch. When a compound task is processed to generate a successor, only the contents of T change. This means that, in Figure 3, nodes **A**, **B**, **C** and **E** all share the same plan Π. When a primitive task is processed, the description on the branch indicates how Π changes. This means that nodes **D**, **F** and **G** all have different plans from the other nodes.

Suppose that, at some point in the planning process that generated Π^{old}, the compound task C_1 was solved using the path in the right-hand side of the figure. Without any changes to the definition of a plan Π, any function $Sim(\Pi^{old}, \Pi_i)$ returns the same results for the nodes **A**, **B**, **C** and **E** because they share the same plan. This means that **F** is the first node in which it can be recognized that a path is being continued that was a part of the optimal solution of Π^{old}. The first change made to the planning algorithm is to redefine the concept of a plan Π to also append compound tasks to Π as they are processed. With this change, it is already possible to recognize in nodes **B** and **E** that they have some similarity with Π^{old} (they all contain the compound task C_1). In node **C**, it can then be recognized that an "incorrect" path is traversed (C_2 has been added which does not occur in Π^{old}), and in node **F** it can be recognized that the "correct" path is continued, leading to different levels of similarity.

The similarity function proposed in this paper is the function that computes the longest *Currently Matching Streak* (CMS). Intuitively, it is the function that finds the length of the longest sequence of consecutive tasks in Π^{old} that also occurs *at the end* of the current (partial) plan Π. More formally, let $\Pi[i]$ denote the i^{th} task in a plan Π. Let m denote the number of tasks in a plan Π. The similarity measure $CMS(\Pi^{old}, \Pi)$ is then defined as the maximum possible value n such that, for some index x, $\Pi^{old}[x] = \Pi_i[m]$, and $\Pi^{old}[x - n + j] = \Pi_i[m - n + j]$ for all j where $1 \leq j < n$. A score of 0 is assigned if $\Pi_i[m]$ does not occur anywhere in Π^{old}.

For example, let $\Pi^{old} = [A, B, C, D, E]$, $\Pi_i = [A, B, C]$ and $\Pi_j = [A, B, C, X, D, E]$. Then $CMS(\Pi^{old}, \Pi_i) = 3$, because the entire sequence of tasks of Π_i also occurs as a consecutive sequence in Π^{old}. Π_j also contains the same sequence, but in Π_j the sequence is followed by a non-matching task X, and then followed by another streak of length 2 that occurs in Π^{old}. Therefore, $CMS(\Pi^{old}, \Pi_j) = 2$.

This similarity measure "rewards" streaks of consecutive tasks that also occurred in the same order in Π^{old}, and also instantly punishes appending a non-matching task to an existing matching streak by resetting the score to 0. It is used to sort nodes for processing as follows. A node N_i is processed before a node N_j if $CMS(\Pi^{old}, \Pi_i) > CMS(\Pi^{old}, \Pi_j)$. If $CMS(\Pi^{old}, \Pi_i) = CMS(\Pi^{old}, \Pi_j) = 0$, the CMS scores of the closest ancestor nodes with non-zero CMS scores are used instead of the CMS scores of the nodes themselves. Finally, ties are broken by using the same ordering as a regular DFS. An example search tree is depicted in Figure 4. The numbers in this figure indicate the order in which parts of the subtree are processed.

C. Domain-Specific Ordering

The approach for reordering branches based on a similarity measure as described above is expected to be better than an arbitrary ordering of branches in cases where the new planning problem is related to the old planning problem. In reality, however, the branches typically are not ordered arbitrarily but are already ordered more efficiently based on domain-specific knowledge. For example, if a compound task is processed to find an item of a specific type somewhere in a map, and

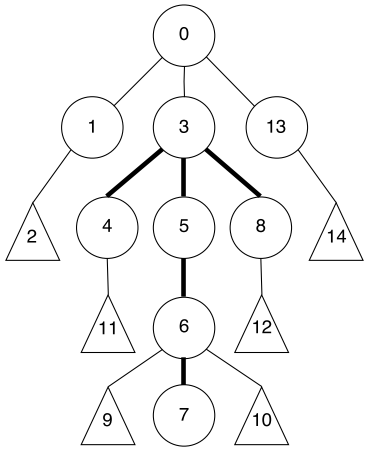

Fig. 4. Example search tree with plan reuse. Circles represent individual nodes, and triangles represent arbitrarily large subtrees. The numbers indicate the order in which nodes or subtrees are processed. Thick lines represent branches where a "correct" choice was made, meaning a task was added that continued a current matching streak or started a new streak.

one valid decomposition is created for every item of that type, these branches can be ordered according to the distance between the agent and those items. Branches for items that are already nearby are then explored first.

When there is already a good ordering of branches based on domain-specific knowledge, Plan Reuse can be detrimental, especially if it is also uncertain if a new planning problem is really similar to an old problem. Two approaches are proposed to reduce the likelihood of Plan Reuse having detrimental effects in the presence of domain-specific ordering, at the cost of also reducing the potential gains of Plan Reuse.

The first approach is the introduction of a parameter M denoting the *Minimum Streak Length* required for branches to be reordered according to their CMS score. Any CMS score that is less than M is simply set to 0. This makes the plan reuse less aggressive, which means there are fewer potential gains, but it is also more likely that a matching streak of tasks can actually be continued when it already has a sufficient length.

The second approach is to make the search probabilistic. This idea is inspired by a probabilistic approach for resuing plans in classical planning [27]. A parameter p is introduced, where $0 \leq p \leq 1$, which defines the probability with which the planner temporarily ignores parts of the search tree that are prioritized according to Plan Reuse, and instead searches parts that are not prioritized in a DFS manner. This idea has been implemented as follows. Whenever the planning algorithm processes a leaf node, the algorithm is set in a mode where it ignores prioritized nodes with probability p, and it is set in a mode where it does not ignore prioritized nodes with probability $1 - p$. Nodes are considered to be prioritized if and only if they are on a path that contains some node with $CMS > 0$. The reason for continuing to run in the same mode until a leaf node is processed is to avoid switching modes too often. Nodes that are pruned by the branch-and-bound optimization are not considered to be leaf nodes. The parameter value $p = 0$ means that the ordering of Plan Reuse always overrides the ordering of domain-specific heuristics, and $p = 1$ means that Plan Reuse is not used. With $0 < p < 1$, lower values for p are more suitable for cases where Plan Reuse is expected to be more reliable than domain-specific heuristics, and higher values are more suitable otherwise.

V. Experiments

This section describes the setup and the results of the experiments that have been carried out to evaluate the performance of the approach for Plan Reuse.[1]

A. Experimental Setup

SimpleFPS [21] is a planning domain that has been designed to simulate planning problems in FPS games. Originally it was defined as a classic planning domain, but it has also been translated into an HTN Planning domain and used for the evaluation of the HTN Planner *SHPE* [24]. Even though *SimpleFPS* is only a simulation of an FPS game, and not a real game, the generated planning problems are not necessarily less complex. With an average optimal plan length of 32 in the experiments described below, the problems can be estimated to be an order of magnitude more complex than those observed in real games [28].

Problems of this planning domain have been randomly generated and used to evaluate the performance of Plan Reuse in comparison to the same planning algorithm without Plan Reuse. The experiments have been carried out inside *UE4*. This means that any overhead involved in implementing and running a planner inside a game engine, as opposed to running it in isolation, is included in the results. The results were obtained using an Intel Core i5 CPU (2.67GHz), running on Windows 7. During the planning processes in these experiments, the memory usage of the entire plugin (including the *SimpleFPS* map data and the constant memory usage of the planner when idle) was at most in the order of 1 MB.

The original version of *SimpleFPS* is deterministic (after random problem generation), and assumes that the agent has access to perfect information. This means that these problems do not require any re-planning. For these experiments, the problems have been changed such that all doors in the maps are assumed to be unlocked initially, and the agent only obtains the information that a door is locked if the agent attempts to move through it. This means that the planner typically finds invalid solutions first, and problems often require re-planning when new information is obtained. To evaluate the performance of Plan Reuse, these "re-planning episodes" have been performed both with and without Plan Reuse. The previous plan is used as Π^{old} for Plan Reuse, but first pre-processed to remove all tasks that have already been executed.

Furthermore, the *SimpleFPS* domain was changed to punish the agent with an extra cost (equivalent to 50 "normal" tasks) for plans in which it chose for a different attacking approach from the original plan, where the three possible attacking approaches are melee, ranged and stealth. This means that if, for example, the old plan involved picking up a knife that turns out to be behind a closed door, it is unlikely that a new optimal plan will instead involve picking up a gun somewhere else. With this change, the likelihood of parts of Π^{old} still being useful for a new optimal solution is increased. It is still possible that there also is a second knife somewhere in a more convenient location, so there also still are problems where Plan Reuse can be detrimental.

[1]The implementation of the planner used in these experiments, and a more detailed description of the implementation, are available at https://github.com/DennisSoemers/HTN_Plan_Reuse.

TABLE I
Effects of Plan Reuse in Total

Parameter Values	ΔNodes Processed	ΔTime
$M = 10, p = 0$	**-7.04%**	**-11.11%**
$M = 10, p = 0.25$	**-11.51%**	**-18.60%**
$M = 20, p = 0$	**-1.70%**	+1.86%
$M = 20, p = 0.25$	**-1.18%**	+3.53%
$M = 30, p = 0$	**-2.81%**	**-2.41%**
$M = 30, p = 0.25$	**-2.43%**	**-0.98%**

Six different variants of Plan Reuse have been tested based on the approach described in Section IV, with different values for the parameters M and p. For M, the values 10, 20 and 30 have been tested. The optimal value for this parameter is domain-specific though, and different values may be more suitable for different problems. The value $M = 1$ has also shortly been tested, but was found to be too aggressive, and has not been included in the results. For p, the values 0 and 0.25 have been tested, where $p = 0$ means the use of Plan Reuse is not probabilistic. The value of $p = 0.25$ was chosen after a smaller number of tests, but is also close to 0.3, which is one of the values used for a similar parameter in [27].

B. Results

A total of 209 problems were completely processed by all variants of Plan Reuse, of which 10 problems were proven not to have any solutions. There were 22 problems that were not solved by any of the variants because they were terminated due to taking too much time. Planning processes were terminated early and declared a failure if no solution was found at all within 75 seconds, or no optimal solution within 150 seconds.

Table I shows the total change in the number of nodes processed and the amount of time spent planning of all the planning problems added together. It shows that especially the two most aggressive variants of Plan Reuse, with $M = 10$, perform well. The variants with $M = 20$ have a weak performance. This is largely caused by one specific problem, which is one of the largest problems in the set, where Plan Reuse with $M = 20$ turned out to be highly detrimental. The mean change in the absolute number of nodes processed by the variant with $M = 10$ and $p = 0.25$ is significant according to a paired, two-tailed Student's t-test with a significance level of 0.05 (p-value ≈ 0.037).

Figure 5 shows the difference in plan quality that Plan Reuse makes as a function of the search effort. In this figure, all planning problems have been mapped to a single measure of search effort and a single measure of plan quality. Informally, if a plot has a point y at $x = 0.5$, that variant of Plan Reuse

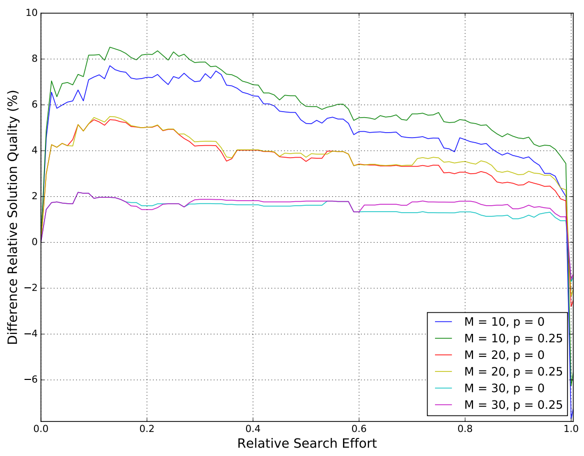

Fig. 5. The change in relative solution quality obtained by adding Plan Reuse as a function of relative search effort.

changes the plan quality by $y\%$ on average if a planning process is interrupted after half the search effort that planning without Plan Reuse would require to find the optimal solution. A point x on the x-axis denotes that $x \times n$ number of nodes have been processed, where n is the number of nodes that were required to find (but not necessarily prove) the optimal solution when planning without Plan Reuse. A point y on the y-axis denotes the difference in the average quality of the best solution found so far between planning with and without Plan Reuse. The quality of a solution is defined as $\frac{C^*}{C} \times 100\%$, where C is the cost of that solution and C^* is the cost of an optimal solution for that problem.

All the plots show a decrease close to $x = 1$. This is simply because all variants of Plan Reuse have at least some problems where Plan Reuse is detrimental, and $x = 1$ denotes exactly the amount of search effort that planning without Plan Reuse requires for all problems. So, $x = 1$ denotes the point in time where it is no longer possible to do any better than planning without Plan Reuse, and it is only possible to do worse. The figure shows that all the variants of Plan Reuse are above the x-axis (indicating an improvement in average plan quality of up to 8%) until x approaches 1. The variants with lower values for M show larger improvements on average. All variants peak with low amounts of search effort, indicating that Plan Reuse is especially beneficial if solutions are required in a short amount of time (for instance, in real-time). For the variants with $M < 30$, the changes in quality are significant (p-value < 0.05). There appears to be less variance in the changes in quality for the variants with $p = 0.25$ (p-value < 0.01).

C. Unchanged Problems Removed

The results for the problems as described above include all the problems on which none of the variants of Plan Reuse made any difference at all in the number of nodes processed compared to re-planning without Plan Reuse. On some of these problems, Plan Reuse cannot have any effect because the value for M is too conservative. This is a side effect of avoiding detrimental cases, and therefore these problems have not been excluded from the results above.

TABLE II
EFFECTS OF PLAN REUSE IN TOTAL - NO UNCHANGED PROBLEMS

Parameter Values	ΔNodes Processed	ΔTime
$M = 10, p = 0$	**-11.57%**	**-18.26%**
$M = 10, p = 0.25$	**-18.92%**	**-29.89%**
$M = 20, p = 0$	**-2.79%**	+1.68%
$M = 20, p = 0.25$	**-1.93%**	+4.26%
$M = 30, p = 0$	**-4.62%**	**-4.81%**
$M = 30, p = 0.25$	**-3.99%**	**-2.75%**

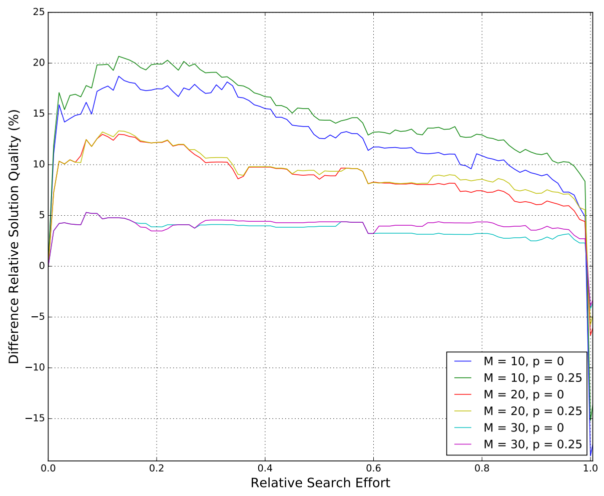

Fig. 6. The change in relative solution quality obtained by adding Plan Reuse as a function of relative search effort. (No unchanged problems)

However, these unchanged problems can also include problems where M was not set too high, but Plan Reuse simply did not result in any significant changes in the ordering, such as trivial problems where the domain-specific ordering is (nearly) optimal. For planning domains where such planning problems are not expected to occur, it can be interesting to look at the results obtained by removing the problems that remained unchanged by all variants of Plan Reuse from the sets.

The results with these unchanged problems (117 problems) removed can be found in Table II and Figure 6. On this set of problems, Plan Reuse reduces the total number of nodes processed by up to 18.92% and the processing time by up to 29.89% (both for $M = 10, p = 0.25$). The peak increase in average plan quality is up to 20% at a relative search effort of 20% ($M = 20, p = 0.25$). The p-values of these results are nearly identical to those of the corresponding results including unchanged problems.

VI. Conclusion and Future Work

In this paper, an approach has been proposed for reusing previously found plans in an HTN Planner when presented with new planning problems that are similar to one that was previously solved. Unlike existing approaches, it does not require conditions and effects of the domain to be specified in a pre-determined form, but allows for them to be implemented in black box functions. The main idea behind the approach is to manipulate the order in which the search tree is traversed by using a similarity function for plans.

Plan Reuse has been shown to be capable of reducing the average number of nodes required to find optimal solutions for *SimpleFPS* planning problems [21] that are likely to have similar solutions to previously solved problems, and also reduce the computation time. It has also been shown to improve the average quality of plans when using the planning algorithm as an anytime algorithm.

For future work, it would be interesting to investigate whether it is possible to estimate whether Plan Reuse will be likely to be beneficial or detrimental for a specific planning problem before the planning process is started. A direction of future research is to do this automatically by, for instance, computing a similarity measure between the old and the new planning problem. If this likelihood can be estimated accurately, Plan Reuse can be turned off in problems where it is expected not to be beneficial, and it can be turned on in problems where it is expected to be beneficial.

Furthermore, it could be interesting to investigate if Plan Reuse finds plans that are more similar to the old plan in cases where there are multiple plans that are all optimal with respect to the cost function. This has been mentioned as a motivation for Plan Reuse in the paper, because it can reduce the likelihood of an agent abruptly changing behavior in a video game and therefore increase the believability of the behavior. It has not been investigated in this paper's experiments because the *SimpleFPS* problems were found to typically have a low number of different optimal solutions. The planner and approach for Plan Reuse have also briefly been tested in an alpha version of the game of *Unreal Tournament*, which uses Unreal Engine 4. These tests are not described in more detail because the planning problems were too simple for Plan Reuse to make a noticeable difference. Experiments with more complex planning problems in later versions of the game, or in other games, could be done in future research.

Finally, a direction for future work would be to look into different variants of similarity measures. For instance, the CMS measure could be changed to have a lower value when appending a non-matching task to an existing streak, instead of resetting the score entirely to 0.

References

[1] I. Millington and J. Funge, *Artificial Intelligence for Games*, 2nd ed. Morgan Kaufmann, 2009.

[2] D. Isla, "Managing Complexity in the Halo 2 AI System," in *Proceedings of the Game Developers Conference*, 2005.

[3] J. Orkin, "Three States and a Plan: The A.I. of F.E.A.R," in *Game Developers Conference*, 2006.

[4] E. D. Sacerdoti, "The Nonlinear Nature of Plans," in *Proc. 4th Int. Joint Conf. Artif. Intell.*, vol. 1. Stanford, CA: Morgan Kaufmann Publishers Inc., 1975, pp. 206–214.

[5] K. Erol, J. Hendler, and D. S. Nau, "UMCP: A Sound and Complete Procedure for Hierarchical Task-Network Planning," in *Proc. 2nd Int. Conf. on Artif. Intell. Planning Syst.*, K. Hammond, Ed. Menlo Park, CA: AAAI Press, 1994, pp. 249–254.

[6] H. Hoang, S. Lee-Urban, and H. Muñoz-Avila, "Hierarchical Plan Representations for Encoding Strategic Game AI," in *Proc. AIIDE*. AAAI Press, 2005, pp. 63–68.

[7] J. P. Kelly, A. Botea, and S. Koenig, "Offline Planning with Hierarchical Task Networks in Video Games," in *Proc. 4th AIIDE Conf.*, C. Darken and M. Mateas, Eds. Menlo Park, CA: AAAI Press, 2008, pp. 60–65.

[8] A. Menif, C. Guettier, and T. Cazenave, "Planning and Execution Control Architecture for Infantry Serious Gaming," in *Proc. of the Planning in Games Workshop of ICAPS 2013*, M. Buro, É. Jacopin, and S. Vassos, Eds., 2013, pp. 31–34.

[9] I. M. Mahmoud, L. Li, D. Wloka, and M. Z. Ali, "Believable NPCs in Serious Games: HTN Planning Approach Based on Visual Perception," in *Proc. IEEE Conf. Comput. Intell. Games*, 2014, pp. 248–255.

[10] S. Ontañón and M. Buro, "Adversarial Hierarchical-Task Network Planning for Complex Real-Time Games," in *Proc. 24th Int. Joint Conf. Artif. Intell.*, Q. Yang and M. Wooldridge, Eds. AAAI Press, 2015, pp. 1652–1658.

[11] T. Humphreys, "Exploring HTN Planners through Examples," in *Game AI Pro: Collected Wisdom of Game AI Professionals*, 1st ed., S. Rabin, Ed. CRC Press, 2013, ch. 12, pp. 149–167.

[12] U. Kuter and D. Nau, "Forward-Chaining Planning in Nondeterministic Domains," in *Proc. 19th Nat. Conf. Artif. Intell.*, A. G. Cohn, Ed. Menlo Park, CA: AAAI Press, 2004, pp. 513–518.

[13] U. Kuter, D. Nau, M. Pistore, and P. Traverso, "Task Decomposition on Abstract States, for Planning under Nondeterminism," *Artificial Intelligence*, vol. 173, no. 5-3, pp. 669–695, 2009.

[14] S. Yoon, A. Fern, and R. Givan, "FF-Replan: A Baseline for Probabilistic Planning," in *Proc. 17th ICAPS*, M. Boddy, M. Fox, and S. Thiébaux, Eds. Menlo Park, CA: AAAI Press, 2007, pp. 352–359.

[15] S. Kambhampati and J. A. Hendler, "A Validation-Structure-Based Theory of Plan Modification and Reuse," *Artificial Intelligence*, vol. 55, no. 2-3, pp. 193–258, 1992.

[16] B. Drabble, J. Dalton, and A. Tate, "Repairing Plans On-the-fly," in *Proc. NASA Workshop on Planning and Scheduling for Space*, 1997.

[17] R. P. J. van der Krogt and M. M. de Weerdt, "Plan Repair as an Extension of Planning," in *Proc. 15th ICAPS*, S. Biundo, K. Myers, and K. Rajan, Eds. Menlo Park, CA: AAAI Press, 2005, pp. 161–170.

[18] N. F. Ayan, U. Kuter, F. Yaman, and R. P. Goldman, "HOTRiDE: Hierarchical Ordered Task Replanning in Dynamic Environments," in *Planning and Plan Execution for Real-World Systems–Principles and Practices for Planning in Execution: Papers from the ICAPS Workshop*, F. Ingrand and K. Rajan, Eds., 2007.

[19] I. Warfield, C. Hogg, S. Lee-Urban, and H. Muñoz-Avila, "Adaptation of Hierarchical Task Network Plans," in *FLAIRS Conference*, 2007, pp. 429–434.

[20] J. Bidot, B. Schattenberg, and S. Biundo, "Plan Repair in Hybrid Planning," in *KI 2008: Advances in Artificial Intelligence*, ser. LNCS, A. R. Dengel, K. Berns, T. M. Breuel, F. Bomarius, and T. R. Roth-Berghofer, Eds. Springer, 2008, vol. 5243, pp. 169–176.

[21] S. Vassos and M. Papakonstantinou, "The SimpleFPS Planning Domain: A PDDL Benchmark for Proactive NPCs," in *AIIDE Workshop: Intelligent Narrative Technologies*, E. Tomai, D. Elson, and J. Rowe, Eds. AAAI Press, 2011, pp. 92–97.

[22] D. Nau, T. Au, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman, "SHOP2: An HTN Planning System," in *JAIR*, M. E. Pollack, Ed. AAAI Press, 2003, vol. 20, pp. 379–404.

[23] K. Erol, J. Hendler, and D. S. Nau, "HTN Planning: Complexity and Expressivity," in *Proc. 12th Nat. Conf. on Artif. Intell.* Menlo Park, CA: AAAI Press, 1994, pp. 1123–1128.

[24] A. Menif, É. Jacopin, and T. Cazenave, "SHPE: HTN Planning for Video Games," in *Computer Games*, ser. Communications in Computer and Information Science, T. Cazenave, M. H. M. Winands, and Y. Björnsson, Eds. Springer, 2014, vol. 504, pp. 119–132.

[25] Y. Kawano, "Using Similar Positions to Search Game Trees," in *Games of No Chance*, ser. MSRI Book Series, R. Nowakowski, Ed. Cambridge: Cambridge University Press, 1996, vol. 29, pp. 193–202.

[26] M. Sakuta, T. Hashimoto, J. Nagashima, J. W. H. M. Uiterwijk, and H. Iida, "Application of the Killer-Tree Heuristic and the Lambda-Search Method to Lines of Action," *Information Sciences*, vol. 154, no. 3, pp. 141–155, 2003.

[27] D. Borrajo and M. Veloso, "Probabilistically Reusing Plans in Deterministic Planning," in *Proc. ICAPS'12 Workshop on Heuristics and Search for Domain-Independent Planning*, P. Haslum, M. Helmert, E. Karpas, C. L. López, G. Röger, J. Thayer, and R. Zhou, Eds. AAAI Press, 2012, pp. 17–25.

[28] É. Jacopin, "Game AI Planning Analytics: The Case of Three First-Person Shooters," in *Proc. 10th AIIDE Conf.* AAAI Press, 2014, pp. 119–124.