Hierarchical Task Network Plan Reuse in First-Person Shooter Games

Dennis J. N. J. Soemers and Mark H. M. Winands
Department of Data Science and Knowledge Engineering, Maastricht University
d.soemers@student.maastrichtuniversity.nl, 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 does not have this requirement, but allows for conditions and effects to be implemented in "black box" functions. 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] are Finite State Machines, Behavior Trees [2], Utility-based decision making systems, and Automated Planning. One of the first well-known applications of Automated Planning in video games is the use of Goal-Oriented Action Planning in *F.E.A.R.* [3], which is based on STRIPS [4].

Hierarchical Task Network (HTN) Planning [5], [6] is another Automated Planning technique that has seen some use in video games. In [7] it is described how an HTN Planner was used to control a team of bots in the game *Unreal Tournament 2004*. HTN planning has also been used in a system that generates scripts offline for the game *The Elder Scrolls IV: Oblivion* [8], in serious gaming [9], [10], and in the adversarial real-time strategy game μRTS [11]. An HTN planning system capable of finding plans under real-time conditions is described in [12]. Examples of commercial video games that are known to use HTN Planners are *Killzone 3* (2011) and *Transformers 3: Fall of Cybertron* [13].

It is not always possible to guarantee that plans can be successfully executed, because HTN Planners are typically not able to correctly predict the actions of other agents or deal with imperfect information and nondeterminism in the environment without incorporating extensions [14], [15]. Existing systems using HTN Planning typically construct new plans from scratch whenever an existing plan turns out to fail during execution in the context of video games [9]. Planning systems outside the context of video games do the same in some cases [16], but there is also work describing more sophisticated approaches [17]–[22]. 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 describes 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 old plans is to speed up the process for 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 a player can see an agent abruptly changing behavior, and therefore can increase the believability of the agent's behavior. It is tested on problems from the *SimpleFPS* [23] domain, using the *Unreal Engine 4 (UE4)* game engine as environment. *UE4* is the latest version of a commercial and widely-used game engine. The engine supports the creation of a wide variety of game genres, but it is best known for First-Person Shooter (FPS) games, such as the *Unreal Tournament* series. 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 using the planning algorithm as an anytime algorithm, and 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 and *UE4*. In Section III, the implementation of the HTN Planning framework in *UE4* is described. The approach used for reusing old plans is described in Section IV. A description of the experiments that have been carried out can be found in Section V. Finally, Section VI concludes the paper and provides ideas for future research.

II. Background

A. Hierarchical Task Network Planning

In this section, the concept of Hierarchical Task Network (HTN) Planning [5], [6] is formalized. Roughly the same formalism and terminology as used in the well-known HTN Planning system SHOP2 [24] are used in this paper.

An HTN Planning Problem 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 that needs to solve

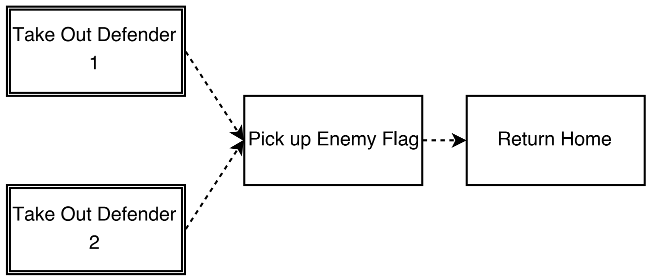

Fig. 1. Example Task Network. An arrow pointing from one task to another task means that the first task is constrained to require execution before the second task (the first task is a predecessor of the second task). Boxes with a double line are compound tasks, boxes with a single line are primitive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

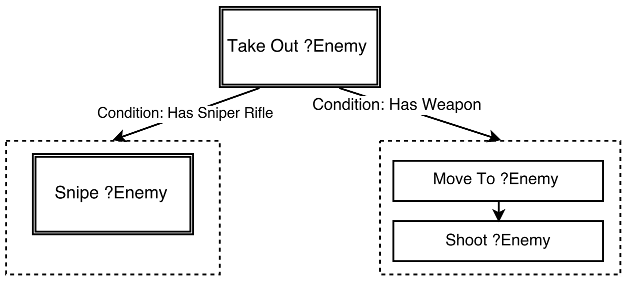

Fig. 2. Example Method. "Take Out ?Enemy" is the compound task for which this method is defined, where "?Enemy" is a variable. The boxes with dashed lines denote task networks. The two arrows pointing away from this compound task point to task networks that the original compound task can be decomposed 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.

the planning problem 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 some tasks can be constrained to require accomplishment before some other tasks in the network. Tasks can be either *primitive*, meaning that they directly correspond to an action that the agent can execute, or *compound*, meaning that they represent a higher-level plan and need to be decomposed into a Task Network during a planning process. S and T can change during the planning process.

Every Operator $o \in \mathcal{O}$ represents a single primitive task t_p in the planning process. The purpose of o is to define conditions that must hold in S for t_p to be applicable in S, and define how S changes if t_p is applied (executed by the agent) in S. If it is desirable for the planning system to minimize the costs of plans, o can also define a nonnegative cost for applying t_p. Similarly, every Method $m \in \mathcal{M}$ represents a single compound task t_c in the planning process. 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 in this case 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, and 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 [25].

B. Unreal Engine 4

Unreal Engine 4 (UE4) is a commercial game engine that can be used to develop video games of a wide variety of game genres. New functionality can be added to the engine relatively easily, for instance through a modular plug-in system. Such functionality can be implemented directly in C++, from which high performance levels can be expected, and also in *UE4*'s visual scripting language *Blueprint*, which is generally expected to have a slower runtime but can in some cases be easier to use. Considering the widespread use of *UE4* and its predecessors in the video game industry, it can also serve as

```
(:operator (!move-to ?old-room ?new-room)
 ()
 ((at ?old-room))
 ((at ?new-room))
)
```

Fig. 3. Example Operator in the JSHOP2 formalism.

a realistic test environment in terms of the memory and CPU overhead involved in running a planner during gameplay.

The only incorporated module for decision-making of agents in *UE4* at the time of this writing is based on Behavior Trees (BT) [2]. In this BT module, *Tasks* and *Conditions* (referred to as "Decorators" in *UE4*) can be implemented in C++ or *Blueprint*, and placed in a tree using a graphical editor. A BT can use a Blackboard [1] for reading relevant information and caching the results of computations, and can also receive events when values in the Blackboard are changed.

III. HTN Planner for Unreal Engine 4

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 define the structures of an HTN Planning problem, such as Operators, using formalisms based on logical expressions. An example of an Operator definition in *JSHOP2* [26], which uses such a formalism, is depicted in Figure 3. This example operator defines a primitive task named "`!move-to`" that can be used to move from "`?old-room`" to "`?new-room`". The effects it has on the world state is that the "`at ?old-room`" predicate is removed from a list of asserted predicates, and the "`at ?new-room`" predicate is added.

The HTN Planner described in this paper does not use such a logic-based formalism, but instead uses a similar approach as *SHPE* [12], which in turn was inspired by *Pyhop* [27]. Instead of defining all the relevant information of a planning problem using logical expressions, it is defined directly using C++ or *Blueprint* functions and variables.

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.

The plugin also contains a base class that can be extended to define domain-specific world states. This can be implemented in any way the user desires to provide any information necessary when passed into the functions described above for tasks. Typically it should be sufficient to simply add variables for any data that is relevant to the planning process, and provide access to these variables.

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, where S_0 and all the tasks in T_0 are implemented as described above, 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* [12] and *SHOP2* [24]. It performs a search through the space of all (partially) decomposed networks, starting from T_0. The intuition behind this search process 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* can be viewed as 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.

The algorithm removes, in each iteration, one node (T_i, S_i, Π_i, C_i) from the *Fringe*, and processes every task $t \in T_i$ that does not have any predecessors. A predecessor of t in this context is a task that is constrained by T_i to require processing before t. This means that all tasks that are allowed to be executed directly according to T_i are processed, and tasks for which conditions exist in T_i that still require other tasks to be executed first 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 (which does not have to be a single task but can be a complete sub-network of tasks with their own ordering constraints).

The *Fringe* in this algorithm can be implemented using different data structures. If it is implemented as a stack, where *Fringe.Next()* pops an element from the top of the stack and *Fringe.Add()* pushes a new element to the top of the stack, the algorithm acts as a depth-first search (DFS). In this case, the

Algorithm 1 The HTN Planning algorithm

```
function Initialize($T_0, S_0$)
    Fringe ← $[(T_0, S_0, \varnothing, 0)]$
    BestCost ← $\infty$
end function

function FindPlan
    while Fringe $\neq \varnothing$ and TimeAvailable() do
        $(T_i, S_i, \Pi_i, C_i)$ ← Fringe.Next()
        if $T_i$ is empty then
            BestPlan ← $\Pi_i$
            BestCost ← $C_i$
            continue
        end if
        for all $t \in T_i$ without predecessors do
            if $t$ is primitive then
                if $t$.IsApplicable($S_i$) then
                    $T'$ ← $T_i$.Remove($t$)
                    $S'$ ← $t$.ApplyTo($S_i$)
                    $\Pi'$ ← $[\Pi_i, t]$
                    $C'$ ← $C_i + t$.GetCost($S_i$)
                    Fringe.Add($(T', S', \Pi', C')$)
                end if
            else
                $\mathcal{D}$ ← $t$.FindDecompositions($S_i$)
                for all $D \in \mathcal{D}$ do
                    $T'$ ← $T_i$.Replace($t, D$)
                    Fringe.Add($(T', S_i, \Pi_i, C_i)$)
                end for
            end if
        end for
    end while
end function
```

loop starting at line 24 should be implemented in reverse order, so that the different decompositions obtained in line 23 are processed in the same order in which they are returned, and the designer can manipulate this order to guide the search. Other planners, such as *SHOP2* and *SHPE*, are often implemented in this way. A DFS is expected to find a first plan (which is not necessarily optimal) quickly, which is especially useful in cases where it is difficult or impossible to define meaningful costs for actions. In such cases the algorithm can also be terminated in line 11 of the pseudocode, instead of continuing the loop.

The *Fringe* can also be implemented as a priority queue instead of a stack. In this case, *Fringe.Next()* removes a tuple (T_i, S_i, Π_i, C_i) such that $\forall(T, S, \Pi, C) \in$ *Fringe*, $C \geq C_i$, and *Fringe.Add()* inserts elements into the priority queue such that it remains correctly sorted. This implementation results in a best-first search similar to Dijkstra's algorithm [28], or breadth-first search in the case of uniform costs for all primitive tasks. This approach is expected to require more time to find a first plan in general, but can still find solutions in cases where a DFS can get stuck in an infinitely long path in badly designed planning problems.

C. Branch-and-bound & Heuristic

To speed up the search algorithm as described above when searching for an optimal solution, a branch-and-bound optimization has been implemented. Immediately after taking a new tuple (T_i, S_i, Π_i, C_i) from the *Fringe* in line 7 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*, it will never be possible to improve on the best solution found so far given the partial solution Π_i, and the algorithm can immediately continue with the next element of the *Fringe*. This is the same branch-and-bound optimization that has been described for *SHPE* [12] and *SHOP2* [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, and enable it to prune parts of the search space earlier. Given such a function, the algorithm can prune partial solutions where $C_i + h(T_i, S_i) \geq$ *BestCost*. In the case of a best-first search, where the *Fringe* is implemented as a priority queue, the heuristic function can also be used to improve the sorting and turn the algorithm into a variant of A* [29].

Such a heuristic function can incorporate domain-specific knowledge, but if two extra restrictions are placed on the problem definition it is also possible define a domain-independent heuristic function. The first of these restrictions is that the cost of executing a primitive task does not depend on the world state in which it is executed. The second restriction is that compound tasks should not 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:

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

In this equation, 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 compound, or Equation 3 for the case where t is primitive. In Equation 2, $\mathcal{D}$ denotes the set of all possible decompositions of t_c in any possible world state.

$$h(t_c) = \min_{D \in \mathcal{D}} h(D) \tag{2}$$

$$h(t_p) = Cost(t_p) \tag{3}$$

D. Plan Execution

The HTN Planner has been implemented in *UE4* as a subclass of *UE4*'s *UBrainComponent* class. This base class provides some basic functionality for starting, pausing and

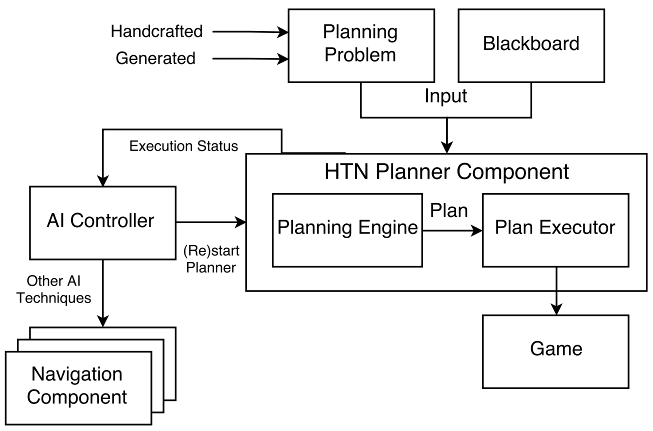

Fig. 4. Architecture of HTN Planning Component in *UE4*.

stopping processing, and provides access to a Blackboard. It is also used in *UE4* as base class of a *Behavior Tree Component* for decision-making based on Behavior Trees.

The architecture of this *HTN Planner Component* and its connections to the game engine are depicted in Figure 4. The *AI Controller* is an object used in *UE4* to provide the connection between an agent's physical representation in the game world and other Artificial Intelligence (AI) techniques, such as a Navigation Component. It is responsible for calling the HTN Planner's functions to take care of the agent's decision-making.

The HTN Planner takes a *Planning Problem* as input as described in Section II. Such a Planning Problem can be procedurally generated at runtime, or can be handcrafted. The HTN Planner also takes input from the agent's Blackboard, which can be used to store any relevant information in memory. The planner can be configured to search for an optimal plan with respect to the costs defined in the problem, or simply search for the first plan it can find. It can also be used as an *anytime* algorithm [30], which returns the best plan found so far whenever time runs out and a plan is required. Because the algorithm's *Fringe* as described above contains all the information that the algorithm needs, the search process can also be interrupted and continued at a later time (for instance to spread out the search over multiple frames without negatively affecting the game's framerate).

The *Plan Executor* takes a plan generated by the HTN Planner as input and ensures the primitive tasks are executed in the order given by the plan. When executed, a primitive task returns a status indicating that is has been successfully finished, is still in progress, or has failed. If a task's execution was successful, the Plan Executor simply continues with the next task. If the task is still in progress, the Plan Executor waits until the task reports a different status. If a task's execution fails, the remainder of the plan is not executed. The agent should make sure that either the Planning Problem or the data in the Blackboard is changed as necessary in cases where the execution of a task fails. If this input does not change in any way, it is likely that the HTN Planner produces exactly the same plan of which the execution previously failed. The AI Controller can interrupt planning or plan execution processes and restart a new planning process at any time (for instance if there are important changes in the environment that are expected to result in finding a different plan, or if the current plan's execution fails or successfully finishes).

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 new 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 [17]–[22] 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}. An example of a situation in a video game where these assumptions are reasonable is if an agent first created some plan that required him to move through a door, but a new plan is required because, during execution, the agent discovers that the door is locked.

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 [31], [32]. 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 interrupted early, and the best plan found so far is executed, that plan will have a higher quality. 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 of the algorithm can prune larger parts of the search space.

In this paper, the focus is placed on using Depth-First Search (DFS) as a baseline, and applying Plan Reuse to the DFS process. This means that branches of the search tree are assigned priorities through Plan Reuse first, and ties are broken by processing in the same order as DFS. The main reason for using DFS as a baseline (as opposed to Best-First Search) is that DFS was found to be faster in general, and also more common among existing planners. The use of Plan Reuse turns the search process into a different variant of Best-First Search, where paths that re-use parts of Π^{old} are prioritized.

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 5. 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 5, $Successors(\mathbf{A}) = \{\mathbf{B}, \mathbf{E}\}$. $Successors(N_i)$ 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.

If $Successors(N_i)$ is non-empty, N_i has at least one branch to a 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 5 are totally ordered.

B. Similarity-Based Branch Reordering

The approach for plan reuse for which the intuition was described above requires a similarity function $Sim(\Pi^{old}, N_i)$, which computes some measure of similarity between the old solution Π^{old} found for a previous planning problem, 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). S_i cannot provide any useful information for the similarity function without making any assumptions about the structure or the form in which a world state is defined. The cost C_i is also not useful for a similarity measure, because there can be many different plans that have the same cost.

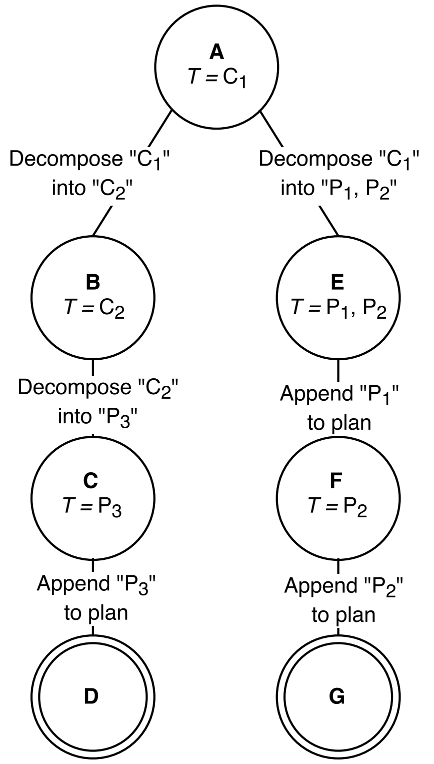

Fig. 5. Example search tree. Every circle represents a node in the search tree. The boldface letters are used to refer to specific nodes in the text. The Task Network of tasks that still need to be solved is shown under this identifier for every node. C_i are compound tasks, and P_i are primitive tasks. 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 (and are therefore solution nodes).

T_i can provide useful information, because it contains tasks that must be solved in all descendants of N_i that lead to solution nodes. However, because T_i is not necessarily totally ordered, and new tasks can be inserted into it during the planning process, it is not trivial to correctly retrieve information from it and T_i has also been ignored in the approach described in this paper. This only leaves Π_i as a source of information, which simplifies the similarity function to $Sim(\Pi^{old}, \Pi_i)$.

The example search tree in Figure 5 shows 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 5, 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 the plan Π changes. This means that nodes **D**, **F** and **G** all have different plans from all 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, the function $Sim(\Pi^{old}, \Pi_i)$ returns exactly 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}. Therefore, 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.

An example of a similarity function that could appear useful at first glance is to treat Π^{old} and Π_i as strings and compute the length of the longest common subsequence (LCS) [33] (similar to computing the length of the longest common substring, but allows for non-matching tasks to be in between parts of a subsequence). However, upon closer inspection, such a function turns out not to have a large impact on the order in which nodes are traversed compared to DFS. The reason for this is that, throughout the planning process, Π_i only grows because tasks are added, and tasks are never removed. This means that for any node N_i and any successor $N_j \in Successors(N_i)$ the following equation always holds:

$$LCS(\Pi^{old}, \Pi_j) \geq LCS(\Pi^{old}, \Pi_i) \tag{4}$$

From this equation it already follows that if a node is assigned a certain priority level based on the LCS, all descendants from that node will be assigned at least the same priority level. Therefore, whenever a node is processed, the entire subtree below it will also be processed before any nodes outside of that subtree can be processed. Furthermore, many siblings in the search tree often have an equivalent LCS value. Any successor node $N_j \in Successors(N_i)$ that results from solving and appending the same task t to Π_i results in an equivalent Π_j, and therefore also an equivalent LCS value. These branches are then still traversed a depth-first manner. Only in cases where T_i is not totally ordered, and N_i has different branches that correspond to appending a different task t to Π_i, can $Successors(N_i)$ have nodes with different LCS values and can the ordering be different from the ordering of DFS. The same is true for any other similarity measure for which an equivalent relation as in Equation 4 holds.

Intuitively this means that it seems preferable to use a similarity measure that does not only reward "correct choices", but also punishes "incorrect choices" afterwards, so that the approach does not always commit to processing a complete subtree when processing the root node of that subtree. The similarity function proposed in this paper, which computes the longest *Currently Matching Streak* (CMS), has this property. 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

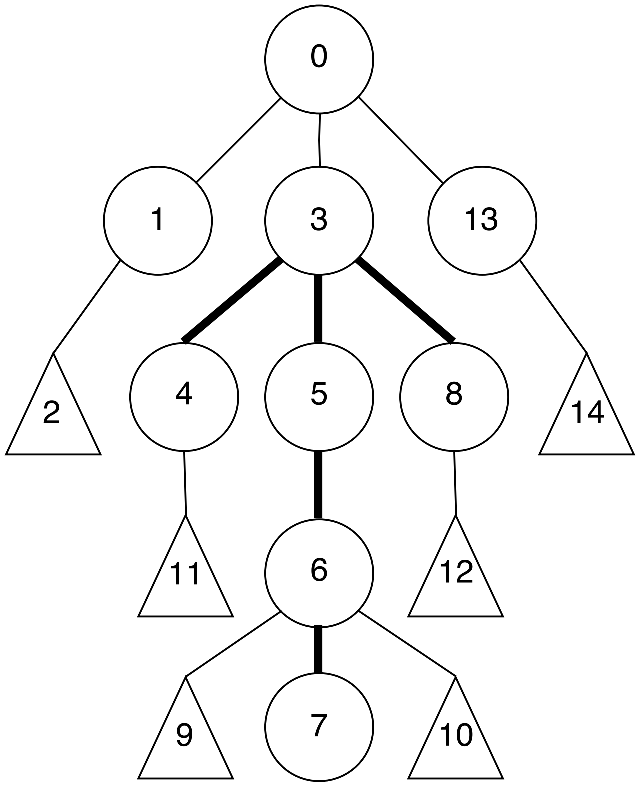

Fig. 6. Example search tree with plan reuse. Circles represent individual nodes, and triangles represent arbitrarily large subtrees. The numbers indicate the order in which nodes or subtrees are processed. Thick lines represent branches where a "correct" choice was made, meaning a task was added that continued a current matching streak or started a new streak.

$\Pi[i]$ denote the i^{th} task in a plan Π. Let m denote the number of tasks in the plan Π_i of a node N_i. The similarity value $CMS(\Pi^{old}, \Pi_i)$ of N_i 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 6. The numbers in this figure indicate the order in which parts of the subtree are processed.

C. Implementation

A straightforward way to implement the ordering described above would be to store the nodes in a priority queue, such as a binary heap. However, this requires some history of ancestor nodes to be stored with nodes in memory in order to determine the CMS score of ancestors when this is necessary, and CMS scores of the nodes should also be stored in memory to avoid re-computing them too often. Insertions and deletions in a priority queue also cannot be done in constant time.

A more complex, but more efficient implementation is described next. Let n denote the length of Π^{old}. For a node N_i, let $P(N_i)$ denote the parent node of N_i and $\mathcal{A}(N_i)$ denote the list of ancestors of N_i. $CMS(N_i)$ is used as a shorthand notation for $CMS(\Pi^{old}, \Pi_i)$, where Π_i is the plan of node N_i. Then, to store nodes during the search process, a collection of data structures is used. Algorithm 2 shows the corresponding pseudocode for the *Add()* and *Next()* functions to store and retrieve nodes for processing in the correct order. The variables starting with "Max" are indices used to keep track of which data structures are non-empty, and the correctness of their values should be verified at the end of every iteration of the main loop of the planning algorithm. The values are correctly increased by the *Add()* function, but may need to be decreased again when their corresponding data structures run empty.

The first data structure is a single stack $S_{Default}$, which contains all nodes N where $CMS(N) = 0$ and $\forall A \in \mathcal{A}(N)$, $CMS(A) = 0$. These are all the nodes for which no matching streaks have been encountered along the entire path yet, and therefore the nodes with the lowest priority. Nodes in $S_{Default}$ are therefore only processed once all of the following data structures are empty. Nodes in this stack are added in line 15 of the pseudocode and removed in line 26.

Secondly, there is a collection of n First-In-First-Out (FIFO) queues, where Q_i denotes the queue intended for containing all nodes N where $CMS(N) = 0$, but $CMS(P(N)) = i$, where $i > 0$. Nodes in these queues have a higher priority than those in $S_{Default}$, but a lower priority than any of the following data structures. For any pair of non-empty queues Q_i and Q_j, nodes in Q_j have a higher priority if $j > i$. Nodes in these queues are added in line 7 of the pseudocode and removed in line 24. Note that it is not required to explicitly compute $CMS(P(N))$, because this is always equal to $MaxCurr$ if nodes are retrieved in the correct order.

The third part is a collection of n stacks, where S_i denotes the stack intended for containing all nodes N where $CMS(N) = 0$, and $CMS(P(N)) = 0$, but the closest ancestor $A \in \mathcal{A}(N)$ for which $CMS(A) > 0$ has $CMS(A) = i$. These are all the nodes that have had some ancestor A in the FIFO queue Q_i previously. Once that node A has been processed, the entire subtree below it should also be processed in a DFS manner (assuming no further matching streaks are encountered), and therefore these stacks take priority over the FIFO queues. Nodes in these stacks are added in lines 10 and 12 of the pseudocode, and removed in line 22. Note that it is not required to explicitly compute $CMS(A)$, because this is always equal to either $MaxParent$ or $MaxAncestor$ if nodes are retrieved in the correct order.

Algorithm 2 Node ordering with Plan Reuse

```
 1: function ADD($N$)
 2:     $c \leftarrow CMS(N)$
 3:     if $c > 0$ then
 4:         $S^*_c$.PUSH($N$)
 5:         $MaxCurr \leftarrow max(MaxCurr, c)$
 6:     else if $MaxCurr > 0$ then
 7:         $Q_{MaxCurr}$.ENQUEUE($N$)
 8:         $MaxParent \leftarrow max(MaxParent, MaxCurr)$
 9:     else if $MaxAncestor > 0$ then
10:         $S_{MaxAncestor}$.PUSH($N$)
11:     else if $MaxParent > 0$ then
12:         $S_{MaxParent}$.PUSH($N$)
13:         $MaxAncestor \leftarrow MaxParent$
14:     else
15:         $S_{Default}$.PUSH($N$)
16:     end if
17: end function

18: function NEXT
19:     if $MaxCurr > 0$ then
20:         return $S^*_{MaxCurr}$.POP()
21:     else if $MaxAncestor > 0$ then
22:         return $S_{MaxAncestor}$.POP()
23:     else if $MaxParent > 0$ then
24:         return $Q_{MaxParent}$.DEQUEUE()
25:     else
26:         return $S_{Default}$.POP()
27:     end if
28: end function
```

Finally, a collection of n stacks is used, where S^*_i denotes the stack intended for containing all nodes N where $CMS(N) = i$, where $i > 0$. These are all the nodes that are currently on a matching streak, and therefore the highest priority nodes. For any pair of non-empty stacks S^*_i and S^*_j, nodes in S^*_j have a higher priority if $j > i$. Nodes in these stacks are added in line 4 of the pseudocode and removed in line 20.

All nodes that are intended to end up in any of the stacks should be pushed in reverse order, so that they end up in the correct order again for a DFS-based tie-breaker. For the same reason, nodes that are intended to end up in one of the FIFO queues should be added to the queues in the original order. This is also the reason for the existence of the FIFO queues, which may seem unnecessary at first glance. If these queues did not exist, and were merged with the collection of stacks denoted by S_i, nodes with different parent nodes with equivalent, non-zero CMS scores would no longer be processed with the DFS ordering as a tie-breaker. Consider, for instance, the root nodes of subtrees 11 and 12 in Figure 6. Without the existence of the FIFO queues, the root node of subtree 11 would be pushed to a stack first, and the root node of subtree 12 would be pushed on top of the same stack later, meaning that those two subtrees would be processed in the incorrect order.

This set-up consists of $3n + 1$ separate data structures, and therefore relies on the assumption that n (the length of the old plan that is being reused) is not too large. In typical video game applications [34], this is indeed the case. The advantages (in comparison to a single priority queue) are that all the used data structures support inserting and deleting in constant time, and it is no longer necessary to store any extra information in the memory of the nodes. All the extra information that a priority queue would require for the same ordering is no longer explicitly stored, but can be derived by looking at which data structures are empty and which are non-empty.

D. *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 instance, in *SHOP2* [24] this is done using the `:sort-by` statement. For example, if a compound task is processed to find an item of a specific type somewhere in a map, and 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, the addition of Plan Reuse can also potentially be detrimental, especially if it is also uncertain exactly how similar the new planning problem really is to the old planning 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 adding 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 (branches can only start being reordered deeper in the search tree), 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 [35], where an approach for reusing plans in classical planning was made probabilistic. A parameter p is introduced, where $0 \leq p \leq 1$, which defines the probability with which the search algorithm should temporarily ignore parts of the search tree that are prioritized according to Plan Reuse and instead search 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 they have a CMS score greater than 0, or if they have an ancestor with a CMS score greater than 0, and considered not to be prioritized otherwise. 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.

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. The implementation of the planner used in these experiments is available at https://github.com/DennisSoemers/HTN_Plan_Reuse.

A. *SimpleFPS*

SimpleFPS [23] 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* [12]. 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 [34].

The *SimpleFPS* domain and its random problem generator have been implemented inside the framework described in this paper. 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. Even though *SimpleFPS* is an abstract domain, which does not correspond to any actual gameplay, the experiments have still 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 of the experiments. 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 in the order of 1 MB, whereas the *UE4* engine and its editor used around 3 GB memory.

The original version of *SimpleFPS* is completely deterministic (after the random generation of the problem), 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 to have imperfect information. More precisely, they 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 (based on the assumption that the door was unlocked) is used as Π^{old} for Plan Reuse, but first pre-processed to remove all tasks that have already been executed.

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 partially depends on the structure of the domain and the expected size of plans though, and different values may be more suitable for different problems. The value $M = 1$ has also shortly been tested, but this value was clearly too aggressive and resulted in too many detrimental cases, and has not been included in the results. The only case that was found where $M = 1$ works well is if the planning problem is unchanged and the planner simply re-plans for exactly the same problem that it has already solved. 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 the value of 0.3, which is one of the values used for a similar parameter in [35].

Two sets of experiments have been performed in this manner. In the first set of experiments, no other changes were made to the *SimpleFPS* domain. In this case, the agent needs to find a new optimal plan. This will sometimes involve finding a key to unlock the door, in which case it is expected that the part of the old plan after moving through the door can still be useful afterwards. However, it also involves cases where a new optimal plan does not use the locked door at all, and simply moves around it. In these cases, Plan Reuse can often be expected to not be beneficial, or even detrimental.

In the second set of experiments, the *SimpleFPS* domain was changed to punish the agent with an extra, constant 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 the 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.

B. Results - Set 1

In the first set of experiments, with no modifications to increase the likelihood of Plan Reuse being beneficial, a total of 163 planning problems were completely processed by all variants of Plan Reuse, of which 4 problems were proven not to have any solutions. Furthermore, 17 problems 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
PROBLEM COUNTS WITH PLAN REUSE - SET 1

Parameter Values	#Failed	#Improved	#Worsened	#Unchanged
$M = 10, p = 0$	7	38	31	90
$M = 10, p = 0.25$	0	36	30	93
$M = 20, p = 0$	1	28	15	116
$M = 20, p = 0.25$	0	27	14	118
$M = 30, p = 0$	0	17	5	137
$M = 30, p = 0.25$	0	15	7	137

TABLE II
EFFECTS OF PLAN REUSE IN TOTAL - SET 1

Parameter Values	ΔNodes Processed	ΔTime
$M = 10, p = 0$	+0.18%	+8.58%
$M = 10, p = 0.25$	+0.07%	+3.30%
$M = 20, p = 0$	+0.53%	+7.75%
$M = 20, p = 0.25$	**-4.94%**	**-4.97%**
$M = 30, p = 0$	**-3.42%**	**-2.79%**
$M = 30, p = 0.25$	**-0.91%**	+0.77%

Table I shows, for every tested variant of Plan Reuse, the number of planning problems that were failed, improved, worsened or unchanged after adding Plan Reuse (in comparison to replanning without Plan Reuse). Failed problems are problems that were solved without Plan Reuse, but where the introduction of Plan Reuse was so detrimental that the processing was terminated due to taking too much time. Problems are considered to be improved or worsened if the algorithm needed to process fewer or more nodes respectively to prove the optimality of the found solution. Unchanged problems are problems where the number of nodes processed did not change. The table shows that all variants have a larger number of problems where Plan Reuse was beneficial than detrimental. It also shows that the addition of probabilistic Plan Reuse helps to eliminate all cases where Plan Reuse is so detrimental that it causes failures, which is otherwise only the case for the most conservative value of $M = 30$.

Table II shows the total change in the number of nodes processed and the amount of time spent planning of all the planning problems added together. Negative numbers mean that Plan Reuse was beneficial (because there were fewer nodes processed or less time spent planning), and positive numbers mean that Plan Reuse was detrimental. The table shows that the three least aggressive variants of Plan Reuse improve the number of nodes processed, and for two of them the improvement is also sufficient to overcome the computational overhead and improve the computation time.

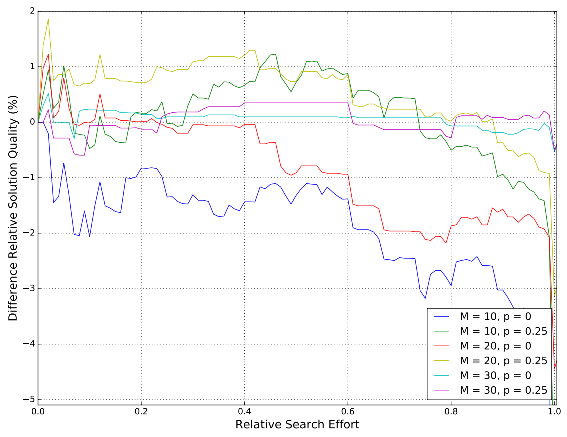

Fig. 7. The change in relative solution quality obtained by adding Plan Reuse as a function of relative search effort. The x-axis represents the quality of plans found without Plan Reuse. Plots above the x-axis indicate an increase in plan quality. (Problem Set 1)

Figure 7 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. 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. This means that, if a plot has a point y at $x = 0.5$, that variant of Plan Reuse 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.

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 variants that were already found to be beneficial in Table II also are above the x-axis most of the time, indicating that typically they can be expected to return better solutions on average if a planning process is terminated early. Some of the highest peaks appear early (with a low amount 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). Additionally, the variant with $M = 10$ and $p = 0.25$ is above the x-axis roughly between $x = 0.2$ and $x = 0.75$, indicating that in this region it can be expected to return higher quality solutions on average, despite its detrimental performance when searching for optimality according to Table II. However, the magnitude of the effects of Plan Reuse in this figure does not appear to be significant.

TABLE III
Problem Counts with Plan Reuse - Set 2

Parameter Values	#Failed	#Improved	#Worsened	#Unchanged
$M = 10, p = 0$	10	49	32	118
$M = 10, p = 0.25$	0	50	32	117
$M = 20, p = 0$	1	30	13	156
$M = 20, p = 0.25$	0	31	14	154
$M = 30, p = 0$	0	16	11	172
$M = 30, p = 0.25$	0	16	14	169

C. *Results - Set 2*

The second set of problems is the set of problems that were modified to increase the likelihood of new optimal solutions having similarities with old optimal solutions, and therefore increase the likelihood of Plan Reuse being beneficial. In this set, 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 failed by all variants of Plan Reuse, and also without Plan Reuse. The same time limits were used as in the first set of experiments.

For this new set of problems, Table III shows the same data as Table I does for the first set. In comparison to Table I, it shows that especially the more aggressive variants (the top rows) have improved. It is still the case that either probabilistic Plan Reuse ($p > 0$) or a conservative value for M (i.e., 30) are required to avoid the most detrimental cases in which Plan Reuse can cause failures.

Table IV shows the same data for this set as Table II does for the first set. It shows that especially the two most aggressive variants of Plan Reuse, with $M = 10$, have a better performance on this set of problems. The variants with $M = 20$ have a weaker performance on this set of problems. This is largely caused by one problem, which is one of the largest problems in the set, and Plan Reuse with $M = 20$ turned out to be highly detrimental for that specific problem. In this set, 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).

Finally, Figure 8 depicts the change in plan quality caused by Plan Reuse as a function of search effort in the second set of problems. In comparison to Figure 7 for the first set of problems, it shows a clear improvement. All plots are now above the x-axis until x gets close to 1, and the plots also reach higher levels of quality increase (up to an 8% increase).

An analysis of the statistical significance of the results in Figure 8 is done using Figure 9. For every variant of Plan

TABLE IV
EFFECTS OF PLAN REUSE IN TOTAL - SET 2

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%**

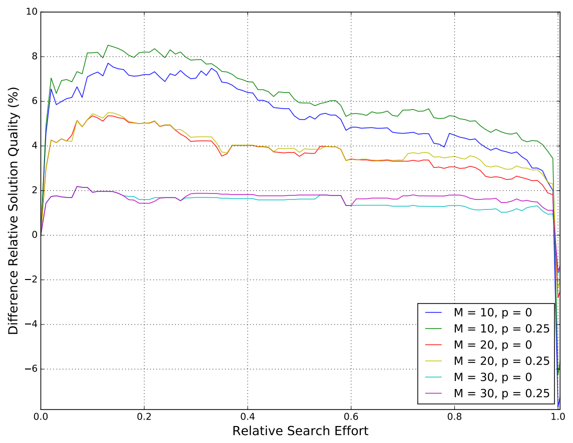

Fig. 8. The change in relative solution quality obtained by adding Plan Reuse as a function of relative search effort. (Problem Set 2)

Reuse, and every value between 0 and 1 with intervals of 0.01 for *Relative Search Effort*, a paired, two-tailed Student's t-test has been performed to test the significance of the change in quality with that amount of search effort. The resulting p-values are depicted in Figure 9. The plots for the two variants of Plan Reuse with $M = 30$ are less stable and often above 0.05 The other plots are almost strictly below 0.05, indicating that these results are significant with a significance level of 0.05. An interesting trend in this figure is that the two probabilistic variants ($p = 0.25$) of Plan Reuse with $M = 10$ and $M = 20$ have much lower p-values in the latter half of the plot than the corresponding non-probabilistic ($p = 0$) variants of Plan Reuse. This indicates that there is less variance in the results of the probabilistic variants.

D. *Unchanged Problems Removed*

The results for both sets of 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.

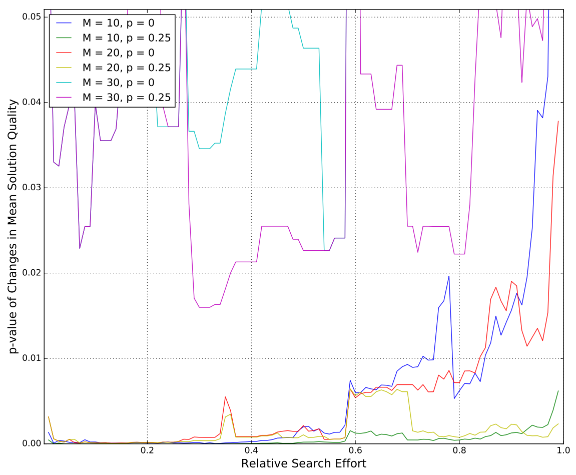

Fig. 9. p-values of paired, two-tailed Student's t-tests to test significance of changes in the mean solution quality. Tests were performed with intervals of 0.01 for *Relative Search Effort* between 0 and 1. (Problem Set 2)

TABLE V
EFFECTS OF PLAN REUSE IN TOTAL - SET 1, NO UNCHANGED PROBLEMS

Parameter Values	ΔNodes Processed	ΔTime
$M = 10, p = 0$	+0.24%	+10.33%
$M = 10, p = 0.25$	+0.09%	+3.69%
$M = 20, p = 0$	+0.72%	+9.35%
$M = 20, p = 0.25$	**-6.70%**	**-6.79%**
$M = 30, p = 0$	**-4.64%**	**-4.03%**
$M = 30, p = 0.25$	**-1.23%**	+0.48%

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 (near)-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.

Table V and Table VI show the total changes in the number nodes process and the amount of time spent planning for the first and second sets of problems respectively, with the unchanged problems removed. The final results are similar to those for the problem sets including unchanged problems, but the magnitudes are larger. In the second set of problems, with unchanged problems removed, Plan Reuse can save close to 20% of the nodes and close to 30% of the planning time.

Figure 10 and Figure 11 depict the changes in plan quality caused by Plan Reuse as a function of search effort for the first and second sets of problems respectively, with the unchanged problems removed. The plots look similar to the corresponding figures where the unchanged problems were not removed, but the scales on the y-axis have changed. In cases where Plan Reuse already was beneficial, it has become more beneficial,

TABLE VI
EFFECTS OF PLAN REUSE IN TOTAL - SET 2, 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%**

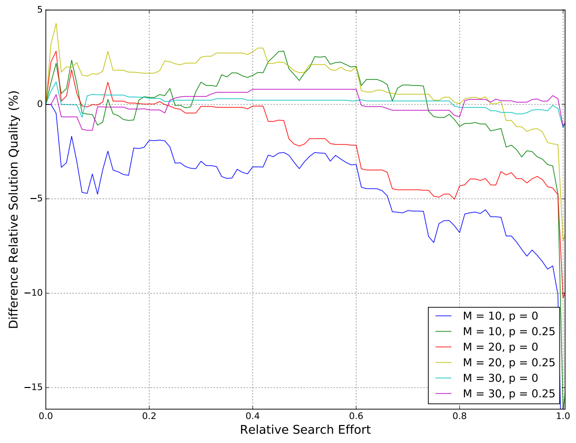

Fig. 10. The change in relative solution quality obtained by adding Plan Reuse as a function of relative search effort. (Problem Set 1, no unchanged problems)

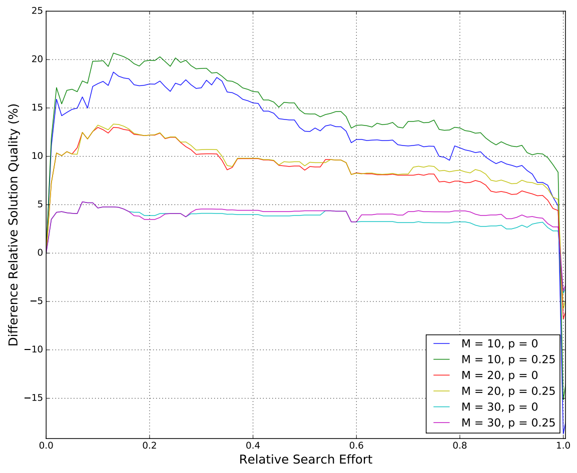

Fig. 11. The change in relative solution quality obtained by adding Plan Reuse as a function of relative search effort. (Problem Set 2, no unchanged problems)

and it has become more detrimental in cases where it already was detrimental. On the second set of problems, Plan Reuse can increase the solution quality by up to 20% on average when interrupting planning processes after processing about 20% of the number of nodes that are required for optimal solutions when planning without Plan Reuse. The significance levels for these results are nearly identical to those including the unchanged problems, and have been omitted to save space.

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 the 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 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 planning problems based on the *SimpleFPS* [23] domain, 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. On problems with a low likelihood of the old optimal solution still resembling a new optimal solution, careful tuning of parameters is required to make the approach robust and avoid the most extreme detrimental cases. When this likelihood is increased, the improvements are more significant and obtained for a wider variety of parameter values.

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, or by looking at the reason for re-planning. For example, if the old plan is still valid, but a re-planning process is started because new information was obtained that may make a different plan optimal, Plan Reuse could be expected to be more beneficial than in a cases like the experiments in this paper, where the old plan is known to fail. 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. It may then also be possible to tune parameters more aggressively to increase the gains in beneficial cases.

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 further in this paper's experiments because the generated *SimpleFPS* problems were found to typically have a low number of different optimal solutions.

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] R. E. Fikes and N. J. Nilsson, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," *Artificial Intelligence*, vol. 2, no. 3-4, pp. 189–208, 1971.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] O. Ilghami, "Documentation for JSHOP2," *Department of Computer Science, University of Maryland, Tech. Rep*, 2006.

[27] D. Nau, "Game Applications of HTN Planning with State Variables," in *ICAPS Workshop on Planning in Games*, 2013.

[28] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[29] P. E. Hart, N. J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *Systems Science and Cybernetics, IEEE Transactions on*, vol. 4, no. 2, pp. 100–107, 1968.

[30] T. L. Dean and M. S. Boddy, "An Analysis of Time-Dependent Planning," in *Proceedings of the 7th National Conference on Artificial Intelligence*. Menlo Park, CA: AAAI Press, 1988, pp. 49–54.

[31] 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.

[32] 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.

[33] V. Chvátal, D. A. Klarner, and D. E. Knuth, *Selected Combinatorial Research Problems*. Computer Science Department, Stanford University, 1972.

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

[35] 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.