Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 April 2017 doi:10.20944/preprints201704.0119.v1

Article

Modified Adversarial Hierarchical Task Network Planning in Real-Time Strategy Games

Lin Sun, Peng Jiao, Kai Xu, Quanjun Yin* and Yabing Zha

College of Information System and Management, National University of Defense Technology; mksl163@126.com (L.S.); crocus201@163.com (P.J.); xukai09@nudt.edu.cn (K.X.); ybzha@nudt.edu.cn (Y.Z.)
* Correspondence: yin_quanjun@163.com;

Abstract: Real-time strategy (RTS) game has proposed many challenges for AI research for its large state spaces, enormous branch factors, limited decision time and dynamic adversarial environment. To tackle above problems, the method called Adversarial Hierarchical Task Network planning (AHTN) has been proposed and achieves favorable performance. However, the HTN description it used cannot express complex relationships among tasks and impacts of environment on tasks. Moreover, the AHTN cannot handle task failures during plan execution. In this paper, we propose a modified AHTN planning algorithm named AHTNR. The algorithm introduces three elements *essential task, phase and exit condition* to extend the HTN description. To deal with possible task failures, the AHTNR first uses the extended HTN description to identify failed tasks. And then a novel task repair strategy is proposed based on historical information to maintain the validity of previous plan. Finally, empirical results are presented for the μRTS game, comparing AHTNR to the state-of-the-art search algorithms for RTS games.

Keywords: HTN planning; real-time strategy game; plan repair

1. Introduction

As a subcategory of video games, the Real-Time Strategy (RTS) game has proposed great challenges for AI research for its large state space, enormous branch factors, limited decision time and dynamic adversarial environment [1]. Regarded as a simplification of real-life environment, RTS games could serve as a test bed for researches like real-time adversarial planning and decision making under uncertainty [2]. Meanwhile, techniques which have been tested and proved effective in RTS games could also be applied in real-world domains [2].

In RTS games, the players need to build economy and military power to defeat the opponents by destroying their army and bases. Compared with traditional board games, the RTS games have some differences as follows [2]:

1. Players do not play in turns. Unlike turn-based games (like Chess), players in RTS games take their actions simultaneously in a short period of decision time.
2. Concurrent actions are allowed. In RTS games, players control multiple units which execute different actions concurrently. This is much more complex than board games where only one action is performed at each time step.
3. The actions are durative. For example, actions like *produce* need a fixed number of time steps before they are finished, while the actions in traditional board games have to be completed within only one time step.
4. The state space and branch factors are usually enormous. For example, consider a typical 128x128 map in *StarCraft*. Generally, there are about 400 units in the map. Just considering the location of each unit, the number of possible states will be about 10^{1685}. If we add the other factors in the game, we obtain even larger numbers.
5. The environment of the RTS games is dynamic.

Because of the differences above, the standard game tree search methods, which perform well in board games like alpha-beta search [3], cannot directly applied in RTS games [4]. Researches have

© 2017 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 April 2017 doi:10.20944/preprints201704.0119.v1

been done to improve the game tree search methods to handle the problems above [4]. Chung et al. study the applicability of game tree Monte Carlo simulations in RTS games in [5]. Balla and Fern apply UCT (Upper Confidence bound applied to Trees) algorithm in RTS game and deal with the problem of durative actions [6]. Reference [7] handles the problems of simultaneous and durative actions by extending alpha-beta search. However, the large branching factors caused by independently acting objects still remain a big challenge. The methods such as Combinatorial Multi-Armed Bandits try to tackle the problem of large branching factors [8][9]. In order to solve the problem of large state space and enormous branch factors, Ontañón combines the HTN planner with the game tree search called Adversarial Hierarchical Task Network planning (AHTN) in [10]. Instead of exploring the whole combination of possible actions, the HTN planning could guide the searching direction based on the domain knowledge. The work in [10] also extends AHTN method to handle the simultaneous and durative actions.

Though AHTN has achieved good performance compared with other search algorithms [10], it still suffers from several weaknesses as follows:

1. The HTN description used by AHTN cannot express complex relationships among tasks and impacts of environment on tasks. For example, assuming that a task has two subtasks; one must be successfully executed while the other doesn't have to be finished successfully. The failure of the latter task only affects the performance of its parent task but would not lead to the failure. This kind of relation cannot be expressed by the HTN description used in AHTN. Relations in situations when subtasks are triggered by the environment or execution results of parent tasks depend on both the environment and its subtasks also cannot be expressed by the HTN description in AHTN.
2. AHTN cannot handle task failures during plan execution. It runs at each frame of the game to generate a new plan whenever idle units exist in the game without considering failed tasks of old plans. Each time when a task of the old plan fails, it would simply remove the failed task without considering its impacts on current or future tasks. Moreover, keeping executing tasks related to the failed one is not only meaningless but also a waste of resource. Thus when one task fails and cannot been repaired, all related tasks in the plan should be terminated. And the AI player should try to repair the plan.

Generally, there are two approaches in task repairs. One is to re-plan the failed task locally. The other approach chooses one plan from multiple ones generated before, and uses it to repair the failed task. Compared to the former one, the second approach takes advantage of the historical information, and thus will reduce the time consumed.

In this paper, we propose a modified AHTN algorithm named AHTNR to handle the two problems above. An extended HTN description is introduced firstly, which enhances the model's capability in handling more complex relationships among tasks and impacts of environment through adding three elements (*essential task*, *phase* and *exit condition*). We then propose a monitoring strategy to identify failed tasks using extended HTN description and a task repair strategy based on historical information. In each decision cycle, the method would try to repair the failed tasks, followed by a new planning process.

The paper is organized as follows. The related work is presented in Section 2, followed by the introduction of the AHTNR planning algorithm in Section 3. Experimental results of the performance comparison between AHTNR and other algorithms could be found in Section 4. Finally, we conclude the paper and further present future works in Section 5.

2. Related work

We start with a short survey of HTN planning, and then move on to a review of its application in RTS games. The basic idea of HTN planning was first proposed by Sacerdoti [11] in 1975. At the same time, the first HTN planner NOAH (Nets of Action Hierarchies) appeared and was designed by Sacerdoti in [12]. After that the planner Nonlin appeared in 1977 [13], SIPE (System for interactive planning and execution) [14] and its successor SIPE2 [15] were introduced in 1984 and 1990 respectively. O-Plan (Open planning architecture) [16] appeared in 1991 with its successor in 1994

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 April 2017 doi:10.20944/preprints201704.0119.v1

[17]. Further, UMCP (Universal method-composition planner) was proposed as the first planner whose completeness had been proved in 1995 [18]. SHOP (Simple hierarchical ordered planner) [19] by Nau et al. was introduced in 1999 and its successor SHOP2 [20] appeared in 2003 as a successful modern planner. SIADEX emerged in 2005 [21]. The differences of theories, concepts, analysis and applications between different planners were compared in [22]. Several extensions were also proposed as the above planners all focus on classical planning. ND-SHOP2 (Non-deterministic simple hierarchical ordered planner 2) was proposed to deal with the nondeterministic effects of actions [23]. Kuter et al. integrated the control strategy of SHOH2 with MDP (Markov decision process) planning algorithms to deal with the probabilistic planning as in [24][25][26][27].

With its searching guided by human knowledge, the HTN planning speeds up dramatically and has been successfully used in many typical video games [28]. Kelly et al. used HTN planner to generate scripts offline for video games [28]. Menif et al. applied SHPE (Simple Hierarchical Planning Engine) to Steam Box, which is a kind of first person shoot game, by an alternative encoding of planning data [29]. Soemers and Winands studied the plan reuse by HTN planners in video games [30]. And there have also been some successful examples of HTN planners used in commercial video games [31]. But as we have known, only a few attempts have applied HTN planners to RTS games, which are also a kind of video games, in previous researches. In [32], a HTN planner was used to provide explanations for human players by querying the reasons leading to current states or events, and interrogating the behavior of a computer player. Laagland presented the design, implementation and evaluation of a HTN planner in an open source RTS game called Spring in 2007 [33], where the planning of the RTS game was divided into three levels with a HTN planner being used in the highest strategic one. Further in [34], Laagland summarized the pros and cons of HTN planning when used in RTS games. Naveed et al. used HTN planners to reduce the size of pathfinding search space in RTS games and tested their algorithm in ORTS game [35] in 2010. Most recently in 2015, Ontañón et al. used the HTN planning in RTS games based on improved game tree search algorithm [10]. In [10], Ontañón made extensions for durative and simultaneous actions and applied HTN planning directly into game tree search to take advantage of standard optimizations.

There are also studies concerning the plan repair based on HTN in many different fields. Garzón et al. studied the repair strategy based on HTN planning in the therapy planning system [36], in which the inference process would be recorded and used in failed plan repair. Gateau et al. used HTN to repair failed plan as local as possible in a high-level distributed architecture for multi-robot cooperation [37]. Ayan et al. designed the extension of SHOP to repair failed plans by introducing the task-dependency graph [38]. In their work, the planner would try to find the least failed task by the task-dependency graph during the plan repair, and thus reduce the repair cost. Kuter and Miller used hierarchical planning and plan repair (HPR) to improve the automated or autonomous system's self-confidence [39].

Our research is based on the work in [10] by Ontañón. Compared to the researches above, our main contributions are: 1) an extended HTN description is proposed to deal with more complex relations among tasks and impacts of the environment; 2) the monitoring strategy using extended HTN description and the repair strategy based on historical information is applied to RTS games.

3. AHTNR Planning Algorithm

The extended HTN description is proposed firstly to enhance the capability in expressing complex relationships among tasks and impacts of environment on tasks through adding three elements *essential task*, *phase* and *exit condition*. Then a monitoring strategy using extended HTN description is used to identify failed tasks. At last, a task repair strategy based on historical information is proposed to repair the failed tasks.

3.1. Extended HTN Description

HTN planning approach generates plans by decomposing tasks recursively into smaller ones [40]. There are two types of tasks in HTN: primitive tasks and compound tasks. Primitive tasks

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 April 2017 doi:10.20944/preprints201704.0119.v1

correspond to actions which can be executed by an agent directly in the world, such as *move*. Compound tasks represent the goals which need to be achieved by a higher plan and decomposed into a Task Network.

A HTN planning problem can be defined as a tuple $P=(S,TN,O,M)$, where:

- S is the set of current world states and consists of all the information that is relevant to the planning process.
- TN is current Task Network.
- O is the set of operators. Each operator $o\in O$ represents the execution of a primitive task. Given S, $\gamma(s,o)$ defines the transition of the state when an primitive task is executed by an agent. If $\gamma(s,o)=\perp$, the operator o is not applicable in s.
- M is the set of methods. Each method $m\in M$ represents a way to accomplish a compound task. $m=(t,C)$, where t is the compound task to which m will be applied. C is the set of preconditions. The method m can be applied to compound task t only when all preconditions in C are satisfied and the tasks preceding t in task network have been fully decomposed.

The result of a parent task depends on the results of its sub-tasks, the relationship among sub-tasks and the impacts from the environment. However the description of HTN above cannot express such relationships and impacts. We extend the HTN description by adding three elements which are *essential task*, *phase* and *exit condition*.

The first two modifications enhance the model's capability in expressing the complex relationships which cannot be expressed by basic relationships, which are *Sequential*, *Parallel*, *And* and *Or* respectively, among tasks. The *Parallel* means the execution time of different tasks is overlapped. The *Sequential* means the execution of tasks has to be one by one. These two relationships could help to control the starting time of sub-tasks. The *And* means the failure of any sub-task will lead to the failure of its parent one. The *Or* means the success of any sub-task will lead to the success of its parent. However some complex relationships cannot be expressed by above four relationships easily. For example, assuming that a task has two subtasks; one must be successfully executed while the other one doesn't have to be finished successfully. The failure of the latter task only affects the performance of its parent task without leading to its parent task's failure. The relationship between these two sub-tasks falls into none of the four relationships above. To describe the above situation, an attribution named *essential task* is added to all sub-tasks. If a task is essential, the success of its parent task only depends on the task's own success. Also, we propose a structure named *phase* to describe the temporal relationships among sub-tasks. The tasks which have *Parallel* relationship will be added to the same phase. While the tasks that have *Sequential* relationship will be separated to different phases. The execution sequence of phases is the same with the execution sequence of tasks. The failure of any phase will lead its parent task's failure. If the tasks of one phase are all defined as essential ones, they would have the *And* relationship. While if all of them are not essential, they have the *Or* relationship.

The third modification to HTN description is *exit condition*. It endows the models the capability in expressing impacts of the environment on the execution and results of tasks. The tasks could be terminated during their execution considering the environment dynamics. The agent could also cancel subsequent tasks due to insufficient conditions of the environment, such as time, position, etc. And some tasks are triggered by the environment. In order to depict the impacts from the environment, we add *exit conditions* to phase. The *exit conditions* could be divided into Sufficient Exit Conditions, Necessary Exit Conditions and Sufficient and Necessary Exit Conditions. The Sufficient Exit Conditions mean that the agent has to terminate current executing phase and determine whether the implementation of current phase is successful when any Sufficient Exit Condition is satisfied. The Necessary Exit Conditions mean that the agent cannot terminate current phase before all Necessary Exit Conditions are satisfied and this kind of Exit Conditions can be used to trigger the tasks belonging to the next phase. The results of the phase would be determined by both of its exit conditions and sub-tasks. There are four kinds of situations which may lead to the failure of the phase:

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 April 2017 doi:10.20944/preprints201704.0119.v1

1. Any Essential Task of the phase is failed.
2. Any Sufficient Exit Condition is satisfied, with several Essential Tasks still executing.
3. Any Sufficient Exit Condition is satisfied and there are no Essential Tasks, however with none of sub-tasks being finished.
4. All sub-tasks are failed with no Essential Tasks.

As discussed above, we extend HTN description as follows:

- The method of HTN is defined as $m = (t, C, Phases)$, where t is the compound task to which m will be applied. C is the set of preconditions. And $Phases$ is the queue of the task execution phase which contains multiple parallel sub-tasks.
- Each phase $phase \in Phases$ is defined as a tuple $phase = (subtasks, EC)$, where $subtasks$ is the set of tasks which belongs to the phase and EC is the set of *exit conditions*

Algorithm 1 shows a standard HTN planning algorithm using the extended HTN description. It works as follows: **Lines 2-4** check whether the current HTN is fully decomposed, and return the solution if this is true. **Line 5** selects a compound task which has not applied a method yet and the task must belong to the first executable phase in the current HTN. **Lines 6-18** select a method to decompose the task. **Lines 9-11** select a method whose preconditions are satisfied. **Lines 12-15** add the tasks belonging to each phase into the HTN in the order of the phase queue. **Line 16** makes sure that only one method will be applied. The algorithm loops until the HTN is fully decomposed.

Algorithm 1. HTNPlanning(s_0, N_0)

```
1.  Loop
2.      If fullydecomposed(N_0) ∧ γ(s_0, N_0) ≠ ⊥ then
3.          Return N_0
4.      End If
5.      pick up a compound task
            t = GetFirstPhaseleave(N_0)
6.      If Applied(N_0, t) = ∅ then
7.          Backtrack
8.      End If
9.      Acquire all methods M of Applied(N_0, t)
10.     For all methods m ∈ M
11.         If all c ∈ C is satisfied then
12.             Acquire the queue Phases from m
13.             For all phase ∈ Phases in order
14.                 N_0 = decompose(N_0, t, m, phase)
15.             End For
16.             break
17.         End If
18.     End For
19. End Loop
```

3.2. Monitoring Strategy using Extended HTN Description

As the environment of RTS game is dynamic, the plan generated in the earlier stage may fail. There are two sources of the plan failures. One comes from the situation when actions executed by units are failed because of being destroyed, or the unit cannot find available resource. The other comes from the situation when Sufficient Exit Conditions, which allow tasks to be executed, are changed under several kinds of situations, as discussed in Section 3.1. Once some actions are failed or any Sufficient Exit Condition is satisfied, the whole plan needs to be checked to find out the least influenced task with available methods.

Algorithm 2 shows the strategy of handling failed tasks. **Line 1-4** shows that the essential task and un-essential task have different handling strategies. **Line 5-14** shows how to handle the failed

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 April 2017 doi:10.20944/preprints201704.0119.v1

compound task. **Line 7-12** shows that if the failed compound task has available methods, it will be saved and repaired in the next decision cycle. Only when the task has no available methods to be applied will it be considered as the failed one. In this way the number of influenced tasks will be limited as small as possible and it will reduce the repairing pressure. **Line 15** shows the strategy of handling the primitive task and compound task without available methods.

Algorithm 2. TaskFailed(t)

1. **If** $Essential(t) = false$ then
2. SubTaskFailed(t)
3. Return
4. End **If**
5. **If** t is Compound task then
6. Acquire all methods M of t
7. **Loop**
8. Pick a method $m \in M \wedge status(m)$!=failed
9. **If** all $c \in C$ is satisfied then
10. Add t into *RepairTaskList*
11. Return;
12. **End if**
13. **End loop**
14. **End if**
15. SubTaskFailed(t)

According to the extended HTN description, each task belongs to one phase except the root node task. The phase to which the failed task belongs needs to be checked whether it is failed or not. Algorithm 3 shows the strategy to handle the related phase according to the principles discussed in section 3.1. **Line 2** shows if the failed task is the root node task, it need not be repaired. If the phase is failed, Algorithm 4 and Algorithm 5 will be triggered in order to cancel the influenced tasks and process the upper task. Algorithm 4 shows the strategy to handle the failed phase. If the task belonging to the failed phase is compound task, its sub-tasks will be processed. If it is primitive task, it will be canceled directly. Algorithm 5 shows the strategy of handling the method to which the failed phase belongs. **Line 2-6** shows that all unfinished phases of the method would be failed because they are sequential. If any Sufficient Exit Condition is satisfied, a method will be triggered to check whether the phase to which the Sufficient Exit Condition belongs is failed or not. The process of the method is similar to Algorithm 3 except that its input parameter is a *Sufficient Exit Condition* not a task.

Algorithm 3. SubTaskFailed(t)

1. acquire the phase p which t belongs to
2. **If** $p = \perp$ **then** return
3. **end If**
4. **If** one of sufficient conditions of p is met **then**
5. **If** any essential tasks of p are failed **then**
6. PhaseFailed(p);
7. MethodFailed(p);
8. **else If** (p has no essential task
9. $\wedge$all tasks of p is executed or do not stat)
10. PhaseFailed(p);
11. MethodFailed(p);
12. **end If**
13. **end If**
14. **else If** any essential tasks of p fail $\vee$
15. (p has no essential task $\wedge$ all tasks of p fail)
16. PhaseFailed(p);
17. MethodFailed(p);
18. **end If**
19. **end If**

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 April 2017 doi:10.20944/preprints201704.0119.v1

Algorithm 4. FailedPhase (p)

```
1.  get all tasks subtasks of p
2.  For all tasks t ∈ subtasks
3.    If status(t) = finished ∨ status(t) = failed then
4.      continue;
5.    End If
6.    If t is Compound task then
7.      Get the method m applied to t
8.      Get all phases Phases of m
9.      For all phase q ∈ Phases
10.       PhaseFailed( q )
11.     End For
12.   Else
13.       Cancel( t )
14.   End If
15. End For
```

Algorithm 5. MethodFailed (p)

```
1.  get the method m which p belongs to
2.  For all phases Phases of m
3.    If q ∈ Phases ∧ q ≠ p ∧ status(q) != finished then
4.      PhaseFailed( q )
5.    End If
6.  End For
7.    Get task t to which m is applied
8.    TaskFailed( t )
```

3.3 Task Repair Strategy based on Historical Information

After detecting an action failure, the least influenced task will be found out and added into the *RepairTaskList*. At each decision cycle, the AI player will try to find an alternative plan for each task in *RepairTaskList*. One kind of repair strategy is to consider the repaired task as a new top task by the planner. But this repair strategy doesn't make use of the information of the previous planning. In this section a task repair strategy based on historical information is proposed. During each task planning period, the AI player will generate some alternative plans with different evaluated values. There are two kinds of differences between the alternative plans: different task decompositions and task parameters. The main purpose of task repair is to find out other decompositions or parameters for the failed task. Thus if all the generated plans in the previous planning could be saved, the AI player would try to search the repair plan from the generated plans instead of making new ones for the failed task directly. This will reduce the time consumed by task repair.

In order to repair the failed task with better performance, it is necessary to try the alternative plans in the order of plan performance. However the plans generated by the AI player are usually out of order except the best one. So when recording the generated alternative plans, all the alternative plans need to be ordered by their evaluated values. At the same time, according to the AHTN algorithm, as all leaves of the game search tree are the alternative plans, the reordering of plans is the same as that of leaf nodes. To solve the above problem, we propose the Theorem 1 and further prove it as follows.

Theorem 1: Following the below two principles, all the leaf nodes are ordered decreasingly from left to right if the root node is a max one; otherwise, they are ordered increasingly.

1. If the parent node is max node, its sub-nodes are ordered decreasingly from left to right by the evaluated values;

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 April 2017 doi:10.20944/preprints201704.0119.v1

2. If the parent node is min node, its sub-nodes are ordered increasingly from left to right by the evaluated values.

Proof:

Given a m level min-max search tree, the ith level has N_i nodes. Each node is represented by n_{ij} $j \le N_i$. If n_{ij} is a max node, the sub-nodes will be ordered decreasingly. And if it is the min node, its sub-nodes will be ordered increasingly.

We use the min-max search tree in Figure 1 as an instance. $n_{i,1,1}, n_{i,1,2}, \cdots n_{i,1,j}$ are sub nodes of $n_{i-1,1}$ and $n_{i+1,j,k}, n_{i+1,j,2}, \cdots, n_{i+1,j,k}$ are sub nodes of $n_{i,1,j}$. We assume that $n_{i-1,1}$ is a max node. The value of $n_{i,1,j}$ is represented by $v_{i,1,j}$ where $v_{i,1,1} \ge v_{i,1,2} \cdots \ge v_{i,1,j}$. The value of sub nodes of $n_{i,1,j}$ is represented by $v_{i+1,j,h}$ where $v_{i+1,j,1} \le v_{i+1,j,2} \cdots \le v_{i+1,j,h}$.

With $n_{i-1,1}$ being a max node, we can get $v_{i-1,1} = v_{i,1,1} = v_{i+1,1,1}$ and $v_{i,1,j} \cdots \le v_{i,1,2} \le v_{i+1,1,1} \le v_{i+1,1,2} \cdots \le v_{i+1,1,h}$. When the best plan node $n_{i+1,1,1}$ is removed, the value $v_{i,1,1}$ of $n_{i,1,1}$ will become $v_{i+1,1,2}$ and the values of other sub nodes of $n_{i-1,1}$ will not change. So the value of $n_{i-1,1}$ will become $v_{i+1,1,2}$ and the best plan node is $n_{i+1,1,2}$. If we go on removing the best plan node, the best plan node will become $n_{i+1,1,3}, n_{i+1,1,4}, \cdots n_{i+1,1,h}$. When all sub nodes of $n_{i,1,1}$ have been removed, the best plan node will become the first sub node of $n_{i,1,2}$. For $n_{i-1,1}$, the order of best plans is the same as the order of $n_{i+1,j,h}$. If $n_{i-1,1}$ is min node, the conclusion will not change. □

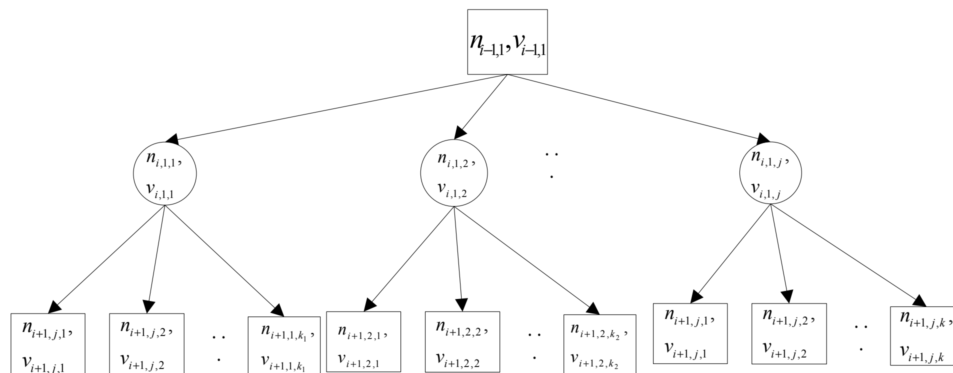

Figure 1. The min-max search tree is used in proof as an example. The rectangle nodes are max nodes which are ordered increasingly from left to right by the evaluated values and the circle nodes are min nodes which are ordered decreasingly from left to right by the evaluated values.

After the above process, the generated alternative plans will be provided to the AI player so as to repair the failed tasks. Before task repair we need to confirm the location of the failed task in the alternative plan and use the same part of the alternative plan instead of the failed task to carry on its execution. As the same task can be used as sub-tasks of different compound tasks, it is necessary to compare both the name and decomposition path to confirm the location of the failed task in alternative plans. When the failed task has been found in the alternative plan, its leaf nodes will be considered as the new actions to be executed.

At the beginning of each decision cycle, the AI player will first check whether there are failed tasks or not. If so, the AI player will try to repair the failed tasks. When all failed tasks have been repaired, the AI player will start a new plan. The main steps of AHTNR method are as follows:

- Step 1: the AI player checks whether there are failed tasks. If so, turn to Step 2, otherwise turn to Step 5;
- Step 2: the AI player removes the last failed task from the *RepairTaskList*;
- Step 3: the AI player tries to find out an alternative plan for the failed task based on the historical information. If there is a suitable alternative plan, repair the failed task and turn to Step 1, else turn to Step 4;

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 April 2017 doi:10.20944/preprints201704.0119.v1

- Step 4: the AI player takes the failed task as a new top task to make a new plan and turns to Step 1.
- Step 5: AI player generates a new plan. After getting a best plan, the AI player first orders and saves all generated plans according to the two principles as in Theorem 1. Then the AI player will get the executing actions from the best plan, and use the task network information to identify possible task failures. At last, the exit conditions will be added to the AI player for checking at each decision cycle.

4. Experiment and Result Analysis

4.1 Experimental Environment and Setting

We compare the performance of our AHTNR with other algorithms using the free software called μRTS [41]. μRTS has been used to evaluate different algorithms applied in RTS games [4][9][10]. The screenshot of μRTS is shown in Figure 2. There are two players (blue and red) competing to destroy each opponent's units. Each player has the same types of units. The small gray circles are workers which can attack enemies, build bases, harvest and transport resources. The Green squares are limited resources which can be harvested. The middle orange circles are light attackers. The yellow circles are heavy attackers. The heavy and light attackers both have larger hit and health points than workers. But they cannot harvest or transport resources. The white squares are bases which can produce new units. μRTS provides six kinds of actions for units in game. They are: move (in any empty one of the four cardinal directions), attack (attack any enemy in range), harvest minerals from resource unit, return minerals to bases, produce new units (some units such as bases can produce new units in any idle one of the four cardinal directions) and stay idle (the unit will do nothing).

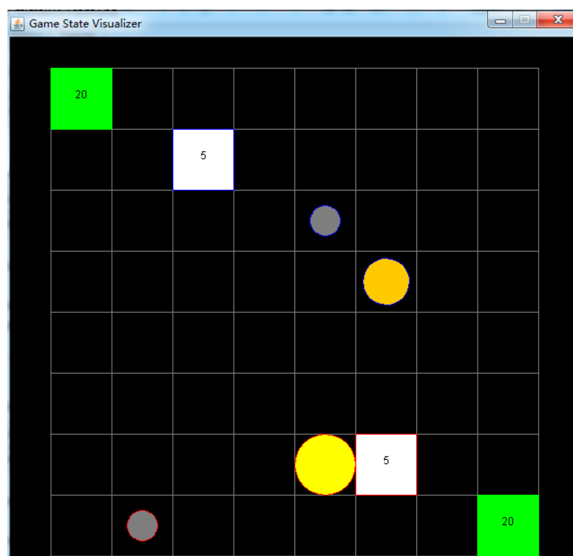

Figure 2. A screenshot of μRTS. The green squares are resources. The white squares are bases. The gray circles are workers, the yellow circles are heavy attackers and the middle orange circles are light attackers. The edges' color of each unit presents its side.

Though μRTS is only a simplification of the RTS games, it is still complex enough and exhibits the challenges of RTS games such as players concurrent moving, durative and simultaneous actions, real-time limitation and large branching factors. The original μRTS is deterministic and fully observable. To test our AHTNR's performance, we crafted two different HTNs for the μRTS domain which work as follows:

1. Low Level: it contains 12 operators (primitive task) and 9 methods for 3 kinds of tasks.
2. Flexible: it contains 12 operators as Low Level. It provides 49 methods and 9 kinds of tasks. It allows methods to have parallel execution tasks.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 April 2017 doi:10.20944/preprints201704.0119.v1

The HTNs used by AHTNR choose the extended HTN description. All tasks except *wait-for-free-unit* and *wait* in HTNs are defined as essential tasks. The Exit Conditions have been added to part of the phases to reflect the influence of environment. **Figure 3** shows an example from the Low Level HTN. The task named *destroy-player-internal* can be decomposed into two sub-tasks by applying the method named *dpi-nextunit* when the condition is satisfied. As the two sub-tasks are parallel, they belong to the same phase named *phase-dpi-nextunit-phase*. Because one sub-task is the same as its parent task, we will focus on the sub-task named *unit-order*. If all conditions of its method named *uo-attack* are satisfied, this task will be decomposed into the sub-tasks named *move-into-attack-range* and *attack*. The sub-tasks belong to different phases as they are *Sequential*. And they both are essential tasks. The first phase named *phase-uo-attack-pahse-1* has an Exit Condition. The task named *attack* in next phase would not be triggered before the Exit Condition is satisfied.

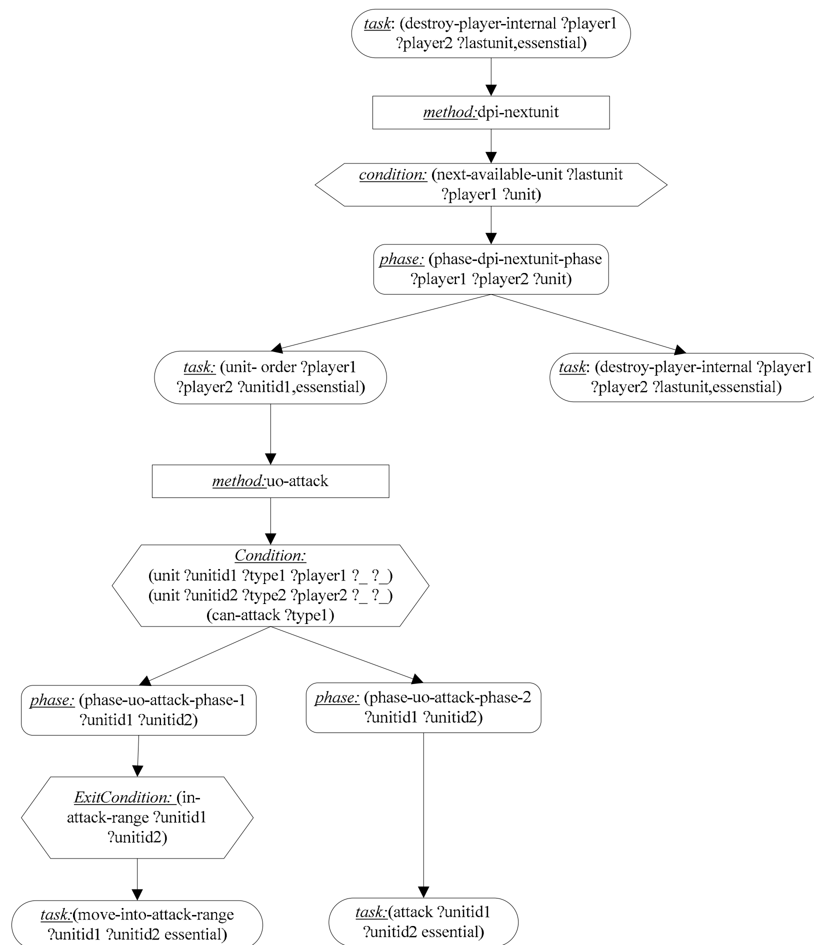

Figure 3. An example from the Low Level definition stating the way of executing the task: *destroy-player-internal*. The phase called *phase-dpi-nextunit-phase* has two parallel tasks and the phase called *phase-uo-attack-phase-1* has an Exit Condition. The next phase will not start until it is satisfied. All tasks except wait-for-free-unit and wait in HTNs are defined as essential tasks

To evaluate AHTNR, we compared it against the following collection of search algorithms:

1. AHTN: It is an algorithm combining game tree search and HTN planning. It cannot handle task failures during plan execution and the HTN description it used cannot describe more complex relationships among tasks and impacts of environment. In the experiments AHTN will use two kinds of HTNs (Low Level and Flexible) without extended HTN description.
2. RandombiasedWorker: The AI player with RandombiasedWorker strategy will create actions randomly.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 April 2017 doi:10.20944/preprints201704.0119.v1

3. LightRush: It is a hard-coded strategy. The AI player will produce light attackers and command them to attack enemy immediately.
4. HeavyRush: It is a hard-coded strategy. The AI player will produce heavy attackers and command them to attack enemy immediately.
5. UCT: It is a variant of MCTS by modifying the strategy of exploring the child node. It is believed to adapt automatically to the effective smoothness of the tree. We use an implementation with extension of handling simultaneous and durative actions in experiments [10].

Before comparing against other algorithms, the parameters following have to be confirmed firstly.

- CPU time: It is the limited amount of time for an AI player using CPU per game frame. In our experiment, we will use different CPU time settings to test the performance of the algorithms.
- Playout policy: The playout policies of AHTNR, AHTN and UCT in our experiments are all *RandombiasedWorker*.
- Playout time: It is the maximum running time of a playout. All playouts are limited to 100 game frames.
- Pathfinding algorithm: The AI player uses the pathfinding algorithm to get the path from current location to destination location. In our experiment, we use A* pathfinding algorithm as pathfinding algorithm.
- Evaluation function: In our experiments, an evaluation function from [10] will be used to compute the reward to get the best plan. The evaluation function from [10] is a variant of LTD2 [42] which not only considers the units' hit-points but also their costs.
- Maximum game time: The maximum game time is limited in 3000 cycles. It means that if both players have living units at 3000th cycle, the game will be over as a Tie.
- Maps: The three maps used in our experiments are: M1 (8×8 tiles), M2 (12×12 tiles) and M3 (16×16 tiles).

In order to evaluate different algorithms, the round-robin tournament will be used. Each algorithm will play 100 games against the others in three different maps with random starting positions. The method to compute the score of each algorithm is as follows: the reward of winner is set to 1 point. 0.5 point is given to both algorithms when there is a Tie. All competitions start with one base, the same count of resource and one worker for both players.

4.2 Experimental Results and Analysis

In this section, we compare the performance of AHTNR and other search algorithms from three aspects: average score, average decision time and average failed task repair rate on three maps against 8 kinds of AI players.

4.2.1. Average Score under Different Parameters Settings on Three Maps

We compare the average scores of each search algorithm against others in 100 games with the CPU time from 20 to 200 milliseconds. **Figure 1**Figure 4-6 show the average scores of three maps respectively.

According to the results in Figure 4-6, we can see that the player of AHTNR-F (the HTN description used is *Flexible*) outperforms other players on all three maps under different CPU time. The performances of AHTNR-F change little with the increasing scale of the maps. Moreover, the average scores of AHTNR-F approximate the maximum average score. AHTNR-LL (the HTN description used is *Low Level*) has similar performance with AHTN-LL on the small map. However it seems to perform worse than AHTN-LL on the large map. Because the larger map has more complex dynamic environment and AHTNR-LL has poor domain knowledge to make suitable repair plan for such complex environment. The performance of AHTNR-LL and AHTN-LL

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 April 2017 doi:10.20944/preprints201704.0119.v1

deteriorates as the size of map grows. Because more units are produced on larger maps and the domain knowledge which AHTNR-LL and AHTN-LL obtain is too simple to generate better plans in such complex game. The scripted methods (*Heavy Rush* and *Light Rush*) seem to perform better on larger maps. This is in fact caused by the AHTNR-LL and AHTN-LL under-performing on larger maps. The performances of all AI players change little under different CPU time settings on three maps. This is because all AI players only cost a few amount of time to make decisions. Moreover, the AI players can save several cycles to make a decision. Because the performances of all AI players are similar under different CPU time settings, we will use one setting of the CPU time in the following.

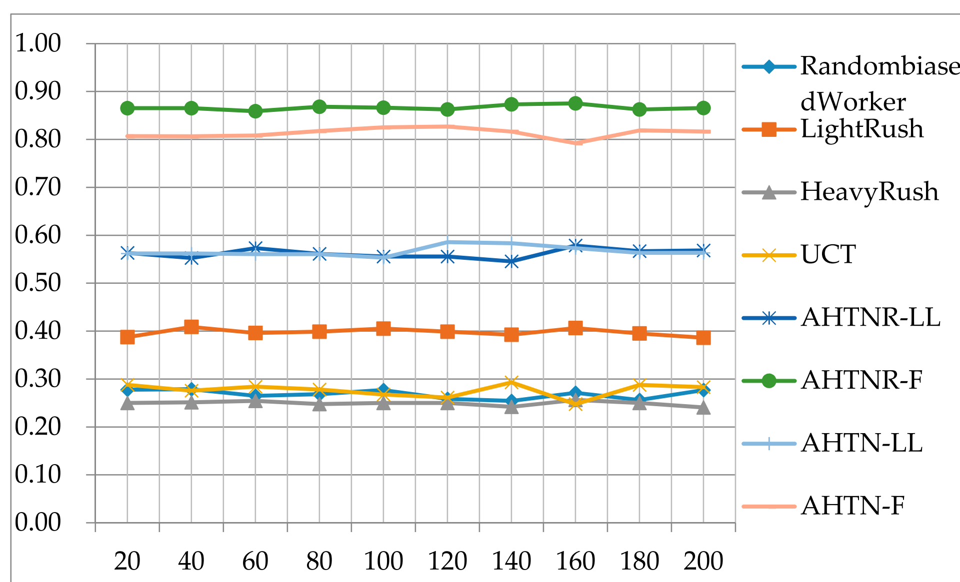

Figure 4. The average score of each algorithm. The horizontal axis is the CPU time from 20 to 200 milliseconds. And the vertical axis is the average score of 800 games. The Playout time is 100 milliseconds. The Max playout is 1. And the map is M1 (8×8 tiles).

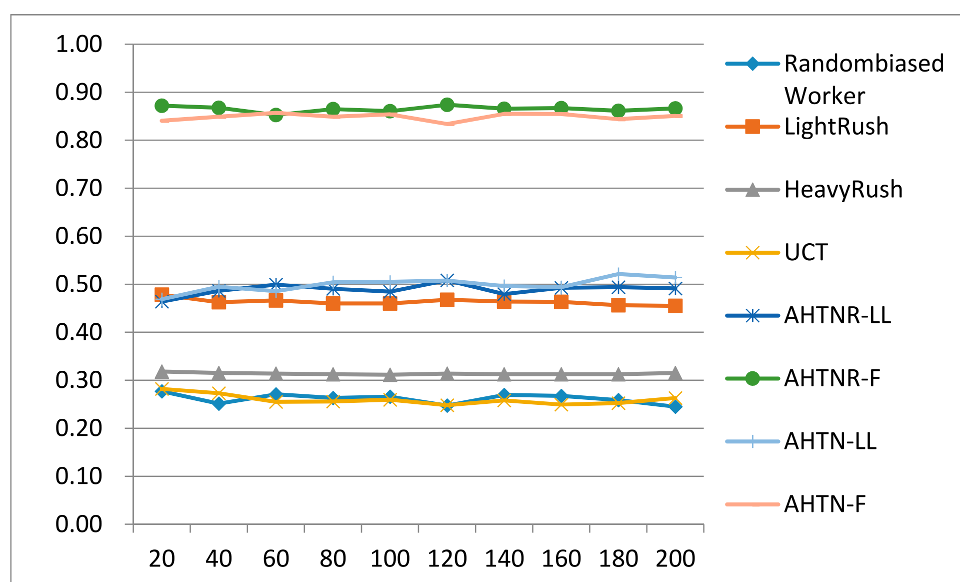

Figure 5. The average score of each algorithm. The horizontal axis is the CPU time from 20 to 200 milliseconds. And the vertical axis is the average score of 800 games. The Playout time is 100 milliseconds. The Max playout is 1. And the map is M2 (12×12 tiles).

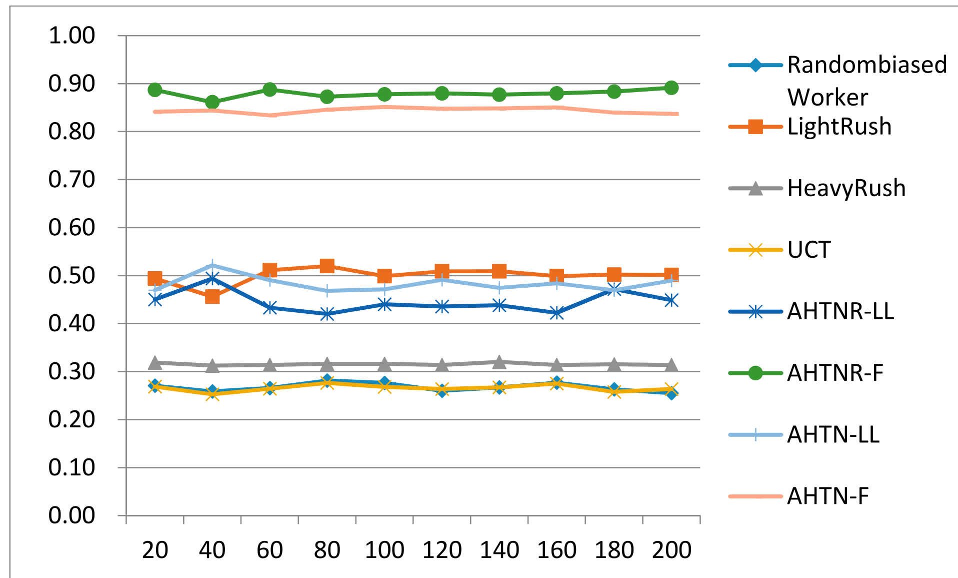

Figure 6. The average score of each algorithm. The horizontal axis is the CPU time from 20 to 200 milliseconds. And the vertical axis is the average score of 800 games. The Playout time is 100 milliseconds. The Max playout is 1. And the map is M3 (16×16 tiles).

4.2.2. Average Decision Time on Three Maps

In RTS games, it requires the AI players to make their decisions as quick as possible. Figure 7-9 show the average planning time of AHTNR and AHTN for one decision process of 100 games on three maps respectively.

From the results in Figure 7-9, we can find that AHTNR usually needs more decision time than AHTN when they have similar domain knowledge. In the decision process, AHTNR repairs failed tasks and sorts all generated alternative plans in search process by their evaluated values. Both of them cost extra planning time. But it improves the performance of AHTNR in dynamic environment by repairing failed plans. Moreover, as shown in Table 1-2, the time of repairing failed tasks and saving alternative plans is short compared with the total time in a decision process. After all, AHTNR costs more decision time than AHTN. But it is in an acceptable range.

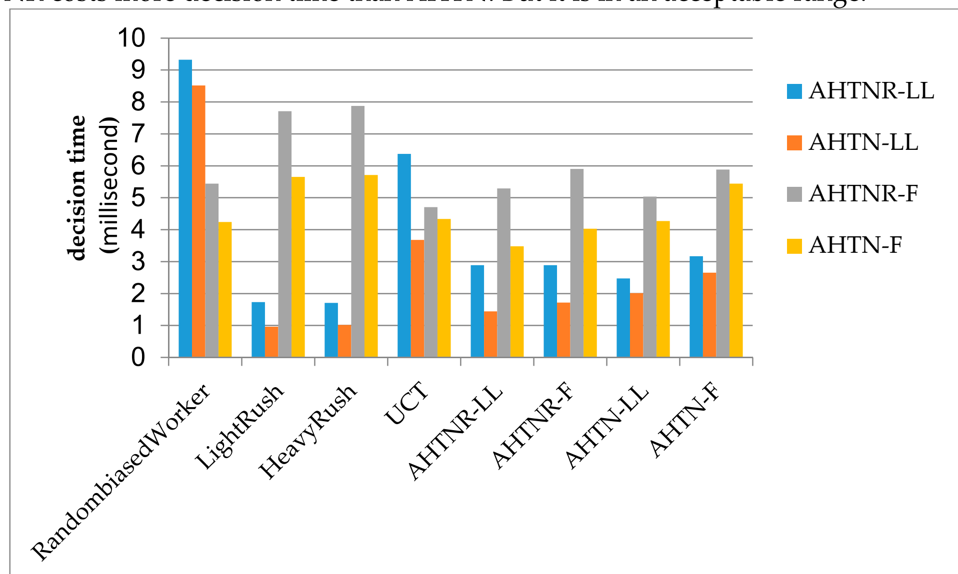

Figure 7. The average decision time of AHTNR and AHTN with two kinds of HTN descriptions of 100 games where CPU time is 100 milliseconds. The Playout time is 100 milliseconds. The Max playout is 1. The map used is M1 (8×8 tiles).

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 April 2017 doi:10.20944/preprints201704.0119.v1

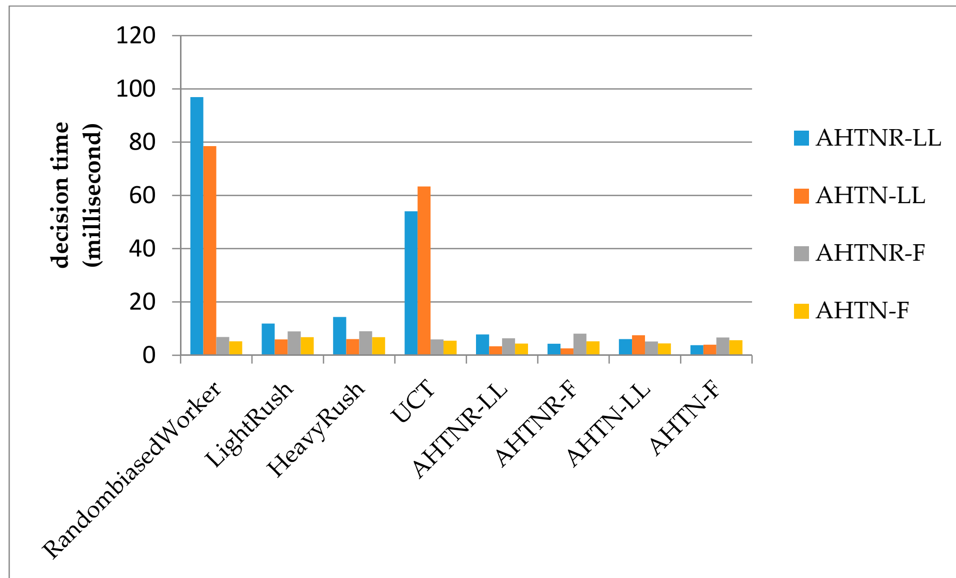

Figure 8. The average decision time of AHTNR and AHTN with two kinds of HTN descriptions of 100 games where CPU time is 100 milliseconds. The Playout time is 100 milliseconds. The Max playout is 1. The map used is M2 (12×12 tiles).

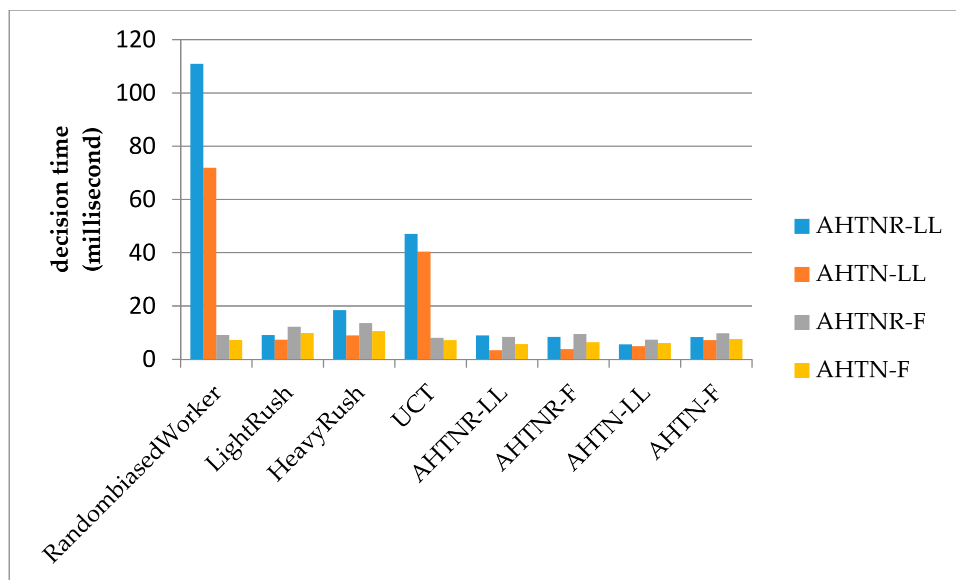

Figure 9. The average decision time of AHTNR and AHTN with two kinds of HTN descriptions of 100 games where CPU time is 100 milliseconds. The Playout time is 100 milliseconds. The Max playout is 1. The map used is M2 (12×12 tiles).

Table 1. The percentage of extra time which consists of repairing failed task and saving alternative plans in total time for a decision process. The games are between AHTNR-LL and 8 kinds of AI players.

AI player	**M1**	**M2**	**M3**
RandombiasedWorker	0.1682	0.0003	0.0003
LightRush	0.0381	0.0078	0.0105
HeavyRush	0.0355	0.0066	0.0070
UCT	0.0498	0.0057	0.0060
AHTNR-LL	0.0202	0.0128	0.0137
AHTNR-F	0.034	0.0188	0.0103
AHTN-LL	0.0221	0.0103	0.0114
AHTN-F	0.0176	0.0174	0.0071

Table 2. The percentage of extra time which consists of repairing failed task and saving alternative plans in total time for a decision process. The games are between AHTNR-LL and 8 kinds of AI players.

AI player	M1	M2	M3
RandombiasedWorker	0.0265	0.0290	0.0228
LightRush	0.0226	0.0259	0.0217
HeavyRush	0.0216	0.0275	0.0219
UCT	0.027	0.0274	0.0275
AHTNR-LL	0.0312	0.0348	0.0391
AHTNR-F	0.0302	0.0353	0.0272
AHTN-LL	0.0232	0.0279	0.0283
AHTN-F	0.0192	0.0263	0.0141

4.2.3. Average Failed Task Repair Rate on Three Maps

In this section, we discuss the average repair rate of AHTNR on three maps against 8 kinds of AI players in 100 games. The results are shown in Table 3 and Table 4, where the domain knowledge is set to be *Low Level* and *Flexible* respectively.

According to the results of Table 3 and Table 4, we can see that AHTNR-F has higher repair rate than AHTNR-LL against all kinds of AI players on three maps. It is caused by the difference between the domain knowledge used by AHTNR-LL and AHTNR-F. The main reason of repair failure is that there are no enough units to execute alternative plans. As AHTNR-F has richer domain knowledge, it can make better and more alternative plans to repair failed tasks than AHTNR-LL.

Table 3. Average repair rate of AHTNR-LL on three maps against 8 kinds of AI players. M1 is 8×8 tiles. M2 is 12×12 tiles. And M3 is 16×16 tiles. The symbol "-" means there are no failed tasks.

AI player	M1	M2	M3
RandombiasedWorker	0.262	0.278	0.227
LightRush	-	0.324	0.037
HeavyRush	-	0.929	0
UCT	0.618	0.591	0.553
AHTNR-LL	0	0	0
AHTNR-F	0.005	0	0
AHTN-LL	0.355	0.343	0.373
AHTN-F	0.236	0.233	0.213

Table 4. Average repair rate of AHTNR-F on three maps against different AI players. M1 is 8×8 tiles. M2 is 12×12 tiles. And M3 is 16×16 tiles. The symbol "-" means there are no failed tasks.

AI player	M1	M2	M3
RandombiasedWorker	0.904	0.886	0.852
LightRush	-	0.753	0.449
HeavyRush	1	1	0.767
UCT	0.945	0.907	0.906
AHTNR-LL	0.008	0.044	0
AHTNR-F	0.016	0.01	0.018
AHTN-LL	0.401	0.618	0.697
AHTN-F	0.407	0.373	0.452

5. Conclusions and future works

AHTN algorithm has addressed the problem of large state spaces and enormous branch factors in RTS games. However, the HTN description it used cannot describe complex relationships among

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 April 2017 doi:10.20944/preprints201704.0119.v1

tasks and impacts of environment. Moreover it cannot handle task failures caused by dynamic environment during plan execution. The AHTNR algorithm we introduce in this paper addresses the above problems by using extended HTN description and incorporating a task repair strategy based on historical information. Compared with state-of-the-art algorithms in μRTS game, the AHTNR shows its capability in repairing task failures using the knowledge of the extended HTN description to handle more complex decision problems.

There are multiple future research directions that can be extended to AHTNR. For instance, refined HTNs could be automatically extracted from thousands of RTS game replays. The more perfect HTN could give great help to improve the performance of AHTNR as shown in experiments. And both AHTN and AHTNR algorithm don't consider indetermination and partial observation, which is frequently appeared in RTS games. This would lead the evolving game states to be different from the forecast during the HTN planning, and affect the validity of the plan solution. Therefore, improved AHTNR algorithm using the techniques of HTN planning under uncertainty will enhance its performance in RTS games which are non-deterministic and partially observable.

Acknowledgments: We want to thank Santiago Ontanón for discussions about plan repairing. The work described in this paper is sponsored by the National Natural Science Foundation of China under Grant No. 61403402 and No. 61473300.

Author Contributions: Lin Sun, Peng Jiao and Yabing Zha proposed the method; Quanjun Yin, Kai Xu and Lin Sun designed and performed the experiments; Lin Sun and Kai Xu analyzed the experimental data; Lin Sun wrote the paper.

Conflicts of Interest: The authors declare that there is no conflict of interests regarding the publication of this paper.

References

1. Buro Michael. "Real-time strategy games: A new AI research challenge." IJCAI. 2003.
2. Ontanón, Santiago, et al. "A survey of real-time strategy game ai research and competition in starcraft." IEEE Transactions on Computational Intelligence and AI in games 5.4 (2013): 293-311.
3. Knuth, Donald E., and Ronald W. Moore. "An analysis of alpha-beta pruning." Artificial intelligence 6.4 (1975): 293-326.
4. Ontañón, Santiago. "Experiments with game tree search in real-time strategy games." arXiv preprint arXiv:1208.1940 (2012).
5. Chung. M., Buro, M.and Schaeffer. J. "Monte carlo planning in rts games." In Proceedings of IEEE-CIG 2005.
6. Balla, R.K., and Fern, A. "UCT for tactical assault planning in real-time strategy games." In Proceedings of IJCAI 2009, 40–45.
7. Churchill, David, Abdallah Saffidine, and Michael Buro. "Fast Heuristic Search for RTS Game Combat Scenarios." AIIDE. 2012.
8. Ontanón, Santiago. "The combinatorial multi-armed bandit problem and its application to realtime strategy games." In Proceedings of AIIDE, 2013.
9. Alexander Shleyfman, Anton´ın Komenda, and Carmel Domshlak. "On combinatorial actions and CMABs with linear side information." In Proceedings of ECML 2014. Springer, 2014.
10. Ontanón, Santiago, and Michael Buro. "Adversarial hierarchical-task network planning for complex real-time games." Proceedings of the 24th International Conference on Artificial Intelligence. AAAI Press, 2015.
11. Sacerdoti, Earl D. The nonlinear nature of plans. No. SRI-TN-101. STANFORD RESEARCH INST MENLO PARK CA, 1975.
12. Sacerdoti, Earl D. A structure for plans and behavior. SRI INTERNATIONAL MENLO PARK CA ARTIFICIAL INTELLIGENCE CENTER, 1975.
13. Tate, Austin. "Generating project networks." Proceedings of the 5th international joint conference on Artificial intelligence-Volume 2. Morgan Kaufmann Publishers Inc., 1977.
14. Wilkins, David E. "Domain-independent planning Representation and plan generation." Artificial Intelligence 22.3 (1984): 269-301.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 19 April 2017 doi:10.20944/preprints201704.0119.v1

15. Wilkins, David E. "Can AI planners solve practical problems?." Computational intelligence 6.4 (1990): 232-246.
16. Currie, Ken, and Austin Tate. "O-Plan: the open planning architecture." Artificial Intelligence 52.1 (1991): 49-86..
17. Tate, Austin, Brian Drabble, and Richard Kirby. "O-Plan2: an open architecture for command, planning and control." Intelligent Scheduling. 1994..
18. Erol, Kutluhan, Dana S. Nau, and Venkatramana S. Subrahmanian. "Complexity, decidability and undecidability results for domain-independent planning." Artificial intelligence 76.1 (1995): 75-88.
19. Nau, Dana, et al. "SHOP: Simple hierarchical ordered planner." Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2. Morgan Kaufmann Publishers Inc., 1999.
20. Nau, Dana S., et al. "SHOP2: An HTN planning system." J. Artif. Intell. Res.(JAIR) 20 (2003): 379-404.
21. de la Asunción, Marc, et al. "SIADEX: An interactive knowledge-based planner for decision support in forest fire fighting." Ai Communications 18.4 (2005): 257-268.
22. Georgievski, Ilche, and Marco Aiello. "HTN planning: Overview, comparison, and beyond." Artificial Intelligence 222 (2015): 124-156..
23. Kuter, Ugur, and Dana Nau. "Forward-chaining planning in nondeterministic domains." AAAI. 2004.
24. Kuter, Ugur, and Dana Nau. "Using domain-configurable search control for probabilistic planning." Proceedings of the National Conference on Artificial Intelligence. Vol. 20. No. 3. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2005.
25. Bonet, Blai, and Hector Geffner. "Planning with incomplete information as heuristic search in belief space." (2000).
26. Bonet, Blai, and Hector Geffner. "Labeled RTDP: Improving the Convergence of Real-Time Dynamic Programming." ICAPS. Vol. 3. 2003.
27. Bertsekas, Dimitri P., et al. Dynamic programming and optimal control. Vol. 1. No. 2. Belmont, MA: Athena Scientific, 1995..
28. Kelly, John-Paul, Adi Botea, and Sven Koenig. "Planning with hierarchical task networks in video games." Proceedings of the ICAPS-07 Workshop on Planning in Games. 2007.
29. Menif, Alexandre, Eric Jacopin, and Tristan Cazenave. "SHPE: HTN Planning for Video Games." Third Workshop on Computer Games, CGW 2014. 2014.
30. Soemers, Dennis JNJ, and Mark HM Winands. "Hierarchical Task Network Plan Reuse for Video Games."
31. Humphreys, Troy. "Exploring HTN planners through examples." Game AI Pro: Collected Wisdom of Game AI Professionals 149 (2013)..
32. Muñoz-Avila, Hector, and David Aha. "On the role of explanation for hierarchical case-based planning in real-time strategy games." Proceedings of ECCBR-04 Workshop on Explanations in CBR. 2004.
33. Spring RTS: springrts.com.
34. Laagland, Jasper. "A HTN planner for a real-time strategy game." Unpublished manuscript.(hmi. ewi. utwente. nl/verslagen/capita-selecta/CS-Laagland-Jasper. pdf) (2008).
35. Naveed, Munir, Diane E. Kitchin, and Andrew Crampton. "A hierarchical task network planner for pathfinding in real-time strategy games." Proc. 3rd Int. Symp. AI & Games. 2010.
36. Sánchez-Garzón, Inmaculada, Juan Fdez-Olivares, and Luis Castillo. "A Repair-Replanning Strategy for HTN-based Therapy Planning Systems."
37. Gateau, Thibault, Charles Lesire, and Magali Barbier. "Hidden: Cooperative plan execution and repair for heterogeneous robots in dynamic environments." Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on. IEEE, 2013.
38. Ayan, N. Fazil, et al. "Hotride: Hierarchical ordered task replanning in dynamic environments." Planning and Plan Execution for Real-World Systems–Principles and Practices for Planning in Execution: Papers from the ICAPS Workshop. Providence, RI. Vol. 38. 2007.
39. Kuter, Ugur, and Chris Miller. "Computational Mechanisms to Support Reporting of Self Confidence of Automated/Autonomous Systems." 2015 AAAI Fall Symposium Series. 2015.
40. Erol, Kutluhan, James A. Hendler, and Dana S. Nau. "UMCP: A Sound and Complete Procedure for Hierarchical Task-network Planning." AIPS. Vol. 94. 1994.
41. Ontañón Santiago. microRTS. https://github.com/santiontanon/microrts, 2016.
42. Churchill David and Buro Michael. "Portfolio greedy search and simulation for large scale combat in StarCraft. " In Computational Intelligence in Games (CIG), 2013 IEEE Conference on pages 1–8. IEEE, 2013.