# HTN Fighter: Planning in a Highly-Dynamic Game

Xenija Neufeld
Faculty of Computer Science
Otto von Guericke University
Magdeburg, Germany,
Crytek GmbH, Frankfurt, Germany
xenija.neufeld@ovgu.de

Sanaz Mostaghim
Faculty of Computer Science
Otto von Guericke University
Magdeburg, Germany
sanaz.mostaghim@ovgu.de

Diego Perez-Liebana
University of Essex
School of Computer Science
and Electronic Engineering,
Colchester, United Kingdom
dperez@essex.ac.uk

***Abstract*—This paper proposes a plan creation and execution system used by the agent *HTN Fighter* in the *Fighting ICE* game framework. The underlying approach implements a Hierarchical Task Network (HTN) planner and a simple planning domain that focuses on sequences of close-range attacks. The execution process is tightly interleaved with the planning process compensating for the uncertainty caused by the delay of 15 frames which the information about the game world state is provided with. Using an HTN and the proposed execution system, the agent is able to follow high-level strategies staying reactive to changes in the environment. Experiments show that *HTN Fighter* outperforms the sample *MCTS* controller and the top three controllers submitted to the 2016 *Fighting Game AI Competition*.**

## I. Introduction

In the last four years, the *Fighting Game AI Competition*[1] became one of the well-known competitions for game-playing agents. Using the *FightingICE* framework that represents many similar commercial fighting games, it provides a highly-dynamic test environment in the field of artificial and computational intelligence in games.

Most of the early agents submitted to the competition were rule-based and followed a simple decision making logic like, for example, the winner of the 2015 competition *Machete* [1]. Later, some approaches tried to adapt their behaviors by predicting the opponent's next action [2],[3] and learning fighting strategies at run-time. Most of the participants of the 2016 competition implemented a variation of *Monte Carlo Tree Search* (MCTS) which is provided with a sample agent [4], [5]. The best three agents of this competition implemented a mixture of a rule-based system and MCTS [6]. Additionally, there are some agents that used other approaches during previous competitions trying to adapt their rule-bases like, for example, *BANZAI* [6] and *CodeMonkey* [7] – the winner of 2014s competition which implemented *dynamic scripting*.

However, all of these approaches are purely reactive. Thus, they search for an action that is optimal for the current game state without taking into account previous actions or future goals. None of these approaches implements high-level strategies using long-term action plans. Although MCTS takes into consideration possible outcomes of actions in future game states, it still provides only *one* action at a time and performs a new search in every frame.

[1]Fighting Game AI Competition: www.ice.ci.ritsumei.ac.jp/~ftgaic

In this paper, we propose using a *Hierarchical Task Network* (HTN) in order to create sequences of actions (plans) that are supposed to provide more advanced behaviors. When using a planner, it is possible to make decisions taking into account long-term goals and high-level strategies providing longer plans instead of single actions. For a fighting game, such strategy could be, for example, keeping the opponent stunned for some time while dealing more damage to him.

Although planning is widely used in other research areas, there is a reason why it is barely applied in such highly-dynamic environments as video games. In contrast to classical planning environments which are static and where a created plan can usually be executed to its end, game environments change quickly. While an agent is still executing a plan step, its opponent might perform a few actions so that the agent's plan becomes invalid and a new plan needs to be created. Thus, planning and plan execution need to be tightly interleaved in order for the agent to act deliberately [8]. This is an even more difficult task for *Fighting ICE* where an agent gets the information about the world state delayed by 15 frames. Thus, the planner needs to rely on a simulation model in order to approximate the present data.

In this work, we implement an HTN planner with a relatively simple planning domain and combine it with the agent controller *HTN Fighter* that is responsible for plan execution. Amongst other behaviors, the planner makes use of the *combo*-system provided by the game and generates plans of multiple combo-attacks. The importance of such combo-attacks in a fighting game is described in [9]. In order to recognize plan failures, the agent controller checks the progress of the previous plan step and the validity of the next one before executing it. When necessary it queries the planner for a new plan.

In order to test whether it is possible to use a planner in such a highly-dynamic game, we let the agent play against the top three agents of the 2016 *Fighting Game AI Competition* and the sample *MCTS* controller. Therefore, we do not only look at the agent's overall performance, but also at his execution of combos in the game.

The rest of this paper is structured as follows: Section II gives some background information on the game framework *FightingICE* and provides some insights into HTN planners. Section III describes *HTN Fighter*'s architecture and its planning domain. Then, Section IV details experiments and their results and finally, Section V concludes the paper proposing some directions for future work.

## II. Background

### A. FightingICE

The *FightingICE* platform offers an environment for research in the area of artificial intelligence in games. This framework presents a fighting game for two players which can be either human players or programmed agents. Since 2013, *FightingICE* is used for the *Fighting Game AI Competition*[10] to which developers can submit their agents as participants of a tournament.

Similar to many commercial fighting games, the setting of *FightingICE* takes place on a spatially limited two-dimensional stage on which the two players can move and perform certain attack and defense actions. The players are represented by one of the three game characters *ZEN, GARNET* and *LUD*, each of which can perform certain skills. However, every character has different requirements for the skills and achieves different effects with them. This information is saved in the so-called *character data*.

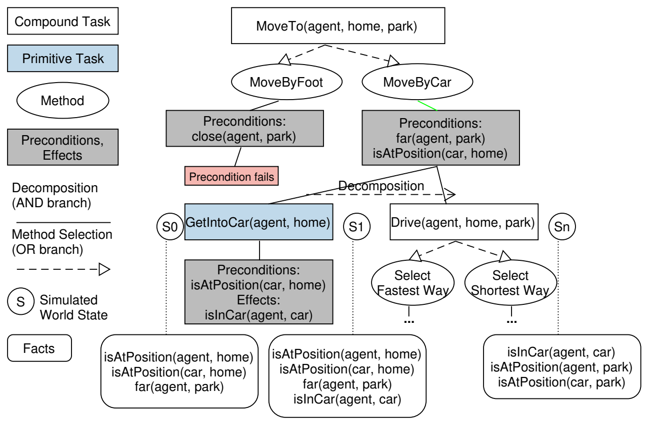
Besides the character data which do not change over time, the agents can access the so called *frame data* which change with every game frame. These data contain information about e.g. positions of the two characters and the amounts of their health and energy points and are used by most of the existing agents for decision making. Since the game runs in $60$ frames per second, in every frame, an agent has $16,67$ milliseconds to perform all necessary computations and to respond to the game environment with an action. However, the agents get the frame data delayed by $15$ frames instead of the current data. This adds more uncertainty to the game and makes it even more difficult for the agents to make decisions.

Imitating the input of a human player, agents are required to perform their actions through simulated key-inputs. Furthermore, performing certain actions in a sequence leads to a so-called *combo*. A successfully performed combo of $4$ combo-attacks causes additional damage to the opponent. However, in order to perform a combo successfully, the time between its attacks is limited to $30$ frames. Furthermore, a combo can be aborted by the opponent through a so-called *combo-breaker* skill which also has to be performed within a given period of time. The fact that sequences of skills build up combos is an additional reason for the usage of an HTN.

### B. HTN

*Hierarchical Task Networks* [11, Chapter 11.5], [12] are often used for planning purposes in the areas of robotics. Also, there are some commercial games that successfully implemented HTNs to define behaviors of non-player characters [13]–[15]. Furthermore, there is some research being done in the field of artificial intelligence in real-time strategy games that uses HTNs for agent behavior [16].

A Hierarchical Task Network planner is a planner that – in contrast to classical planners – does not search a space of world states in order to find a goal world state. Instead, it takes a high-level task that needs to be accomplished by an agent and searches for possible decompositions of this task into a sequence of sub-tasks. Thus, it uses a network of tasks that build a hierarchy.



Compound Task
Primitive Task
Method
Preconditions, Effects
Decomposition (AND branch)
Method Selection (OR branch)
S Simulated World State
Facts
MoveTo(agent, home, park)
MoveByFoot
MoveByCar
Preconditions: close(agent, park)
Precondition fails
Preconditions: far(agent, park) isAtPosition(car, home)
Decomposition
S0
GetIntoCar(agent, home)
S1
Drive(agent, home, park)
Sn
Preconditions: isAtPosition(car, home) Effects: isInCar(agent, car)
Select Fastest Way
Select Shortest Way
...
isAtPosition(agent, home) isAtPosition(car, home) far(agent, park)
isAtPosition(agent, home) isAtPosition(car, home) far(agent, park) isInCar(agent, car)
isInCar(agent, car) isAtPosition(agent, park) isAtPosition(car, park)

Fig. 1: HTN planning example.

Tasks that can be decomposed into smaller parts are called *compound tasks* and those, that are the leaves of the network representing basic actions of an agent, are called *primitive tasks*. Primitive tasks contain *operators* which correspond to parametrized descriptions of agent actions [17]. Operators are defined by *preconditions* under which they might be performed and *effects* that they have on the world. In Figure 1, an agent needs to move from *home* to *park* and thus, his top-most compound task is *MoveTo*.

For the planner to be able to check preconditions and to apply effects during the planning process, it requires an inner representation of the world state and a simulation model. Usually, a world state, preconditions and effects are represented by so-called *facts* which describe properties of the world by some functions and variables. In our example, one of the facts that are true in the initial world state *S0*, is *isAtPosition(agent, home)* which describes the agent's current position. After getting into the car, the task's effect is applied and the fact *isInCar(agent, car)* is added to the next world state *S1*.

In order to describe how a compound task can be decomposed, the so-called *methods* are used. It is possible that a compound task can be decomposed in multiple ways using different methods. For example, the goal task *MoveTo* could be accomplished by either moving by foot or by car. So, in order to decide which method to use for a task decomposition, methods also have preconditions defining when they are applicable. For example, moving by foot is only possible for short distances and moving by car is not possible if there is no car available. These preconditions are usually checked by the planner for every method in the order that the methods are defined in the *planning domain* which contains the description of all tasks, methods and facts. In graphical representations, the order is usually from left to right (see Figure 1). As soon as an applicable method is found, it is used to decompose a compound task. Usually, the preconditions of further methods are not checked, unless the selected method fails to fully decompose the task.

A method leads to further subtasks which can be primitive or compound. If a primitive task such as *GetIntoCar* is reached, it is added to the final plan. The process of task decomposition continues until the plan contains only primitive tasks. Well-known approaches to HTN planners are the *Simple Hierarchical Ordered Planner* (SHOP) [18] and his successor SHOP2 [19]. These implement the *total-order* decomposition of tasks meaning that tasks are decomposed and added to the plan in the same order that they will be executed later on. However, it is also possible to use *partial-order* decomposition, defining the order for only some of the tasks [20].

In contrast to many reactive approaches that are usually used to define agent behavior in video games, planners take long-term goals into account and plan further in advance. They are well applicable in the most video games, since "many game AI problems can be formulated as planning problems" [21]. Even though an HTN planner does not always provide the *optimal* plan, because it does not search through the whole search space, it is usually sufficient for a game environment, as it is efficient and delivers *some* plan that is feasible and leads to the goal. Furthermore, especially because of the hierarchical decomposition of a problem into subproblems, an HTN is more similar to human reasoning, so that the planning domain can be easily constructed by developers.

## III. HTN Fighter

In order to create a good planning domain for an HTN planner, it requires good knowledge about the environment that the planner has to operate in – namely the game. One important aspect to consider are the preconditions of methods and primitive tasks. With well-defined preconditions at higher levels of a hierarchy, decisions can be made earlier, cutting away unfeasible parts of the search space. Additionally, the order of methods plays an important role if the search is guided by this order only, without using any heuristics as is the case for the work described here.

Keeping these aspects in mind, we created a relatively small domain for *HTN Fighter* ordering methods at the higher levels of the hierarchy by the priority we considered best. These hierarchy levels are shown in Figure 2. Here, the top-most task for the agent is always *Act*. This task can be decomposed by six methods. After some observations of the agents submitted to previous competitions, we noticed when an agent might become vulnerable and for that reason, assigned the highest priority to the first two methods *Avoid Projectiles* and *Escape From Corner*. Similar considerations were made for the rest of the methods. Even though all methods are important for the game play, most of them represent *single* actions and do not contribute to any *complex* behavior.

With the methods *Use Combo* and *Knock-Back Attack*, however, the agent is supposed to show some strategic close-range behavior following plans of multiple actions. These methods were implemented to test whether it is possible to use a planner in such a highly-dynamic game using the execution system described later in this chapter. As already mentioned in Section II-A, *FightingICE* allows for execution of *combos* which consist of four attacks and give additional damage to the opponent. Using the method *Use Combo*, the planner can create a plan of up to four attack actions. Additionally, method *Knock-Back Attack* decomposes the task *Use Attack Skill* and consist of three sub-tasks: *Knock-Back Attack* which is used twice and *Knock-Down Attack*. Following this strategy, the agent keeps the opponent stunned (uncontrollable) for several frames dealing more damage without getting hurt.

In contrast to the high levels of the hierarchy, we do not predefine the order of low-level tasks in the hierarchy. Instead, the methods of distinct attack actions are added dynamically at the beginning of a game by checking each attack for its parameters. Furthermore, these methods are sorted by the damage the corresponding attacks deal. That way, the preconditions of the attacks with higher damage are checked first, giving the character the chance to always execute the most powerful attack applicable in the current situation. Adding methods dynamically provides the following advantage: there is no need to create a distinct planning domain for every character. Since the approach checks for the actions' parameters (which are different for every character), it assigns the correct action methods to compound tasks for every character.

For the same reason, the preconditions of primitive tasks are defined in a generic way. Instead of predefining that, for example, action *STAND_A* should only be executed when the opponent is within the distance of 40 units, the planner checks whether the *hit box* of an action (which is provided by the game for the current character) intersects with the opponent's *hit box*. This way, it is possible to use the same preconditions for all attack actions accessing the corresponding action parameters.

As already mentioned in section II-B, an HTN planner usually has its own simulation model of the environment and uses predefined effects of tasks to simulate changes in the world state caused by these tasks. However, there is no need to pre-define such effects for *FightingICE*. Instead, it is possible to use the simulator provided by the framework and to simulate plan tasks directly on copies of the frame data.

A big challenge when using a planner in such a highly-dynamic real-time environment as a video game is the correct execution of plans, while staying reactive to changes in the environment. This is an even bigger challenge when knowledge about the planning environment is delayed by 15 frames as is the case for *Fighting ICE*. For that reason, the underlying architecture should provide a possibility to interleave planning and execution, recognizing plan failures and re-planning at runtime as described in [22]. The architecture used in this work contains two main loops the *planning loop* and the *execution loop*. The *execution loop* is the same that is used by every agent implemented for *FightingICE*. Here, in every game frame, the *agent controller* gets *frame data* that is delayed by 15 frames from the *game framework* and provides *Key Input* data in order for the agent to perform an action.

In the *planning loop* the agent controller queries the *HTN planner* for a plan. This loop is only updated when a new plan is required which happens either when the previous plan

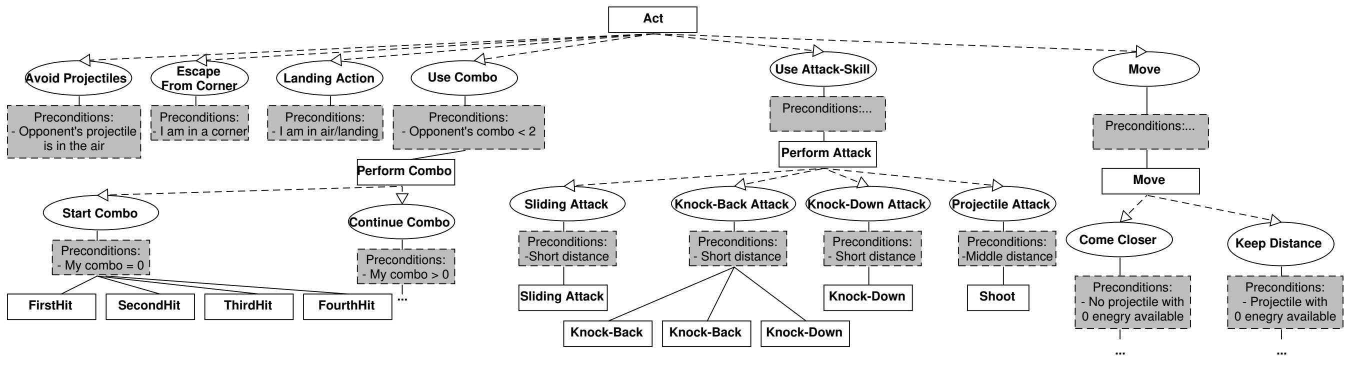

Act
Avoid Projectiles
Preconditions: - Opponent's projectile is in the air
Escape From Corner
Preconditions: - I am in a corner
Landing Action
Preconditions: - I am in air/landing
Use Combo
Preconditions: - Opponent's combo < 2
Perform Combo
Start Combo
Preconditions: - My combo = 0
FirstHit
SecondHit
ThirdHit
FourthHit
Continue Combo
Preconditions: - My combo > 0
...
Use Attack-Skill
Preconditions:...
Perform Attack
Sliding Attack
Preconditions: -Short distance
Sliding Attack
Knock-Back Attack
Preconditions: - Short distance
Knock-Back
Knock-Back
Knock-Down
Knock-Down Attack
Preconditions: - Short distance
Knock-Down
Projectile Attack
Preconditions: -Middle distance
Shoot
Move
Preconditions:...
Move
Come Closer
Preconditions: - No projectile with 0 enegry available
...
Keep Distance
Preconditions: - Projectile with 0 enegry available
...

Fig. 2: High-level HTN for HTN Fighter.

ends (thus, is successfully executed) or when a plan failure occurs. For the recognition of plan failures, the agent controller performs the following two checks: first, it checks whether the *previous* plan step was actually executed by the agent and second, it checks whether the preconditions of the *next* plan step still hold before executing it. If there is no plan failure, the agent controller executes the next task from the plan converting it into the corresponding *Key Input* data.

The check for the previous action is necessary in *Fighting-ICE* especially due to the delay of 15 frames. Since the agent controller does not have full knowledge about the agent's and the world state when executing an action, it cannot be certain about the action actually being executed or aborted. Most of the previous agents submitted to the competition use the class *CommandCenter*(CC) provided by the game framework to check whether a new command/action can be executed. The CC returns false if the character is not controllable, for example, when still executing a previous command or playing a hit-animation. However, the delay of 15 frames is also valid for the CC. Thus, as shown in Figure 3, the CC only recognizes that the command *STAND_A* which was sent in frame 1, is executed after the delay and thus shows the character as controllable for 15 more frames until frame 16.

This is not a problem for agents that make decisions in every frame since they compute the optimal move for the current situation and do not take into account their previous or next actions. Thus, they are in no disadvantage even if they send a command to the CC when the character is actually not controllable and the command gets lost (frame 2 – 15). However, this is obviously a problem for a plan execution system that needs to execute plan steps in the correct order and with the right timing. If the system relied only on the feedback of the CC, it would try to send all the commands of a plan one after another in the first frames having them lost. To prevent this, we added an additional approach to the execution system of *HTN Fighter*. Remembering the time the last command was sent, the underlying architecture does not send a new command in the 15 following frames (unless the previous action is shorter than 15 frames). Only in frame 16, it relies on the feedback of the CC. When the CC shows the presumable end of *STAND_A* in frame 19 and shows the agent as controllable, does the controller send the next command *STAND_B*. This way, the correct execution timing is achieved.

However, if the character is hit in frame 8 and plays the *STAND_RECOV* animation, the CC gets this information only in frame 23 and the command *STAND_B* is still lost. After frame 23 the hit is shown by the CC and, knowing the length of the *STAND_RECOV* animation, the CC knows that the character is uncontrollable until frame 36. At this point the actual character state and the information known by the CC are synchronized again. In frame 36, the CC shows the character as controllable again and this is where the agent controller recognizes a plan failure because the previous action executed (*STAND_RECOV*) is different from the previous command (*STAND_B*). It re-plans and repeats the command *STAND_B* achieving the execution of plan steps in the correct order. The following commands are executed with the correct timing and order, since the character is not interrupted again.

## IV. Experiments and Results

In order to test *HTN Fighter*, we compared him in 100 games against each of the top three agents of the 2016 competition – *Thunder01*, *Ranezi* and *MrAsh*. Additionally, we performed tests against the *MCTS sample controller* which all three agents are based on. With every opponent agent, *HTN Fighter* fought 50 games as player one and 50 games as player two. Every game consisted of 3 rounds. The agents played as character *ZEN* and started with 400 health points (HPs). According to this year's competition rules, a round ended either when one of the agents had zero HPs (and thus lost the round) or after 60 seconds. In this case, the winner of the round was the agent with the higher number of HPs.

The results of the described experiments are shown in Figure 4. As we can see, even with a quite simple planning domain, *HTN Fighter* was able to win more than half of the games against all opponents almost reaching two thirds against the top three opponents of the last year's competition.

In addition to the win rates of *HTN Fighter*, we recorded the average number of combos performed by *HTN Fighter* and

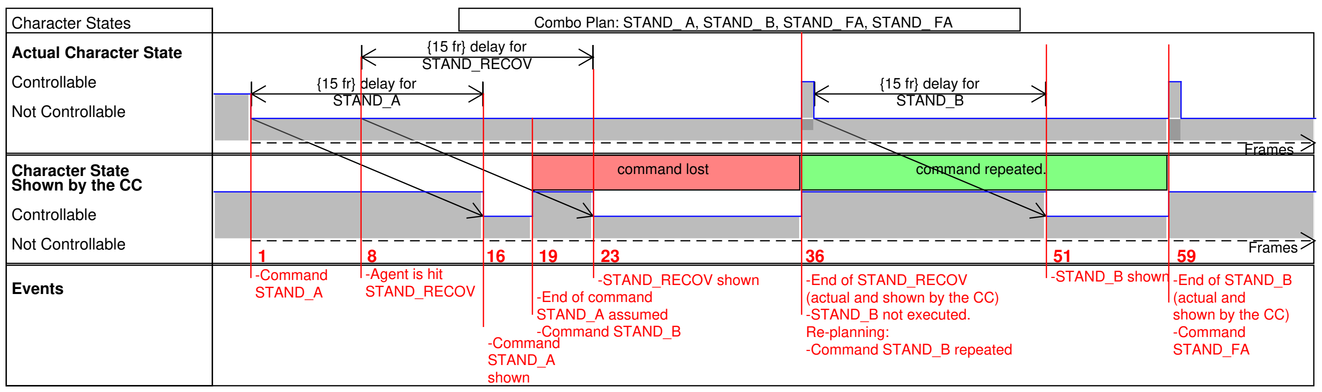

Character States
Combo Plan: STAND_ A, STAND_ B, STAND_ FA, STAND_ FA
Actual Character State
Controllable
Not Controllable
{15 fr} delay for STAND_RECOV
{15 fr} delay for STAND_A
{15 fr} delay for STAND_B
Frames
Character State Shown by the CC
Controllable
Not Controllable
command lost
command repeated.
Frames
1
8
16
19
23
36
51
59
Events
-Command STAND_A
-Agent is hit STAND_RECOV
-Command STAND_A shown
-End of command STAND_A assumed
-Command STAND_B
-STAND_RECOV shown
-End of STAND_RECOV (actual and shown by the CC)
-STAND_B not executed.
Re-planning:
-Command STAND_B repeated
-STAND_B shown
-End of STAND_B (actual and shown by the CC)
-Command STAND_FA

Fig. 3: Time-line with the actual character state (in terms of controllability) and the state shown by the *CommandCenter* to the agent controller.

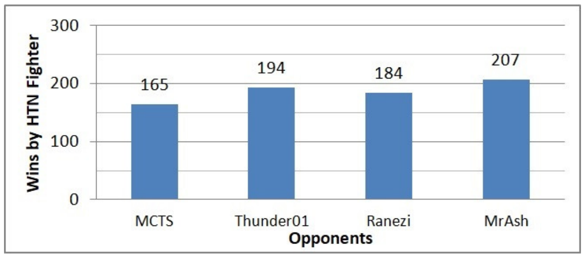


Fig. 4: Number of games won by HTN Fighter against opponent AIs in 100 games (300 rounds).

each opponent.These values should show whether and how often the agent was able to fully or partially execute plans of multiple actions. As already mentioned, an agent can execute a combo of 4 attacks. However, if his opponent executes a *combo-breaker* attack after the second combo-attack, the combo is canceled. This also happens if the agent does not execute two successive combo-attacks within 30 frames.

Figure 5 shows the average numbers of chains of 1 – 4 combo-attacks executed by *HTN Fighter* and every opponent AI throughout the 300 game rounds played against each other. As expected, none of the opponents ever performed a full combo (4 attacks) and there were only a very few cases when the opponents hit 3 combo-attacks and slightly more chains of 2 attacks. Only *MrAsh* executed multiple chains of 2 combo-attacks. For all four opponents, the number of only 1 combo-attack is very high which means that in most cases, the agents did not continue the combo after this attack and performed a different action.

In contrast to the opponent agents, we can see that *HTN Fighter* was able to perform full combos in some rare cases. Also, the higher values for chains of 2 attacks show that the agent tried to perform combos more often. However, the visible difference in the numbers of chains of 2 and 3 attacks shows that in many cases the combo was aborted. This happened because sometimes the opponents recognized *HTN Fighter*'s intention to perform a combo and broke it with a combo-breaker.Although, in most cases, having performed 2 combo attacks, the agent pushed its opponent back, so that the preconditions of the third attack did not hold anymore. At this point, a plan failure was recognized and a new plan was created. This led to an interesting emergent behavior when the new plan contained a *sliding attack* which knocked the opponent down. In combination with the sequences created by the method *Knock-Back Attack* described in section III, the agent was able to keep the opponent uncontrollable for multiple seconds which gave him a big advantage in close-range fights.

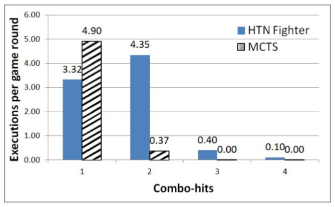
We assume that such close-range attacks were the reason why *HTN Fighter* performed worse against the *MCTS* agent than against the other three opponents. Although the three agents are based on *MCTS*, all of them implement additional logic to approach their opponents. However, *MCTS* lacks this logic and often stays far from its opponent. This gave a disadvantage to the *HTN Fighter* which did not have a special strategy for long-range fights.

Although the numbers of longer combo-chains are quite low, they show that it is possible to create and execute plans of multiple actions even in such a highly-dynamic environment without having complete knowledge about it (due to the 15 frames of delay). Monitoring the plan execution progress and interleaving planning and execution processes enabled us to keep the agent reactive while following plans.

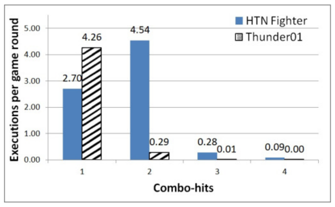
## V. Conclusions and Future Work

This work proposes a *Hierarchical Task Network* (HTN) Planner that is used by the agent *HTN Fighter* in the *Fighting ICE* game framework. Even though the game is very dynamic, this work shows that planning and execution can be interleaved in order to recognize plan failures and re-plan at run-time. The agent shows the ability to execute plans of multiple actions and to act deliberatively.

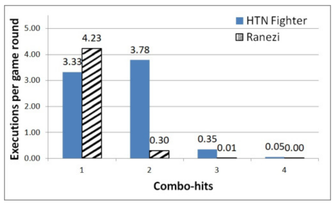
Although the planner uses a relatively simple planning domain, the agent already outperforms the *MCTS* controller and



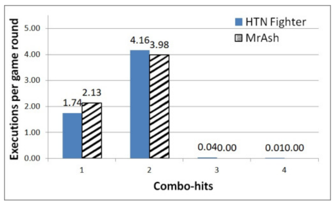
(a) HTN Fighter vs. MCTS



(b) HTN Fighter vs. Thunder01



(c) HTN Fighter vs. Ranezi



(d) HTN Fighter vs. MrAsh

Fig. 5: The average number of successfully performed chains of combo-hits of the length 1 – 4 for each agent pair.

the top three opponents from the 2016 competition. We believe that with a more detailed planning domain even better results can be achieved with this approach. Thus, building a more complex planning domain with better high-level strategies is one of the main tasks for future work. Such strategies could involve, for example, differentiating between the beginning, the middle and the end of a game round and selecting behaviors of different aggressiveness levels accordingly. Alternatively, an agent could decide between different strategies depending on whether the opponent prefers long-range or short-range attacks. For now, we focused on close-range attacks.

Additionally, in order to improve the simulation process during the planning, the opponent's actions could be predicted in a similar way to [2], [3]. Also, in order to execute *waiting* or *movement* actions, the execution part of the system should allow for parameterizing plan tasks, instead of just executing an action once. For example, an agent controller should know how far he should move or for how long he should wait.

Finally, there is scope for other techniques to be used when creating the planning domain. For example, instead of defining the order of HTN methods manually, it might be detected through exploration. For this purpose, the Upper Confidence Bounds might be used as is currently done in many implementations of *MCTS* [4], [5]. Going further, the preconditions of HTN methods [23], [24] or even the methods themselves [25] might be adapted for the different game characters through learning, for example, from re-play data of other (human) players.

## References

[1] T. FightingICE, "2015 fighting game artificial intelligence competition." [Online]. Available: http://www.ice.ci.ritsumei.ac.jp/~ftgaic/index-R15.html

[2] K. Yamamoto, S. Mizuno, C. Y. Chu, and R. Thawonmas, "Deduction of fighting-game countermeasures using the k-nearest neighbor algorithm and a game simulator," in *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*. IEEE, 2014, pp. 1–5.

[3] Y. Nakagawa, K. Yamamoto, and R. Thawonmas, "Online adjustment of the AI's strength in a fighting game using the k-nearest neighbor algorithm and a game simulator," in *Consumer Electronics (GCCE), 2014 IEEE 3rd Global Conference on*. IEEE, 2014, pp. 494–495.

[4] S. Yoshida, M. Ishihara, T. Miyazaki, Y. Nakagawa, T. Harada, and R. Thawonmas, "Application of Monte-Carlo tree search in a fighting game AI," in *Consumer Electronics, 2016 IEEE 5th Global Conference on*. IEEE, 2016, pp. 1–2.

[5] M. Ishihara, T. Miyazaki, C. Y. Chu, T. Harada, and R. Thawonmas, "Applying and improving Monte-Carlo tree search in a fighting game AI," in *Proceedings of the 13th International Conference on Advances in Computer Entertainment Technology*, no. 27. ACM, 2016.

[6] T. FightingICE, "2016 fighting game artificial intelligence competition." [Online]. Available: http://www.ice.ci.ritsumei.ac.jp/~ftgaic/index-R.html

[7] ——, "2014 fighting game artificial intelligence competition." [Online]. Available: http://www.ice.ci.ritsumei.ac.jp/~ftgaic/index-R14.html

[8] D. S. Nau, M. Ghallab, and P. Traverso, "Blended planning and acting: Preliminary approach, research challenges." in *AAAI*, 2015, pp. 4047–4051.

[9] G. L. Zuin, Y. Macedo, L. Chaimowicz, and G. L. Pappa, "Discovering combos in fighting games with evolutionary algorithms," in *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference*. ACM, 2016, pp. 277–284.

[10] F. Lu, K. Yamamoto, L. H. Nomura, S. Mizuno, Y. Lee, and R. Thawonmas, "Fighting game artificial intelligence competition platform," in *Consumer Electronics (GCCE), 2013 IEEE 2nd Global Conference on*. IEEE, 2013, pp. 320–323.

[11] M. Ghallab, D. Nau, and P. Traverso, *Automated planning: theory & practice*. Elsevier, 2004.

[12] I. Georgievski and M. Aiello, "An overview of hierarchical task network planning," *arXiv preprint arXiv:1403.7426*, 2014.

[13] R. Straatman, "Killzone 2: Multiplayer bots." [Online]. Available: http://files.aigamedev.com/coverage/GAIC09_Killzone2Bots_StraatmanChampandard.pdf

[14] M. Kurowski, "Dying Lights zombies and HTN planning in open worlds." [Online]. Available: http://aigamedev.com/premium/interview/dying-light/

[15] T. Humphreys, "Planning for the Fall of Cybertron: AI in Transformers." [Online]. Available: http://aigamedev.com/premium/interview/planning-transformers/

[16] S. Ontanón and M. Buro, "Adversarial hierarchical-task network planning for complex real-time games," in *Proceedings of the 24th International Conference on Artificial Intelligence*. AAAI Press, 2015, pp. 1652–1658.

[17] D. Nau, "Game applications of HTN planning with state variables," in *Planning in Games: Papers from the ICAPS Workshop*, 2013.

[18] D. Nau, Y. Cao, A. Lotem, and H. Munoz-Avila, "SHOP: Simple hierarchical ordered planner," in *Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2*. Morgan Kaufmann Publishers Inc., 1999, pp. 968–973.

[19] D. S. Nau, T.-C. Au, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman, "SHOP2: An HTN planning system," *J. Artif. Intell. Res.(JAIR)*, vol. 20, pp. 379–404, 2003.

[20] D. Nau, H. Munoz-Avila, Y. Cao, A. Lotem, and S. Mitchell, "Total-order planning with partially ordered subtasks," in *IJCAI*, vol. 1, 2001, pp. 425–430.

[21] M. Cavazza, "AI in computer games: Survey and perspectives," *Virtual Reality*, vol. 5, no. 4, pp. 223–235, 2000.

[22] D. S. Nau, "Current trends in automated planning," *AI magazine*, vol. 28, no. 4, p. 43, 2007.

[23] H. H. Zhuo, H. Muñoz-Avila, and Q. Yang, "Learning hierarchical task network domains from partially observed plan traces," *Artificial intelligence*, vol. 212, pp. 134–157, 2014.

[24] O. Ilghami and D. S. Nau, "Camel: Learning method preconditions for htn planning," 2002.

[25] C. Hogg and U. Kuter, "Learning methods to generate good plans: Integrating htn learning and reinforcement learning." 2010.