Offline Planning with Hierarchical Task Networks in Video Games

John-Paul Kelly
Department of Engineering
Australian National University
Canberra, ACT

Adi Botea
NICTA and
Australian National University
Canberra, ACT

Sven Koenig
Computer Science Department
University of Southern California
Los Angeles, CA

Abstract

Artificial intelligence (AI) technology can have a dramatic impact on the quality of video games. AI planning techniques are useful in a wide range of game components, including modules that control the behavior of fully autonomous units. However, planning is computationally expensive, and the CPU and memory resources available to game AI modules at runtime are scarce. Offline planning can be a good strategy to avoid runtime performance bottlenecks.

In this work, we apply hierarchical task network (HTN) planning to video games. We describe a system that computes plans offline and then represents them as game scripts. This can be seen as a form of generating game scripts automatically, replacing the traditional approach of composing them by hand. We apply our ideas to the commercial game THE ELDER SCROLLS IV: OBLIVION, with encouraging results. Our system generates scripts automatically at a level of complexity that would require a great human effort to achieve.

Introduction

Artificial intelligence (AI) can have a dramatic impact on the quality of video games. Non-playing characters (NPCs) are examples of entities that are under the control of the computer. Traditionally, NPCs were used with the only goal to populate the background of game worlds, and their behavior was limited to basic animations, with little or no interaction with the rest of the game world. In contrast, intelligent NPCs should act according to meaningful plans and interact with each other. AI techniques, such as search and planning, can make the behavior of NPCs look intelligent. However, planning is computationally expensive. Plans have to be available in real time, and the CPU and memory resources available to game AI modules at runtime are limited. A typical approach for achieving intelligent NPC behavior is therefore to use scripts, that is, plans that are composed offline by hand and cached to be used at runtime. A significant advantage of scripts is that they can then be used at runtime with little CPU and memory overhead. However, many scripts need to be generated by hand to cover a reasonable range of situations but they require lots of human effort to generate and the process is error-prone.

Copyright © 2008, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

We address this issue by presenting a system that uses an offline planning approach for the automated generation of scripts. Our system guides planning with hierarchical task networks (HTNs) (Sacerdoti 1975), hand-coded structures that encode knowledge about the domain. While scripts are plans, HTNs correspond to many plans because they abstract details of specific planning problems away. Planning takes as input an HTN and information about the specific planning problem and outputs a plan. Designing either scripts or HTNs can be a time-consuming task. Fortunately, HTNs can be reused to generate many scripts within one game and potentially also across games. Our system can be integrated into games that implement scripting functionality. Our system converts plans from the plan-representation language, such as PDDL, into the scripting language accepted by the game. The resulting scripts are then handled by the game engine just like regular scripts. Our system currently handles only linear scripts (as opposed to conditional tree-shaped scripts) due in part to the availability of more advanced planning technology for sequential planning as compared to contingency planning. We apply our ideas to Bethesda Softworks' THE ELDER SCROLLS IV: OBLIVION, a popular commercial game, with encouraging results. Our system generates scripts automatically at a level of complexity that would require a great human effort to achieve.

Related Work

Scripts and finite state machines are traditional approaches for controlling the behavior of NPCs in video games. Finite state machines have been used from first-person shooters, such as ID Software's QUAKE series, to real-time strategy games, such as Blizzard Software's WARCRAFT III. Some recent games, such as Bungie's HALO 2, use hierarchical finite state machines (Isla 2005). However, the complexity of using finite state machines grows quickly when the behavior of NPCs and the world become more complicated or the number of NPCs grows. Their use is therefore considered problematic in current generation games (Orkin 2003). Scripts have been used from role-playing games, such as Bioware's NEVERWINTER NIGHTS, to first-person shooters, such as Epic Games' UNREAL TOURNAMENT. Scripts are composed offline in a high-level game-specific language and have similar properties as finite state machines. In particular, the complexity of using scripts grows quickly when the behavior of NPCs and the world become more complicated or the number of NPCs grows, which makes it attractive to generate them semi-automatically. SCRIPTEASE, for example, aims at simplifying the process of writing scripts through pattern templates (McNaughton *et al.* 2004). It allows game developers to create relatively complicated behavior via a graphical user interface (GUI), without having to program them explicitly. The game developers must still manually choose for each NPC which behavior to initiate and when to initiate them, which means that the complexity of using SCRIPTEASE still grows quickly when the behavior of NPCs and the world become more complicated or the number of NPCs grows.

AI planning techniques have recently been used in games to control NPC opponents in first-person shooters, such as Epic Games' UNREAL TOURNAMENT or Monolith's F.E.A.R. Simple behavior (which does not need planning) was acceptable in early games, such as moving between ammunition caches and chasing all encountered opponents. More complicated behavior (which needs planning) is expected in modern games, such as moving as coordinated squads and performing tactics (including flanking or sending for backup) (Orkin 2006). While on-line planning might be able to handle dynamic environments better than off-line planning, it suffers from the problem that planning is computationally expensive. Plans have to be available in real time, and the CPU and memory resources available to game AI modules at runtime are limited. Off-line planning generates plans off-line and then uses them at runtime with little CPU and memory overhead. It is easier to implement than online planning and requires no changes to the game software. Off-line plans are typically generated by hand and then represented as game scripts. For example, an NPC might go every day for lunch at 12pm and then work until 4pm, resulting in realistic behavior. However, many scripts need to be generated by hand to cover a reasonable range of situations, they require lots of human effort to generate, the process is error-prone, and it is difficult to integrate new NPCs into a game. For example, if an NPC needs to buy merchandise from a store, then the NPC owner of the store needs to be there. Thus, game designers need to study the scripts for all NPCs to make sure that a change does not cause conflicts. The complexity of manual planning therefore grows quickly when the behavior of NPCs and the world become more complicated or the number of NPCs grows. We address this issue in the following by describing a system that uses SHOP2 (Nau *et al.* 2003), a modern HTN planner, to automatically compute plans offline and then represents them as game scripts. HTN planners were devised in the 1970s (Sacerdoti 1975) and have been advocated as knowledge-based planning approach for complex applications (Wilkins and desJardins 2001), such as production line scheduling (Wilkins 1988) and the game of bridge (Smith *et al.* 1998).

Our Testbed

We use Bethesda Softworks' THE ELDER SCROLLS IV: OBLIVION, a role-playing game, as testbed for our system. In this game, each NPC can be assigned a set of AI packages and a script. AI packages implement atomic behavior, such as eating, working and sleeping, and have activation preconditions that can include time ranges. For example, a sleep package could be active from 10pm to 8am. Setting the activation preconditions of appropriate AI packages corresponds to manual planning and does not require scripts, which are repeatedly executed in a loop and mostly used for dialogue and quest-related activities.

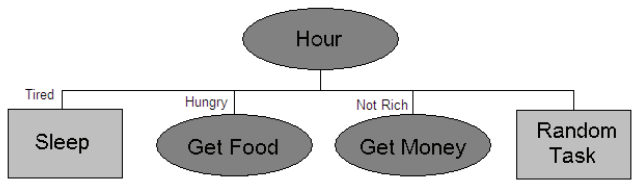
Our System

In our system, the game developers design an HTN that encodes knowledge of the game world, create a set of AI packages for all actions of the HTN, and specify several planning problems together with their initial world states. Each planning problem addresses the daily planning objectives of all NPCs. The planner is run on each planning problem. It takes as input the HTN, the initial world state and the planning problem and outputs a plan for all NPCs. The planner cannot plan for each NPC separately since NPCs can interact. The script generator converts the plan, whose actions correspond to AI packages, into a script for each NPC. Thus, we extend the use of scripts to encode daily plans of NPCs. The resulting scripts are then handled by the game engine just like regular scripts. We now describe the various components in turn.

The Hierarchical Task Network

The planning problem is encoded as an abstract task in HTN planning. Plans are action sequences that are refinements of the abstract task. HTN methods encode how abstract tasks (ovals) can be repeatedly refined into subtasks and eventually into actions (rectangles). When an abstract task is refined, the available methods are tried from left to right until a method is found whose preconditions match the current world state. A world state contains information such as a list of all locations, a list of all stores, a list of items traded in each store, the owner of each store, the number of deer that populate the game, and the NPC state of each NPC. An NPC state contains information such as the wealth, hunger and tiredness level of the NPC, its current activity and location, its inventory, its house, and a list of actions that it can perform.

Part of our HTN for THE ELDER SCROLLS IV: OBLIVION is illustrated in Figure 1. It encodes plans for a game day, divided into 24 game hours. If desired, plans that cover several game days can be obtained by chaining plans together that correspond to one game day each, by enforcing that the final world state of a plan is equal to the initial world state of the next plan. The picture on top illustrates four alternative methods that refine the abstract task of spending one game hour. If the preconditions of the first three methods are not satisfied, then an action is chosen at random, to avoid NPCs being idle. The picture at the bottom illustrates four alternative methods that refine the abstract task of obtaining food, namely directly by hunting or shopping or indirectly by learning how to hunt or making money, which will help one to obtain food in the future. The precondition of shopping is that the shop is open. After the NPC has waited



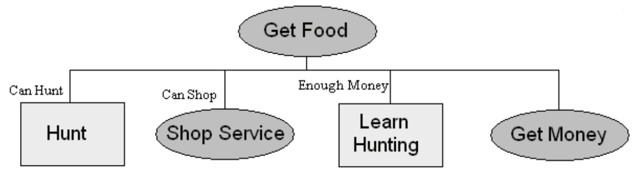


Figure 1: Part of the HTN.

for a sufficiently long time at the store, the precondition is considered to be unsatisfied and the next method is tried.

The Planner

We use JSHOP2 (Nau *et al.* 2003), an off-the-shelf Java implementation of SHOP2 that is publicly available and can handle numerical variables, which is important to model aspects of world states such as hunger levels.

The Script Generator

The script generator converts the linear plans (action sequences) from the plan-representation language PDDL into the scripting language used by THE ELDER SCROLLS IV: OBLIVION. Scripts check that the initial world state of a game day matches the initial world state of their plan. Scripts activate AI packages to implement actions. The AI packages thus have no preconditions. Each script uses conditional statements to ensure that the actions of the NPC are performed to completion, in the correct game hour and in the correct order since the game repeatedly executes each script in a loop. These conditional statements use variables that keep track of the number of actions performed during the current game hour as well as the last game hour that had all of its actions completed. Each script uses flag-based message passing to encode the exchanges of goods that occur between NPCs. The script of an NPC sends a request message to the script of another NPC to start an exchange. The script of the second NPC performs its part of the transaction and then sends a confirmation message. The script of the first NPC then performs its part of the transaction.

Example

We use a small example with three NPCs named Bob, Stan and Fred to illustrate parts of our system. Stan and Fred own one store each and thus interact with Bob via the exchanges of goods. For simplicity, we focus our attention on Bob for a few game hours and do not present all details.

```
(cost food 5.0)                  (cost skins -4.0)
(cost goods -50.0)               (cost learn-hunt 30.0)
(cost rent 1.0)                  (shop skin-shop skins Stan)
(shop skin-shop goods Stan)      (shop food-shop food Fred)
(place Bobs-house)               (place forest)
(place skin-shop)                (place food-shop)
(deer 4.0)
(asleep Bob)                     (money Bob 7.0)
(hungry Bob 7.0)                 (sleepy Bob 4.0)
(have-goods Bob)                 (at Bob Bobs-house)
(lives Bob Bobs-house)
(available-task Bob purchase-food sedate-hunger)
(available-task Bob wander wander)
(available-task Bob skin get-money)
(available-task Bob sleep sleep)
(available-task Bob sell-goods get-money)
```

Figure 2: Part of the initial world state.

```
(!sleep Bob)
(!increment-person Bob)
(!wake Bob)
(!move Bob food-shop)
(!wait Bob food-shop)
(!increment-person Bob)
(!transaction Fred Bob Food)
(!eat Bob)
(!increment-person Bob)
(!move Bob forest)
(!skin Bob)
(!move Bob skin-shop)
(!transaction Stan Bob Skins)
```

Figure 3: Part of a plan.

```
if (theTime >= 10 && theTime < 11 && finished != 10)
  if current == 0
     AddScriptPackage TravelDeerForest
     set current to current + 1
  if current == 1
     if bobVar != 60
        set bobVar to 60
        set skinning to 1
        set skinDeer to 0
        AddScriptPackage Skin
  if current == 2
     AddScriptPackage TravelSkinShop
     set current to current + 1
  if current == 3
     if bobVar != 70
        set bobVar to 70
        set selling-skin to 1
        AddScriptPackage SellSkins
  if current == 4
     set current to 0
     set finished to 10
     set bobVar to 0
```

Figure 4: Part of a script.

Figure 2 shows part of the initial world state but skips the NPC states of Fred and Stan. The planning problem is *((day Fred Stan Bob))*. Figure 3 shows a few steps of the resulting plan for all three NPCs over a game day but skips actions that do not involve Bob. Bob sleeps, then wakes up and goes to buy food from Fred's store. (He can satisfy his hunger this way since he has money.) Bob waits for the store to open. Fred shows up before Bob gives up, allowing Bob to buy the food and eat. Bob walks to forest to skin a deer and then goes to sell its pelt at Stan's store (to make some money). The repeated call *!increment-person* makes Bob more tired and hungry over time.

Figure 4 shows part of the script that corresponds to the last four actions of the plan in Figure 3 although a detailed explanation of the code is beyond the scope of this paper. The first line ensures that each action of the plan is executed at the appropriate time. The variable *current* ensures that the actions are performed in the correct order by keeping track of how many actions have been completed in the current game hour. The variable *bobVar* ensures that each action is performed only once (although we skip its implementation details). The AI package *Skin* implements the action of killing and skinning a deer. The script of Bob sets the variable *skinning* to one. The script of the deer makes a pelt appear and increments the variable *current*, which allows Bob to execute the next action. Bob cannot increment the variable himself since he does not know exactly when his action completes. The AI package *SellSkins* initiates the sale of the pelt. The script of Bob sets the variable *selling-skin* to one. The script of Stan then performs the transaction and increments the variable *current*, which allows Bob to execute the next action. The last section of the script ensures that Bob does not execute the actions again once he has completed them.

Discussion

We now discuss the advantages and disadvantages of our system.

First, automated planning has advantages over manual planning. We have already argued that the complexity of manual planning grows quickly when the behavior of NPCs and the world become more complicated or the number of NPCs grows. Automated planning increases the game development speed and thus allows our system to compute more complex plans, which cover longer game periods and have more interactions between NPCs, resulting in a better gaming experience. Automated planning requires game developers to encode knowledge of the game worlds, in our case in form of HTNs. This should not be a problem for game developers since HTNs are similar to scripts. It is an open question whether game users, a community which is very large and heterogeneous in terms of programming skills, would welcome our system to add user-created content to off-the-shelf games. Scripting has gotten end users accustomed to using structured languages, and HTNs are not harder to understand than current sophisticated forms of scripts, such as conditional scripts. If desired, GUIs could be designed to help game developers and end users generate HTNs. Eventually, a standardized system similar to ours might get used across games, similar to current graphics engines, networking code or physics simulators.

Second, offline automated planning has advantages over online automated planning. We have already argued that online planning is computationally expensive, and the CPU and memory resources available to game AI modules at runtime are scarce because sophisticated graphics and sound algorithms need them. Offline planning avoids this overhead and thus allows our system to compute more complex plans, which cover longer game periods and have more interactions between NPCs. Offline planning is less versatile than online planning since offline plans need to account for many contingencies and could thus be both difficult to compute and of large size in game worlds with incomplete or uncertain information (such as dynamic or partially observable domains). In contrast, online planning can make assumptions and replan when the current plan can no longer be executed. Our game world, however, can be modeled as completely observable and static by giving up only a small amount of realism, since the human player is the only unknown. For example, one can put virtual laws in place that prevent the human player from interfering with the NPCs to make their plans unexecutable, for example, by preventing the human player from killing NPCs that are vital for plan execution. Thus, offline planning is often sufficiently powerful, which is one of the main reasons why scripting, a form of offline planning traditionally performed by hand, is so popular in modern games.

Third, interfacing an offline automated planner via a scripting interface to a game has advantages. We have already argued that this makes offline planning easier to integrate into games than online planning since it requires no changes to the game software. This is an important issue in an industry with tight deadlines, where game companies tend to ignore new ideas unless they can easily be added to the game currently under development (Hopson 2006).

Comparison with SCRIPTEASE

To the best of our knowledge, the work on SCRIPTEASE (McNaughton *et al.* 2004) is the only research on automating script generation previously reported in the games literature. SCRIPTEASE is a tool that allows a user to generate scripting code through a graphical front end. Scripting can be performed even by users with no programming skills. The process is faster and safer, since manual coding introduces bugs that can be hard to fix.

As an important difference from our work, SCRIPTEASE makes no attempt at automating the planning side of scripting. The user is responsible to select all actions that compose a script, possibly starting from a predefined pattern. Arguably, SCRIPTEASE could be seen as focused on the software engineering side of scripting rather than the AI side. The authors state that AI capabilities, such as learning, would be a good addition to the SCRIPTEASE tool (McNaughton *et al.* 2004). In addition, we believe that our work and SCRIPTEASE can be combined into a tool that would provide access to a planning engine through a sophisticated and intuitive GUI that would not require users to have deep planning knowledge. The GUI and all the machinery behind

NPCs	Time (sec.)	Nodes	Plan Length
3	6.12	1108	293
4	6.49	1452	381
5	7.07	1798	469
6	7.56	2142	557
7	7.94	2486	645
15	10.83	6332	1351
20	12.47	8062	1797
40	28.57	14982	3566

Table 1: Summary of results for 24 game hours when the problem size varies from 3 to 40 NPCs.

Game Hours	Time (sec.)	Nodes	Plan Length
4	12.31	2634	700
8	15.80	4570	1416
12	17.49	6566	2134
16	20.45	9086	2857
20	24.01	11126	3591
24	25.32	13614	4308

Table 2: Summary of results for 40 NPCs when the covered time period varies from 4 to 24 game hours.

it could be used to design HTNs and express the initial world state and the abstract task to be refined into a plan. Based on such input data, the planning engine could compute plans and translate them to the scripting language.

Evaluation

We ran two experiments with the HTN described earlier, which consists of 12 abstract tasks, 19 actions and 40 methods and requires 180 lines of code. The initial world state for 3 NPCs over 24 game hours uses 49 boolean atoms.

In our first experiment, we varied the problem size by increasing the number of NPCs from 3 to 40. Each plan covered a period of 24 game hours. In the second experiment, we varied the covered time period by increasing the number of game hours from 4 to 24. Each plan was for 40 NPCs. The two experiments were run on a PC with an AMD Athlon 64 X2 4200+ processor and 1 GB RAM. The initial NPC states were generated randomly. Tables 1 and 2 summarize the results of both experiments. They show the planning effort (measured in both CPU time and number of nodes) and the length of the produced plans.

The data in Tables 1 and 2 show that the planning effort grows at most linearly in both the number of NPCs and the number of game hours, which is evidence that the HTN guides planning well. Note that the numbers reported in the last row of each table are different, even though the number of NPCs and the number of game hours are the same in each case. This is because the problem sets for the two experiments were generated independently of each other. The NPCs tend to sleep slightly more in the plans computed in the first experiment, and agents that are asleep execute only one sleep action during several game hours while agents that are awake typically execute several actions during each game hour. As a result, plans where NPCs sleep more tend to be shorter.

The size of each script for 40 NPCs varies between 80 and 500 lines when the number of game hours varies from 4 to 24. The plans are quite long but contain adjustment steps for state variables that can be compiled out by the script generator. For example, a script does not need to represent how hungry an NPC is. It needs to execute only the eating action. The runtime of the script generator is less than 30 seconds for 40 NPCs over 24 game hours. Manual planning and writing scripts of 500 lines for all 40 NPCs, in comparison, would be extremely tedious and time-consuming.

We have observed that automated planning makes the NPCs behave more realistically because there is more variation in their daily routine and both more and more meaningful interactions among them. Manual planning, for example, makes it attractive for the NPC owner of a store to keep its store open exactly from 9am to 5pm each day since such rigid rules make manual planning easier. Automated planning, in contrast, makes it easy for the NPC owner of a store to leave early if it is tired. Even though this action can influence the actions of other NPCs in complicated ways, the planner can compute a plan that will handle all details correctly. In addition, automated planning chains the actions of NPCs in meaningful ways. For example, an NPC attempts to buy food only if it has enough money available. In contrast, NPCs in the original game get food regardless of the amount of money they have.

Adding a new NPC to a game requires only slight modifications to the HTN and the initial world state. For example, one needs to add the NPC state for the new NPC to the initial world state. Similarly, changing the behavior of an NPC requires only slight modifications to the initial world state. For example, one needs to change the list of actions that the NPC can perform in its NPC state. In our experience, the editing takes only a couple of minutes in both cases before one can run the planner and script generator again.

Conclusion

Planning is a powerful but potentially resource-intensive technology to add intelligent behavior to NPCs in video games. In this paper, we have introduced an approach to offline HTN planning in the domain of video games. Our implementation generates scripts automatically, replacing the traditional approach of creating scripts manually. We applied our ideas to THE ELDER SCROLLS IV: OBLIVION, a popular role-playing game from Bethesda Softworks. In experiments, scripts are generated at a level of complexity that would require a great human effort to compose and debug. Future work includes applying offline conformant planning and offline contingent planning to games.

Acknowledgments

John-Paul Kelly was a summer student when this research was performed, and Sven Koenig was on sabbatical at NICTA. NICTA is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Research Centre of Excellence programs.

References

R. P. Goldman and M. S. Boddy. Expressive Planning and Explicit Knowledge. In *Proceedings of AIPS-96*, pages 110–117, 1996.

J. Hopson. We're Not Listening: An Open Letter to Academic Game Researchers, 2006. `gamasutra.com/features/20061110/hopson_01.html`.

D. Isla. Handling Complexity in the Halo 2 AI. In *Proceedings of GDC-05*, 2005.

M. McNaughton, M. Cutumisu, D. Szafron, J. Schaeffer, J. Redford, and D. Parker. ScriptEase: Generative Design Patterns for Computer Role-Playing Games. In *Proceedings of IEEE ASE-04*, pages 88–99, 2004.

D. Nau, T. Au, O. Ilghami, U. Kuter, J. Murdock, D. Wu, and F. Yaman. SHOP2: An HTN Planning System. *JAIR*, 20:379–404, 2003.

J. Orkin. Constraining Autonomous Character Behavior with Human Concepts. In *AI Game Programming Wisdom 2*, pages 189–198, 2003.

J. Orkin. Three States and a Plan: The A.I. of F.E.A.R. In *Proceedings of GDC-06*, 2006.

M. Peot and D. Smith. Conditional Nonlinear Planning. In *Proceedings of AIPS-92*, pages 189–197, 1992.

E. Sacerdoti. The Nonlinear Nature of Plans. In *Proceedings of IJCAI-75*, pages 206–214, 1975.

S. Smith, D. Nau, and T. Throop. Computer Bridge: A Big Win for AI Planning. *AI Magazine*, 19(2):93–105, 1998.

D. Wilkins and M. desJardins. A Call for Knowledge-Based Planning. *AI Magazine*, 22(1):99–115, 2001.

D. Wilkins. *Practical Planning: Extending the Classical Planning Paradigm*. Morgan Kauffman, 1988.