University of California

Los Angeles

Creating a Cognitive Agent in a Virtual World: Planning, Navigation, and Natural Language Generation

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

William Redington Hewlett II

2013

Abstract of the Dissertation

Creating a Cognitive Agent in a Virtual World: Planning, Navigation, and Natural Language Generation

by

William Redington Hewlett II

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2013

Professor Demetri Terzopoulos, Chair

Creating believable virtual humans for use in interactive video games and other computer graphics applications is a serious challenge. Much research has focused on how to create human models that exhibit realistic appearance and movement. This dissertation investigates how to create virtual humans that *act* like real people. In particular, we develop human agents that make plans, navigate through complex environments, and communicate with one another. Despite their autonomous behavior, our agents can be tightly controlled by content designers who wish to script their virtual world behavior. Many virtual environments, particularly those used in interactive games, have tight restrictions on memory and frame rate, and we show how judicious offline computation can yield significant runtime performance gains. We demonstrate a virtual world with agents that can plan, navigate, and communicate in English.

The dissertation of William Redington Hewlett II is approved.

Alphonso Cardenas

Glenn Reinman

Eddo Stern

Demetri Terzopoulos, Committee Chair

University of California, Los Angeles

2013

To my wife Kimberly

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my advisor Professor Demetri Terzopoulos, who proposed a vision of this work when I first visited UCLA. I would also like to acknowledge my committee, Professors Glenn Reinman, Alphonso Cardenas, and Eddo Stern. Many other UCLA professors helped me on my journey, but I am particularly indebted to Professors Petros Faloutsos, Richard Korf and Adam Meyerson.

I want to thank my close friends, coauthors, and coworkers in the Magix lab: Brian Allen, Kresimir Petrinic, Gabriele Nataneli, Shawn Singh and especially Wenjia Huang for helping produce the simulations using her virtual human animation system. Special thanks to Indiana University Professor Randall Beer, who allowed me to work in his lab in Indiana for two years while my wife completed her MBA degree, and Zach Haga, a fellow researcher from Professor Beer's lab. I would also like to thank Intel Corp., Microsoft Corp., and AMD/ATI Corp. for their generous support through equipment and software grants.

Finally, I want to thank my parents, who encouraged me on the way, my children Will and Guinevere, both of whom were born during my studies, and most of all my wife Kimberly.

Vita

1976	Born, Palo Alto, California, USA.
1995–2001	B.S. with Honors (Symbolic Systems), Stanford University M.S. (Computer Science), Stanford University
2001–2003	Software Developer, The 3DO Company
2003	Software Developer, Pirate Games
2003–2006	Software Developer, Electronic Arts
2007	Summer Intern, SRI International
2009	Summer Intern, Blizzard Entertainment
2012	Summer Intern, Google Inc.

Publications

William Hewlett. Partially Precomputed A*, IEEE Transactions on Computational Intelligence and AI in Games, June 2011, Volume 3(2), pages 119-128

Mubbasir Kapadia, Shawn Singh, William Hewlett, and Petros Faloutsos. Egocentric Affordance Fields in Pedestrian Steering. In Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games, I3D 2009, pages 215 to 223, New York, NY, USA, 2009. ACM.

Shawn Singh, Mubbasir Kapadia, William Hewlett, Glenn Reinmann, and Petros Faloutsos. A Modular Framework for Adaptive Agent-Based Steering. In Proceedings of the 2011 Symposium on Interactive 3D Graphics and Games, I3D 2011. pages 1 to 9, New York, NY, USA, 2011. ACM..

Mubbasir Kapadia, Shawn Singh, William Hewlett, Glenn Reinman, and Petros Faloutsos. Parallelized Egocentric Fields for Autonomous Navigation. The Visual Computer, pages 1 to 19, January 2012, Volume 28(12), pages 1209-1227

Willaim Hewlett, Michael Freed. Automatic Response Composition, Workshop on Enhanced Messaging, AAAI, July 2008

CHAPTER 1

Introduction

The goal of virtual worlds, such as interactive video game worlds, is to create animated scenarios that are realistic while conforming to the visions of their content creators. To this end, impressive work has been done, particularly involving graphics, animation, and physics simulation, but the realism of animated characters in virtual worlds has lagged behind. Animated agents in modern virtual worlds such as games are typically controlled by finite state machines or rule-based systems. These systems are simpler for programmers to create, but they often require that content creators script every possible situation ahead of time, which is work intensive and usually cannot adequately represent every situation of interest.

Speech in video games and other virtual worlds is even more scripted. While exceptions exist (such as Facade (Mateas and Stern, 2004)), the vast majority of video games use prerecorded speech exclusively. Some recorded content can be used multiple times, such as simple commands and exclamations, but most audio clips are used only once. As modern video games become bigger and more complex, the amount of scripted audio is rising quickly, but it still lacks adequate coverage.

This dissertation presents a system for creating artificial intelligent agents in virtual worlds. The agents have two primary cognitive abilities, the ability to plan how to meet their goals, and the ability to communicate to other agents using natural English. Planning and speech allow our agents to act realistically in

situations unanticipated by the content authors, while still allowing control of the agents in predicted situations. Shao and Terzopoulos (2007) develop autonomous pedestrians by breaking down the control of an animated intelligent agent into multiple levels, including cognitive, behavioral, and motor control levels. This thesis focuses on the cognitive model of an autonomous agent, which is both the highest level of control and the least developed.

It is important to limit the scope of our work in order to make it tractable. Our system does not create a completely intelligent "agent". In particular, agents are not able to communicate in English with humans in the real world. The planning and speech of each agent is autonomous; agents do not see the plans of other agents and conversations are generated on an agent-by-agent basis rather than planning all conversational roles using a conversation manager. Furthermore, agents do not understand the English language generated by other agents; they communicate via normal programming channels of messaging and function calls, rather than interpreting the English that they are receiving.

Agents in our system are for the most part "embodied"; each agent does its own planning and speech generation, but they are not "embodied conversational agents" because they do not communicate with each other through human speech or with human non-verbal messages. For example, if Agent Bob knows that an airplane flight is canceled, then Agent Alice will not get this information until she approaches and converses with him, but while the conversation between Agents Bob and Alice appears to be in English, their "speech" actually represents computer messages sent between them at the same time as their conversation is generated.

The vast majority of work in natural language processing and natural language generation is focused on enabling computer-human discourse; communication in English between computer controlled agents is not a well-studied problem. Computer-computer discourse avoids many of the problems that plague natural

language processing, such as interpreting the wide range of utterances that a human can make. While creating a machine that can talk with a human is one of the most important problems of artificial intelligence, creating agents that can talk to each other in English would be an important breakthrough for simulations such as virtual worlds and games. Recently, a few other researchers have become interested in conversation between computer agents, for example Hernault et al. (2008) describes a system where agents discuss a newspaper article.

This project caters to three important groups, the system creators, the content authors, and the users. The system creators construct the overall system including modules such as a planning module or a natural language module, which can be used to create many kinds of specific agents, while the content authors tailor agents in a specific world with specific constraints and rules. For example, the system creators create a planning system which allows agents to plan from their current state to a goal state, and a natural language system that gives agents the ability to communicate with one another in natural English. The content authors could use the system to create an airport with agents that play the roles of tourists, business people, and security officers. On the planning side, the content authors choose the agents, the goals of the agents, the actions that the agents can take, and the effect of those actions. On the natural language side, the content authors create sets of sentences by specifying a few general concepts, and link together these statements with conversational state machines. The users interact with and experience the system once it has been built.

The remainder of this dissertation is organized as follows: There are four additional chapters followed by our conclusion. In Chapter 2, we describe how our virtual agents plan and utilize a system of state machines. In Chapter 3, we show how the agents can speak to each other in English given some direction from content creators. Chapter 4 describes a low level path finding algorithm that scales well to large numbers of agents and huge worlds. In Chapter 5, we present

simulations that incorporate some of the previously developed techniques.

CHAPTER 2

Control of Artificial Agents: Planning and State Machines

2.1 Overview

2.1.1 Why Plan?

The most common technique for controlling artificial characters in commercial virtual worlds is finite state machines (FSM). There are a number of reasons for this. First of all, FSMs are easy for system creators to implement. They have low performance costs and are the simplest agent control structure for content creators. The problem with state machines is that every possible situation to which an artificial agent must react must be represented explicitly in the state machine.

The goal of this thesis research is to maximize the realism and coverage of the artificial intelligence while minimizing the work of the content authors. A content author could write out exactly what a particular agent in a particular situation should say or do, and this would provide maximum realism in that situation, but that content would not necessarily be applicable by other agents or in other situations. This specificity is known as lack of coverage. If the content author did this for every situation and every agent, the amount of work involved would be enormous. While techniques such as finite state machines and rule-based systems can lower the authoring burden, Orkin (2004) argues that planning systems are

much more effective at reducing this burden in the game development cycle.

The virtual world system described in this dissertation uses state machines extensively, but it also incorporates a planning system for higher-level processes. One of the traditional weaknesses of planning systems in virtual worlds is that they have expensive performance costs. Our system takes a novel approach to minimizing performance costs while still providing the flexibility of a planning system.

2.1.2 Planning in a Finite Space

The vast majority of planning algorithms work in infinite abstract spaces. For example, imagine an agent that can move one unit in any of the four cardinal directions on a two dimensional plane. Assuming the agent's state is an (X, Y) coordinate, we can easily describe the actions as changes in the state; for example, MoveNorth $= (X, Y+1)$. Both the start point and end point in this system would be a (X, Y) coordinate, and any planner could solve for a plan in this space.

Now imagine that instead of an infinite plane, the plane is only 1000×1000, with $1,000,000$ nodes. The actions have to change slightly (as you cannot go past 1000 in either direction), but its easy to map actions from the infinite domain to the finite one. Of course, this new domain cannot handle searches between nodes that are more than 1000 nodes apart in one direction, but it is difficult for both computers and humans to manage search depths greater than 1000 anyway, especially in real time. One of the keys of our planner is the fact that it works in a finite space. This allows it to create plans on very large graphs much faster than other planning algorithms, using techniques from Chapter 4.

Of course, many planning domains have more dimensions than the two in the very simple planning domain above. There are two factors that allow us to constrain our planning to a finite space. The first is the insight that most people

are not able to plan more than a few steps ahead, and the second is that virtual worlds tend to have a very limited number of actions that their computer agents can take. Most agents can move according to a recorded animation, move to a location, or pick up and use an item. Our agents can also engage in conversations, but each conversation has very specified planning input and outputs (see Chapter 3, Section 3.7). If we constrain our planning locations to specified locations rather than (X, Y) coordinates (for example "In Front of the Ticket Counter"), we remove a significant number of nodes from the planning graph. This is particularly important for interaction between agents that move, because potentially every combination of locations needs to be represented. In most video games and virtual worlds, computer agents are mostly constrained to a particular area, and only interact with other agents in that area.

A tricky problem with constraining our plans to finite size is an exponential explosion in planning states. For example, suppose there are a planning agents in the world, each of which could be in b locations, and each of which could have some set of c items. If we set up this space in a naive way, we will have$ab2^c$ possible states. Instead, it makes sense to constrain our states to those that are actually important. If we frame each planning problem by what is important to a single planning agent, we can minimize the state-space explosion. For example, most planning agents only care about a very small subset of what other agents carry, and only need to worry about where other agents are only if they are near to them.

2.1.3 Planning with Multiple Agents

Given that there are potentially many agents that want to plan at the same time, the planning system creates a single planning graph that is used by all agents. Each plan is performed with respect to the agent currently using the planning system. In some ways the planning system is similar to an emergency plan for a

building, where a person in the building might read directions that say "1) Take cover. 2) If you are a manager locate all of your staff. 3)..." Of course, each planning operation involves some search of the PPA* (Partially Precomputed A*) data structure, as opposed to a written plan. A more formal way to frame this multiagent planning is that given an agent and a goal of that agent, we first determine the state of the agent. This state corresponds to a node in a graph. The goal may correspond to a single state or a set of states. PPA* then searches the graph until it reaches one of the goal states, and the path that it finds corresponds to a plan. (See Chapter 4, Section 4.5). Often, the plan will make assumptions about states that are outside the agent's immediate surroundings. For example, an agent might assume an airplane is leaving from Terminal 1 only to arrive there and find that it is leaving from Terminal 3. Then the agent calls the planner again to replan with a different starting node. In this case, being wrong about its assumptions causes the agent to make believable mistakes.

One aspect of multiple agent planning for which our system has only minimal support is anticipation, or planning about other agents' planning. For example, in a chess game a planning agent might consider each of the possible actions of a competitor and attempt to find the best move given the options of the opponent. Similarly, we could imagine that one agent in a team might figure out what each other agent on the team will do to maximize an expected value function. While this functionality can be approximated by content designers by specifying certain goals and actions in a plan, generally agents in our system do not reason about the plans of other agents. Reasoning about other planning agents requires either an exact or approximate model of their planning system and quickly leads to a combinatorial explosion in the size of the planning state space.

2.2 Related Work

Orkin's work on the FEAR game (Orkin, 2005) is one of the first applications of planning systems in a successful virtual world. This work is interesting for virtual worlds because, unlike many planners, it runs in real time while using limited computing resources. There are many lessons to be learned from the FEAR system. For one thing, the author draws a focus to the game designers and the expressiveness of the system, which is a clear parallel to our content authors and our goal of maximizing realism and coverage. There are other techniques in the FEAR system that might be incorporated into our system. For example, to achieve real-time performance, the authors hash action symbols into a look-up table so that determining what actions are available is as fast as possible.

Agents (or "NPCs" in the game literature nomenclature) in FEAR also limit their planning by focusing on a particular individual and only updating knowledge about that individual. This is important because certain tests, such as line-of-sight checks, are very expensive to perform at run-time. For example, suppose a particular agent could talk to any agent in the simulation, but one of the pre-conditions of talking was that one had to maintain line-of-sight to the agent to whom one was speaking. Naively, one could test each agent in the system with a line-of-sight check, but that would be expensive. Instead if one keeps track of the current "target" to which an agent is speaking, one can converse with that target and only test against that target until a higher-level planner decides to switch targets. In some ways, this is similar to hierarchical planning techniques such as SHOP (Nau et al., 1999), which break down large planning tasks into several smaller planning problems, except that in this case the scope of the hierarchy is explicitly set to be interactions with the target. The open source SHOP2 (Nau et al., 2003) is the modern version of the SHOP architecture, and it is a possible alternative to finite space planning.

2.3 Input

How content creators interact with a planning system is one of the key features that distinguish planning systems from each other. In this work, the planning system has a similar authoring philosophy as the Conversation State Machines (see Chapter 3, Section 3.5). The basic idea is that there are two tiers of content creation, the plans themselves, which are created in a very simple planning language, and the implementation of the planning actions and constraints, which are programmed individually. This is similar to the Conversation State Machine sub-system, where the logic is created in a simple language, but the states and transitions are either created automatically or hand programmed. The advantages of this system is that content creators can work on content creation without being responsible for programming tasks.

In Figure 2.1 we can see a simple example of a plan for an agent that wants to travel by airplane. Each action has a set of constraints and a set of effects. The planning system performs a search of the space and arrives with the plan found in Figure 2.3. The agent travels to a ticket counter, asks which gate to go to, purchases a ticket, goes through security, waits for her plane, and finally boards the plane.

2.4 State Machines

2.4.1 State Machines and Planning

Even though planning is employed by agents in our simulation, traditional finite state machines are used as well. First of all, there are some agents that are not complex enough to require planning. For example, a ticket agent might stay behind his desk throughout the simulation and run a set of Conversation State Machines with planning agents that approach it. Second, even for the planning

Figure 2.1: Simple Action List Example

#<ActionName>

#<Constraint1> <Constraint2> ... <ConstraintN>

#<Effect>

Get_On_Plane

Have_Ticket At_Gate Correct_Time

Travel

Go_To_Gate

At_Past_Security Know_Gate

At_Gate

Wait_At_Gate

At_Gate Not_Correct_Time

Correct_Time

Ask_For_Gate

At_Ticket_Counter

Know_Gate

Go_To_Ticket_Counter

At_Main_Terminal

At_Ticket_Counter

Ask_For_Ticket

At_Ticket_Counter

Have_Ticket

Figure 2.2: Simple Action List Example (cont)

Go_To_Security

At_Main_Terminal

At_Security

Pass_Through_Security

At_Security Have_Ticket

At_Past_Security

Go_To_Main_Terminal

At_Ticket_Counter

At_Main_Terminal

Check_Gate

At_Wrong_Gate

Know_Gate

Go_To_Correct_Gate

At_Wrong_Gate Know_Gate

At_Gate

Figure 2.3: Results of Plan from Figure 2.1

State: Has_Ticket: 0, Location: 3, Time: 0, Traveled: 0, Know: 0,

State: Has_Ticket: 0, Location: 2, Time: 0, Traveled: 0, Know: 0,
Action: Go_To_Ticket_Counter

State: Has_Ticket: 0, Location: 2, Time: 0, Traveled: 0, Know: 1,
Action: Ask_For_Gate

State: Has_Ticket: 1, Location: 2, Time: 0, Traveled: 0, Know: 1,
Action: Ask_For_Ticket

State: Has_Ticket: 1, Location: 3, Time: 0, Traveled: 0, Know: 1,
Action: Go_To_Main_Terminal

State: Has_Ticket: 1, Location: 1, Time: 0, Traveled: 0, Know: 1,
Action: Go_To_Security

State: Has_Ticket: 1, Location: 4, Time: 0, Traveled: 0, Know: 1,
Action: Pass_Through_Security

State: Has_Ticket: 1, Location: 0, Time: 0, Traveled: 0, Know: 1,
Action: Go_To_Gate

State: Has_Ticket: 1, Location: 0, Time: 1, Traveled: 0, Know: 1,
Action: Wait_At_Gate

State: Has_Ticket: 1, Location: 0, Time: 1, Traveled: 1, Know: 1,
Action: Get_On_Plane

agents there are some small tasks which involve state machines rather than the planner. Each planning Action can be associated with a state machine that actually performs the action. For example, the Pass_Through_Security action from Figure 2.3 might involve a number of mundane actions such as removing shoes or taking your laptop out of its bag, but these actions do not have to be represented in the plan of that agent. It is important to note, however, that these small State Machines can end in a failure state. For example Pass_Through_Security might fail because of some unexpected or outside action, such as a security breach that shuts down a line, and in this case the State Machine would fail and the planner for this agent would be restarted with a different starting node (which involves different assumptions about the world).

2.4.2 State Machine Authoring

There are many different ways to manage state machine authoring by content authors. One strategy is to write the state machine logic in an intermediate-level computer language such as C++. If content authors also happen to be expert programmers, then this allows them the most power and flexibility, but typically content authors do not have such programming backgrounds. Another method is to have content authors script their logic with an existing high level scripting language, such as Python. This middle road requires that content authors have some programming ability, but isolates them from critical low-level code. The third option is to separate state machine authoring from programming languages by having a specialized format for state machines that does not involve programming. This thesis takes the third approach. States and Transitions in the state machine are either created automatically by the Dialog System, or specially created by C++ programmers. State Machine content authors construct the state machines out of a library of possible states and transitions, and do not have to concern themselves with programming. One of the advantages of this division of

labor is that state machines can be authored by end users without worrying that such content authoring will break the entire system. Such an approach may not work for all projects; the right approach depends significantly on the abilities of the content creation team.

2.4.3 State Machine Specifics

2.4.3.1 Beyond Finite State Machines

Our state machines are similar to traditional Finite State Machines (Harrison, 1965), with a number of modifications. First, because each state can represent a piece of code (see Subsection 2.4.2), our state machines are effectively Turing Complete (Turing, 1936). Additionally, our State Machines encode additional information in the State Machine description. For example, a traditional State Machine has only states and transitions between states. We add an "Optional Modifier" for a state, which is an argument for the code that represents that state. For example, in Figure 2.4, the Goto1 State has a modifier of "TicketCounterLocation." This means that this state should run the "Goto" state code with "TicketCounterLocation" as a parameter for that code, which allows us to have several states which reference the same code. Furthermore, we could have separate states Goto1 and Goto2 (or more generally Goto#) which utilize the Goto state code but are separate states in the state machine. This allows content authors to create states that link to these matching Goto states while keeping them separate in the state space.

2.4.3.2 State Library

There are a number of different states that content creators can use to build state machines, and some of them represent code written by project specific programmers. To create these states, these programmers use a state library. Each

programmer-created state has a name and optional initialization, update, and cleanup functions. The name of a state is used by content creators to reference the state. When a state machine transitions into a state, the state's initialization function is run, if it exists. Similarly, when the state machine transitions out of a state, the cleanup function for that state is called. For each frame, the state machine system is invoked with the time since the last frame. Then the update function for each current state is run (if it exists). Often, states will only have update functions defined, but there are circumstances where cleanup or initialization functions are required. In our system, state transitions do not have code associated with them, but it is possible to add such functionality in the future.

During system initialization, each state machine text file (which is prepared by content creators) is read in, and then the state machines are created and linked. Certain states have parameters (see Subsubsection 2.4.3.1) or child state machines (see Subsection 2.4.4), and each state created this way is a separate instance of the state code. For example, a "Goto Home" state is different from a "Goto Church" state; each has a separate copy of the a Goto "Update" function which sends a Agent that uses the state machine to a different location.

2.4.3.3 Messaging System

A number of messages need to be sent between state machines and actors. In general, messages are passed between state machines owned by the same Agent or between Agents (but not between state machines of different agents). For example, consider a conversation between two agents, Agent Alice and Agent Bob. After Alice makes a statement, the expectation is that Bob will make the next statement. Humans use verbal and non-verbal clues to handle this turn-taking behavior. In our system, this is handled by a messaging subsystem. After Alice finishes her statement, her conversational state machine will message its parent state machine (see Subsection 2.4.4) that it has finished its statement and is awaiting a response.

The parent state machine might be an animation state machine that would finish her talk animation and pass a message to Agent Alice. Then Agent Alice would send a message to Agent Bob, who would in turn message his state machines in descending order until the message was processed, and Agent Bob's state machines would begin forming his next reply.

2.4.4 Hierarchical State Machines

State Machines in our system can be hierarchical, which means that a state in a state machine can be represented by an entirely separate State Machine. For example, consider the state machine found in Figure 2.4.

StandSellTicket is a state machine which describes the behavior of a ticket seller behind an airline counter. The agent moves into the correct location, and then waits for a customer to start a conversation. Once the conversation begins, the agent invokes the SellTicket state machine, a Conversational State Machine (see Chapter 3, Section 3.5). Notice the TalkStateMachine state and the SellTicket parameter on the ninth line of text in Figure 2.4. The TalkStateMachine state is a generic state which handles the animation of a talking character while it runs a child state machine. In this example, the child state machine is the SellTicket state machine, which is the passed parameter on this line. By separating the TalkStateMachine state from the child state machine, we can reuse the TalkStateMachine for other conversations. Once the SellTicket state machine is finished, the StandSellTicket state machine does some post-talk cleanup, and then loops back to traveling to the correct location. Thus, the agent running this state machine can talk to a number of different buying agents in series. Hierarchical State Machines can be very effective because they can compartmentalize state machines implementations from each other. In this example, when the SellTicket State Machine is created, its authors can assume that the buying and selling agent are properly configured for a conversation. Furthermore, the Sell-

Figure 2.4: StandSellTicket State Machine

#<StateName>, <Optional Modifier>

#<TransitionName>, <EndState>

Start

Empty, Goto1

Goto1, TicketCounterLocation

ReachedLocation, WaitForConversation

WaitForConversation

ConversationCoupled, TalkStateMachine

TalkStateMachine, SellTicket

EndStateMachine, FinishTalkState

FinishTalkState

FinishTalk, Goto1

Ticket state machine can be reused by a different parent state machine that sets up its required conditions correctly. This hierarchical compartmentalization is very effective when there are multiple state machine authors on a team, because more general, lower-level state machines can be used as building blocks for many parent state machines, and team members can work on authoring different state machines at the same time.

2.5 Planning Graph Creation

To create a planning graph, where each node is a *State* and each transition is an *Action* that transitions between states. The user specifies a starting state. This starting state is simply a state in the graph that can be reached in a normal situation. From there, we use a breath first node traversal to create every reachable state similar to Dijkstra's algorithm (Dijkstra, 1959). As each transition of a node is expanded, we perform the *Action* associated with that transition to produce a new *State*. We check that each new State has not been created before to prevent loops, and if it is new, we add it to a queue of available states. In each iteration, we take the top state from the queue and expand it, and once the state queue is empty we have reached and created all possible states.

Because arbitrary actions and states can induce infinite graphs, we have a limit to the number of states that can be created. One thing we would like to add in the future is a state explosion debugger, which attempts to diagnose why a particular plan graph becomes too large. For example, we could count the number of times each type of action is expanded, although in some cases this may not reveal the problem. For example, suppose we add a simple +1/-1 polarity to the example in Section 2.1.2. The agent can now be at a (X, Y) location but now has the polarity $+1$ or -1, with an action to switch polarities. If we run a breadth first search over this space we will eventually reach our state limit. If we count the number

of Switch Polarity actions, we find that there are just as many as the number of Travel North actions, even though the unconstrained Travel North action is a problem and the Switch Polarity action is not.

2.6 Planning under Uncertainty

One aspect of modern planning systems that is not considered in this thesis is planning under uncertainty. While planning for conditions which might occur may be more realistic, it adds considerable computational expense to planning operations. While plans in our system are deterministic, agents do not have an omniscient view of the world. For example, an agent might plan to enter a room but upon arriving at the door to the room, find it locked. The agent made a plan that involved entering the room, but once the room is discovered by the agent to be locked, a new plan must be made (which might involve searching for a key). Unlike some planning systems that reason under uncertainty, agents in our system do not explore areas of uncertainty (for example, trying every door to see if it is locked), but plan given their current view of the world, update knowledge as circumstances permit, and re-plan when a plan is broken by new information. By ignoring uncertainty, but allowing for agents to possess incorrect knowledge, realistic simulations can be created which are computationally affordable. Furthermore, since our finite space planning system is faster than many infinite space planning systems, it is feasible to re-plan often.

CHAPTER 3

Natural Language Generation

3.1 Overview

The Natural Language Generation (NLG) system creates English sentences from input composed by content authors. For example, consider the sentence "I would like to buy a ticket to Boston." The content designer specifies the nouns {I, ticket, <destination>} and the verbs {like, buy}, and the NLG system creates a sentence from those words. The sentences are then ranked using data from a large n-gram corpus. Finally, this collection of possible sentences is one state in a set of conversational state machines. Each computer agent can employ one of these state machines in order to communicate with other agents. The state machine will proscribe when the agent should talk or listen, and after a conversation the agent may have attained certain knowledge, gotten permission, or received an item that will help it achieve one of its planning goals.

3.2 Related Work

The method of generating sentences used in our thesis is most similar to the Nitrogen system (Langkilde and Knight, 1998), which also overgenerates sentences and uses an n-gram corpus to rank them. There are two major differences between the Nitrogen system and our sentence generation system. First of all, the Nitrogen system uses a relatively small Wall Street Journal n-gram corpus, with only bi-grams and uni-grams. Second, the Nitrogen system uses a more complex input

structure where content authors list concepts in a formatted structure rather than nouns and verbs. In some ways, specifying semantic content can be more powerful, but it requires a greater amount of linguistic sophistication from content authors. One capability that Nitrogen possesses that our sentence generation does not is the ability to use word synonyms to expand the breadth of possible outputs. For example, the Nitrogren system might be given the input concept "airplane", but would also try to build sentences with synonyms of "airplane", such as "plane" or "aircraft". Even though it is easier to create synonyms with semantic concepts rather than words, expanding our system to use synonyms would increase the range of generated sentences and is something we are exploring. One technique would be to use synonym information from VerbNet (Schuler, 2005) and WordNet (Miller et al., 1990) and run all possible combinations of synonyms through surface realizer. An additional alternative would be to first use our full NLG system to find a set of grammatical sentences with the base words, and then generate synonym variations and score them with the n-grams in a second pass, to avoid a combinatorial explosion of possible sentences.

Another system that combines a surface realizer with n-gram sentence selection is the OpenCCG system (White; Baldridge and Kruijff, 2002; White and Baldridge; Hockenmaier et al., 2004). We experimented with using OpenCCG for this work but eventually choose RealPro as our surface realization system (see Section 3.4.2). OpenCCG takes a categorical grammar (Ajdukiewicz, 1935; Bar-Hillel, 1953), and grades sentences composed with the categorical grammar using an n-gram database, but writing a categorical grammar for a broad range of possible English sentences is a daunting task, and in practice RealPro had a much broader coverage of the English language out of the box. As OpenCCG becomes more widespread and more grammars get written, it may be an attractive alternative to RealPro, which is a proprietary system.

There are many natural language generation systems and it is generally dif-

ficult to compare between different ones. DeVault et al. (2008b), DeVault et al. (2008a) and Traum et al. (2003) present an interesting technique where the words in sentences from training data are matched one-to-one to semantic meanings by a content author. Then this training set is used to learn a system of constructing novel sentences from new semantic input. We attempted to recreate this system, but after consulting with the author we determined that the published work provided insufficient information for an implementation.

The language generation system developed by Chen et al. (2002) uses a combination of different techniques to perform online sentence generation. First, semantic input is fed into SPOT (Walker et al., 2001), a trainable sentence planner that composes plans of several sentences and uses training data to choose how to order and parse data between the sentences. These sentence plans are then fed through FERGUS (Bangalore and Rambow, 2000) a surface realizer that is a successor to RealPro. While FERGUS is in some ways more advanced than RealPro, it is no longer available according to the author, and also lacks the broad English coverage that RealPro enjoys. Using a sentence planner such as SPOT (Walker et al., 2001) is a possibility for our NLG system, but because our sentences are in conversations, we decided to have content authors create conversation state machines (see Section 3.5) rather than attempting to automatically generate an entire conversation from a single input. In some ways, this is more realistic since human conversations are created by two agents rather than by a single planned dialog. Nevertheless, it remains an interesting strategy for future work, since removing the burden of authoring conversational state machines might be worth the loss of realism.

Total Words	1,024,908,267,229
Total Sentences	95,119,665,584
Unique Words	13,588,391
Bi-grams	314,843,401
Trigams	977,069,902
Four-grams	1,313,818,354
Five-grams	1,176,470,663

Table 3.1: Google N-Gram Dataset: Basic Parameters

3.3 N-grams

3.3.1 N-gram Dataset

For this thesis work, we used the Google n-gram data set (Thorsten Brants, 2006). An n-gram is a sequence of n consecutive words. For example "Four score and" is a tri-gram, or three word n-gram. The Google n-gram data set is approximately all publicly available English pages on the internet of January 2006. From these web pages, 95 billion sentences were culled, with more than 1 trillion total words. Words that appeared less than 200 times are replaced with the token <UNK>, and sentence beginnings and ends are marked with the tokens <S> and </S>. Figure 3.1 shows the basic size statistics of the Google n-gram data set.

3.3.2 Preparing Data

The Google n-gram data set is too large to be used efficiently; part of the work of this thesis was to shrink the effective size to something more manageable. Additionally, the data set contains a large amount of "nonsense" words, such as misspelled words, non-English words, or machine created content. For example, consider the words before the word "after" in Table 3.2. "Aften" is a Danish and Dutch word, "aftenoon" is a misspelling of "afternoon", and "aftention" is a

Table 3.2: Google N-Gram Dataset: Words near “after”

aften	12195
aftenen	671
aftenoon	2283
aftenposten	712
aftenposten.no	536
aftention	221
afteor	379
after	277057138

misspelling of “attention.”

In order to fit n-gram data in memory, we should shrink our dictionary size to ignore useless words. Our first attempt was to pick a threshold for the number of times a word must appear to be in the dictionary. This was found to be an ineffective threshold because commonly misspelled words are so common. For example, the word “teh”, a misspelling of “the”, is found 1.2 million times, whereas the word “agony” is found 1.0 million times. Instead, we used the Scowl dictionary (Atkinson), which is a large spell check dictionary of configurable size. For the Scowl dictionary, after some experimentation we settled on the “60/100” sized dictionary, which had 118 thousand words and includes words such as “legitimatized” and “superposition” but not such words as “anticapitalist” and “magnetizer”. This specific configuration contained about 18% of the 650,000 words in the scowl lexicon (some of which did not appear in the Google lexicon). To this list we added a number of punctuation characters and additionally the special characters "<S>" and "</S>", which are used in the Google data set to signify the beginning and end of a sentence. All told, our dictionary has around 117 thousand unique words.

Given a reasonable dictionary, the next step is to create a tri-gram database that contains only those characters. Furthermore, we can shrink the size of our

Table 3.3: N-Gram Database: Compressed

	Original Bi-grams	Original Tri-grams	Compressed Bi-grams	Compressed Tri-grams
Size	4.96GB	19.0GB	1.51GB	6.56GB
Number of Files	32 files	98 files	6 files	26 files
Number of Grams	315 million	977 million	113 million	440 million
Space/Gram	16.9 bytes/gram	20.9 bytes/gram	12 bytes/gram	16 bytes/gram

n-gram database by replacing strings with the dictionary index of the word. So instead of the string "are" we can store the integer 4970, which saves some space and makes fast searching feasible.

To create the compressed N-Gram files, we first read an uncompressed file line by line and removed all capitalization. In the Google N-Gram data set, capitalized words are stored separately, but this does not work because a word might be found at the beginning of a sentence and not match the same word elsewhere in a sentence. After capitalization is removed, we check if every word in the n-gram is found in our dictionary. If it is, we temporarily add this n-gram to our database. Once all of the tri-grams or bi-grams are loaded, we then sort them and subsume duplicates. The duplicates exist because we removed the capitalization, but we add them together rather than deleting them. In the C language on Windows systems, it is difficult to read and write to files on disk that are larger than 300MB, so once all the n-grams are loaded and sorted, we write them out to a number of files. One entry for a tri-gram, for example, will be 3 integers, each of which correspond to a word, followed by the count for that tri-gram. Table 3.3 shows the size difference between the original n-grams and our compressed versions.

If our dictionary was smaller, for example less than 65 thousand unique words rather than 117 thousand unique words, and if we rounded our n-gram counts (since all but a few digits are relatively insignificant), it is conceivable that we might store each tri-gram in 16 bit shorts rather than 32 bit unsigned ints. This

would half our space/gram and the total space used to store the grams in memory. We are exploring this additional compression in hopes of adding the four-grams and five-grams to our n-gram database.

3.3.3 Real-Time Lookup

Given a set of sentences, we would like to rank them on how similar they are to our data set. To do this, we use n-gram perplexity.

$$\log_2((\prod_{i=0}^{n-2}(T_i/B_i))^{1/n}) \tag{3.1}$$

The perplexity of a sentence is given by Equation 3.1, where T_i is the tri-gram starting with word i, B_i is the bi-gram starting with word i, and n is the number of words in the sentence. For example in the sentence, "You are booked on a flight to Boston", the inner product would include C(you are booked)/C(you are), where C(you are)is the count of the number of times the bi-gram "you are" appears in the data set.

One interesting problem that occurred during implementation of the real-time lookup system was that loading the tri-gram data set from disk at startup took too long, even with the optimizations. To counteract this problem, we created a separate tri-gram "server" that rates sentences. Each client process sends a sentence to the tri-gram server, which performs the perplexity calculation above and then returns the perplexity to the client process. This compartmentalizes the n-gram process and allows debugging of the sentence generation system.

Another interesting implementation decision was the data structure used for storing and retrieving the tri-grams. Given that the basic operation is to look up a tri-gram from a large set of tri-grams, the natural data structure to use would be a hash table. While a hash table would give us an $O(1)$ lookup time (or arguably $O(k)$ where k is the size of the key), unfortunately it uses significantly more space, which is a limiting factor. It is possible that a perfect hash function could be used

especially because the tri-gram table is constructed completely offline; the best perfect hashing functions use around 2.5 bits/key (Wikipedia, 2013), or about 128MB of extra space for just the 440 million tri-grams. Instead of a hash table, we keep the tri-grams sorted and do a binary search of the data, for a $O(\ln(n))$ search time.

3.3.4 N-Gram problems

There are a few sentences that are rated incorrectly by the N-Gram model in our testing. For example, the sentences "You are booked on a flight to Boston" and "You are booked on the flight to Boston" are rated slightly lower than the ungrammatical sentence "You are booked a flight to Boston" (See Figure 3.1). The reason for this is that, in this case, the tri-grams do not span enough of the sentence for the incorrect grammar to be scored correctly. This would be ameliorated if we used four-grams and five-grams, but it is unclear how to integrate them into a perplexity model that also includes smaller sentences that might not have five-gram representations. Furthermore, there is not enough main memory to hold all five N-Gram datasets. If we knew in advance our entire lexicon and the lexicon was small enough, we could use that lexicon instead of the scowl dictionary and this would significantly lower the memory requirements, allowing us to use four-gram and five-grams, but the preprocessing time of generating usable datasets that fit in memory necessitates that this be an offline calculation.

3.4 Sentence Generation

3.4.1 Sentence Input

The Sentence Generator takes a small input of nouns and verbs for each sentence. Figure 3.2 shows a set of possible user created sentence states. There are a number

Figure 3.1: Incorrect Ngrams

You are booked on a flight to Boston

$$\frac{\text{(are booked on)[9822]}}{\text{(are booked)[113311]}} x \frac{\text{(booked on a)[17476]}}{\text{(booked on)[124857]}} x \frac{\text{(on a flight)[113187]}}{\text{(on a)[175586178]}} = 7.8x10^{-6}$$

You are booked a flight to Boston

$$\frac{\text{(are booked a)[1222]}}{\text{(are booked)[113311]}} x \frac{\text{(booked a flight)[8829]}}{\text{(booked a)[251133]}} = 3.8x10^{-5}$$

of additional modifiers that content creators can add. For example, the QUESTION modifier informs the sentence generation system that the desired sentence is a question, which is followed by an interrogative word, such as "when" or "who". Similarly, the modifier PAST before a verb informs the system that the verb in question should be in the past tense.

One of the more interesting modifiers is the VARIABLE modifier, which means that the following term should be looked up at run time and will be changed by the conversation state machine. For example, in Figure 3.2 the state WantTicketToX has a VARIABLE CityName, so that the same state can be used for a ticket to Boston or a ticket to San Francisco.

3.4.2 Real Pro Sentence Realization

Realization in Natural Language Generation is the process of turning a syntactic grammar structure into a sentence. *RealPro* (Lavoie and Rambow, 1997) is a realization software creates sentences from a deep syntactical structure in an XML file. Two interesting properties of Real Pro are that it is relatively fast at sentence generation, and that it is completely deterministic. Unfortunately, while a large percentage of English sentences can be produced with RealPro, a large percentage of RealPro inputs will not produce English sentences. By using RealPro as a piece

Figure 3.2: Sentence State Input

#<StateName>

#<verb1>, <verb2>, ... <verbN>

#<noun1>, <noun2>,... <nounN>

ConfirmFlightX

be, PAST book

you, flight, VARIABLE CityName

WhenTravel

like, travel

you QUESTION when

WantTicketToX

like, buy

I, ticket, VARIABLE CityName

LeaveAtX

like, leave

I, VARIABLE Time

Figure 3.3: RealPro Example Input

```
<dsynts>
 <dsyntnode lexeme="have" class="verb" rel="nil" question="+">
  <dsyntnode lexeme="you" pro="pro" rel="I" person="1st"/>
   <dsyntnode lexeme="ticket" class="noun" rel="II">
    <dsyntnode lexeme="your" person="2nd" pro="poss" rel="-1"/>
   </dsyntnode>
  <dsyntnode lexeme="can" rel="-1"/>
 </dsyntnode>
</dsynts>
```

in a sentence generation system, we are able to reduce our input from sets of all words in English to a relatively structured input. Figure 3.3 shows an example of an input of the RealPro system that produces the output "Can I have your ticket?" Each node in Figure 3.3 refers to a single word in the output sentence, and the attributes describe how the specified word relates to other words in the sentence. For example, the line

<dsyntnode lexeme="you" pro="pro" rel="I" person="1st"/>

refers to the word "I", which is a *1st* person *pronoun* form of the word "*you*", and is the subject of the sentence, which is why it has the relation *1* to the verb node above it. Our sentence generation system generates inputs similar to those found in Figure 3.3 to create sentences.

3.4.3 Sentence Creation

Before using RealPro, the NLG system first takes the provided nouns and verbs (see Figure 3.2) and creates a list of variations on them. Most of these variations will not be grammatical sentences. There are several variations, including noun article type, noun particle type, and the relationship between nouns and verbs.

Each of these variations is then converted into a RealPro XML representation of a sentence (see Figure 3.3). These representations are then passed through Real Pro, which creates actual sentences (most of which are ungrammatical). Some of the new sentences are repeats of previous sentences. In the previous example, the 864 possible variations of "I would like to buy a ticket to Boston" become only 306 unique sentences. The reason that these repeats exist is that there can be many RealPro inputs that create the same output sentences.

Sentence variation has two important properties. The first is that each sentence state designed by content authors should have at least one grammatical sentence. The other desired property is that there are many correct sentences with the same meaning as the original sentence. Each of these properties push us toward generating as many variations as possible. There are two problems with having a large number of generated sentences. The first is a performance issue. Each potential sentences has to be generated by RealPro (see Section 3.4.2) and checked by the n-gram sentence chooser (see Section 3.4.4). But the second, more insidious problem with having too many generated sentences is that a grammatical sentence might be generated that has a meaning different than the desired meaning. For example, the sentence "Bob hit Sam" has a significantly different meaning than the sentence "Sam hit Bob", and it would be undesirable for both sentences to be generated from the same input. The N-Gram Sentence chooser is usually not able to choose between two grammatical sentences, so there is no way to recover from this error. On the other hand, sometimes generated sentences can have similar meanings. Consider the first two sentences of Figure 3.4. Both of these sentences are grammatically correct and while they have slightly different meanings, a speaker can usually use them interchangeably without sounding incorrect.

Perplexity="-9.146734" Text="I would like to buy a ticket to boston."

Perplexity="-9.888337" Text="I would like to buy the ticket to boston."

Perplexity="-10.313545" Text="I like to buy a ticket to boston."

Perplexity="-10.688484" Text="I would like to buy ticket to boston."

Perplexity="-10.725578" Text="I would like to buy a ticket to the boston."

Figure 3.4: "I would like to buy a ticket to Boston" Perplexities

3.4.4 Sentence Choice

For each sentence input, a large number of possible sentences are generated, many of which are not grammatical. To choose a sentence, we run each of them through the n-gram ranker (see Section 3.3.3) and prioritize them. While tri-gram perplexity does a very good job of choosing which sentences are correct by ranking, it is difficult to ascertain a probability of sentence correctness from perplexity values. For example, compare Figure 3.4 which shows the perplexities of the top 5 of 306 sentences generated from a sentence state, with Figure 3.5, which is generated from a different sentence state. Ideally, the perplexities would reveal that the first sentence was used $A\%$ of the time, the second $B\%$, and the rest were of so low perplexity that we could ignore them altogether. With such a distribution, we could have a computer agent say the first sentence most of the time and the second sentence infrequently to provide variety. Unfortunately, the perplexity values do not reveal which sentences are correct and which are not. Additionally, there does not seem to be a perplexity value for which good sentences are above the perplexity and bad sentences are below, and there does not seem to be any way to match probability to perplexity. Our system uses a weighted average to pick from the first two entries for each state.

Perplexity="-11.586642" Text="You are booked a flight to boston."

Perplexity="-11.717254" Text="You are booked on a flight to boston."

Perplexity="-12.336688" Text="You are booked on the flight to boston."

Perplexity="-12.838062" Text="You are booked on flight to boston."

Perplexity="-12.961783" Text="You would be booked on a flight to boston."

Figure 3.5: "You are booked on the flight to Boston" Perplexities

3.5 Conversation State Machines

3.5.1 Conversational State Machine Example

Conversation State Machines are used to create conversations. For example, consider a conversation where a customer wants to buy a ticket from an airline agent (See Figure 3.6). In this simple state machine, the buyer starts the conversation with a salutation such as "Hello," then requests a ticket to a location, tells the selling agent what time to leave, and receives the appropriate ticket.

There are three kinds of states in a conversational state machine: dialog states, one-off states, and special states. In Figure 3.6, Salutation is a one-off state that holds certain greetings like "Hi" and "Hello". These utterances do not use the Sentence Generation module, but merely pick from a list of appropriate statements. Dialog States do use the Sentence Generation module, and reference states created in a document similar to Figure 3.2. Finally, Special States, such as Start and Finish, are state machine specific and do not involve any dialog.

3.5.2 Paired State Machines

Conversational State Machines are unusual compared to most virtual world state machines because they involve two agents simultaneously. While each agent is represented with their own state machine, and does not have information about

#<StateName>, <Optional Modifier>
#<TransitionName>, <EndState>

Start Empty, Salutation

Salutation FinishStatement, WantTicketToX

WantTicketToX WaitForResponse, LeaveAtX

LeaveAtY WaitForResponse, Finish

Finish

Figure 3.6: Simple Buy Ticket State Machine

other agents' state machines, the state machines of agents in a conversation are tightly choreographed. In most conversations we see turn taking behavior, where the first agent will say something while the second waits, and then their roles are reversed. In practice, what happens in the state machines is that one agent will be in a "listening" state until it receives a state machine message signaling it to talk. In human conversations there are many techniques for passing priority during conversational turn taking behavior, for example ending a sentence with certain changes in pitch or changes in body language, but our system does not simulate these hints.

Conversation is a relatively low priority activity which is easily interrupted. For example, if during a conversation one of the conversing agents has its bag stolen, they might abruptly end the conversation to pursue the thief or call for help. This means that any conversation state machine has to be able to handle

interrupted conversation at any point. While it is common that State Machines in virtual worlds have to deal with certain interruptions (for example being knocked over), conversation state machines have to deal with interruptions of the other conversing agent, and they may not have a list of ways that that agent can be interrupted.

Special States (see Subsection 3.5.1) can also involve non-dialog actions that have effects outside of the state machine. For example, during a Buy Ticket Transaction, the seller agent might give the buyer a ticket. The normal way to handle this type of situation would be for the Buy Ticket State Machine to handle the transfer of the ticket after the Finish state. However, the Buy Ticket State Machine might have additional dialog actions that occur after the physical transaction, so it is important for the transfer of the ticket to happen inside of the state machine rather than after it, in case later states are ignored through interruption.

3.6 Speech Synthesis

Natural Language Generation is one of many problems that need to be solved in a complete Natural Language system. Once text dialog is generated by our Natural Language Generation system, a Speech Synthesis system turns the text into audible speech. Speech Synthesis (also known as "Text-To-Speech") is a well studied problem that is not a focus of this dissertation. We use the Microsoft Speech SDK version 5.0 (SAPI 5.0) as our Text-To-Speech solution. There is only one high quality voice included in the SAPI system, so we also used MikTex 2.9 to create the male voice used in our simulations. Additionally, it is difficult to integrate SAPI 5.0, a managed C++ program, with our system, which requires the tighter memory control of unmanaged C++. In practice, when we want to create speech audio we launch a command line program in the background which

launches a separate project that generates and plays the audio. This series of systems makes it difficult to perform other desirable tasks, such as integrating lip synching animations into our Natural Language Generation system, and we are exploring other solutions to the Speech Synthesis problem.

3.7 Integration with Planning

Each Conversation State Machine (see Section 3.5) can be evoked from an *Action* in a plan. For example, an agent that wants to buy a ticket would perform the *Ask_For_Ticket* action, which would invoke the *Buy Ticket* Conversation State Machine (see Figure 3.6). In practice, there might be an intermediate state machine that would be invoked by the *Ask_For_Ticket* action, which would wait in line and move to the ticket counter before invoking the *Buy Ticket* Conversation State Machine (see Chapter 2, Section 2.4.4). Additionally, both the intermediate state machine and the Conversation State Machine might end in a failure state, which would mean that the agent would have to replan. It is important that content creators consider failures when constructing State Machines, because certain failures might be easier to handle at the State Machine level then at the Planning level. For example, suppose an agent has a plan to talk to a second agent, but the second agent is busy with a different conversation. It is more appropriate for the State Machine to handle waiting for the blocking agent to be finished instead of reporting a broken plan and forcing a replan. In this case the original plan is sufficient, the agent just needs to wait until it can continue its conversation.

CHAPTER 4

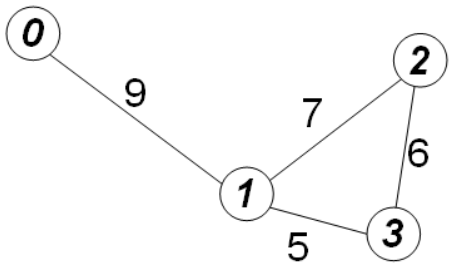
Path Finding

4.1 Overview

One of the challenges that virtual world developers face is how to program virtual characters that are capable of navigating to their goals. This problem is typically implemented by searching for a shortest path in a graph that represents the reachable terrain of a virtual world. A* (Hart et al., 1968), the computationally optimal technique for finding a shortest path, is a frequently used method for path planning in video games and other virtual worlds because of its simplicity and the size of navigational graphs. However, A* is becoming unmanageable for the extensive levels and large numbers of characters in today's video games. In this thesis we introduce *Partially Precomputed A** (Hewlett, 2011), a practical method for achieving fast real-time search times in large graphs.

The simplest precomputation strategy is to calculate every possible shortest path offline and store them in a large matrix. This technique is known as *All Pairs Shortest Paths* (APSP). If there are n nodes in the graph, then the matrix is n^2 in size, where entry i, j contains a reference to the next node after i in a shortest path from i to j (Figure 4.1). The time required to calculate the node to travel to next is $O(1)$, but the space required is $O(n^2)$. PPA* uses much less memory than a full APSP matrix, and is faster than A* at run-time.

PPA* consists of both a precomputation technique and a search technique. In the precomputation phase, an input graph is clustered into a hierarchy of



	0	1	2	3
0	0	9	16	14
1	9	0	7	5
2	16	7	0	6
3	14	5	6	0

Distance(Shortest Path(0, 2)) = 16

Figure 4.1: All Pairs Shortest Path (APSP) distance matrix

many highly connected small subgraphs, and APSP calculations are made upon each subgraph. To perform searches of the input graph after the precomputation phase, PPA* performs an A* search on a related small generated graph, where a shortest path in the generated graph exactly corresponds to a shortest path in the large graph. Like A*, PPA* always returns the shortest path if one exists. Our experiments reveal that PPA* definitively outperforms A* on a test set of real world graphs, and that PPA* has a more pronounced advantage on larger graphs. Furthermore, it is easy to integrate PPA* into modern code bases because the in-game search code is simply performing A* on a precalculated graph. PPA* is most valuable when you have time to perform precalculations offline, when A* isn't fast enough to search graphs at run-time, and when APSP requires too much memory. Video games or large virtual worlds are a good examples of domains where PPA* is extremely valuable, because they have all three of these characteristics.

4.2 Related Work

A* (Hart et al., 1968) is provably the optimal algorithm for finding a single path in an unknown graph, but if multiple searches are performed over the same graph

then other algorithms are competitive. Floyd-Warshall (Floyd, 1962; Warshall, 1962), and Johnson's Algorithm (Johnson, 1977) are both techniques that find *every* shortest path in a graph, but are too slow as run-time algorithms for graphs with thousands of nodes. PPA* uses techniques from both A* and Johnson's Algorithm to quickly find single paths using precomputed results.

Several algorithms use a hierarchical approach to perform *approximate* shortest path computation. For example, the HPA* algorithm (Botea et al., 2004) produces paths that are within 1% of optimal on maps for video games, with searches that are 10 times as fast as A*. Sturtevant (2006) implements an alternative approximate search algorithm for the game Dragon Age^{TM}which expands 100 times less nodes than A*, while only using 3% additional memory. Given that game graphs are usually approximations of the space that can be navigated, game developers typically are not concerned with optimality, as long that the quality of the result is close to optimal. Comparisons between PPA* and A* suggest that our work is competitive with these approaches, while always producing the optimal path, and presumably developers would prefer an optimal path over a nearly optimal path if path generation speeds are similar. It is difficult to do measured comparisons between optimal and near-optimal approaches, since near-optimal approaches can typically trade off optimality for path generation speed. For this reason in this paper we will compare PPA* to techniques that like PPA* always produce the true shortest path.

A number of approaches are orthogonal to PPA* in that they could potentially be applied in combination for additional performance enhancement. In (Sturtevant et al., 2009), APSP matrices are partially computed and then used to produce a better heuristic function for A*. In Section 4.4, we find that PPA* searches faster and expands less nodes than this technique while using a comparable amount of memory. Approaches such as that of Bjornsson and Halldorsson (2006) and Hierarchical A* (Holte et al., 1996) also use clustering to create better

heuristic functions, but they improve performance over A* by less than a factor of 2. PPA* can use any admissible heuristic, and in our tests we use the straight line distance heuristic function for both A* and PPA*.

Cazenave (2007), and Louis and Kendall (2006) list a number of very useful practical optimizations to A* algorithms. Of these, we use lazy cache optimization, preallocation of memory, and maintaining the best path for each visited point in the node in both our PPA* and A* algorithms. Because performing PPA* at runtime involves performing an A* search, and because that A* search comprises the majority of the runtime of PPA*, any technique which improves the speed of A* will improve the speed of PPA* as well.

IDA* (Korf, 1985) trades performance in order to use less memory during a search, so it is unlikely to be useful for games; however, in certain domains, such as grid maps for games, algorithms such as Fringe Search (Bjornsson et al., 2005) use a combination of A* and IDA*, promising a 10-40% speedup over A*. An open question is whether a technique such as Fringe Search or heuristic methods such as Sturtevant et. al. could be combined with PPA* to achieve further performance gains.

There are some search techniques, such as D* Lite (Koenig and Likhachev, 2002), which search at a similar speed to A* but after changes to a graph can replan faster than performing another search. While PPA* offline structures can be dynamically altered if small changes are made to a graph (See Section 4.4), replanning techniques such as D* Lite are faster if a large number of dynamic changes are expected.

In the navigation search domain, there are a number of techniques with very fast search times, but it is difficult to compare them to PPA*. For example, Bast et al. produce shortest paths in an average of 5 milliseconds on a road map graph of the entire US with 24 million nodes. These paths are for the *travel time road map*, where each edge length is the amount of time to traverse that edge at posted

speed limits, rather than the *distance graph*, which is the embedded planar graph and is used in our results. The travel time road map is a very different graph topologically than the distance graph, since shortest paths in the travel time road map use freeways almost exclusively, whereas shortest paths on the distance graph are much more varied, often using surface streets. Because of this, techniques that attempt to establish "corridors", certain edges that are always traversed to get to certain regions, are much more effective on travel time graphs. Navigation search techniques are concerned about how long it will take to drive to a location, but in virtual worlds travel times are mostly uniform over distance and the distance graph is more relevant. It is unclear exactly how much faster it would be in general to search travel time graphs, but for one subsection of the search in (Bast et al.) (finding the total "distance" of the shortest path), searching the distance graph is 8 times slower than searching the travel time graph.

Bast et al. (2007) uses Dijkstra's single-source shortest path algorithm (Dijkstra, 1959) to determine a *highway hierarchy*, a set of edges that are commonly on shortest paths between nodes far away from each other. Like all precomputed hierarchical approaches, it relies on performing searches on successively larger and larger regions of the graph, but unlike PPA* which uses A* to search a simple generated graph, it relies on a bidirectional Dijkstra's search with lookups into tables which contain the distances of the highway edges. As a speed optimization, Bast et al. uses an APSP matrix for lookups of its highest graph, while PPA* uses APSP matrices at every level of the hierarchy. ((Sanders and Schultes, 2005) is a good introduction to this line of algorithms, based on approaches from the graph theory community.)

Our approach is most similar to the HEPV algorithm (Jing et al., 1996) which also comes from the Navigation Search literature. Like PPA*, HEPV uses clustering and APSP matrices to produce a precomputed data structure, but the run-time search is different. Instead of searching the reduced graph using A*,

HEPV searches it by beginning with the start and goal nodes and recursively expanding nodes connected to them. This effectively expands every node in the entire reduced graph, and in experiments it lags in performance behind A* on the original graph.

The problem with high speed search on large graphs appears in many different research communities. This paper bridges the gap between the navigation search and virtual world search communities by providing a competitive precomputed search algorithm that is easy for programmers to integrate into existing game code because it reuses existing components.

4.3 Approach

Conceptually, there are three different steps in PPA*.

1. Partition a graph into subgraphs
2. Calculate APSP matrices for the subgraphs
3. Perform searches

In practice, the first two steps are performed offline and repeated at different levels of the search data structure hierarchy. For ease of explanation, we will first describe the full approach with a single hierarchical layer, and then explain how the clustering and finally the search is expanded to multiple layers.

4.3.1 Single Layer PPA*

To describe single layer PPA* we will consider an example based on a 1913 map of Venice (Baedeker, 1913). We would like to perform fast searches over this map, for example from the triangle (Ex Campo Di Marte) to the star (Giardini Pubblici) in Figure 4.2. First, each island in the city is marked with a node and

Figure 4.2: Map of Venice

each bridge corresponds to an edge as in Figure 4.3. Next, the nodes are clustered as in Figure 4.4. The nodes in a cluster and the edges between these nodes form a *subgraph*. The goal of the clustering is to minimize the number of nodes that have edges that cross clusters. These nodes that have edges into different clusters are called *border nodes* and the edges that cross into other clusters are *border edges*. These special nodes and edges can be seen in Figure 4.5. As a precomputation step, after clustering we calculate the APSP matrices for each *subgraph*, so that at runtime we can lookup the shortest distance from any node in the subgraph to any other node in the *subgraph*.

In addition to the lower level *subgraphs*, there is a higher level *parent subgraph* that consists of all of the *border nodes* of each of its *child subgraphs*. Edges in this higher level subgraph consist of the *border edges* as well as *virtual edges*. *Virtual edges* are edges that we add to the higher level subgraph which represent connectivity in the lower subgraphs. Each *border node* in a lower level subgraph has a *virtual edge* to each other border node in its subgraph which represents the shortest path between the two nodes as seen in Figure 4.6. This *virtual edge* has a length of the precomputed shortest path. The entire highest level subgraph can be seen in Figure 4.7. We will also precompute the APSP matrix for this highest

Figure 4.3: Graph of Venice

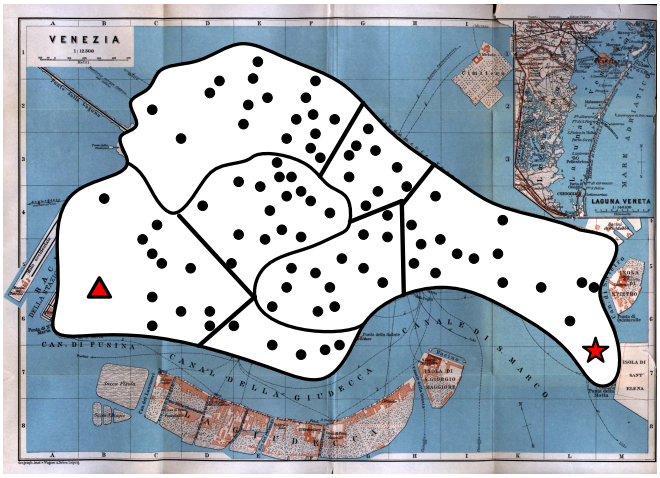


Figure 4.4: Nodes Clustered

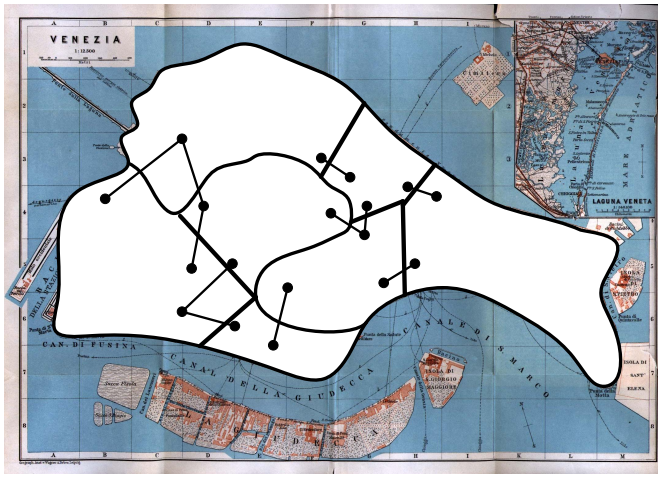


Figure 4.5: Border Nodes and Border Edges

level subgraph.

At runtime, we would like to perform a search on the graph of Venice. First we construct virtual edges between the start node and the border nodes of the starting node's subgraph. The lengths of those edges will be the distances recorded in the APSP matrix associated with the starting node's subgraph. We can construct similar virtual edges in the goal node's subgraph as seen in Figure 4.8.

Finally, we can construct virtual edges between each of the border nodes of the starting node's subgraph and each of the border nodes of the ending node's subgraph. The lengths of these virtual edges can be found in APSP matrix of the higher level subgraph. Putting these virtual edges together with the virtual edges of the start and end subgraphs for this search we get the final search graph, as seen in Figure 4.9. Instead of performing an A* search of the original graph with almost 100 nodes, we perform a search on this constructed graph with only 7 nodes.

To search a Single Layer PPA* data structure, PPA* search behaves much like A* search except that it searches over a different, smaller graph. The nodes of this smaller graph are a subset of the nodes of the original graph. Like A*, PPA*

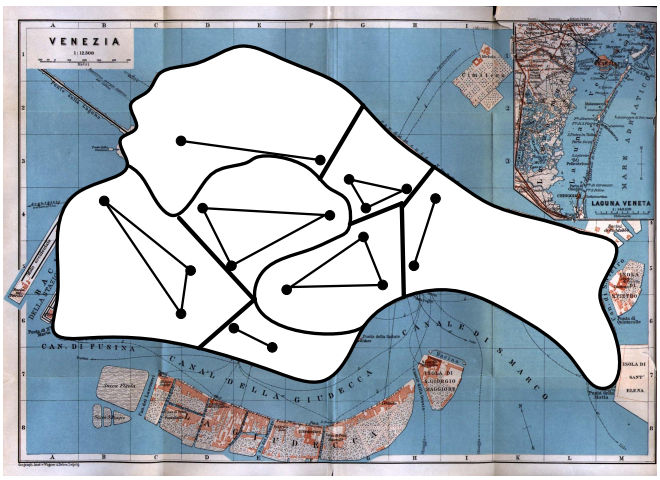


Figure 4.6: Virtual Edges

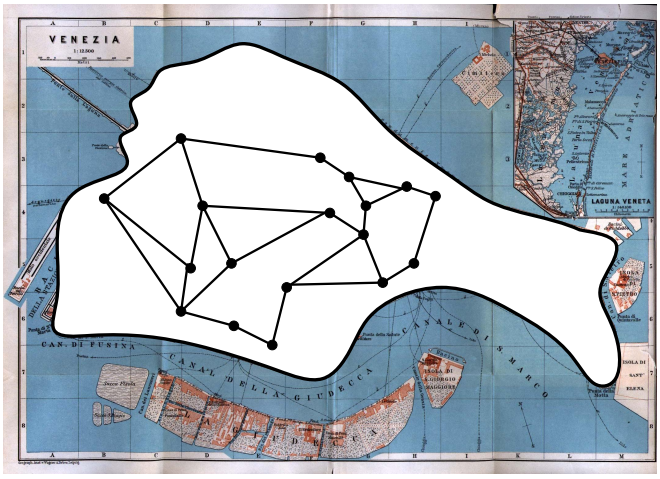


Figure 4.7: Parent Subgraph

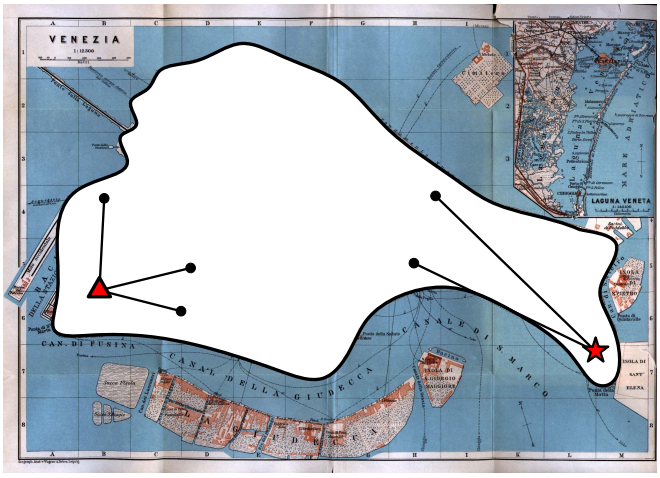


Figure 4.8: Start and Goal Subgraphs: Virtual Edges

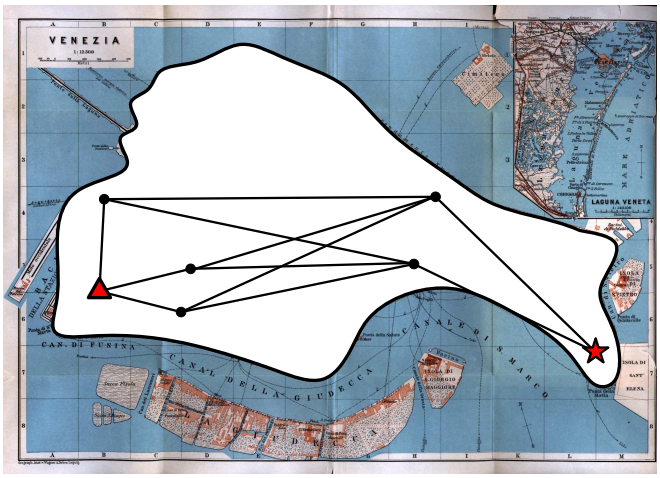


Figure 4.9: Final Searchable Graph

expands a start node, generates neighbors, and expands nodes until it expands the goal node. Each node expanded by PPA* is a node that will be expanded by A* and the nodes that are expanded by PPA* are expanded in the same order as the nodes are expanded by A*. Furthermore, PPA* uses the same $h()$ and $g()$ functions that A* uses, and when it evaluates a node, it obtains the same outputs from the $g()$ and $h()$ functions for that node that A* does. The paths returned by A* and PPA* are exactly the same. To verify the correctness of our implementation, after each search in a testing run we compare the A* calculated path against the PPA* calculated path.

4.3.2 Single Layer PPA*: Clustering and Analysis

In the simple single layer case, first a clustering algorithm (Karypis and Kumar, 1995) is used to cluster the original graph into a series of disjoint subgraphs. After clustering, we identify *border nodes*, which are nodes in a subgraph with edges to nodes in other subgraphs. Given a set of subgraphs, the ideal clustering of the original graph will minimize the number of border nodes while allocating similar numbers of nodes to each subgraph. In the planar Venice example (Figure 4.4), our clustering is simple, but in general a clustering technique for PPA* should not use the physical location of the nodes. Instead, minimizing the number of border nodes is the only important clustering criterion.

Two nodes which are far apart may be in the same cluster, and clusters can overlap in "physical space," as long as each node is contained by a single cluster and the number of border nodes per cluster are minimized. The PPA* algorithm works on the graphs that A* works on; it requires a graph with no negative cycles and an admissible heuristic. In many other precomputed techniques (especially non-optimal techniques such as HPA* (Holte et al., 1996) and (Sturtevant, 2006)), a single node represents a region of space and an entire child subgraph. In the PPA* data structure, a parent subgraph consists of all border nodes of each child

subgraph, which is normally several nodes contributed per child subgraph.

Assume that the original graph is clustered into f subgraphs each with k nodes. For each subgraph, an APSP matrix is generated. The precalculated matrices are each of size k^2, so the asymptotic memory of storing these matrices is $O(fk^2)$. We would like to estimate the algorithmic cost of memory in terms of n rather than k and f so that PPA* can be compared to other techniques. First we will estimate the memory costs of Single-Layer PPA*, in Section 4.3.3 we will extend these results to Multi-Layer PPA*.

Suppose there are c border nodes in each subgraph. We will form a new subgraph from these nodes, called the *parent subgraph*. Each of the disjoint subgraphs has the parent graph as a parent, but the parent graph *only* contains information about the border nodes of its child subgraphs (as opposed to all nodes in the original graph). The total number of nodes in the parent graph will be fc. We will construct an APSP matrix for the parent graph as well. So the total space for all of the APSP matrices of this graph will be $O(fk^2 + (fc)^2)$. As f and k approach the square root of the total number of nodes, $\sqrt{n}$ (since $fk = n$), the space cost approaches $O(n\sqrt{n} + nc)$, or $O\left(n\sqrt{n}\right)$ if c is small. This Single-Layer PPA* memory cost is too expensive, so we will reduce it by using multiple levels of hierarchy (see Section 4.3.3). All results (see Section 4.4) use Multi-Layer PPA*.

4.3.3 Multi-Layer PPA*: Clustering and Precalculation

The offline precalculation step of PPA* consists of two sub-steps, partitioning the original graph into a hierarchy of subgraphs and then calculating the APSP matrix for each subgraph. To cluster a graph with a levels of hierarchy, we first cluster the nodes of the original graph to produce several distinct child subgraphs and then cluster each of the child subgraphs recursively. This process is similar to that of a quad tree decomposition. Eventually, we reach the lowest level of

subgraphs, and the set of the lowest level of subgraphs will contain every node of the original graph, with each node belonging to exactly one lowest level subgraph. For the sake of clarity we will call these lowest level subgraphs *level-0 subgraphs*. Like the parent graph in the single layer case, a parent subgraph of a set of level-0 subgraphs will contain only the *border nodes* of those subgraphs (recall that border nodes are those that have edges to other nodes outside the subgraph). These level-1 parent subgraphs are disjoint from each other but not every node in the original graph is contained by them, since many nodes are not border nodes. We repeat the process of associating border nodes of level-1 subgraphs with their level-2 subgraph parents, and so forth to the top graph, of level-$(a-1)$. This top graph will only contain nodes which have edges that cross our first clustering of the graph. While the top graph only contains a fraction of the total nodes of the original graph, for any node we can get its level-0 subgraph, find the parent of that subgraph, the parent of that subgraph and so forth up until the highest level subgraph. In effect, each node has a list of "ancestor" subgraphs, but only the lowest, level-0 subgraph in this list is guaranteed to contain that node.

There exist a plethora of graph clustering algorithms. For this work we chose the Metis clustering algorithm (Karypis and Kumar, 1995). The Metis algorithm clusters graphs quickly (an order of magnitude faster than other algorithms we tested) and generally produces clusters which work very well with PPA*. The Metis algorithm attempts to divide a graph into f subgraphs where each subgraph has approximately the same number of nodes and where the number of edges that has to be cut to separate the subgraphs (in effect, the border edges) is minimized. Our metric is different; we want to find evenly sized subgraphs with as few *border nodes* as possible; however, in practice the Metis clustering does a fair job of meeting our metric.

Finally, after all subgraphs are determined, APSP matrices are computed intra-subgraph. Lower level subgraph shortest paths are calculated first, so that

virtual links can be constructed in higher subgraphs to properly calculate the shortest path distances in those subgraphs. Since graphs in games and other virtual worlds are relatively sparse, we use Johnson's algorithm (Johnson, 1977) to calculate the APSP matrices. Johnson's algorithm has a running time of $O(V^2 \log V + VE)$, which compares favorably to the $O(V^3)$ running time of Floyd–Warshall (Floyd, 1962; Warshall, 1962),for graphs where E is closer to V than V^2.

We can model the algorithmic memory usage of PPA* by assuming as before that the graph is broken up into subgraphs each with k nodes. We will assume that we have a perfect hierarchical graph, where the number of border nodes c per subgraph and the number of nodes k in a subgraph is constant at each of the a levels of the hierarchy. To calculate the asymptotic bound we multiply the total number of subgraphs by the memory of each subgraph, which is proportional to k^2. There is one highest level subgraph, and it has k nodes in it, like all subgraphs. The number of subgraphs in the next level down of the hierarchy is k/c, since each lower level subgraph has c nodes that they contribute to the highest level subgraph. Similarly, the level below that will have k/c graphs for every graph in the second highest level, or $(k/c)^2$ total subgraphs. This chain can be extended down until the lowest level, which will have $(k/c)^a$ subgraphs. We can then compute the total number of subgraphs as sum of all of these quantities multiplied by the size of each or $O(k^2((k/c)(a+1)-1)/(k/c-1))$. The total number of nodes in the graph is n, which is also equal to the number of subgraphs at the lowest level of the hierarchical tree, multiplied by k, since every node appears exactly once in the lowest level. Since there are $(k/c)^a$ lowest level subgraphs, $n = k(k/c)^a$. Solving for a we obtain $a = \log(N/k)/\log(k/c)$. We can then plug this back into the original memory equation to solve for the total amount of memory used by this ideal hierarchical approach. Fortunately, it simplifies neatly to $(k(n-c)/(k-c)$. If k/c, or the ratio of border nodes to nodes in a subgraph is constant (which is

common in our experiments), we can simplify this to $O(n)$ memory for PPA*.

It is simple to make small changes to the overall graph at runtime. This is important for virtual worlds that might have dynamic events such as a fallen bridge or a traffic jam, which might alter the in-game graph. There are four types of possible dynamic graph changes, node insertion, node deletion, edge insertion, and edge deletion. While each type of change uses a similar amount of computation, our experiments focus on edge deletion because it involves the smallest changes to the data structure of the graph. To perform an edge deletion, first the edge is removed, then the subgraph which contains the edge is recalculated. Each parent subgraph above this subgraph is also recomputed, so deleting a high level edge is faster than deleting a low level edge. For example, if a map has a number of islands connected by bridges, removing a bridge will be faster than removing a street on an island, since when a bridge is deleted only the "bridge-level" subgraph has to be recomputed, whereas when a street is removed both subgraph which contains the island as well as the bridge subgraph have to be recomputed, since the "bridge level" subgraph contains virtual links over the island between bridges which might use the removed street. To improve update times, at each level we check the distances between each border node of a subgraph before and after a dynamic change. If the distances between border nodes do not change, we cease recursing and avoid recalculating higher level graphs. While recompute times of about 0.20 seconds (see Section 4.4) are reasonable for some applications, if there are large scale dynamic changes then it is likely that a complete re-clustering will be necessary to preserve fast search speeds.

The precalculation phase is readily amenable to parallization. Precalculation consists of two stages, clustering and APSP calculations. Each clustering operation is independent of every other clustering operation, although top level clustering operations must be performed before lower level clustering operations. The calculation of APSP matrices is embarrassingly parallel; that is, each matrix

can be calculated completely independently, although lower level APSP matrices must be calculated before their parents. All of our tests were done with a single processor, but it should be noted that with k processors we expect a precalculation speedup of nearly k.

4.3.4 Multi-Layer PPA*: Search

Performing searches on hierarchical graphs with more than one level is similar to the single layer case (Figure 4.3.4). An important difference from the single layer case is that subgraphs will have higher level parent super-subgraphs that contain them. Each node has a subgraph that immediately contains it, which in turn is contained by a higher level subgraph, and there is a such a parent for each level of the hierarchy. So if there are four levels of hierarchy then each node has four subgraphs which are associated with it. Higher level subgraphs only contain APSP distance information for border nodes of their immediate child subgraphs, not every node contained by those subgraphs. Given a start node in a lowest level subgraph, PPA* generates the border nodes of that subgraph. The distance to each of the border nodes is the precalculated distance to them in the APSP matrix for the subgraph. To expand one of these border nodes, consider the parent graph containing it. This parent graph will have its own border nodes, which are nodes that connect to border nodes in other parent graphs. The distance to each parent border node is available in the parent graph APSP matrix. This process generates and expands nodes toward the uppermost parent graph. The search also expands nodes downwards towards the goal node. If a node is being expanded and one of its subgraphs is in the set of subgraphs associated with the goal node, PPA* generates border nodes for the child subgraph associated with the goal which is below the common subgraph. Finally, if a node is in the same immediate subgraph as a goal node, PPA* can generate the goal node. (See Pseudo-code 1)

For example, consider a street search from the UCLA campus in Los Angeles,

```
for i=0 to levelsOfHierarchy do
    # Subgraph(n,i) returns ith ancestor subgraph of n  currentSubgraph =
    Subgraph(currentNode, i)
    startSubgraph = Subgraph(startNode, i)
    goalSubgraph = Subgraph(goalNode, i)
    if (currentSubgraph == startSubgraph) then
        foreach borderNode in currentSubgraph do
            Add borderNode to Open List
        end
    end
    if (currentSubgraph == goalSubgraph) then
        if (i == 0) then
            Add goalNode to Open List
        end
        else
            ls = Subgraph(goalNode, i-1)
            foreach borderNode in ls do
                Add borderNode to Open List
            end
        end
    end
end
```

California to the Carnegie Mellon campus in Pittsburgh, Pennsylvania (See Figure 4.3.4). Suppose that this is a three level hierarchy where each city has its own subgraph, each state has its own subgraph, and the highest subgraph is the entire United States. The search will first expand the start node, which will generate the border nodes for Los Angeles. These border nodes for Los Angeles are the nodes which have connections outside of Los Angeles. When one of these Los Angeles border nodes is expanded, it will generate the border nodes for California, since California is higher level subgraph for the Los Angeles subgraph. Border nodes of the California subgraph are contained in the United States subgraph, but in this example the United States does not have border nodes, since it is the highest subgraph. Since the United States subgraph is an eventual parent subgraph of the Carnegie Mellon goal node, we expand downwards into the child subgraph of the United States containing Carnegie Mellon, which is the Pennsylvania subgraph. When a California border node is expanded, each border node of Pennsylvania is generated. Note that border nodes from Nevada will not be expanded, even though the eventual path might travel from California through Nevada, since Nevada is not an ancestor subgraph of Carnegie Mellon. Since Pennsylvania is an ancestor of the immediate subgraph containing Carnegie Mellon, when expand the border nodes of Pennsylvania we generate the border nodes of the Pittsburgh subgraph. Finally, when one of the Pittsburgh border nodes is expanded we generate the Carnegie Mellon goal node, and when we expand the Carnegie Mellon goal node we are finished. The path is our UCLA start node, a Los Angeles border node, a California border node, a Pennsylvania border node, a Pittsburgh border node, and finally our Carnegie Mellon goal node. The actual path is constructed using the information stored in the APSP matrices.

In order to construct a complete path from a PPA* search, it is necessary to find the nodes that connect each PPA* node together. This path is always contained in a set of existing APSP graphs. There are two possible methods of

recomputing this path: store paths in APSP matrices or perform a secondary search. In our experiments we always performed these secondary searches to construct full paths, and our APSP matrices only contain distance information. To perform a secondary search for an APSP graph, one can run A* on subgraph, but one must use the APSP distance values as the heuristic function, since the APSP values are the exact distance to the goal. Because this is an exact heuristic function, the A* algorithm will only expand nodes along the direct path to the goal. Some edges in a subgraph are "virtual" in that they do not exist in that subgraph but are contained in a lower level subgraph and APSP matrix. We recursively perform A* on these links in the lower level subgraphs.

In this paper we use a simplified version of A* called ResolvePath for this step. (See Pseudo-code 2) Since the heuristic is perfect there is no need to maintain an open list or "expand" nodes, instead for each node we look through each of its neighbors and the correct "next" node is immediately calculated.

In our example above (See Figure 4.3.4), we had a California border node and a subsequent Pennsylvania border node which were produced by our search. The actual best path might travel from California through the neighboring state Nevada on the way to Pennsylvania. In this case the optimal path in the United States subgraph would travel first to a Nevada border node on the California side of Nevada to a Nevada border node on the other side of Nevada. The edge across Nevada between these border nodes would be virtual, since information about the path would be stored in the Nevada subgraph rather than the United States subgraph. We would then run our ResolvePath function between these nodes in the Nevada subgraph, which might recurse into a Nevada city subgraph. Each node in the overall path is visited only once and in practice ResolvePath consumes less than 20% of the total search time. All reported results include this path resolution step in the time required to generate a path.

There are a few edge cases which are unintuitive. For example, suppose the

Procedure **ResolvePath**(c, n, s, p)

Input: c = The current node

Input: n = The next node

Input: s = The subgraph containing c and n

Input: p = The path between c and n

Output: p = The path between c and n

while $c \neq n$ **do**

$edge.cost = edge.length + \textbf{distance}(edge.to, n)$

$bestEdge = \arg\min_{edge} edge.cost$; edge $\in c$

if $!bestEdge.virtual$ **then**

Append$(p, bestEdge.to)$

end

else

$ls = \textbf{LowerSubgraph}(s, n)$

 # ls is child of s, contains node n **ResolvePath**$(n, bestEdge.to, ls, p)$

end

$c = bestEdge.to$

end

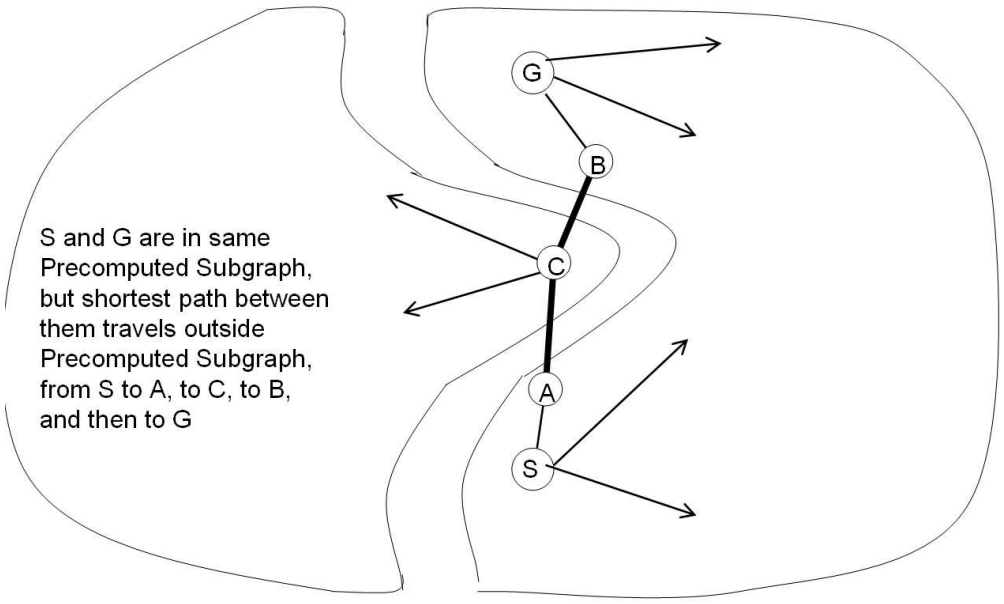


Figure 4.10: Difficult edge case

start node and the goal node are in the same graph (Figure 4.10). It might appear that PPA* could just use the precomputed path between them. Unfortunately, while PPA* can generate the goal node in this case, it might be that the shortest path between the start and goal nodes travels through border nodes and in different subgraphs. The APSP matrix only stores a shortest path contained in the subgraph and does not consider paths that travel outside the subgraph. In this example, the goal node and border node A are generated from the start node. Assuming an admissible heuristic function, PPA* will next expand border node A. Node A is contained by an ancestor graph of the goal graph, so PPA* looks for members of this super graph which are also members of the lower goal subgraph. The lower subgraph from the super graph on the ancestor list is the goal subgraph, so PPA* generates B, a border node of the goal subgraph. Since $g() + h()$ for node B is less than $g() + h()$ for the goal node through the start node, PPA* expands node B. Finally, PPA* generates the goal node a second time, and once it expands the goal node it will have a shortest path from the start to the goal. To produce the actual set of nodes in the final shortest path, PPA* consults the APSP matrix for each subgraph it traverses. In this example, when resolving the edge from A to B, the shortest path is in the super graph: $A \Rightarrow C \Rightarrow B$. The final shortest path is $start \Rightarrow A \Rightarrow C \Rightarrow B \Rightarrow goal$. Unlike approximate shortest path algorithms, PPA* always returns the shortest path.

The PPA* search of a graph is an A* search on every border node in the ancestor subgraph lists for the start subgraph and the goal subgraph. The order of expansions is based on the $g()$ and $h()$ functions, so PPA* might expand one node downwards and later expand another node upwards, just as A* will sometimes expand nodes that are two steps away from the start node before expanding all of the nodes that are one step away. As the number of border nodes per subgraph increases, the performance of PPA* decreases. If there are c border nodes per subgraph and a levels of hierarchy the number of nodes that PPA* needs to

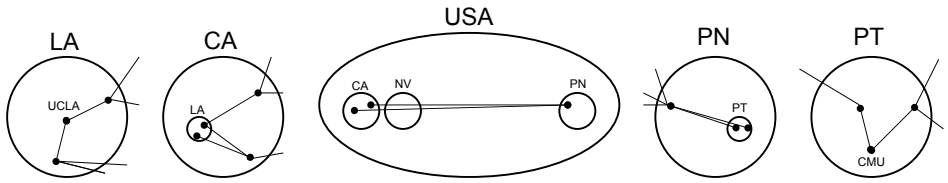


Figure 4.11: Example of PPA* Search

expand in the worse case is c^{2a}. In effect, A* is being performed on a graph with a depth of $2a$ and a branching factor of c. In Section 4.4, we will show experimental results which indicate that PPA* outperforms A* on real world graphs by a wide margin.

One concern that software engineers may have when confronted with a complex algorithm like PPA* is that it is too difficult or dangerous to integrate into an existing codebase. While the offline precomputation step is intensive, the in-game search algorithm can easily be integrated into an existing A* algorithm. The heuristic function and the overall logic of the A* algorithm are exactly the same, but the Add Neighbors to the Open List function is different. In A*, the edge length is the distance of an outgoing edge, but in PPA* the edge length is calculated by consulting the stored APSP matrix. Similarly, the successors to a node in A* are merely the neighbors of a node, but in PPA* they are the border nodes of the appropriate subgraph. To reduce complexity (at the cost of memory), the successor nodes as well as distances can be stored in APSP matrices, otherwise an algorithm such as ResolvePath should be used.

4.4 Results

Our results consist of two sets of experiments. For one set of experiments, we used road maps of California from the US Census Bureau. These maps vary in size from 11383 to 200288 nodes, and our experiments on them show the performance of PPA* on a range of different map sizes with relatively irregular node place-

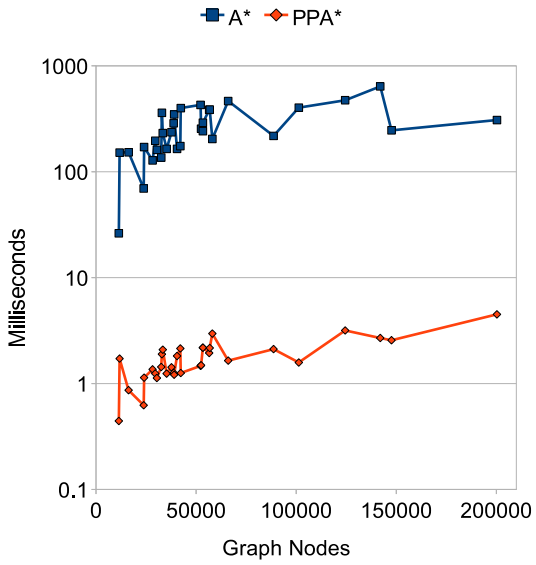


Figure 4.12: Average search times (logarithmic scale)

ment. The second set of experiments are performed on $512x512$ "room" maps of around 200000 nodes that are used by Sturtevant et al. (2009) and are publicly available. This second set of experiments clearly demonstrate the value of PPA* in comparison to a heuristic based search approach.

Each search consisted of picking two vertices at random and then finding the shortest path between them. The same set of vertices were used for both PPA* and A*. For each test, we performed 1000 such searches and averaged the results. The searches were evaluated on a 2.6 GHz Core i7 using a single thread. The average search time for PPA* was significantly less than for A* (Figure 4.12). Like all of the graphs in the results section, this graph uses a logarithmic scale to allow for comparison. It should be noted that the time listed in this graph is the total time to produce the entire optimal path for each method.

A method for comparing search algorithms that is independent of the hardware is to consider the number of nodes that the search expands or generates. The best way of explaining the difference between expanded nodes and generated nodes is to use A*, since PPA* uses A* to search its smaller graph. In A*, the start node

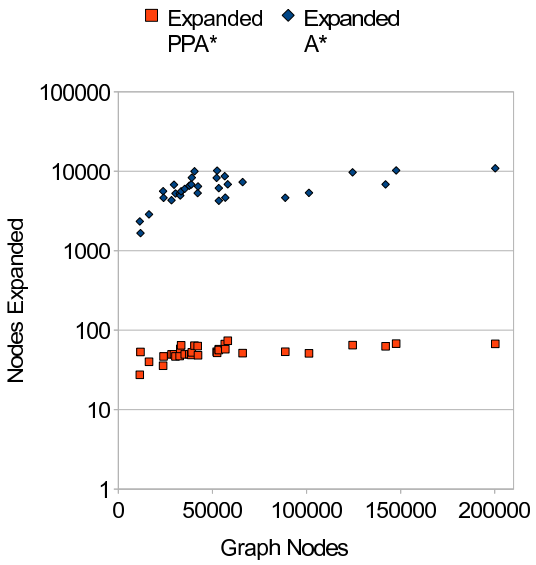


Figure 4.13: Expanded nodes per search (logarithmic scale)

is first placed on the open list, which is a priority list of nodes that need to be expanded. The top node of the open list is then expanded, and its neighbors that have not been expanded yet are added to the open list. The search ends when the goal node is expanded. The generated nodes are nodes that are added to the open list, while the expanded nodes (a much smaller subset of the generated nodes) are only the nodes that get expanded. Figure 4.13 shows the large difference in the number of expanded nodes between A* and PPA*. This discrepancy can be explained by the fact that a path in the PPA* graphs is much shorter than a path in the A* graph.

The number of nodes generated by both A* and PPA* is closer (Figure 4.14), but PPA* still outperforms A* by an order of magnitude. The reason that the generated nodes are closer is that the PPA* graph is highly connected, so each time a node is expanded, a large number of nodes are generated. In effect, the branching factor of A* is smaller than PPA*, but the number of steps required to travel from the start to the goal node is so much lower in PPA* that the number of total nodes generated (and the search time) is much lower in PPA*.

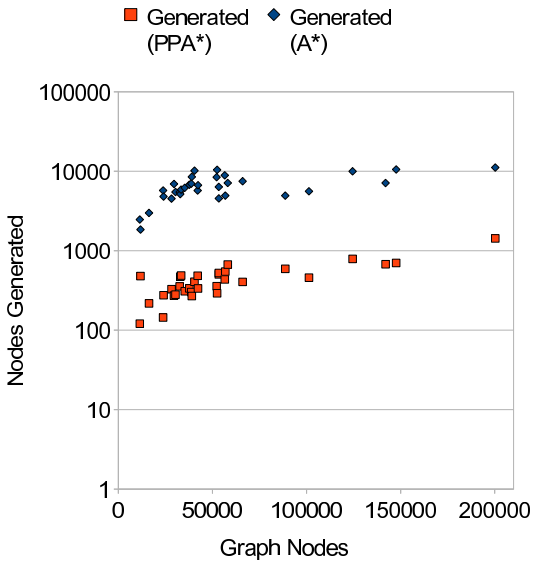


Figure 4.14: Generated nodes per search (logarithmic scale)

Table 4.1: Comparison vs. Sturtevant et. al. on 200000 node room graph

	Average Search Time (milliseconds)	Average Nodes Expanded Per Search	Precomputed Memory	% of APSP memory	Precomputation Time (seconds)
Sturtevant 2009	54	3479	N/A	~0.1%	N/A
PPA*	1.4	53	176 MB	0.1%	213

Additionally, most of the work done in each search is per expanded node rather than per generated node, and most search algorithms are compared by the number of expanded nodes rather than the number of generated nodes.

The second set of experiments also consisted of 291 random searches performed on a $512x512$ "room" map, the same map which was used by (Sturtevant et al., 2009) These searches all had a solution length between 256 and 512 nodes. With an optimized distance heuristic with advanced placement (the most effective search in Sturtevant et. al.), an average of 3479 nodes were expanded and each search took an average of 0.054 seconds. On the same map, PPA* expanded an average of 53.25 nodes and each search took an average of 0.0014 seconds, significantly better in both nodes expanded and execution time (See Table 4.1). It should be

noted that the number of nodes expanded by PPA* is usually smaller than the total path length of a search in the original graph because PPA* is effectively searching a much smaller graph. Additionally, in these experiments PPA* used a straight line heuristic, which while correct is less effective than the octile heuristic used by Sturtevant et. al. for this problem.

Precomputation consisted of two stages, clustering and computing APSP matrices. It took 469 seconds to precalculate our largest roadmap graph, which has 200284 nodes (Figure 4.15). Computing APSP matrices dominated the time spent on precomputation. Because clustering was such a small percentage of overall precomputation time, a more specialized clustering technique might be more effective for PPA* both in precomputation time and for run-time performance. Graphs of more than 50000 nodes had 5 levels of hierarchy with 4 subgraphs per parent subgraph for a total of 1365 subgraphs, of which 1024 were level-0 subgraphs.

PPA* has substantial storage benefits. An alternative to PPA* would be to calculate the APSP matrix for the entire graph during precomputation time. A search at run time of this matrix would be very fast, but the matrix would be unacceptably large. Assuming that each entry used four bytes, the fully precomputed matrix for 205373 nodes would take 169GB of space, which is too large for most practical applications. The same $512x512$ "room" map with 205373 nodes and four levels of hierarchy requires 176MB of space for PPA*. Figure 4.16 shows the dramatic memory savings of PPA* compared to full APSP matrices. PPA* has a competitive memory footprint with Sturtevant et al. (2009), who reported that their technique used 1/1000 of the total memory needed for the full APSP matrices (See Table 4.1).

It is relatively inexpensive to recalculate the APSP matrices after small dynamic changes. Using the 200000 node "room" graph of Sturtevant et. al., we built our normal precomputed graph which took 4.99 seconds to cluster and 207.87 seconds to build all 1365 APSP matrices, for a total precomputation time of 212.86

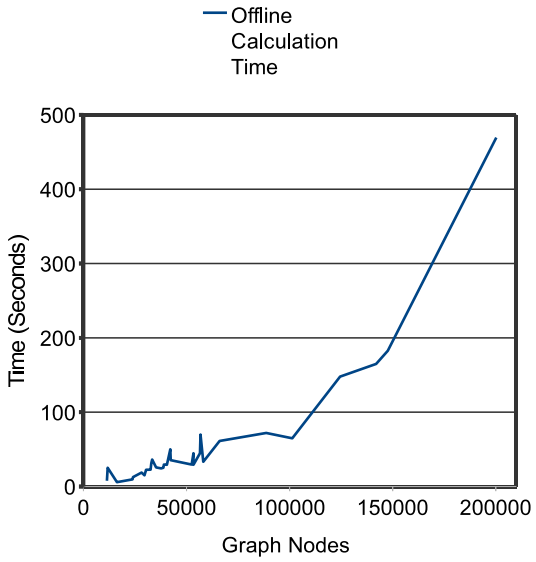


Figure 4.15: Precalculation Times

seconds. Then we deleted 100 edges at random, rebuilding the APSP matrices after each edge deletion. It took an average of 0.197 seconds to perform all required APSP matrix rebuildings after each edge was deleted.

4.5 Integration with Planning

Chapter 2, Section 2.5 describes how our planning space can be mapped onto a graph, with *States* that correspond to *Nodes* and *Actions* that correspond to *Transitions*. Once this planning graph is generated, we create a PPA* data structure that contains this graph.

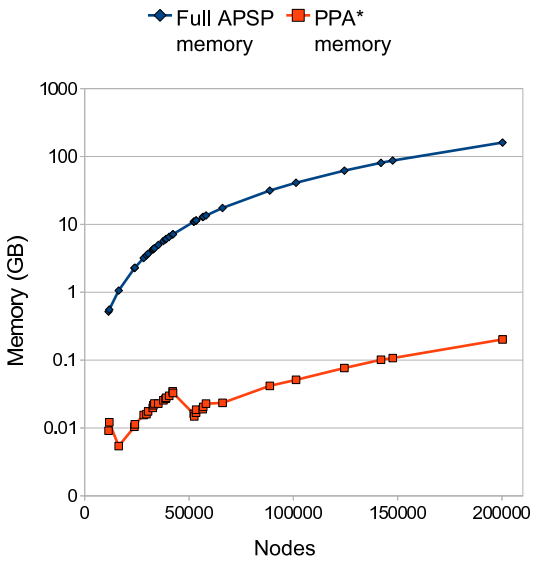


Figure 4.16: Precomputed memory costs (logarithmic scale)

One feature that our planning system requires is that a search may have multiple goal nodes (with a single start node). When there is more than one goal node, we are using a modified version of A* rather than PPA*. In this multi-goal A*, there are two differences. The first difference is that in the inner A* loop where we would normally check if an expanded node is the goal node we now have to check a set of goal nodes rather than just a single one. The second difference is that any heuristic function now has to predict the distance to the nearest goal node, rather than a distance to a unique goal node. This is tricky because the heuristic function is not admissible if it overestimates the distance to the nearest goal node, even if some of the goal nodes are further away than its prediction.

Fortunately, our A* implementation and our PPA* algorithm make no assumptions on the heuristic function and instead require the user to implement one (or return 0, which is always admissible because of the lack of negative distance transitions). It should be noted that many planning algorithms distinguish themselves by how effective their heuristic is, and techniques that improve heuristics could be added to our system to make it faster.

CHAPTER 5

Simulation

5.1 Introduction

In order to demonstrate some of the techniques presented in this thesis, we have created a virtual airport where agents use natural language generation, planning, and state machines to perform human-like behaviors and make human-like decisions. We selected two scenes from this simulation, both of which demonstrate hierarchical state machines and our natural language generation systems, while the second also demonstrates a "failed" plan.

Our simulation does not demonstrate the partially precomputed A* technique introduced in Chapter 4, because the virtual world is not large enough to require this speedup. Additionally, Figure 4.1 and Figure 4.12 in Chapter 4 show that PPA* outperforms alternative techniques on random graphs by a large margin.

In the first scene (Figure 5.1), computer animated customers move through a number of lines to communicate in English with airport ticketing counter attendants. During these dialogs the customers communicate with the ticket sellers to choose destinations and travel times. Finally, the customers receive their tickets and exit the lines. In the second scene (Figure 5.2), which takes place in a different section of the airport, the featured traveler has proceeded to her gate, but she realizes that the gate is closed. This means that her original plan to travel has failed. She then re-plans and talks to an airline representative in order to find her new departure gate.

Figure 5.1: First Scene: Ticketing

5.2 Graphics and Animation

Our simulation uses the Ogre (2013) open source rendering engine to help manage its graphics and animation systems. Figure 5.1 is a visual example of our characters engaging in dialog during the simulation.

We extended the OGRE engine with the animation systems used in Singh et al. (2011) and Kapadia et al. (2009) so that the animations can smoothly blend into one another. This animation system was further developed by Huang and Terzopoulos (2013) to create talking animations and to use inverse kinematics for the footwork. We used the steering solution described in (Singh et al., 2011) to guide our agents through the simulation. It is important to note the difference between a path finding system (see Chapter 4) and a steering solution. Path finding locates a path to a goal while steering chooses a realistic looking local path near this correct global path. Singh et al. (2011) uses A* as a sub-component to

Figure 5.2: Second Scene: Airport Gate

its steering system, and it may be possible to replace this A* sub-component with PPA* to improve performance in future versions of our system.

5.3 Planning and State Machines

The characters in the simulation use state machines to determine their behavior. For example, in the first scene (Figure 5.1) the ticket sellers run state machines similar to the one found in Chapter 2, Figure 2.4. Each character is running two hierarchical state machines; the parent state machine handles the animation and movement while the child state machine handles the natural language generation. In the first scene, the customers are performing the Ask_For_Ticket action described in Chapter 2, Figure 2.3, and do not need to invoke the planner because there are no problems with the current plan.

In the second scene (Figure 5.2), the featured agent also actively engages the planning system. Once she discovers that she is at the wrong gate, she updates her planning state to reflect that she is at the wrong gate and that she does not know from which gate her flight is leaving. Then she re-plans based on this new information. If we visualize the planning space as a graph, where each node is a planning state and each edge is an action between planning states, a plan is a path through that graph. A planning agent traverses the plan graph following the plan path from its current node (state) to a goal node (state). Occasionally, external factors such as a changed gate will cause a re-plan. In this case, the agent effectively believes that it is in a different node (state) then it actually is. Once this incorrect assumption is corrected, the agent adjusts which node it is in on the planning graph and re-plans a new path to the goal. After the featured agent re-plans, she talks to the gate agent to determine her new gate, because learning her new gate is the first step in her new plan.

One feature of our planning system is that agents can hold incorrect beliefs,

which can increase their realism. For example, if the gate for a flight is changed but not announced over a public address system, agents will not immediately change direction to travel to the correct gate. Instead, each agent will continue with its incorrect assumptions until they learn of this new information. Once they detect an inconsistency, they change their believed planning node (state) and re-plan from this new node. Because of the high performance of the finite planning space, such re-planning can be performed quickly.

5.4 Natural Language Generation

Figure 5.3 and Figure 5.4 show the dialog generated by our system for each scene of the simulation. In both scenes there are two paired state machines that create this dialog, one for each agent in the conversation. In the first scene, the buyer's state machine is similar to the state machine in Chapter 3, Figure 3.6, while the sentences are generated from data similar to that found in Chapter 3, Figure 3.2. From this content author data, large numbers of variations are generated as described in Section 3.4.3. Each of these variations are run through the RealPro software to generate possible sentences, and these sentences are ranked by their tri-gram perplexity as described in Section 3.3.3. For each dialog state in the state machine, we generate speech from the top ranked sentence generated from the original data.

To generate the speech audio we use the techniques described in Section 3.6. We have muted the other agents to avoid a cacophony of spoken interactions. Multiple agents talking over each other in many different conversations might be desirable for certain virtual worlds, but would draw attention away from the generated dialog in our simulation.

The final statement of the first scene "You are booked a flight to Boston" is an example of a mistake our system can make (see Chapter 3, Figure 3.1). We

Customer: Hello.

I would like to buy a ticket to Boston.

Ticket Agent: OK.

When would you like to travel?

Customer: I would like to leave at seven.

Ticket Agent: OK. You are booked a flight to Boston.

Figure 5.3: Scene 1 Dialog Example

have included this error in our simulation to demonstrate that while our Natural Language Generation system can make mistakes, they may not be so glaring in a conversation as to break realism.

In the second scene (Figure 5.4), the traveling agent is trying to find out where her new gate is, by talking to the gate agent. There are some interesting issues with this conversation. First of all, in the sentence "What happened to the flight to Boston?", the natural language generation system picks the correct sentence, but it should be noted that a grammatical sentence with the wrong meaning is generated. The third sentence (in order of tri-gram perplexity) is "What happened on the flight to Boston?", as shown by Figure 5.5 While this sentence is grammatically correct, it is not what the content author meant to express, and even if the perplexities correctly distinguished grammatical sentences from ungrammatical sentences, this sentence would not be removed. This type of error is noted in Section 3.4.3. Finally, in the sentence "Gate was changed", shown in Figure 5.6, we can see that tri-gram perplexity is not enough to separate the correct sentence "The gate was changed" from the grammatically incorrect sentence produced. Fortunately, this error is similar to the problem with the

Traveler: Hello

What happened to the flight to Boston?

Gate Agent: Gate was changed

You need to go to Gate Two.

Traveler: OK

Thank you very much

Figure 5.4: Scene 2 Dialog Example

<Sentence Perplexity="-11.935594" Text="What happened to the flight to boston?" />

<Sentence Perplexity="-12.059346" Text="What happened on a flight to boston?" />

<Sentence Perplexity="-12.194072" Text="What happened on the flight to boston?" />

Figure 5.5: "What happened?" Perplexities

sentence "You are booked a flight to Boston", in that while the produced sentence is not grammatical, it is not wildly incorrect enough to break the realism of the dialog. These sort of perplexity errors can be fixed by the techniques mentioned in Section 6.2.

<Sentence Perplexity="-10.415640" Text="Gate was changed." />

<Sentence Perplexity="-11.052237" Text="Gate would be changed." />

<Sentence Perplexity="-12.695483" Text="The gate was changed." />

Figure 5.6: “The gate was changed” Perplexities

CHAPTER 6

Conclusion and Future Work

In this dissertation, we introduced three interrelated systems for creating believable virtual human agents in virtual worlds. The first is the Natural Language Generation system, which allows content developers to script conversations between human characters. The second system is PPA*, an algorithm for calculating shortest paths that is well suited to the virtual world domain and significantly better than existing techniques. Finally, our planning and state machine environment uses the first two systems to build artificial agents that can plan, navigate, and talk. The planning, state machine, and NLG systems are hand tunable by content authors with no programming experience, and the system scales well to real time simulations with large numbers of agents.

6.1 Planning and State Machines

A finite space planning algorithm has advantages and disadvantages. The main advantage is extremely fast plan resolution that allows many agents to plan simultaneously at runtime and also perform frequent replanning operations. There are two disadvantages. The first is that there is somewhat of a loss of realism since, unlike humans, finite space planners cannot make general plans, and plans in one domain do not help in others. The second problem is that even in finite spaces, it is too easy to create state graphs that grow too quickly relative to the number of agents, the number of locations, or the number of items. We would like to explore techniques for avoiding such combinatorial explosions in graph size; for example,

by placing certain restrictions on plans.

We also would like to explore more natural authoring of planning problems. Using a hierarchical planning system like SHOP2 (Nau et al., 2003) is a highly attractive method of approaching the authoring problem, but it is unclear how to integrate hierarchical planning into our finite space planner. Finally, something that we would like to investigate in the future is a hybrid planner that sometimes uses fast planning over a finite search space and at other times uses a relatively slow but bounded search over infinite or arbitrarily large search spaces.

The interaction of State Machines and Planning systems is a relatively unexplored area of research. Currently, in our system, state machines must be hand linked to planning nodes. For example, a programmer connects a Goto planning action and a Goto state machine. If State Machines published their planning inputs and outputs, then there could be a state machine library that could be accessed by the planning algorithms in the same way that a content author chooses a state from a library of possible states when constructing a state machine.

6.2 Natural Language Generation

Our Natural Language Generation system gives content authors a high degree of control over formulating conversations, without requiring programming experience or linguistic expertise. It is designed to fit into a system of goals and plans that create a believable facsimile of human intelligence, while allowing relatively tight control over content.

There are two improvement directions that we would like to explore with the NLG system. First, we would like to greatly expand the number of correct sentences that are generated, without compromising either the runtime or introducing incorrect meanings. The most promising strategy is to introduce synonyms after a first pass of sentence generation. The second problem is that occasionally the

n-gram model scores an incorrect sentence too high. The solution to this is to use four-grams and five-grams, in addition to the uni-grams, bi-grams, and tri-grams that we are already using. There are a number of techniques that we are exploring to lower the memory requirements of the entire tri-gram system in order to include higher n-grams.

While much of sentence generation is done automatically, conversational state machine creation is left mostly in the hands of the content authors. It would be better to have conversational templates as an alternative to state machines for most routine conversations. These templates would ultimately resolve into state machines, but would encapsulate frequently used conversational state machine patterns. It may be the case that sentence planning techniques could allow content authors to script only general outlines for a conversation and the natural generation system would create the rest of the dialog.

6.3 PPA*

Many developers use A* or completely calculated shortest paths to navigate agents in virtual worlds. As the size and number of agents in these worlds increase, these algorithms will be too expensive in compute time and memory. We introduced the Partially Precomputed A* (PPA*) algorithm. PPA* makes large performance and memory improvements on these existing algorithms and is simple and safe to integrate into existing game engines.

We would like to make further improvements in our search algorithm. One promising avenue is to use bidirectional search, which begins at both the start and goal node instead of searching from the start node to the goal. Since the middle of the PPA* graph is much more dense than the ends, this technique promises dramatic speed gains.

In the longer term, we would like to extend PPA* to situations where APSP

matrices are not precomputed, but are gradually filled in as more searches are completed. This is similar to Lifelong Planning A* (Koenig et al., 2004), which performs successive searches of a graph with different start nodes and a fixed goal node, except we would allow arbitrary start and goal nodes. The intuition is that certain PPA* subgraphs are likely to be traversed by a number of searches, and shortest paths to and from the border nodes of these subgraphs are gradually developed.

All of the experiments done with PPA* use a straight-line heuristic, which is correct but can be improved. Most other techniques for improving A* focus on improved heuristics. Since PPA* is heuristic-independent and accepts any admissible heuristic, it follows that these other techniques could be used to speed up PPA* just as they speed up A*. It is unclear at this time which heuristics are most effective on the condensed graph that PPA* uses; this is an active area of our research.

We hope to expand PPA* functionality to include multiple goals, but this is much more complicated than the multi-goal A* we are using for planning. For example, consider Pseudo Code 1 from Chapter 4, Section 4.3.4. In normal PPA*, we build a set of all of the subgraphs which are ancestors of the goal subgraph. In multi-goal PPA*, we would need to create one of these sets of goal subgraphs for each of our goals. In normal PPA*, when we expand a node that is in a goal subgraph ancestor A, we add the neighbor nodes of the goal subgraph $A-1$ contained by A that is one subgraph down from A in the goal subgraph chain, as we expand downwards towards the goal node. In multigoal PPA*, we might have to add neighbor nodes in different graphs when we expand downwards, because different goal nodes might share the same subgraph ancestor A but not the same subgraphs $A-1$. Using multi-goal A* is much slower than using PPA*, but it still produces a correct path and therefore a correct plan. Fortunately, our multi-goal A* algorithm can search a PPA* data structure because all of the normal graph

information is still present.

We are interested in finding applications where many A* searches need to be done quickly, without a separate precomputation time, which might be good candidates for PPA*. PPA* is clearly useful in settings where precomputation is not an issue and computational performance is vital.

Bibliography

Ajdukiewicz, K. (1935). Die syntaktische konnexität. *Studia Philosophica*, 1(1):27. 22

Atkinson, K. SCOWL. version 7.1. Available at http://wordlist.sourceforge.net/. 25

Baedeker, K. (1913). Northern Italy handbook for travelers. 43

Baldridge, J. and Kruijff, G.-J. M. (2002). Coupling ccg and hybrid logic dependency semantics. In *IN PROC. ACL 2002*, pages 319–326. 22

Bangalore, S. and Rambow, O. (2000). Exploiting a probabilistic hierarchical model for generation. In *Proceedings of the 18th conference on Computational Linguistics-Volume 1*, pages 42–48. Association for Computational Linguistics. 23

Bar-Hillel, Y. (1953). A quasi-arithmetical notation for syntactic description. *Language*, 29(1):47–58. 22

Bast, H., Funke, S., Matijevic, D., Sanders, P., and Schultes, D. In transit to constant time shortest-path queries in road networks. In *9th Workshop on Algorithm Engineering and Experiments (ALENEX'07)*, New Orleans, USA. 41, 42

Bast, H., Funke, S., Sanders, P., and Schultes, D. (2007). Fast routing in road networks with transit nodes. *Science*, 316(5824):566. 42

Bjornsson, Y., Enzenberger, M., Holte, R., and Schaeffer, J. (2005). Fringe search: Beating A* at pathfinding on computer game maps. *Proceedings of the IEEE Symposium on Computational Intelligence in Games*, pages 125–132. 41

Bjornsson, Y. and Halldorsson, K. (2006). Improved heuristics for optimal path-finding on game maps. *Proceedings of the Second Artificial Intelligience and Interactive Digital Entertainment Conference*, pages 9–14. 40

Botea, A., Muller, M., and Schaeffer, J. (2004). Near optimal hierarchical path-finding. *Journal of Game Development*, 1:7–28. 40

Cazenave, T. (2007). Optimizations of data structures, heuristics and algorithms for path-finding on maps. In *Computational Intelligence and Games, 2006 IEEE Symposium on*, pages 27–33. IEEE. 41

Chen, J., Bangalore, S., Rambow, O., and Walker, M. A. (2002). Towards automatic generation of natural language generation systems. In *Proceedings of the 19th international conference on Computational linguistics-Volume 1*, pages 1–7. Association for Computational Linguistics. 23

DeVault, D., Traum, D., and Artstein, R. (2008a). Making grammar-based generation easier to deploy in dialogue systems. In *Proceedings of the 9th SIGdial Workshop on Discourse and Dialogue*, pages 198–207. Association for Computational Linguistics. 23

DeVault, D., Traum, D., and Artstein, R. (2008b). Practical grammar-based NLG from examples. In *Proceedings of the Fifth International Natural Language Generation Conference*, pages 77–85. Association for Computational Linguistics. 23

Dijkstra, E. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271. 19, 42

Floyd, R. W. (1962). Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345. 40, 52

Harrison, M. A. (1965). *Introduction to switching and automata theory*, volume 65. McGraw-Hill New York. 15

Hart, P., Nilsson, N., and Raphael, B. (July 1968). A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107. 38, 39

Hernault, H., Piwek, P., Prendinger, H., and Ishizuka, M. (2008). Generating dialogues for virtual agents using nested textual coherence relations. In *Intelligent Virtual Agents*, pages 139–145. Springer. 3

Hewlett, W. (2011). Partially precomputed A*. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(2):119–128. 38

Hockenmaier, J., Bierner, G., and Baldridge, J. (2004). Extending the coverage of a CCG system. *Journal of Language and Computation*, 2:165–208. 22

Holte, R. C., Perez, M. B., Zimmer, R. M., and MacDonald, A. J. (1996). Hierarchical A*: Searching abstraction hierarchies efficiently. In *AAAI/IAAI, Vol. 1*, pages 530–535. 40, 49

Huang, W. and Terzopoulos, D. (2013). Door and doorway etiquette for virtual humans. 68

Jing, N., Huang, Y.-W., and Rundensteiner, E. A. (1996). Hierarchical optimization of optimal path finding for transportation applications. In *CIKM*, pages 261–268. 42

Johnson, D. B. (1977). Efficient algorithms for shortest paths in sparse networks. *J. ACM*, 24(1):1–13. 40, 52

Kapadia, M., Singh, S., Hewlett, W., and Faloutsos, P. (2009). Egocentric affordance fields in pedestrian steering. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 215–223. ACM. 68

Karypis, G. and Kumar, V. (1995). A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report TR 95-035. 49, 51

Koenig, S. and Likhachev, M. (2002). Improved fast replanning for robot navigation in unknown terrain. In *in Proceedings of the International Conference on Robotics and Automation*, pages 968–975. 41

Koenig, S., Likhachev, M., and Furcy, D. (2004). Lifelong planning A*. *Artif. Intell.*, 155(1-2):93–146. 78

Korf, R. E. (1985). Depth-first iterative-deepening: an optimal admissible tree search. *Artif. Intell.*, 27(1):97–109. 41

Langkilde, I. and Knight, K. (1998). Generation that exploits corpus-based statistical knowledge. In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics - Volume 1*, volume 1 of *ACL '98*, pages 704–710, Montreal, Quebec, Canada. Association for Computational Linguistics. 21

Lavoie, B. and Rambow, O. (1997). A fast and portable realizer for text generation systems. In *Proceedings of the Fifth Conference on Applied Natural Language Processing*. 29

Louis, S. J. and Kendall, G., editors (2006). *Proceedings of the 2006 IEEE Symposium on Computational Intelligence and Games (CIG06), University of Nevada, Reno, campus in Reno/Lake Tahoe*. IEEE. 41

Mateas, M. and Stern, A. (2004). Natural language understanding in Facade: Surface-text processing. *Technologies for Interactive Digital Storytelling and Entertainment*, pages 3–13. 1

Miller, G., Beckwith, R., Fellbaum, C., Gross, D., and Miller, K. (1990). WordNet: An on-line lexical database. *International journal of lexicography*, 3(4):235–312. 22

Nau, D., Au, T., Ilghami, O., Kuter, U., Murdock, J., Wu, D., and Yaman, F. (2003). SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research*, 20(1):379–404. 9, 76

Nau, D., Cao, Y., Lotem, A., and Muftoz-Avila, H. (1999). SHOP: Simple hierarchical ordered planner. pages 968–973. 9

Ogre (2013). Ogre 3D open-source graphics rendering engine. http://www.ogre3d.org/. 68

Orkin, J. (2004). Symbolic representation of game world state: Toward real-time planning in games. In *Proceedings of the AAAI Workshop on Challenges in Game Artificial Intelligence*. 5

Orkin, J. (2005). Agent architecture considerations for real-time planning in games. *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment*. 9

Sanders, P. and Schultes, D. (2005). Highway hierarchies hasten exact shortest path queries. *Algorithms–ESA 2005*, pages 568–579. 42

Schuler, K. (2005). *A BROAD-COVERAGE, COMPREHENSIVE VERB LEXICON*. PhD thesis, University of Pennsylvania. 22

Shao, W. and Terzopoulos, D. (2007). Autonomous pedestrians. *Graph. Models*, 69(5-6):246–274. 2

Singh, S., Kapadia, M., Hewlett, B., Reinman, G., and Faloutsos, P. (2011). A modular framework for adaptive agent-based steering. In *Symposium on Interactive 3D Graphics and Games*, pages PAGE–9. ACM. 68

Sturtevant, N. R. (2006). Memory-efficient abstractions for pathfinding. *Proceedings of the Third Artificial Intelligience and Interactive Digital Entertainment Conference*, pages 31–36. 40, 49

Sturtevant, N. R., A, F., M, B., J, S., and N, B. (2009). Memory-based heuristics for explicit state spaces. *International Joint Conference on Artificial Intelligence*, pages 609–614. 40, 61, 63, 64

Thorsten Brants, A. F. (2006). Web 1t 5-gram version 1. 24

Traum, D., Fleischman, M., and Hovy, E. (2003). *Nl generation for virtual humans in a complex social environment*. Defense Technical Information Center. 23

Turing, A. M. (1936). On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 42(2):230–265. 15

Walker, M. A., Rambow, O., and Rogati, M. (2001). SPoT: A trainable sentence planner. In *Proceedings of the second meeting of the North American Chapter of the Association for Computational Linguistics on Language technologies*, pages 1–8. Association for Computational Linguistics. 23

Warshall, S. (1962). A theorem on boolean matrices. *J. ACM*, 9(1):11–12. 40, 52

White, M. OpenCCG: The opennlp ccg library (openccg.sourceforge.net). 22

White, M. and Baldridge, J. Adapting chart realization to CCG. In *IN: PROC.* 22

Wikipedia (2013). Perfect hash function — Wikipedia, the free encyclopedia. [Online; accessed 2-May-2013]. 28