NORTHWESTERN UNIVERSITY

Real-time Decision Making for Adversarial Environments
Using a Plan-based Heuristic

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Computer Science

by

Robert Shaun Thomas

EVANSTON, ILLINOIS

June 2003

ABSTRACT

Real-time Decision Making for Adversarial Environments
Using a Plan-based Heuristic

Robert Shaun Thomas
Northwestern University
2003

Throughout human history games have served a cultural entertainment role. With the advent of the computer, games have taken on new dimensions and become a part of our mainstream culture. One of the new dimensions the computer has brought to game playing is the potential for interactive game play through the use of non-player characters (NPCs). Traditionally these NPCs have been implemented in a variety of game types (adventure, strategy, first-person shooter, role-playing) through the use of scripts. These scripts however are limiting in their rigidity. In order to create more flexible and compelling characters with sophisticated behavior what is needed is a control technique that allows an NPC to decide a course of action given its current goals and situation. AI planning systems have been designed to address these very issues, but have yet to be used in any commercial games. This is a result of the disconnect between the goals of the AI researcher and the real world constraints faced by the computer game producer.

This thesis is about an agent designed to play the board game Settlers of Catan. In designing this agent our goal was to create a control technique that allowed our agent to decide a course of action given its current goals and situation. Additionally, by using a computer game environment we were forced to address some of the real world constraints faced by computer game producers. These included limited computational resources, and having our agent able to make decisions within a reasonable time frame while maintaining compelling and sophisticated behavior.

Our solution involves using a decision-making strategy that uses a rough plan to guide reactive behavior. In addressing the negotiation aspect of the game we used strategies adapted from business negotiation literature. To test our ideas we have created an on-line version of Settlers of Catan called Java Settlers. This environment allows us to collect data on the behavior of human and computer controlled players.

ACKNOWLEDGEMENTS

This dissertation and the work that it represents would not have been possible without the help, support, and generosity of many people. It has been a long road so its hard to know where to start. Robin Hunicke and Jay Budzik both began their graduate careers around the time when I started and were there when I discovered Settlers of Catan. I remember playing on the floor of the computer lab at the University of Chicago where we started as graduate students. Robin's support, feedback, and patience with me through the entire process has been invaluable. Jay has always been great to talk to and I am indebted to him for helping me make the networking code robust enough to handle hundreds of simultaneous connections without deadlocking.

Josh Flachsbart and Dave Franklin have also taken the amazing journey that is graduate school along with me and I have learned a lot from both of them. Josh has an amazing depth of knowledge of everything from programming languages and computer vision techniques to world history and religion. Dave is multi-talented and in addition to being a great jazz pianist, also has a knack for creating really fun poker variants. David Wilson and Shannon Bradshaw have been valuable friends both for being great people to bounce ideas off of and for playing a mean game of poker.

Julie Dubiner and Vidya Setlur have both been great officemates. Julie helped me brainstorm ideas for what to name the project and helped write a general description of it for a non-technical audience. Her sense of humor and attitude helped me keep perspective when things seemed to be at their worst. Vidya is an inspration, she always has a positive attitude and is one of the nicest people I know.

I am grateful to the people in the lab who showed the most enthusiasm for what I was working on, Louis Lapat and David Ayman Shamma. Lou would always be interested in what problem I was trying to solve and would always ask lots of great questions. Ayman I think is the biggest fan of Java Settlers in the lab and his stories about Shakey are hilarious. Or at least I find them funny. His enthusiasm for my work helped me get through the times when I would get stuck writing and couldn't imagine why anyone would want to read this thing, so I am very grateful for his feedback.

I also need to thank everyone else in the Intelligent Information Laboratory, since we are like a big family and help each other out in any way we can. I appreciate all of the conversations we have at lunch and after work. So a big thank you goes to Andy

Crossen, Sanjay Sood, Jared Dunne, Sara Owsley, and Kate Lockwood. I could tell stories about all of them, but I wouldn't want to get myself into trouble.

Donna Szymanski is the office manager for the Infolab and I can't thank her enough for all of her help over the years at Northwestern. She is not simply a manager, but for lack of a better term, a morale officer for the lab, and an excellent cook. I'm sure I speak for all the students in the lab when I say we love you Donna.

I want to thank Larry Birnbaum and Ian Horswill for giving me valuable feedback on drafts of this dissertation and advice for how to improve it. Their comments have been insightful and this dissertation is all the better due to their ideas and suggestions.

I need to thank Janice Muhr who also was very helpful during the writing of this dissertation. She helped me in numerous ways throughout the writing process and I am grateful for her advice on how to get through it and for her feedback on early drafts.

My advisor Kris Hammond has known me since before I became a graduate student. I am indebted to him for giving me the chance to do the kind of research I felt most passionately about, and for always supporting it. He has created an incredible laboratory of some of the smartest people I've ever encountered and I feel privileged to have been a part of it.

My mom has been an inspiration for me since the beginning. She has been nothing but supportive through the entire process and I don't know how I could have possibly have done it without her generous love and support. Her knowledge of the business world and negotiation has also been helpful in generating some of the ideas that were used in my research.

Finally I have to thank Heidi Kasper for her infinite patience with me during the writing of this work. She has helped me in so many ways during this process I'm sure I won't be able to think of them all. I am grateful for all of her feedback on many drafts of this work, as well as her help with how best to present some of the more complicated ideas. She has also been extremely supportive and caring and I love her with all my heart.

Contents

List of Figures

List of Tables

Chapter 1
Introduction and Motivation

Many years ago, when I was very little, I was taken to a science museum. This was the kind of museum where the exhibits were interactive and encouraged children to learn about science by doing little experiments that were set up for them. One thing that I remember most at this museum was my first experience with a computer. This was the late 70's, early 80's, so personal computers were just becoming available and this is what the museum had. I remember that it had a black and white screen and the graphics were of the ASCII variety. What was on the screen was a simulation of a lunar lander and the object was to control its thrusters in order to land safely. I remember having no idea what I was supposed to do or how to make anything work but I was still fascinated with this interactive thing that wasn't like anything else in the museum. Soon I was taking workshops for children where we were taught how to program in BASIC and I learned that you could make the computer do just about anything you could think of as long as you could write a program for it. From then on I knew that whatever I did with my life, it would involve computers in some way and most likely programming them.

As I grew up I took more classes on programming and eventually owned my own personal computer. I read magazines and books on programming and wrote many small programs for fun. In high school, when I was deciding what I wanted to study in college, I decided that I wanted to study artificial intelligence (AI). It seemed to me that the programmers who would work on the software of tomorrow would need to know at least some AI techniques.

Just before graduating college, I was an intern for Kris Hammond at the University of Chicago. There I worked on a number of AI projects including a car recommendation system called Car Navigator and a movie recommendation system called Pick-A-Flick (Burke 1997). Later, Kris became my advisor. The reason I became one of Kris' students is because I agreed with his philosophy of how to do research in finding useful AI techniques to create software that would improve people's lives.

Shortly after becoming a graduate student, I needed to come up with an idea for a project to work on. I wanted to work on a project that would expose me to a number of different areas of AI and I was also interested in applications that would be used for entertainment, either an authoring tool or a piece of entertainment software. At the time Joseph Bates' Oz project (Bates 1991) was getting attention in the AI community and I was partly inspired by that work. The main goal of the Oz project was to create tools that would allow artists to create believable characters for interactive fiction. I found this work interesting because one of their hypotheses was that creating a lifelike character didn't require that a lot of deep knowledge about the world be programmed into it. When I thought about how people would participate in these interactive fiction systems that they wanted to create, it struck me that people would at first probably cooperate and do what was expected. But as soon as that became uninteresting, the user would try to find the boundaries of the environment either in terms of actual geometric boundaries or more likely the boundaries of what was possible in that artificial world. This type of exploration would be contrary to the goals of an author who devised some kind of plot structure for the story that the interactive fiction would portray. So I thought that I would take a shot at addressing this problem and ways to steer a user back onto the author's intended track without breaking the users suspension of disbelief. I didn't get very far with this idea because I needed an environment in which to do the research that was simple enough to create in a relatively short time and still be complex enough to demonstrate my ideas. But thinking about how to create or find an environment with the qualities I wanted made me consider games as a possible direction.

Both parlor games and computer or video games have the quality that they are abstractions of some situation that can be found in the real world. This level of abstraction varies greatly from games that are very abstract like Go to realistic computer simulations like Microsoft Flight Simulator. The connection to real world situations can vary greatly too. Some games try to reproduce specific historical conflicts like World War II or the American Civil War; others are based on running a business in a competitive environment. Other games may have a less clear connection to a real world situation like Backgammon or Checkers, but if they didn't require the use of at least some skill or intuitive knowledge that most people have, people would have a hard time relating to the game. Due to the abstract nature of games, they are much simpler in terms of what kinds of actions the participants are allowed to take as compared to their real world counterpart. Because most games are a simplification of a real world situation or event, these games make good candidates for being used as environments for studying human behavior as well as the behavior of intelligent agents.

I started thinking about what game would make a good environment for research and about the same time I picked up a new board game called Settlers of Catan. In this game four players compete to be the first to successfully colonize the small, uninhabited island of Catan. Each player does this by gathering resources from the land and then using them to build roads, settlements, and cities. While playing this game I observed that players would use strategic planning, resource management, diplomacy and negotiation in order to play the game competently. I figured if I attempted to build an intelligent agent that could competently play Settlers of Catan I would have to learn about a number of different issues addressed by the artificial intelligence community and at the end I would have an artifact that other people could use for fun and entertainment.

Less is More

The thesis claim of this dissertation is "Less is More". What I mean by this is the following: When people approach a seemingly complicated problem, like designing an intelligent agent for a complex domain, they sometimes assume that the design of the agent must also be complex. This is not necessarily true. When designing agents to act intelligently in a real-time system, using simple techniques that get you 80% of the behavior you want suffices.

From our experience, designing an agent that performs 80% of the desired behaviors can be done with relatively simple methods. Utilizing simple methods whenever possible has many advantages. They are easier to understand and debug in addition to being easier to modify and maintain. Also simple designs tend to run more efficiently. In situations where computational resources are shared with other systems, this efficiency can become a critical factor.

Throughout this dissertation, I will attempt to show how our process and results support this claim. In a nutshell, our process can be described as follows: Implement the simplest idea first and see where it fails. Make the smallest change possible to get the desired effect and repeat. Our results will show that by using this methodology we were able to develop a robust system that supports hundreds of simultaneous users and provides them with challenging intelligent agents which to interact.

Settlers of Catan

Now I will discuss the Settlers of Catan game in more detail. As I briefly stated before, Settlers of Catan is a game where four players compete to be the first to establish a successful colony on the newly discovered island of Catan. There are no native inhabitants on the island, so the only people the players can interact with are themselves. Also, the only resources that each player can use to build up their colony are ones that they can harvest from their plot of land, or what they can get through trade. The success of a colony is measured in Victory Points, and the first player to gain 10 or more Victory Points is the winner (Smith 1996).

The game itself is very colorful and tactile. The island is represented with hexagonal cardboard tiles, each of them about four inches in diameter. On each of them is a drawing depicting one of six different types of land: plains, meadows, mountains, hills, forest, and desert. These tiles are shuffled at the start of the game and laid out in a large hexagonal pattern. Around the edge of the main island another set of tiles is shuffled and laid out which represent ports of call. Each player gets a bunch of colored wooden pieces that represent their roads, settlements, and cities. When a player expands their colony, they place these pieces on the board. Cards are used to represent the resources that players use when building. Each card has a picture depicting one of the five different kinds of resources: shafts of wheat for grain, sheep for wool, mountains for iron ore, hills for clay, and forests for wood. When a player receives resources, they are given a number of cards of the appropriate type. Players then spend these resources to build roads, settlements, or cities to expand their colony. There is another set of cards called Development Cards that represent other ways the colonies can develop. They are bought with resources in the same way roads and settlements are. Some of these cards allow a player to build two roads without paying the resource cost; others allow a player to claim a monopoly on a resource; and some of the developments are worth Victory Points. Most of the Development Cards are knights that allow the player to move the Robber Baron (a piece that prevents a hex from producing resources) and steal a resource at random from an opposing player. These knights are then added to that player's army.

Decisions to make during the game

There are many interesting and difficult choices that a player must make during the course of a game. On a player's turn, after she has rolled the dice and the resources for that roll have been distributed, she must decide what she wants to build next to expand her colony. There are many factors that affect this decision.

Long-term planning

The player must look at how building a particular piece like a road or settlement will affect her overall goal of achieving 10 Victory Points. In order to do this, the player must formulate a rough long-term plan for how she will attain the 10 points and then evaluate how building a particular piece contributes to this plan. This plan is rough because the environment is too unpredictable to be able to make a dependable long-term plan that a player could follow move for move for the rest of the game.

Short-term planning

In addition to the long-term goal of winning the game, a player also has many short-term goals such as reaching a spot on the board with her roads before another player, gaining the Largest Army award, or building a settlement on the coast to gain access to a trade port. Some of these short-term goals are less pressing, such as gaining the Largest Army award, as opposed to others like racing another player to a spot on the board. It can be difficult to decide which short-term goals can wait because a player won't know exactly what resources the other players have, what their short-term goals are, or how the dice will play out over the next few turns. But a player can make an educated guess by remembering what resources the other players have gained recently and watching their actions to predict what they're likely to do next. A player can also increase her knowledge by trading with other players to see what kinds of resources they want, and there's typically a lot of table talk during the game that may either clarify or obfuscate the other players' intentions depending on how skilled they are in the art of deception.

What to build next

To restate what I said before, in Settlers of Catan players are continually forming and modifying rough long-term plans for how to win the game. These long-term plans guide the composition of short-term plans to achieve subgoals in service of achieving the ultimate goal of winning the game. On a player's turn, the player must decide what action to take next. This is usually dictated by what that player wants to build next and his available resources. The decision of what to build next is determined by the player's current plans and his resources. Players must perform a tough balancing act between building something now using their available resources, or holding out and waiting for more resources to build something better later. If the player builds immediately, he can have the satisfaction of making a contribution to the size and effectiveness of his settlement even though it may be small compared to what he could build with more resources. If the player waits and holds on to his resources, he risks losing his resources to the robber (if a seven is rolled or another player plays a Knight card) or he may fall behind in the game because he underestimated how long it would take to get the resources to build what he wanted. Here's an example: It's Bob's turn to play and after rolling the dice he sees that he has three ore and a wood. Because Bob has a settlement on a 3:1 port, he could trade in the three ore for a clay which would allow him to build a road right now. On the other hand he could not build anything and wait until he collected two wheat so that he could build a city, which would increase his Victory Point total by one and increase his resource production. Knowing when to build and when to wait for a better opportunity is an important skill for playing Settlers of Catan.

Evaluating the other players' positions

In Settlers of Catan, it's important to be able to accurately evaluate the other players' positions to determine how close each one is to winning and what they need to do in order to win. This comes into play during negotiation when deciding on the value of an offer. A player doesn't want to give another player resources that will allow her to get significantly further ahead. This can happen when one player underestimates how close another player is to winning. Another reason it's important to be able to evaluate accurately is because sometimes it's more important to take actions that interfere with the current leader's plans to slow her down rather than concentrating on just expanding your own colony. This can take many forms including building roads to cut the leader off from part of the island, or buying Development Cards to try to take away the Largest Army Award. To estimate another player's position, you are trying to see what they need to do in order to win, and then guess at how long it will take them to do it. There are many reasons why this is difficult, the other player's resources and Development Cards are hidden (some Development Cards are Victory Points which remain hidden until the end of the game), and the outcome of the dice is unknown. But the layout of the other player's colony is public knowledge which provides all or most of their Victory Point total and by looking at where the other player's settlements and cities are built, one can guess at what kinds of resources they'll be receiving and how often.

Sometimes a player can see that another player is in the lead and will remain in the lead until the end of the game, justifying a change in priority to slowing that player down. But that player isn't in a position to make moves that interfere with the lead player. In this case, the player must look to see if another player is in a position to interfere and then decide if it's advantageous to help that other player take down the leader. If so, then he must convince the other player to cooperate and then figure out the best way to assist her without putting her too far in the lead.

What to trade for and whom to trade with

In order to build something, a player must have a specific combination of resources in her hand, but during the game these resources are distributed randomly. This means trading is necessary to build what you want quickly. It's possible to make trades with the bank or ports if you have access to them, but sometimes a better deal can be made with another player. Even though you are competing against the other players, each player has a different plan for how to win and also has different needs depending on his placement on the board. These facts mean that there is a lot of trading and negotiation during a game of Settlers of Catan.

There are a couple different strategies for deciding what kinds of offers would be beneficial to you and reasonable enough that another player would accept it. The easiest is to first determine what you want to build, what resources are necessary to build it, and what resources you have left over with which to trade. Then you simply ask if anyone is selling what you need. If so, you offer them some of your left over resources making sure not to make a deal that is too favorable for the other party. Also it's important to know what your alternatives are if your offer isn't accepted. For example, all players can trade four of one kind of resource for another with the bank, so it wouldn't make sense to offer four wheat to another player for one clay since you could make that trade with the bank and not put those resources in another player's hands.

A better strategy is to have a couple options in mind for what you want to build so that if no one is selling what you need to build your first choice, you can try to make a deal to build something else. For example, Sarah has a sheep, a wheat, and two wood and wants to build a settlement, so she offers to trade a wood for another player's clay. Another player says that they want two wood for the clay which would leave Sarah without wood and unable to build the settlement until she got more. Sarah changes the offer to two wood for an ore, which would allow her to buy a Development Card.

Finally, one can simply trade resources that are easy to come by for resources that are hard to come by without having a particular building piece in mind. This can be useful when it's not your turn and you don't have a clear plan. Since all of the resources are useful in some way, this strategy is a good fallback.

Sometimes players will get creative and make offers that involve things other than just resources. One possibility is to make a promise in addition to offering resources. These can take the form of declarations to not build on a certain part of the island, to not place the robber on a player's settlement, or to make a specific trade when it's their turn. This last kind of trade happens because a trade can only occur between the player whose turn it is and another player. Other players can't trade between themselves or the bank and ports, so if Bob wants a wheat and has two wood to trade he can trade it to Sarah who has a 2:1 wood port but she must promise to trade the wheat to him on her turn after she's uses the port to turn the two wood into a wheat.

Directing attention away from yourself

In Settlers of Catan, alliances form and break between the players during the course of the game. Usually three of the players will join forces to slow down the leading player. This means that it's important to not look like you are in the lead until you are in a position where the other players can't do much to stop you. There are a couple ways to do this.

One technique is to divert attention away from your colony by pointing out how well another player is doing and try to convince the other players that that player has the best potential for winning. This is usually done by describing a scenario that would allow the other player to win and then convincing the other players that this scenario will likely happen if they don't do anything to stop it. Here's an example of a typical argument: "Hey, Joe is set up to win. Look, he's already got six points and once he has the sheep port he'll be able to use his six and nine on sheep to get a bunch of clay to take Longest Road, since he already has good access to wood. So that's one point for the settlement on the port, two points for longest road, and then he just builds a city and he's won. We need to stop him from getting to that port."

Another technique is to gain points that aren't obvious to the other players. Some Development Cards are worth one Victory Point and are hidden from the other players until the end of the game. Also a player can build a long path of roads but leave breaks in it on purpose so that she doesn't receive the Longest Road award until she quickly connects them near the end of the game.

Most seasoned players use both techniques to try to covertly gain the ten points needed to win the game. A player who takes an obvious lead early will have to deal with the other players robbing him often, building roads and settlements to cut him off from other parts of the island, and refusing to trade with him.

Motivation

Throughout human history games have served a cultural entertainment role. With the advent of the computer, games have taken on new dimensions and become a part of our mainstream culture. One of the new dimensions the computer has brought to game playing is the potential for interactive game play through the use of non-player characters (NPCs). Traditionally these NPCs have been implemented in a variety of game types (adventure, strategy, first-person shooter, role-playing) through the use of scripts. These scripts however are limiting in their rigidity. In order to create more flexible and compelling characters with sophisticated behavior what is needed is a control technique that allows an NPC to decide a course of action given its current goals and situation. AI planning systems have been designed to address these very issues, but have yet to be used in any commercial games. This may be due to the complexity of these systems and large amount of computational resources needed to run them. This is a result of the disconnect between the goals of the AI researcher and the real world constraints faced by the computer game producer (Tozour 2002).

My view in this thesis is that by developing AI techniques that can be used to create agents that are flexible and can respond in a reasonable time frame given limited CPU resources, computer games can be made more compelling.

Decision-Making Using a Rough Plan

Our solution involves using a decision-making strategy that uses a rough plan to guide reactive behavior. In addressing the negotiation aspect of the game we used strategies adapted from business negotiation literature. The following is an example of how we use a rough plan to guide decision-making in the agents we have designed to play Settlers of Catan. In this example the patterned player on the top of figure 1.1 is deciding where to build next.

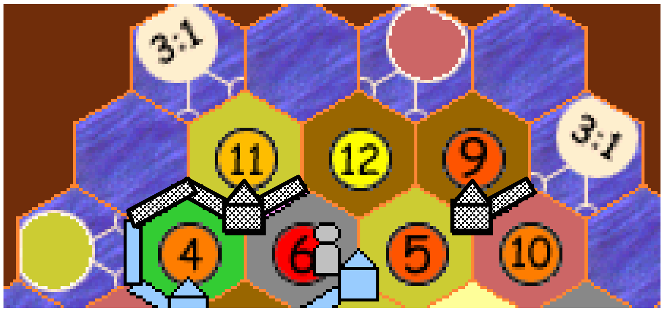


Figure 1.1 Example of deciding where to build

It is considering building either a settlement on the port on the left side, a road between the 6 and 12 hexes, or a city on the settlement at the 11-6-4 intersection. To determine where to build, the agent first estimates how long it will take to achieve its long-term goal of gaining 10 Victory Points. It does this by composing a plan consisting of abstracted steps. Then it estimates how long each of these steps takes to implement and takes the sum to arrive at an estimate of how long it will take to implement the whole plan. This calculation doesn't take interference from other players into account, but this will be addressed shortly. Table 1.1 shows the estimated time to gain 10 Victory Points for the patterned player.

Original Estimate	
Build 2 Cities	33 turns
Build 2 Settlements	18 turns
Take Longest Road	14 turns
Take Largest Army	15 turns
Total Estimate	80 turns

Table 1.1 Estimated Time to Complete Goal

Build a Settlement: 0 turns		Build a Road: 0 turns		Build a City: 21 turns	
Build 2 Cities	33 turns	Build 2 Cities	33 turns	Build Settlement and City	18 turns
Build 2 Settlements	22 turns	Build 2 Settlements	18 turns	Build 2 Settlements	22 turns
Take Longest Road	8 turns	Take Longest Road	7 turns	Take Longest Road	8 turns
Build a City	8 turns	Take Largest Army	15 turns	Build a City	8 turns
Total Estimate	71 turns	Total Estimate	73 turns	Total Estimate	77 turns

Table 1.2 Estimates After Building

Table 1.2 shows the estimates for gaining 10 Victory Points after building each choice that the player is considering. The player has the resources to build a settlement or road on this turn, but must wait to build a city. By comparing the difference in total estimates before and after each choice, the agent can get an idea of which move is best. In this case, building a settlement reduces the estimated time to complete the plan by the most, and the player can build a settlement now, so it is the best choice. We also want to consider how our choice will affect other players. We can use the same technique to estimate the impact of this move on other players. Table 1.3 shows estimates for the other player in figure 1.1 before and after the patterned player builds a settlement on the left side. Because that player was counting on building on that spot, after the settlement is built a new plan must be used. This new plan takes longer to implement than the original, so by building a settlement the patterned player is slowing the other player down.

Before		After	
Build a Settlement and City	29 turns	Build 2 Cities	33 turns
Build 2 Cities	24 turns	Build 2 Settlements	28 turns
Take Largest Army	15 turns	Take Largest Army	15 turns
Take Longest Road	18 turns	Build 2 Cities	16 turns
Total Estimate	86 turns	Total Estimate	92 turns

Table 1.3 Estimates for Another Player

If after deciding what we want to build, our agent doesn't have the necessary resources, it can negotiate with other players in an attempt to get those resources. It determines what it will do if it can reach an agreement with anyone and then uses that to guide evaluation of offers that are put on the table. If an offer is good enough and the other player isn't in a position to win right away or build something that will slow the agent down, it will take it.

We have implemented and tested these ideas on an on-line version of Settlers of Catan called Java Settlers. By conservative estimates, Java Settlers currently receives over 10,000 visitors a month who play at least one full game. A competition ladder has also been created by fans of the site, and it has been featured in PC World magazine (Aquino 2002). Using this platform, we have gathered data on thousands of games played with every combination of number of people verses agents. Our results show that when one person is playing against three agents, on average the person will win about half of the time. We believe this is a good level of competence especially since about half of the people that use the system play against three agents. The other half tends to play only with other people. When negotiation is not permitted, a person playing with three agents will win less than half the time. We believe this is evidence that our initial approach to negotiation hinders our agent's ability to win rather than being an expected enhancement. Further work is required to improve this aspect of the agent's behavior.

Challenges in the Java Settlers project

The Java Settlers project is the name I have given to the endeavor of translating the board game Settlers of Catan into a version that can be easily played online with other people, intelligent computer opponents, or a combination of both. The project also entails creating data recording utilities in order to observe the behavior of people using the system as well as the behavior of the computer opponents and the system itself. There were many challenges that had to be addressed during the creation of the system.

At the core of the Java Settlers project is a large-scale online multiplayer game. Creating a stable multiplayer game capable of supporting hundreds of users simultaneously has many challenges of its own. These include creating a robust client-server architecture that can handle different traffic loads gracefully, handling synchronization issues between different players, making a client that can be used on many different platforms and looks and acts the same across these platforms, designing an interface that is easy to use and delivers all of the information that is normally available when playing the game face-to-face.

In addition to being a source of entertainment, this multiplayer game is an environment for doing AI research. This research involves the creation of intelligent computer opponents that can hold their own against other people playing Settlers of Catan using the Java Settlers system. Creating these opponents has its own challenges that come from the kinds of behavior that the game demands of its players. These are behaviors like managing a limited supply of resources, making and following plans in a hostile dynamic environment, negotiating with other self-interested agents, hypothesizing about other agents' goals and plans, and making and understanding arguments about these hypotheses with other players.

Finally after the system has been created, there are challenges as to how to best use the system. Can it be used to train people to become better negotiators, or to become better strategic thinkers? How can the system be used to develop better, more sophisticated AI programming techniques?

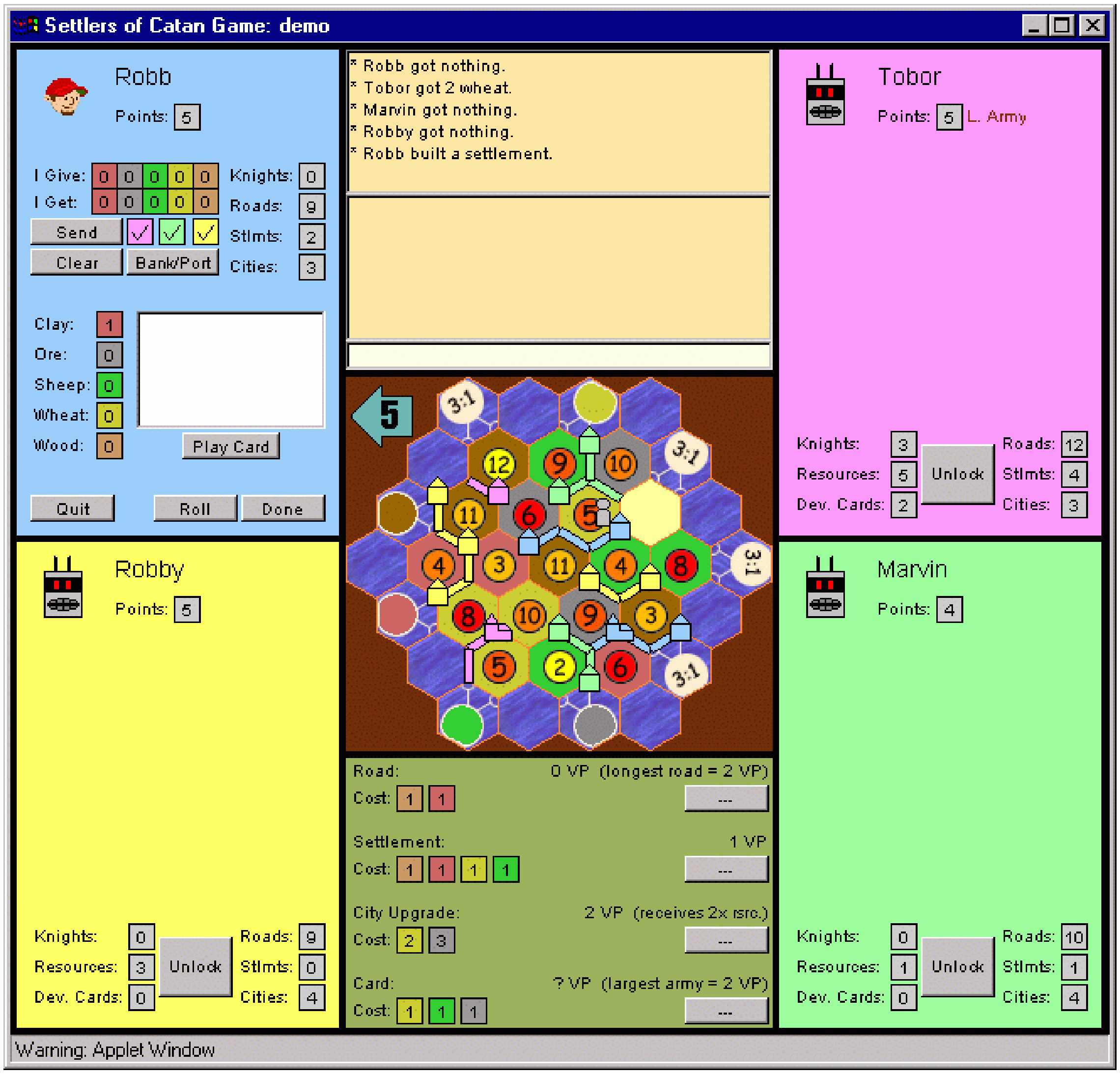
Overview of this thesis

The rest of this dissertation will describe the Java Settlers project on a technical level including the design process that went on during its creation, how each of it's parts works, and the methods used to create the intelligent computer opponents for the system. I will also attempt to answer some of the questions I have just sketched out in the previous section. Chapter 2 describes the implementation of the Java Settlers system. Chapters 3, 4 and 5 describe the implementation of a competent player for Settlers of Catan. These chapters also include experimental results from testing the algorithms employed in the agent. Chapter 6 addresses related work in the areas of reactive planning and utility theory. Chapter 7 summarizes the results of the project and discusses future work.

Chapter 2
Design and Implementation of a Large-scale Online Multiplayer Game

Java Settlers is meant to recreate the experience of playing the board game Settlers of Catan. Our design goals were to allow people to easily access the system from any platform that supported a web browser and an Internet connection, and to get people playing with a minimum of time spent learning the interface. This chapter describes the user interface for the player client as well as the representations used in the Java Settlers system and its overall architecture. By using an already existing chat program as a starting point, we were able to quickly implement a system that allowed us to gather data on people playing against each other. We also discovered that small changes to the interface that make actions more prominent in the appropriate context have a noticeable affect on user behavior.

The Graphic Interface



Settlers of Catan Game: demo

Robb

Points: 5

I Give: 0 0 0 0 0

I Get: 0 0 0 0 0

Knights: 0

Roads: 9

Stlmts: 2

Cities: 3

Clay: 1

Ore: 0

Sheep: 0

Wheat: 0

Wood: 0

* Robb got nothing.
* Tobor got 2 wheat.
* Marvin got nothing.
* Robby got nothing.
* Robb built a settlement.

Tobor

Points: 5 L. Army

Knights: 3

Resources: 5

Dev. Cards: 2

Roads: 12

Stlmts: 4

Cities: 3

Robby

Points: 5

Knights: 0

Resources: 3

Dev. Cards: 0

Roads: 9

Stlmts: 0

Cities: 4

Marvin

Points: 4

Knights: 0

Resources: 1

Dev. Cards: 0

Roads: 10

Stlmts: 1

Cities: 4

Road: 0 VP (longest road = 2 VP)

Cost: 1 1

Settlement: 1 VP

Cost: 1 1 1 1

City Upgrade: 2 VP (receives 2x rsrc.)

Cost: 2 3

Card: ? VP (largest army = 2 VP)

Cost: 1 1 1

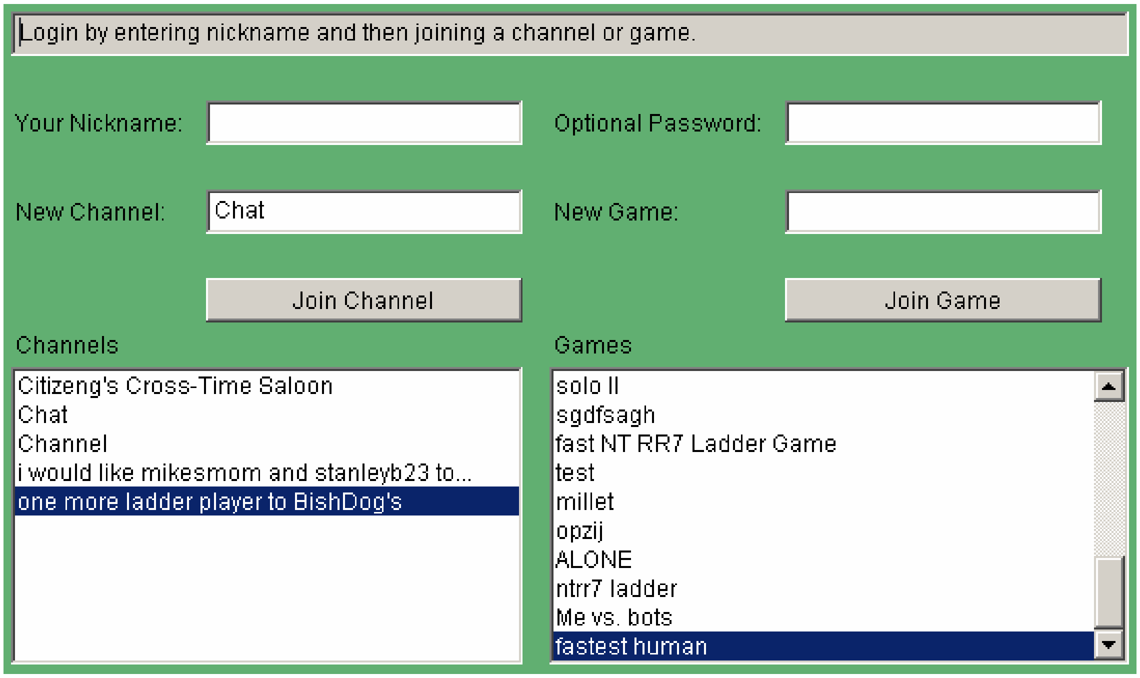
Warning: Applet Window

Figure 2.1 Game Interface After a Few Turns

The Login Area

When someone loads the client applet into their web browser, the first thing they see is the login area (figure 2.2). Before a user can join or create a game or channel, she must first enter a nickname for herself in the field next to the label “Your Nickname:”. If this person has reserved a nickname for herself using the account client, then she also needs to enter her password in the field labeled “Optional Password:”. I didn’t want to force users to create an account before using the system because I wanted to allow people to play with the least amount of work on their part. If they liked the system and wanted to keep playing under the same name, they could quickly reserve their nickname using the account client.

After entering a nickname, the user can create or join a game or channel. To join an existing game or channel, the user can select the name of the area she wants to join and press the appropriate join button, or she can simply double-click on the name. To create a new game or channel, the user types in the name of the new area and clicks the appropriate join button.



Login by entering nickname and then joining a channel or game.

New Channel: Chat

Channels

- Citizeng's Cross-Time Saloon
- Chat
- Channel
- i would like mikesmom and stanleyb23 to...
- one more ladder player to BishDog's

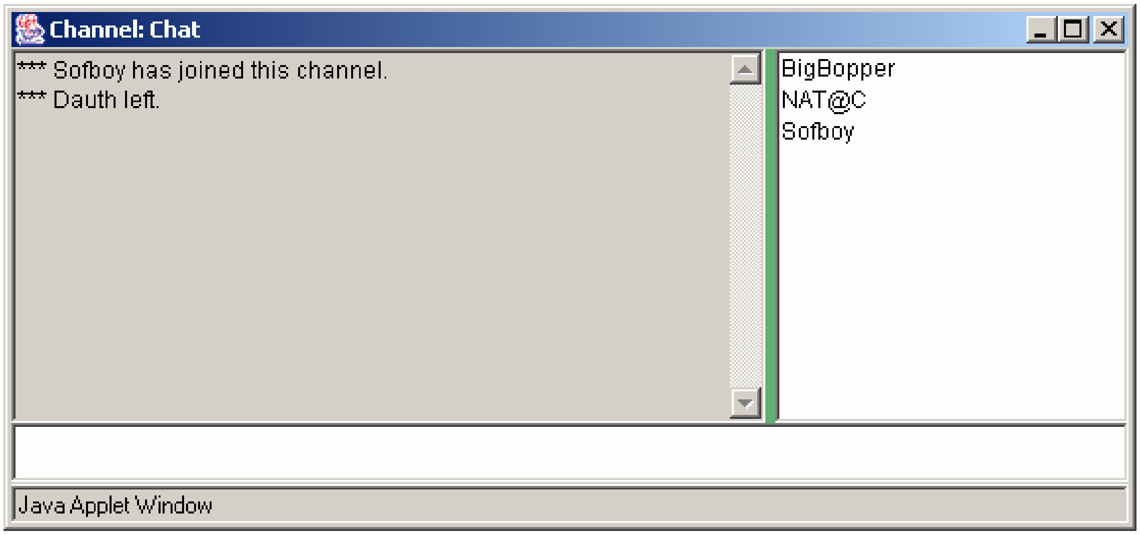
Games

- solo II
- sgdfsagh
- fast NT RR7 Ladder Game
- test
- millet
- opzij
- ALONE
- ntrr7 ladder
- Me vs. bots
- fastest human

Figure 2.2 Login Area

The Chat Area

If the user joins a chat area, a separate window is displayed showing the chat interface (figure 2.3). To talk to the other people, the user just types in the bottom field and hits the Enter key. Closing the window leaves the chat channel. This is typically used as a place for players to gather before starting a game, or for a place to just chat.



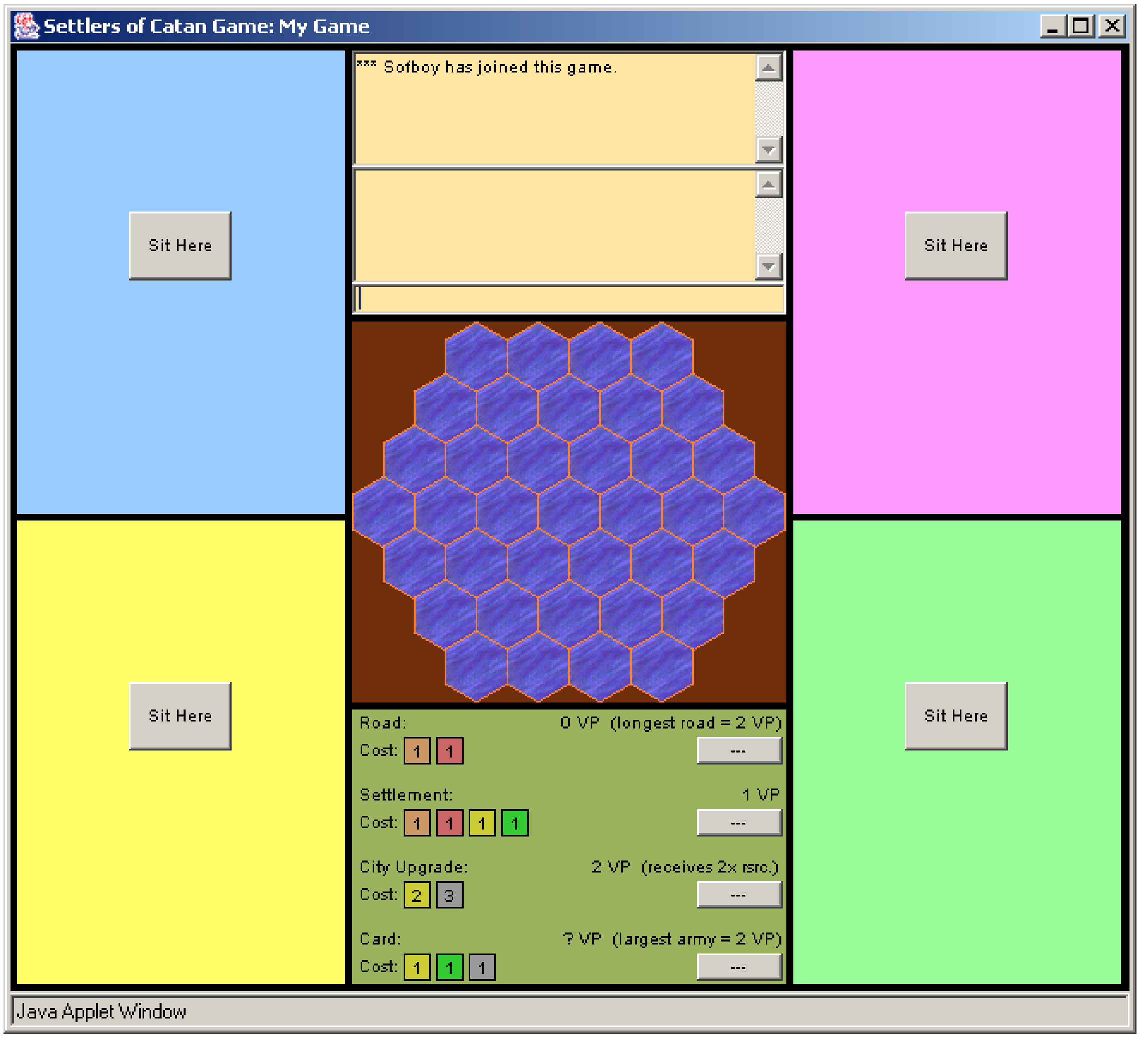
*** Sofboy has joined this channel.
*** Dauth left.

BigBopper
NAT@C
Sofboy

Figure 2.3 Chat Area

The Game Area

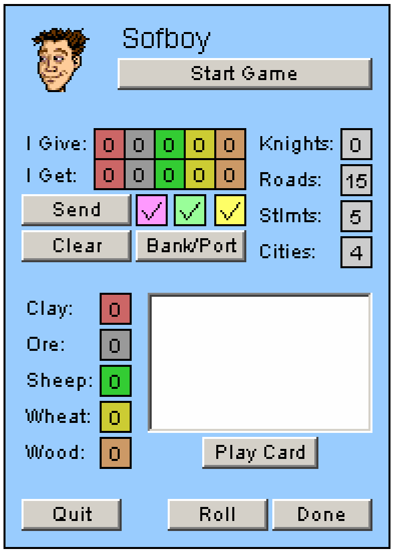
If the user joins a game area a separate window is displayed showing the game interface (figure 2.4). The window is divided into a couple different regions. In the four corners are the player information regions. This is where information about the player such as how many pieces they have left to play and how many cards they have is displayed. Before the game starts, there is a button in the middle of each area that is vacant labeled "Sit Here". If the user wishes to play the game rather than observe, they choose their color by pressing the "Sit Here" button, after which they are shown their player information and also presented with some more buttons (figure 2.5).



*** Sofboy has joined this game.

Figure 2.4 Game Area

In the center of the game window is the game board. Before the game begins all of the hexes are water hexes. After someone starts the game by clicking the "Start Game" button, the board is created according to the rules by shuffling the land hexes, laying them out in a large hexagon surrounded by the port hexes, and then placing the numbers on the land hexes. The details of how the board is laid out can be found in appendix B. Figure 2.1 shows what the game interface looks like after a few turns.



I Give: 0 0 0 0 0
I Get: 0 0 0 0 0
☑ ☑ ☑

Knights: 0
Roads: 15
Stlmts: 5
Cities: 4

Clay: 0
Ore: 0
Sheep: 0
Wheat: 0
Wood: 0

Figure 2.5 Your Player Area

Above the game board is the chat and information region (figures 2.1 and 2.8). The upper section displays messages from the server about the game and who is entering or leaving the game area. The lower section displays chat messages from other people in the game area. To send a message, the user types into the text field below the chat display section and presses the Enter key.

Below the game board is the building region (figure 2.6). This part of the window shows what resources are required in order to build or buy a development card. It also serves as the interface to perform those actions. The colored boxes with numbers in them state what combination of resources is needed to buy each item. The color of the box corresponds to the type of resource and the number indicates how much. These colored boxes are used throughout the interface. On the right side of this region are buttons with context-sensitive labels that tell the player what items are available to him. These buttons are labeled according to the kind of resources he has, if he has pieces left to play, and if there is a spot on the board to place that kind of piece. If the player meets all of the necessary preconditions to buy an item, the button next to it is labeled “Buy”, otherwise it’s labeled with dashes.

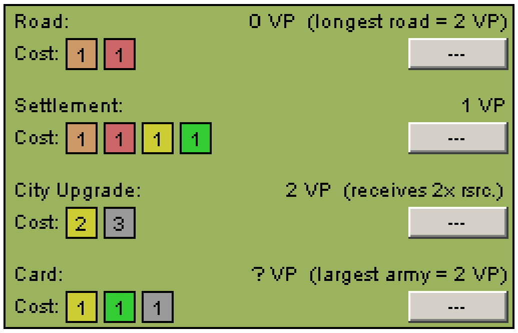


Figure 2.6 Building Area

Your Player Area

I will now describe the player area in more detail. If you are playing the game rather than observing it, your player area will look similar to figure 2.5. This region of the game area displays the player's resources, development cards, building pieces, the size of her army and how many victory points she has. If she has the Longest Road or Largest Army awards, they will appear here also. The player's resources are displayed using the same colored boxes with numbers that are found in the building region. This display also doubles as a key to show what colors correspond to which resource. The development cards are shown as a list next to the resources. To play a development card, the player selects one from the list and clicks the "Play Card" button, or simply double-clicks on the card name. Below this there are three buttons labeled "Quit", "Roll", and "Done". If the player wants to leave the game, she clicks the "Quit" button.

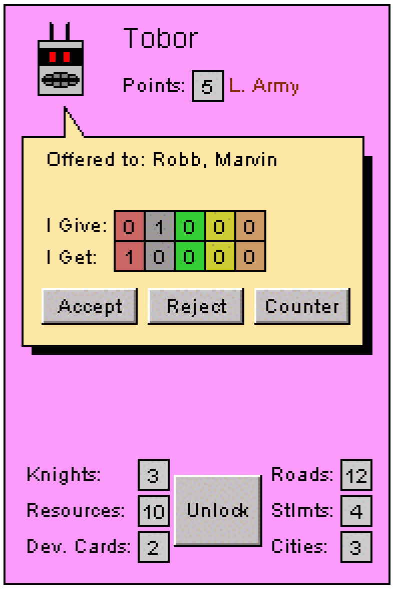
When it’s her turn to roll the dice, she clicks the “Roll” button. This isn’t automatically done at the start of a player’s turn because one can play a development card before rolling the dice. When the player is done with her turn, she clicks the “Done” button.

Above the resource and development card displays is the trading interface, a list of how many pieces they have available to place on the board and the size of their army. The trading interface provides a way to create trade offers and send them to the other players with a few mouse clicks. At the top of the trading interface are two rows of colored squares with numbers in them. These are similar to the colored boxes with numbers that are found in other parts of the interface. The top row of boxes is labeled, “I Give:” and represents what resources this player is offering in exchange for what he wants. The second row is labeled, “I Get:” and represents what resources this player wants to receive. To create an offer, the player clicks on the boxes to increment the values of the numbers. For example, to make the offer, “I’ll give you two wheat for one clay”, the player would click the top yellow box twice and the bottom red box once. If the player makes a mistake, he can click the “Clear” button to reset the offer. To make the offer to all of the other players, the player clicks the “Send” button. Sometimes a player will only want to make an offer to certain players. The three colored checkboxes next to the send button determine whom the offer will be sent to. A check means that the offer will be sent to the player with the corresponding color. If it's not the sending player's turn, the offer will automatically be made to the player who's turn it is, since trades can only be made with that player. Finally, if the player wants to trade with the bank or a port, he simply makes a legal offer and clicks the "Bank/Port" button. If the offer isn't a legal one, he's notified in the game message window.

At the top of the player area is the player's icon, nickname, victory point total and any awards they have. A player can change their icon by clicking on it. Clicking on the right half of the image cycles forward through the available faces while clicking on the left half cycles backwards.

Another Player's Area

The player areas for the other players look similar, but only display publicly available information (figure 2.7). This includes the size of their army, how many pieces they have available to place on the board, total number of resources, total number of development cards, and their victory point total minus any victory point cards they have.



Tobor

Points: 5 L. Army

Offered to: Robb, Marvin

I Give: 0 1 0 0 0

I Get: 1 0 0 0 0

Knights: 3

Resources: 10

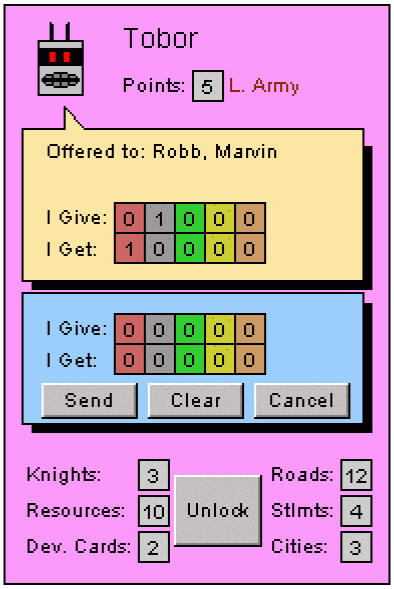
Dev. Cards: 2

Roads: 12

Stlmts: 4

Cities: 3

Figure 2.7 Other Player's Area



Tobor

Points: 5 L. Army

Offered to: Robb, Marvin

I Give: 0 1 0 0 0

I Get: 1 0 0 0 0

I Give: 0 0 0 0 0

I Get: 0 0 0 0 0

Knights: 3

Resources: 10

Dev. Cards: 2

Roads: 12

Stlmts: 4

Cities: 3

Figure 2.8 Making an offer

When another player makes an offer, a speech balloon appears to show which player is making the offer and what it is. In figure 2.8, Tobor has made an offer to trade one ore for one clay from either Robb or Marvin. The buttons below the offer allow a player to quickly respond in one of three ways: accept, reject, or make a counter offer. If the "Reject" button is clicked, the other players see a speech balloon appear in your player area with the message, "No Thanks." If the "Accept" button is clicked, the deal is made and a message appears in the game message window describing who got what. If the "Counter" button is pressed, a copy of the trading interface appears below the offer.

This interface works in the same way as the one in the player's player area. This counter button was added as an experiment in response to the observation that people were not making as many counter offers as people playing the game face-to-face. Before the addition of the counter button, people made an average of .01 counter offers per offer. After this button was added, counter offers increased by 30%. This is evidence that providing the user with choices appropriate to the context will affect their behavior. This experiment and its implications on negotiation are discussed in more detail in chapter 5.

Game Representation

The next important piece to creating a multi-player game is how the game elements are represented. The way these elements are represented in code determines how easy or difficult it will be to manipulate them when players take actions in the game environment. Also, picking the right representations will shape the way the AI opponents will be built, because the kind of information the AI system has access to and how quickly it can be accessed is determined by its representation.

Board Representation

The most challenging piece of representation was making a coordinate system for the board. Since the board is made up of hexes, a Cartesian coordinate system where the axes are at right angles would not be the most intuitive to use. Also, we were not only interested in the location of a hex, but also of the edges and nodes between the hexes, and we needed to be able to calculate the coordinates of any hex, edge, or node relative to another hex, edge, or node.

We started by labeling the hexes using two axes at a 120° angle starting at the left most center hex on the board. One axis is in the northeast direction and the other is in the southeast direction (figure 2.9). We assumed that we would be doing a lot of calculations using this coordinate system, so we wanted the representation to be compact and allow calculations to be performed using a minimum amount of computation. With these goals in mind, we created a numbering scheme where all coordinates were represented by two hexadecimal digits. Distance along the Southeast axis is represented by the left or most significant digit while distance along the Northeast axis is represented by the right or least significant digit (figure 2.9). Using this scheme we can only represent boards of a certain size, but the Settlers of Catan board is within this limit. If we needed to represent a larger board with more hexes, we would use a pair of integers for each coordinate.

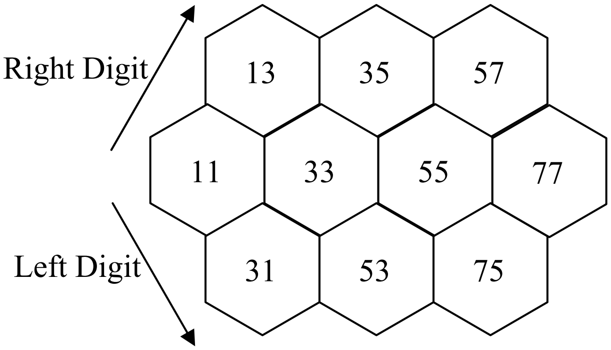


Figure 2.9 Hex Coordinate System

For the hex coordinates, the increment along each axis is by two. For example, if you want to get the coordinates of the hex immediately to the Southeast of hex 11, you add 20. Incrementing by two simplifies the task of calculating the coordinates of an edge or a node given the coordinates of a hex. Edges are labeled in a similar way where the left digit represents distance in the Southeast direction and the right represents distance in the Northeast direction (figure 2.10).

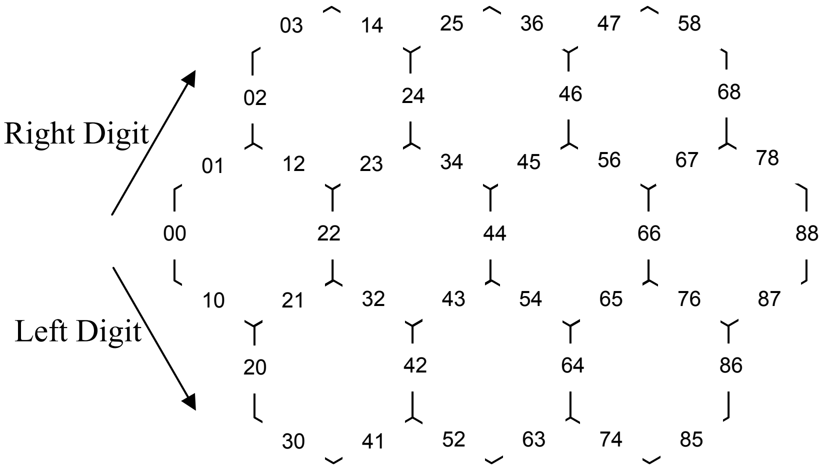


Figure 2.10 Edge Coordinate System

Finally, the nodes are labeled using the same method (figure 2.11).

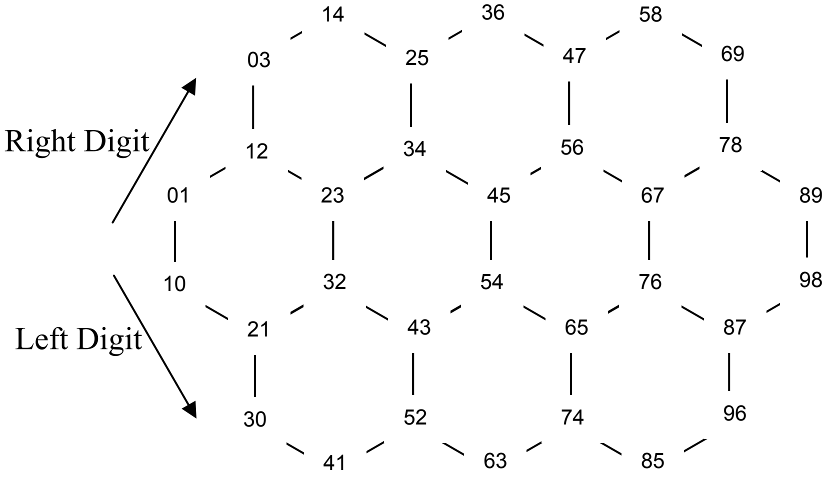


Figure 2.11 Node Coordinate System

Using this coordinate scheme it's very easy to calculate the coordinates of a hex, edge, or node relative to another hex, edge, or node. This is important for a number of reasons in the game. For example, after rolling the dice players receive resources if they have a city or a settlement touching a hex with the matching number. One way to figure out who gets resources is by looking at the coordinates of a hex that has a number matching the dice and then checking the coordinates of each of the players' settlements and cities to see if they are touching that hex. Figure 2.12 shows how to calculate the coordinates of any node or edge around a hex by adding or subtracting from the left and right digits of the hex coordinate. For example, to get the coordinates of the top most node of a hex at coordinates 11, just add 0 to the first digit and 1 to the second to get 12.

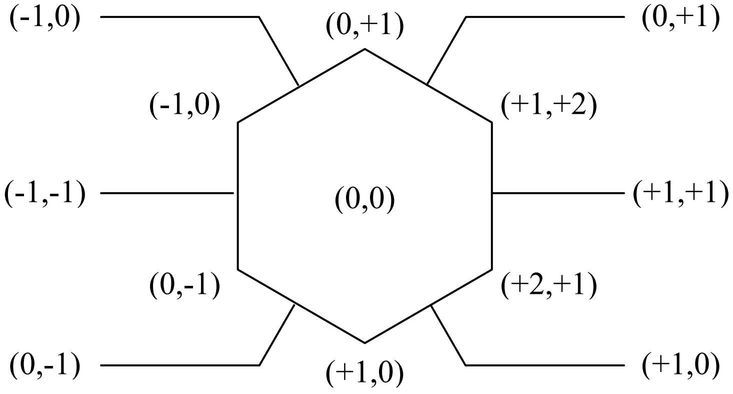


Figure 2.12 Relative Coordinates

Similarly, the coordinates of any one element can be calculated given the coordinates of another. Appendix A contains a summary of how this is done as well as a coordinate diagram of the Settlers of Catan board.

Other game elements

After settling on a coordinate scheme for the board, representing the rest of the game elements was simple. The board is an object that contains two lists of integers representing the layout of the hexes and numbers. There is a third list that translates from a spot in the list to its hex coordinate. This makes looking up a hex land type or number simple and fast. The board object also tracks the coordinates of the robber. Since only the location of the robber is important, this seemed sensible.

For the player's pieces, we made a class for all of the pieces and then made subclasses for roads, settlements, and cities. Each of these contains a coordinate and a reference to the player that owns the piece.

Resources are represented as sets. This made the most sense because of the way resources are used. For example, to build a road a player needs to see if her hand contains the subset {clay, wood}, if so then she has the resources to build a road. By subtracting the resources needed to build something from your hand, you can see what resources are available for trading. Sets are also used when trading with another player, e.g. “I’ll give you a sheep and a wheat for your two clay”.

Development cards are also represented as sets. One thing that makes development cards different than resources is that it’s important to distinguish between cards that have been bought on this turn and cards that have been bought on previous turns. We keep track of this by marking each card as either new or old. New cards have been bought on this turn and old cards have been bought on previous turns. At the beginning of a turn, a method is called on the current player’s development card set that marks all new cards as old.

The dice are represented as part of the game object. The game object tracks the state of the game and also contains methods for performing game actions. The state of the game determines what kinds of actions a player can take. For example, at the beginning of a player’s regular turn, she can either play a development card from her hand or roll the dice but nothing else. A finite state machine is used to transition between the states as the players take different actions. This means that the game object must be notified about any actions that affect the game. Since this is the case we made a game object method for every possible kind of action a player could take. These methods update the game state and make any necessary changes to the game elements. The game object also contains methods for checking if an action is legal before trying to perform it.

In addition to having variables for tracking the state of the game and the value of the dice, the game object contains a board object and four player objects. The board object was described earlier. The player object keeps track of everything that has to do with a particular player. This includes the player's resources and development cards, how many victory points they have, how many knights they've played, the length of their longest road, and legal places to build. The resources and development cards are represented as sets that were described earlier. Victory points are calculated by counting the number of settlements and cities that player has built, adding one point for settlements and two points for cities. Then if this player has the longest road and/or the largest army, he gets two points for each of those. Finally any victory point cards in the player's hand are added to a separate private victory point total, this is to distinguish it from the public victory point total. Calculating the length of the longest road and tracking where that player can build is a little more involved.

The length of the longest road for a player is calculated by finding the longest single chain of continuous road pieces and then counting the number of pieces in that chain. This chain can be broken by another player if they build a settlement on a node along the path of the road. Figure 2.13 illustrates an example. Here, the length of Amy's longest road is 7 going from the node marked A to the node marked C. This is longer than Betty's longest road which has a length of 6. If Betty builds a settlement at the node marked B, Amy's road will be cut leaving her with a longest road length of 4.

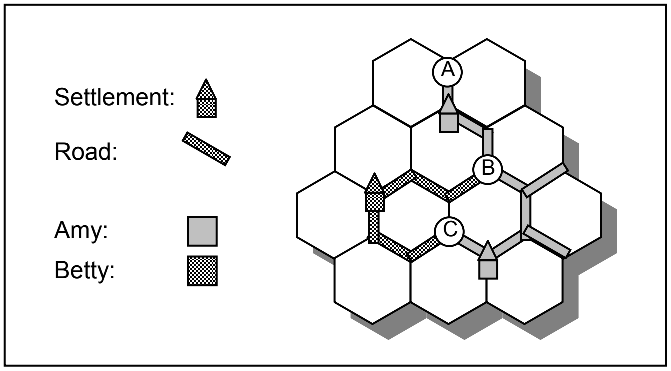


Figure 2.13 Example of Longest Road

The algorithm for calculating a player's longest road performs a depth-first search of all possible paths starting from every node that touches one of the player's roads. The reason we need to search starting from every node is because the length of the longest path changes depending on where you start the search. For example, in figure 2.13 if we start all of our paths at the node marked B, the length of the longest path for Amy is 4.

In addition to knowing the length of a player's longest road we also need to know where they can build. This information is used by the player client to highlight legal spots on the board when a player is about to place a piece. Also, the server uses it to prevent illegal actions, and the computer players use it to determine where to build. This information is represented by arrays of bits in the player object. For each kind of thing a player can build there is an array A such that A[c] = 1 if c are the coordinates of a place where that player can build that thing, otherwise A[c] = 0. These arrays we'll call potential arrays because they track where the player may potentially build. This is also to distinguish them from the arrays that track the places on the board where a player could never build. These arrays are called the legal arrays. When a player object is created, the potential and legal arrays are initialized. If the game hasn't started yet, the potential road and city arrays are set to all zeros and the potential settlement array is set to match the legal settlement array. This means that that player can build a settlement on any legal spot on the board. As players place their initial settlements, the legal and potential settlement arrays are updated. When a settlement is placed, the node where the settlement was built and nodes adjacent to the settlement are no longer legal places to build, so those entries in the legal and potential arrays are set to zero. Also, the entries for the edges touching that settlement in the potential road array are set to one as long as there is a one in the same entry in the legal array. Finally the entry in the potential city array with the same coordinates as the settlement is set to one.

After the initial placement phase, the potential settlement array is reset to all zeros because players must now build roads before building their next settlement. During the rest of the game, as players build the potential arrays are updated according to the rules of the game. When a player builds a road, the nodes adjacent to the road edge are set to one in the potential settlement array as long as the corresponding nodes in the legal settlement array are one. Also the legal and potential road arrays are updated so that a player can't build twice on the same edge. When a player builds a settlement, the legal and potential settlement arrays are updated in the same way as during the initial placement phase. When a player builds a city, the entry for that city in the potential city array is set to zero.

System Architecture

There are many different kinds of online multiplayer games and each of them has different requirements (Smed 2002). Java Settlers is meant to allow people to have the experience of playing a board game, so the latency and bandwidth requirements are fairly low. What is important is that the order of the messages is preserved and that all of the players in a game have an accurate representation of what's happening. A simple client/server model is sufficient for the purpose of simulating a multiplayer board game across a network.

The Game Server

The main job of the Java Settlers server is to keep track of the two types of areas where users connected to the server can interact, and to handle messages in each area in the appropriate way. The two types of areas are games and channels. Game areas are where users can play the game Settlers of Catan or watch others play. Channels are areas where people can gather before starting a game or just to chat. New games and channels are created by users as they need them, and are destroyed when everyone leaves a particular game or channel. Message handling for both areas is pretty simple. In the channel area, incoming text messages are echoed to the other clients in that area. The same thing is true for text messages in the game area. In addition, messages for game actions are first checked to see if they're legal. Then if they are, the appropriate method for taking that action is called on the game object for the area. Finally the results of the action are reported to the clients in that game area.

The server also controls the dispatch of the computer controlled players. If a user starts a game with fewer than four people sitting at the game area, the server will notify a number of computer players to fill the remaining seats. Computer players connect to the server as regular players do, but they use a different client. Once a computer player receives a request to join a game, it goes to that game, sits down and plays. A computer player will also receive a request to play if a human player leaves the game before it's over.

Human players can replace computer controlled players by clicking a button in the computer player's area. This ability to boot a computer player can be disabled by anyone that is already seated. This was added because a number of people only wanted to play against the computer and were frustrated by people who would boot a computer player despite being asked not to.

The Player and Robot Clients

The client that people use to connect to the Java Settlers server is called the player client. There is another client for computer controlled players called the robot client. Both clients handle messages in the same way. When a message comes in, the appropriate function is called to handle the message. This usually results in a call to a game object to update its data and a call to an interface object to update its display. When a user presses a button or types into a field in the client interface, a message is sent to the server to indicate what the user is doing.

The difference between the player and robot clients is that the robot client doesn't join any game it wants to, rather it waits for requests from the server. When a request message is received, the robot client sends a message to the server requesting to join the game it was asked to join. After the robot client receives authorization, the robot client creates a robotBrain object to play that game and joins it. The robotBrain object monitors the messages received by the robot client for that game and decides what actions to take based on the current state of the game and it's current goals. When a robot client is asked to join a game in progress, the client receives information about the current state of the game and decides what to do from there as if it was playing the game from the beginning.

Networking Code

In order to make an online multiplayer game, you need a way to pass messages between the people playing, and a way to track the state of the game. The two most popular architectures for this type of networked application are peer-to-peer and client/server (Smed 2002). We decided that a client/server model was the best choice because we wanted to be able to collect and store data on what went on during games that people played. Having a central server made the data collection task easier. We also wanted to allow people to access the game in the simplest way possible. By making the client a Java applet embedded in a web page, anyone who wanted to access the system could do so by simply accessing the Java Settlers page using a web browser.

We started development by finding a chat program that had already been developed in Java (Christian 1996). This quickly took care of how to pass messages between the client and the server. The code for the chat system is relatively small and was written so that it could be used for applications other than chatting. This made it easy to adapt it for my purpose.

The chat program consists of three basic parts: a server application, a Connection class, and a client applet. The server application is an implementation of an abstract Server class. The Server class is a thread that creates a socket on which it listens for clients connecting to it. Once a client has connected to the listening socket, the server creates a new Connection object containing a new socket for sending messages between the client and the server and adds it to a list of Connections that are connected to the Server. The Connection object is a thread that reads messages off of its socket and then calls the Server's *treat* method with that message as an argument. The Server's *treat* method is an abstract method that does something with the message it receives. In the case of the chat server the *treat* method sends the text of the message to the other clients connected to the server. To send a message to the client on the other end of a Connection, the server application simply calls the *put* method of that Connection with the message as an argument.

If a problem occurs while trying to read or write to a Connection's socket, the Connection object calls the Server's *removeConnection* method passing itself as an argument. This method removes the Connection from its list of connected Connection objects, calls the Connection's *disconnect* method to close the socket and then calls the *leaveConnection* method which is an abstract method that does what is necessary when a connection leaves. In the case of the chat server it sends a message to the other clients stating who left the chat server.

The client applet isn't based on an abstract Client class. Rather it is a simple program that opens a socket to the server and then sends messages based on the user's interactions with the interface. Because the client code isn't dependent on any of the Server or Connection classes, it makes it easier to create a client using a language other than Java. The only thing that needs to agree between the client and the server is the format of the messages.

This code provided a good starting point for creating an online multiplayer board game, but as the Java Settlers site became more popular and traffic increased, the code needed to be modified to handle the load. One thing that would happen was that the server would get into a deadlocked state and freeze. This was due to a couple of reasons. One was when a client broke its connection with the server, the *leaveConnection* method would call other methods that would try to send messages across the Connection that was leaving. In the Connection class, the *put* method used to send messages across the Connection was synchronized. Also, this method would call the *removeConnection* method in the Server when it had a problem sending a message. The *removeConnection* method was also synchronized. All of these synchronized methods created a situation where deadlock could occur. To handle this problem and the increased traffic load, we added threads that would maintain queues of incoming and outgoing messages for each

Connection. This solved the deadlock problem by not making it necessary to hold the lock on the Connection when calling the *leaveConnection* method.

Summary

In this chapter I've described how the Java Settlers server works along with the player and robot clients. I've also described the underlying representation of the game elements as well as a detailed explanation of the client's user interface. The basic method for creating the system was to start with a working chat system and then adopt it to fit the needs of the Java Settlers project by incrementally adding more and more features. This method works well when the code that you're starting with has been designed to be extensible. The core code may have to be modified to handle situations, like a greater load, than the code was originally designed for, but starting with a solid core program can save development time.

Take Aways

- Adapting code that has already been written for a similar purpose can speed up development time.
- Small changes to the user interface that make actions more prominent in the appropriate context have a noticeable affect on user behavior.

In the next chapter I will describe how the AI for the computer opponents works and the design decisions that were made when creating it.

Chapter 3
Agent Implementation Part I: Determining Options & Resource Estimation of Time

This chapter lays the ground work for creating a competent Settlers of Catan player. In Settlers of Catan, a player's time is spent mostly thinking about where to build next. Knowing what you want to build next helps you decide what you want to trade for or if you want to trade at all. But deciding what to build next and where to build it involves many factors. In this chapter we will outline some of the basic strategies for how a player decides where to place her initial settlements and roads. We will show that these strategies are related by the fact that they seek to maximize the speed at which a player can build. From this we conclude that the ability to estimate a players building speed is critical to creating a Settlers of Catan player. We also show that a relatively simple technique that gives us relatively accurate results quickly is preferable to a slower technique that provides more precise estimates.

Legal places to build

Before deciding what and where to build, one must know all of the possible options that are under consideration. In Settlers of Catan the set of all legal places to build is determined by a few simple rules (Smith 1996):

- Cities can only be built on your existing settlements.
- Settlements can only be built on intersections that touch one of your roads and are not adjacent to another settlement or city.
- Roads can only be built on edges that are touching one of your roads, settlements, or cities.
- You cannot place a piece where there is already a piece except in the case of one of your own cities replacing a settlement.

Because pieces stay on the board once they're placed (except in the case of a city replacing a settlement), we can build up a representation of legal places to build and update it as necessary. This will save us a lot of computation time as opposed to recalculating all legal moves every time we want to make a decision about where to build next. If the board were more dynamic, i.e. the legal places to build changed from moment to moment; recalculating might be better than incrementally updating a persistent representation.

The SOCPlayer object maintains this representation because where a player can build next is a property of that player. For both settlements and roads, two lists are maintained: a list of legal places and a list of potential places to build that piece. Legal places are spots on the board where it is possible to build that kind of piece if the building prerequisites are met. Potential places are legal spots on the board where the building prerequisites are met. Here's an example to illustrate the difference between legal and potential building spots: all edges on the board that don't have a road on them are *legal* spots to build a road; legal edges that are touching your roads are *potential* spots for you to build. We don't need to maintain these lists for cities because they are equivalent to our list of settlements on the board.

The lists are implemented as arrays of Boolean values. If a spot on the board is legal or potential, then the array cell with an index matching the coordinates on the board will contain the value "true", otherwise it contains the value "false". This representation can provide an answer to the question of weather a player can build on a particular spot very quickly. Updating this representation so that it's accurate is a simple process. When a game is started, the legal lists for roads and settlements are initialized so that the list of potential spots for roads is empty and the list of potential settlements is equivalent to the legal list of settlements. This is because at the beginning of the game, a player can place an initial settlement anywhere on the board. After a player places a settlement, each player's legal and potential lists are updated according to the following rules:

- *All players*: The legal list for settlement spots is updated so that the intersection where the settlement was built and the adjacent intersections are marked illegal.
- *The player who just built*: that player's potential road list is updated to include the edges touching that settlement that are also legal.

Figure 3.1 illustrates legal places to build settlements and roads after building.

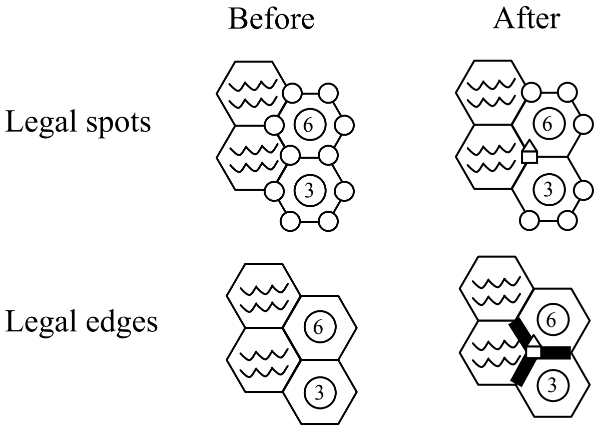


Figure 3.1 Legal Building Spots

This data is useful in a number of situations. The client uses it to highlight legal places to build when the player is about to place a piece. The client also uses it to highlight the "Buy" buttons in the building interface when the player has enough resources, an available piece in her hand, and a potential place on the board to put it. The server also uses this information to verify that building requests from clients are legal. This is to prevent someone from building another client that allows them to cheat. And as was suggested at the beginning of the section, the robot players use this information when planning a course of action and when considering where to build next.

Picking a spot

So now that the robot player has a way to figure out the possible options for what and where we can build on the board, how does it decide which option is best? Basically the robot player assigns a utility to each possibility and then picks the best one. The hard part about doing this in Settlers of Catan is that there are usually a number of reasons for building something. This means that there are many different ways to measure the utility of a possible building option. One simplification that comes to mind is to just look at the ultimate goal of the game, being the first to gain 10 Victory Points, and measure an options utility in those terms. But that simplification doesn't help us answer some questions about the usefulness of building something. For example, is it better to slow down another player by building in their way, or is it better to improve our situation? Or, is it better to build a road that will increase our longest road length and give us an opportunity to build a settlement, or is it better to save up for a city, which will increase our resource production and give us an additional Victory Point. All of these actions improve our chances of winning, but they do so in different ways.

Let's simplify the problem of deciding where to build by considering the situation at the start of the game. When the game starts, each player gets a chance to place two settlements on any legal spot on the board. After placing each settlement, a road must be placed on an edge touching the settlement just placed. In order to compensate for the advantage of having first pick, each player is only allowed to place one settlement, except for the last player who gets to place two. After that player places his second settlement, the second to last player places her second settlement. This continues until finally the starting player places his second settlement.

The placement of these two settlements has a large impact on a player's chances of winning. It will determine the type of resources produced for that player and the probability that they will receive resources when the dice are rolled. Also, since further development must be connected to a player’s existing structures, the initial placement determines where a player can build in the future.

The first thing that players look at when considering where to build an initial settlement are the numbers surrounding each spot on the board (Croxton 2001). They are looking for combinations of numbers that will be rolled with high probability. By building on these numbers, a player is increasing his chances of receiving resources on every turn. The probabilities for rolling a particular number are shown on the table below.

Number	Probability
2, 12	3%
3, 11	6%
4, 10	8%
5, 9	11%
6, 8	14%
7	17%

Table 3.1 Dice Result Probabilities

In Java Settlers, the backgrounds of the numbers on the board are colored to indicate their relative probability. The most probable are colored red while the least probable are shaded yellow. Numbers in between are colored a shade in between depending on where they lie on the scale. These visual aids help a player see at a glance which spots on the board are most likely to produce something when the dice are rolled. People usually consider these spots first because maximizing production is a winning strategy.

The way the board is set up, there is no way for a six and an eight to be next to each other and therefore touching the same spot. It's also not possible for two of the same number to be next to each other. The result is that there are a lot of spots on the board with the similar probability distributions. How does a player choose between the highest probability spots? The answer is in what resources are produced at each spot. The best pair of settlements will produce all five types of resources in great quantity. It's rare to find such a pair of spots due to the random nature of the board and the fact that three other players are competing for the same goal. Since this is the case, each player must decide which subset of the five resources she would like.

The subsets can be roughly grouped into three strategic categories: road-building, city-building, and monopolizing (Croxton 2001). A road-building subset concentrates on wood and clay production which are used in road an settlement building. This is useful because being able to build roads early allows a player to expand before the other players get a chance to. This gives that player access to the good settlement spots first as well as reducing options for other players. A player using this strategy will also be in a good position to take the Longest Road award and keep it. The disadvantage of this strategy is that later in the game it will be harder to build cities after the limited supply of five settlements is exhausted.

A city-building subset concentrates on wheat and ore production which are used to build cities and buy Development cards. Building cities early will quickly increase that player's resource production since cities produce two resources as opposed to a settlement's one. Wheat and ore are two of the three ingredients needed to buy Development cards, which are instrumental in taking the Largest Army award. In the standard game, five of the 24 Development cards in the deck are worth one Victory Point. This means that it is possible to win the game with only two cities, the Largest Army award, and four Victory Point cards. Winning in this way is unlikely, but using this strategy in addition to building a settlement or another city is a powerful one. The drawback to the city-building subset is that it makes expansion difficult. This means that unless you start the game with access to a trade port, you probably won't get to one until much later in the game. A player without access to a trade port is in a weaker trading position than a player who has access.

A third common strategy is to try to monopolize a resource and have access to the matching two-for-one port. This strategy is less common in it’s purest form than the other two because a player needs to have access to a lot of one kind of resource to make it work. Also, it’s likely that it would be a true monopoly in that at least one other player will have some access to the resource that is being “monopolized”. The advantage of this strategy is that it gives the player flexibility in deciding what to build. Also, the player will be in a strong negotiating position as long as there’s demand for the resource she controls. In order for this strategy to be successful, the monopolizing player must receive enough resources to keep pace with the other players. The next section will describe how to estimate a player’s ability to build given what resource sources and ports they have access to.

Estimating Building Speed

So how does one decide which of these strategies to use? What does a player do when none of her options seem to fit one of these possible strategies? More importantly, can the answers to these questions help us build a competent AI opponent? To answer these questions, let’s take a closer look at how to decide if the monopoly strategy is feasible.

As I described in the last section, the monopoly strategy is only feasible if the monopolizing player expects to receive enough of the monopolized resource to trade at a rate that gives that player a decent chance at winning the game. But what does this really mean, and is there a way to compute this for a player? To arrive at a definition of what it means to have a decent chance at winning the game, we need to look at the object of the game and what is required to achieve it. The object of Settlers of Catan is to be the first to gain 10 Victory Points. This means that the game is a race and we need to know how fast we can get these 10 points and whether its fast enough to beat the other three players. So how does a player gain victory points? By building settlements and cities, taking the Longest Road or Largest Army awards, and by possessing Victory Point cards. If we could estimate how fast we could build those things we could get an idea of how fast we could gain 10 Victory Points. To estimate how fast we can build something, we need to know how fast we can get the necessary resources to build it.

The simplest way to do this is to ignore the possibility of trading and just consider how often we will receive resources given the types of hexes our settlements and cities are touching and the numbers on those hexes. By calculating how often these numbers are rolled, we can estimate how long it will take to get a specific set of resources. I'll illustrate with an example. In this instance we have built two settlements as shown in figure 3.2 and we want to know how long it will take to get the resources to build a road starting with nothing.

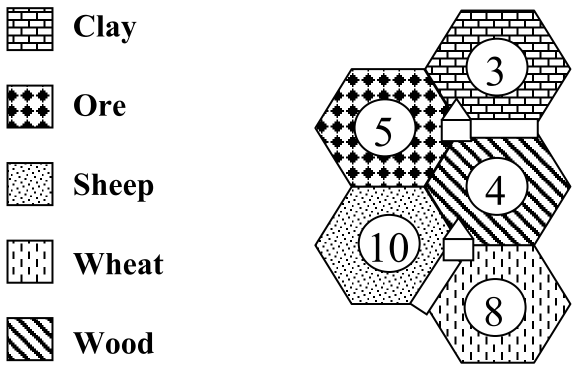


Figure 3.2 Settlement Placement Example

First, we make a table of how frequently we receive each type of resource. We do this by looking at the board and noting what numbers our settlements and cities are touching. If multiple settlements are touching the same number for the same type of resource, we note it multiple times. Also numbers that our cities are touching will be noted twice because they produce twice as much as a settlement. The first column of table 3.2 lists the numbers for the configuration in figure 3.2. Next for each type of resource, we want to estimate the probability of getting a single resource. We do this by taking the sum of the probabilities of rolling each number for that resource type. The second column of table 3.2 lists the probabilities for the configuration in figure 3.2. Note in our example that this does not give an accurate probability of receiving a wood resource on a roll of the dice. This inaccuracy will be addressed shortly. Continuing the example, our original goal was to create a frequency table, so we simply take the inverse of the probability rounding to the nearest integer to finish the table.

	Number	Probability	Frequency
Clay	3	.06	17
Ore	5	.11	9
Sheep	10	.08	13
Wheat	8	.14	7
Wood	4, 4	.16	6

Table 3.2 Resource Production Frequency

After building the frequency table, we can use an iterative algorithm to estimate how many rolls it will take to get a specific set of resources. For our example, we want one clay and one wood. The algorithm is as follows:

```
rolls := 0
do forever {
   if ourResources contains targetResources
      return rolls
   rolls := rolls + 1
   for every type of resource {
      if rolls MODULO frequencyTable[resourceType] == 0
ourResources[resourceType] := ourResources[resourceType] + 1
   }
}
```

This algorithm generates the sequence in table 3.3, with the estimate of 17 rolls to get a clay and a wood starting with no resources.

Roll	Clay	Ore	Sheep	Wheat	Wood
0	0	0	0	0	0
1	0	0	0	0	0
2,3,4,5	0	0	0	0	0
6	0	0	0	0	1
7	0	0	0	1	1
8	0	0	0	1	1
9	0	1	0	1	1
10, 11	0	1	0	1	1
12	0	1	0	1	2
13	0	1	1	1	2
14	0	1	1	2	2
15, 16	0	1	1	2	2
17	1	1	1	2	2

Table 3.3 Resource Production Sequence

A Word About Accuracy

This algorithm is not meant to get an accurate estimate of when a player will be able to build something in the sense that in so many rolls of the dice that player should expect to have the resources she needs to build. What we really have is an estimate that is accurate relative to other estimates we make. For example if our estimate tells us that it takes much longer to build a city than a road, this should be true in reality as well. The other important quality of our estimation algorithm is that it is affected by the same factors that are relevant in reality. In this case we are estimating how fast we can build something. This depends on our ability to produce resources, which is a result of where we've built our cities and settlements.

Now I will address the inaccuracy that I pointed out earlier. Recall that we had two settlements built on a 4 wood hex and so we recorded 4 twice in our number table under wood. This resulted in a probability of .16 for receiving wood on the roll of the dice (.16 is 2 times .08, the probability of rolling a 4). Then we went on to assume that the frequency with which we would receive one wood resource was the inverse of .16, which is 6.25 rounding to 6. I argue that even though this is inaccurate, it is good enough for our purposes. Why is it inaccurate? Because we should really expect to receive 2 wood resources for every 12 rolls of the dice, not 1 wood for every 6. How does this affect our calculations? In the last example it doesn't because the limiting factor was the rate at which we could get clay, 1 every 17 rolls, which is longer than 12. But if the situation were changed so that we received clay faster then once every 12 rolls, then is can be argued that our estimate is inaccurate because we will have to wait longer to receive wood than we think. How will this affect the behavior of our computer player? The answer to that question is, "it depends". This is because this estimation is used to make many different kinds of decisions, most of which haven't been discussed yet. But to give a concrete example, remember that at the start of this discussion we were trying to estimate how long it would take for a player to win. In order to make this estimate, we need to create a rough plan describing what will be built in what order. The algorithm that I'll describe later prefers building things that take less time to build. In this instance our underestimation of the time it takes to build a settlement may cause the computer player to choose to build a settlement over something else when creating this rough plan. This in turn may cause our estimate of how long it will take a player to win to be an underestimate. Later I will describe how we use comparisons of these estimates before and after a possible piece is placed on the board to get an idea of the utility of that piece. If placing the possible piece doesn't change our resource production, then this effect will cancel out since the underestimation will be in both the before and after estimates. If placing the possible piece does change our resource production, it can only make it better, meaning that our resource production frequencies will be the same or less. If this reduction is in our clay production, we will keep underestimating by the same amount. If the reduction is in our wood production, we will reduce the amount by which we are underestimating. Either way, we will recognize possible pieces that will get us closer to winning by either reducing the length of our plan or by increasing the rate at which we can build. In this example the inaccuracy of the estimate didn't affect the behavior in a noticeably negative way. This is true for other cases where the building speed estimate is used. Briefly they are: deciding where to place the robber; deciding what to trade for; and how much to discount the utility of a possible piece due to the time it takes to build it.

The Effect of Trading

Recall the two criteria for our building speed estimation: the estimate should be accurate relative to other estimates made by the same algorithm; and it is affected by the same factors that are relevant in the real situation. An important factor that our current algorithm doesn't take into account is the possibility of trading with the bank or ports of trade. Without this factor taken into account, our algorithm won't be able to capture the value of having access to ports, and resources we don't have access to will seem impossible to get. Figure 3.3 is a slight modification of our previous example so that the illustration of the point will be clearer. The difference is that we no longer have access to wheat and in its place we have access to a 2:1 wood port. Table 3.4 shows the new resource frequencies.

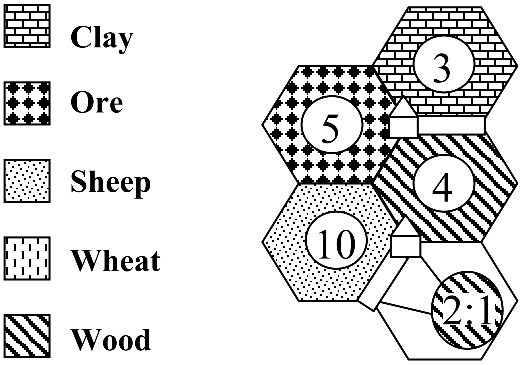


Figure 3.3 Alternate Placement

	Number	Probability	Frequency
Clay	3	.06	17
Ore	5	.11	9
Sheep	10	.08	13
Wheat	-	0	∞
Wood	4, 4	.16	6

Table 3.4 New Resource Frequencies

Ignoring the port for now, let's look at how to estimate how long it would take to buy a Development Card. The resources required to buy a Development card are 1 ore, 1 sheep, and 1 wheat. Without doing any calculation we can see that we will never be able to buy a Development Card unless we have a way to trade for wheat. Luckily there is a way, apart from trading with another player; anyone can trade four of one type of resource for one of another with the bank. To take this fact into account we modify our algorithm so that when we have four resources of the same type that we don't need, we trade them for one that we do need. Table 3.5 shows the estimate for getting 1 ore, 1

Roll	Clay	Ore	Sheep	Wheat	Wood
0	0	0	0	0	0
1,2,3,4,5	0	0	0	0	0
6	0	0	0	0	1
7,8	0	0	0	0	1
9	0	1	0	0	1
10, 11	0	1	0	0	1
12	0	1	0	0	2
13	0	1	1	0	2
14 .. 17	0	1	1	0	2
18	1	2	1	0	3
19 .. 23	1	2	1	0	3
24	1	2	2	0	4
24	1	2	2	1	0

Table 3.5 Card ETA Without a Port

Roll	Clay	Ore	Sheep	Wheat	Wood
0	0	0	0	0	0
1,2,3,4,5	0	0	0	0	0
6	0	0	0	0	1
7,8	0	0	0	0	1
9	0	1	0	0	1
10, 11	0	1	0	0	1
12	0	1	0	0	2
12	0	1	0	1	0
13	0	1	1	1	0

Table 3.6 Card ETA With a Port

sheep, and 1 wheat starting with nothing. If we take the wood port into account, we can trade two wood resources for one resource of our choice. This gives us the estimate in table 3.6.

Here is the algorithm with the trading elements added:

```
rolls := 0
do forever {
   for resourceType := every type of resource {
      if we have a port of type resourceType
         tradeRatio := 2
      else if we have a 3:1 port
         tradeRatio := 2
      else
         tradeRatio := 4

numberOfTrades := (ourResources[resourceType] –
targetResources[resourceType]) / tradeRatio
      for numberOfTrades {
         // Trade to get the rarest resources first
         mostNeededResource := -1
         for resourceType2 := every type of resource {
            if ourResources[resourceType2] <
            targetResources[resourceType2]
                  if mostNeededResource == -1
                        mostNeededResource := resourceType2
                  else if frequencyTable[resourceType2] >
                  frequencyTable[mostNeededResource]
                        mostNeededResource := resourceType2
         }
         // Make the trade
         ourResources[resourceType1]:=
         ourResources[resourceType1] – tradeRatio

         ourResources[resourceType2]:=
         ourResources[resourceType2] + 1
      }
   }

   if ourResources contains targetResources {
      return rolls
   }
   rolls := rolls + 1
   for resourceType3 := every type of resource {
      if rolls MODULO frequencyTable[resourceType3] == 0
      ourResources[resourceType3]:=
      ourResources[resourceType3] + 1
   }
}
```

This algorithm is relatively time and space efficient. The time for running is $O(n^2 \bullet r)$ and the space requirement is $O(n)$ where n is the number of resource types and r is the number of rolls. In Settlers of Catan there are five different types of resources and it's rare that anything would take more than 40 rolls of the dice to build.

As has been stated before, the main requirement of the algorithm we use to estimate building speed is that it is consistent with reality. This requirement motivated the addition of a trading phase in the algorithm. But this trading phase actually makes the algorithm report an estimate that is inconsistent in certain situations. Luckily these situations are rare and the estimates can be made consistent with a simple fix. Figures 3.4 and 3.5 illustrate one of these situations. It shows a configuration of pieces before and after building a city on the top-most spot. Tables 3.7 and 3.8 show the resource frequencies for both configurations. Tables 3.9 and 3.10 show that the estimate to build a city starting from no resources is longer in the second case, which is inconsistent with what we should expect to actually happen. By building a city we increase our resource production, so our time to build a city after that should decrease or stay the same. Because the trading phase of our algorithm doesn't look ahead, it makes the mistake of trading four clay for a wheat instead of an ore (table 3.10). This causes the estimate for the time to build a city to increase even though we've increased our resource production. This is inconsistent with reality and can create misleading results when we are looking at the impact placing a piece has on a player's ability to win the game.

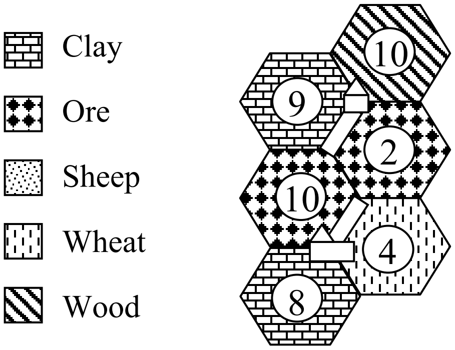


Figure 3.4 Before Building a City

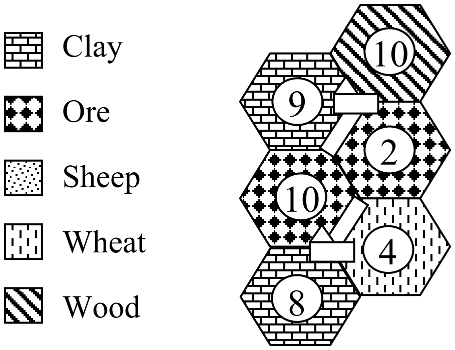


Figure 3.5 After Building a City

	Number	Probability	Frequency
Clay	8, 8, 9	.39	3
Ore	10,10,2	.19	5
Sheep	-	0	∞
Wheat	4, 4	.16	6
Wood	10	.08	13

Table 3.7 Frequencies Before

	Number.	Probability	Frequency
Clay	8, 8, 9, 9	.5	2
Ore	10,10,2,2	.22	5
Sheep	-	0	∞
Wheat	4, 4	.16	6
Wood	10, 10	.16	6

Table 3.8 Frequencies After

Roll	Clay	Ore	Sheep	Wheat	Wood
0	0	0	0	0	0
1, 2	0	0	0	0	0
3	1	0	0	0	0
4	1	0	0	0	0
5	1	1	0	0	0
6	2	1	0	1	0
7, 8	2	1	0	1	0
9	3	1	0	1	0
10	3	2	0	1	0
11	3	2	0	1	0
12	4	2	0	2	0
12	0	3	0	2	0

Table 3.9 City Estimate Before

Roll	Clay	Ore	Sheep	Wheat	Wood
0	0	0	0	0	0
1	0	0	0	0	0
2	1	0	0	0	0
3	1	0	0	0	0
4	2	0	0	0	0
5	2	1	0	0	0
6	3	1	0	1	1
7	3	1	0	1	1
8	4	1	0	1	1
8	0	1	0	2	1
9	0	1	0	2	1
10	1	2	0	2	1
11	1	2	0	2	1
12	2	2	0	3	2
13	2	2	0	3	2
14	3	2	0	3	2
15	3	3	0	3	2

Table 3.10 City Estimate After

The remedy for this problem is simple. When comparing building time estimates where the only change is to increase our resource production and the new estimate is longer than the original, treat it as equal to the original. This is reasonable because the minimum effect that increasing resource production can have is no effect at all. Another situation where this error might have an effect is when we use the building time estimate to discount the utility of placing a piece. As long as the error is small, the only harm is that the utility will be discounted a little more than it would have otherwise. Based on an empirical study, this error is between 1 and 3 and occurs in .03% of actual cases.

Another Approach

Though I have stated that the algorithm described is suitable for our purposes, we did experiment with an alternate approach in an attempt to test the hypothesis that a more accurate building time estimate would result in an improvement in game play. Unfourtunately this approach proved to be too slow to be useful in practice, but it is presented here as possible motivation for future work.

Earlier I showed an example where a player had built two settlements touching the same hex (figure 3.3). There I described how the building time algorithm multiplied the probability of that hex producing by 2 to reflect that there are two settlements on that hex. This is an inaccurate representation of how often that player should expect to receive resources from that hex. The following modification to the original algorithm is an attempt to get a more accurate estimate. The problem lies in the restriction that in the original algorithm, the player can only receive at most one of any type of resource on a roll. To fix this we need to keep track of how many resources of a particular type a player receives with a given frequency. We do this by extending our resource frequency table to record how often a player receives n resources of a given type, where n is an integer between 1 and 10. The reason we can stop at 10 is because that is the most anyone can receive from one roll of the dice. To fill the table we use the same method as before of recording all the numbers that the player’s settlements are touching for each resource type, and recording them twice for cities. But this time instead of adding the probabilities for rolling those numbers, we find the probability of rolling a single number and put it in table entry [type, n], where type is the resource type and n is the number of resources that player will receive when that number is rolled. Figure 3.6 shows a possible board situation and table 3.11 is the associated resource frequency table.

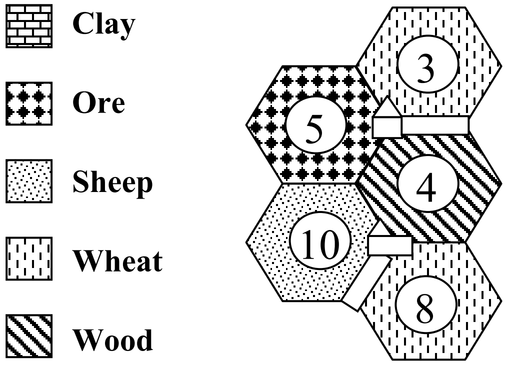


Figure 3.6 Possible Situation

Resource	Amount	Num.	Prob.	Freq.
Ore	1	5	.11	9
Sheep	1	10	.08	12
Wheat	1	3	.06	17
Wheat	2	8	.14	7
Wood	3	4	.08	12

Table 3.11 Resource Frequencies

To estimate how many rolls it will take before a player can gather a particular set of resources we use the same method as before but with the expanded resource frequency table. For each roll we look up what type of resource and how much should be produced and add it to our current set of resources. Then we perform any possible trades and continue until our resources contain the target set.

Even though this algorithm is theoretically more accurate than the algorithm described earlier, it has a longer run time that makes it impractical in a resource-bounded system. The runtime for this algorithm is $O((n^2 + 10n) \bullet r)$, where n is the number of resource types and r is the number of rolls. Since this algorithm is used repeatedly for making decisions, it is important that is has a very short execution time. To get an idea for how much this affects the run time, the average length of time to choose an initial settlement using this algorithm is 163.8 seconds compared to 1.5 seconds using the original algorithm.

Summary

Settlers of Catan is a game of resource management, and a race between four players. The combination of these features means that the ability to use resource production as an estimate of time is important. This ability is important in other domains as well. For example imagine that our agent is in a simulation where entrepreneurs are competing to be the first to establish a controlling portion of franchise businesses. That agent would have to perform a similar form of estimation using the resources that were important in that environment. In this chapter we developed an algorithm for estimating resource production in the Settlers of Catan environment. We proposed two options for achieving this end, one in which the amount of time needed to make the estimation was a limiting factor and one that would be more suitable in an environment in which precision is more important than computational time.

Here the less is more principal is illustrated by the comparison of these two algorithms. We began with an algorithm that would provide us with a rough estimate of the time required to obtain a specific set of resources. While not an exact measure this algorithm was simple and computationally cheap. When we looked at a more complicated algorithm that would allow us to more accurately predict the time required to obtain a specific set of resources it became clear that the cost to computational time was too expensive and therefore not usable as a solution.

Take Aways

- The first step in building a competent agent for a competitive environment is identifying what information is crucial for developing a strategy for acting in that environment. In Settlers of Catan, estimating the time it takes to build something given a particular board position is crucial to determining a good strategy.
- When developing algorithms to obtain this information from the environment, precision is not always the most important consideration. In our case a simple algorithm that provided a rough estimate and was computationally cheap was the better solution.

Chapter 4
Agent Implementation Part II:
Making a Plan and Deciding What to Build

In the previous chapter we laid the ground work for designing an algorithm for deciding what and where to build next with an algorithm that can estimate building time. In Settlers of Catan, deciding what and where to build can be a challenge. Should I build a road that would extend my longest road length and get me closer to building a settlement on that 3:1 port? Or should I wait and save my resources to build a settlement on one of the spots I already have access to. Maybe I should trade to get resources to buy a card since that other player looks like she’s trying to take the Largest Army award, which would put her in the lead. When considering what to build next a player takes many factors into account: cost in terms of time and resources; which actions perform multiple duties; how an action will impact other players; and risk of postponing to build. This section will discuss the challenges of designing an algorithm for deciding what to build next that takes these factors into account and makes an intelligent choice.

Choosing Initial Settlements

Now that we have established a way to estimate how long it will take to build something, how can we use it to decide where to place an initial settlement? Recall that when experienced players are considering where to place an initial settlement, they are looking for spots that will produce often and also spots that produce useful combinations of resources. The reason both of these features are desirable is because together they allow a player to build quickly, which in turn gives that player a better chance of winning the game. Now that we have a way to estimate how fast a player can build, we can use it to find a good spot to place an initial settlement.

One possibility would be to pretend to place a settlement on the board and then estimate how long it would take to build each possible building type (road, settlement, city, and Development Card) starting with no resources. We will refer to this sum of these estimates as the building speed of a player. We would then repeat this for all of the legal settlement spots on the board and choose the one with the fastest building speed.

If two or more spots have the same building speed, we can take the sum of the probabilities for the numbers touching each spot and pick the one with the greatest sum. Figure 4.1 shows a board with some building speed estimates for spots on that board. Table 4.1 lists the building speed estimates for the top individual spots on the board.

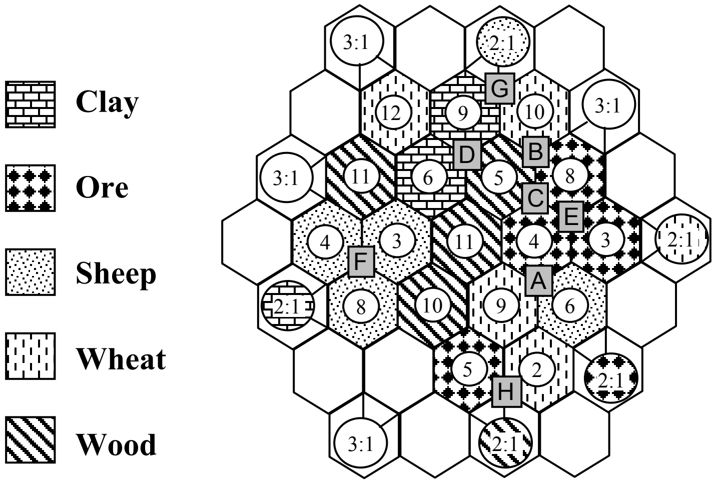


Figure 4.1 Possible Starting Settlements

Spot	Road	Settlement	City	Card	Total	Total Prob.
A	36	45	28	13	122	.33
B	28	45	26	35	134	.33
C	20	45	36	36	137	.33
D	9	36	64	36	145	.36
E	32	64	44	36	176	.28
F	32	52	80	36	200	.28

Table 4.1 Individual Building Speed Estimates

Spots	Road	Settlement	City	Card	Total	Total Prob.
A, E	24	24	18	9	51	.61
B, E	9	16	14	13	52	.61
D, H	9	27	36	24	63	.5
F, G	9	13	26	13	61	.47

Table 4.2 Paired Building Speed Estimates

This seems to be a good metric because it favors spots that have a mixture of good resource types and some high probability numbers. But it doesn't take into account the fact that we will have an opportunity to place another settlement before the setup phase is over. We can address this by considering all pairs of legal settlement spots. Then after choosing a pair, pick the better of the two spots to place first by using our original metric. Table 4.2 shows the building speed estimates for the top few pairs of settlements. Note that this wouldn't affect where the first two players place their first settlements at spots A and B, but the third player, instead of choosing D prefers F. This is because it has a faster building speed when paired with G. Notice also that the pair (F, G) is an example of the monopoly strategy.

In these examples we've been simply adding the estimates of all of the building types and taking the lowest total. If we wanted to find pairs of building spots that support the road-building or city-building strategies described earlier, we would take a weighted sum of the building speed estimates according to what building types we wanted to build. This hasn't been explored in the current version of Java Settlers and has been left for future work.

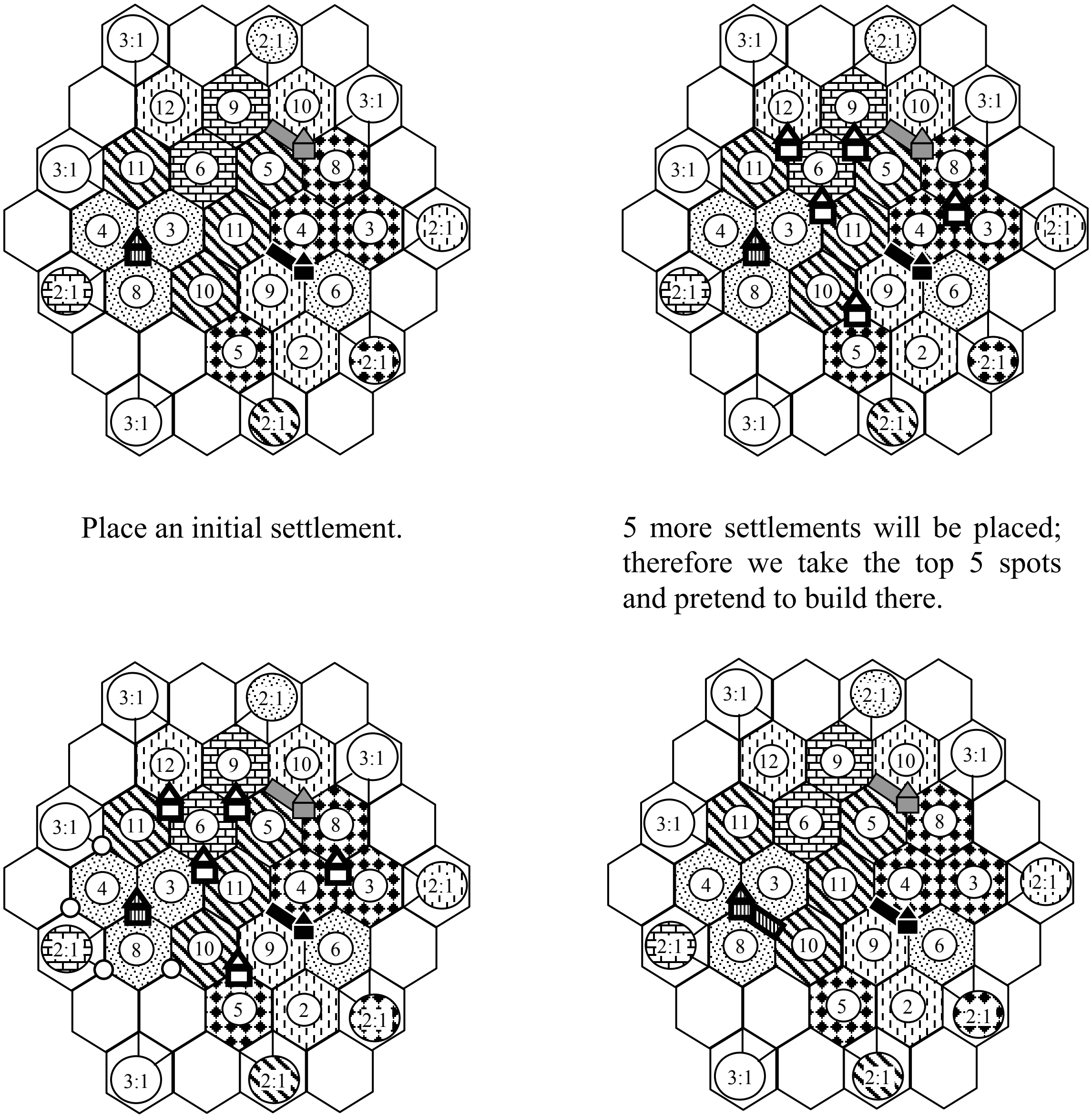
Choosing Initial Roads

After a player places an initial settlement, he must place a road on an edge touching that settlement. This road is important because it almost always determines where that player will build in the future. What we want to do is place the road so that it helps us build another good settlement after the initial placement phase is over. But we must also take into account where other players are likely to place their initial settlements. If we aim our road toward a really good spot someone else will probably claim it before we get a chance to build there and our road will be wasted (Croxton 2001).

For our algorithm we want to do the following: guess where players ahead of us are going to build; pretend to build settlements on those spots; out of the remaining legal settlement spots which are near the settlement we just placed, pick the best one; find the edge that leads to that spot from our settlement and place our initial road there.

One possible way to guess at where the players ahead of us are going to build is to execute the algorithm described above for choosing initial settlements for each player. It turns out that this is too slow for practical use. To give the reader an idea of why this is the case, consider that this algorithm is executing the building speed algorithm $n^2/2$ times where n is the number of legal settlement spots. At the beginning of the game there are 80 legal spots to build a settlement. On an 800 MHz Pentium III it takes approximately 4 seconds to execute the initial settlement placement algorithm. If we needed to guess at where three players where going to place their settlements, it would take about 12 seconds before we could decide where to place our initial road. This does not seem to be a reasonable amount of time to wait for the computer to make its move. This would also break the illusion of an intelligent opponent because people typically don’t take long to decide where to place their road after they’ve placed their initial settlement. In order to keep pace with the typical player and not test their patience, we will use a simpler method for guessing where other players will place their initial settlements.

Our original settlement-choosing algorithm where we only consider single spots is faster, so we’ll use that. We can use it to generate an ordered list of legal settlement spots from best to worst. Then all that needs to be done is to figure out how many settlements will be placed by other players before the end of the initial placement phase and pretend to place that many of the top settlements on the board. Then we can look at the remaining legal settlement spots that are two nodes away from the settlement we just placed, pick the best one by pretending to build at each of these spots and calculating our new estimated building times. Once we’ve picked the best spot, it is easy to calculate the coordinates of the edge leading to that spot thanks to our coordinate system. Figure 4.2 illustrates these steps.



Place an initial settlement.

5 more settlements will be placed; therefore we take the top 5 spots and pretend to build there.

Then we consider settlement spots near the one we just placed.

After picking the best one, we pick the edge that leads to it to place our road.

Figure 4.2 Choosing an Initial Road

Deciding What to Build

How do people decide where to build? Because there are many elements in Settlers of Catan which have an impact on a player's performance but are impossible to predict precisely (dice, Development Cards, the actions of other players), there are no hard and fast rules for determining the best possible move. This is not surprising since it is this very aspect of the game that makes it interesting and fun to play. Instead, a player must consider the situation she is in and use general strategies or rules of thumb to help determine the best possible move.

One of these general strategies that is more important in the early stages of the game is to build things which increase resource production. Increasing resource production by building settlements or cities allows a player to build faster, which in turn gives them the ability to increase their building rate, etc. Failure to increase resource production means falling behind the other players who are.

Another general strategy is to build as soon as possible rather than wait to build a better thing later. Earlier, the road-building and city-building opening strategies were described. In both cases, a player will want to increase their resource production as much as possible in the early stages of the game, but what resources they have access to will determine the best way to do that. The best option for the road-building player would be to build one road off of an existing road and then a settlement. The city-building player should build a city. There are two reasons for this; one is that by increasing production sooner, the player can reap the benefits sooner. Secondly, in settlers of Catan it is risky to hold onto resources for too long. One of the rules states that when a seven is rolled, any player with more than seven resources in their hand must discard half of their resources. So if building something requires you to hold onto too many resources before you collect enough to trade with the bank or a port because no one else will trade you the resource you need, you may never get there because you need to keep discarding. Another rule states that when a seven is rolled and after players have discarded, the player that rolled the dice moves the robber and steals a resource at random from a player that is touching the hex that the robber is now on. So even if you have seven or less resources and don't need to discard, there is a chance that someone will set you back by taking a resource that you need.

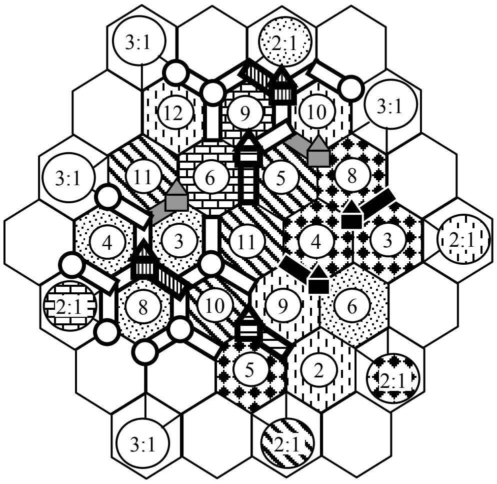
A third general strategy is that in order to win the game, a player will probably need the Longest Road or Largest Army awards (Croxton 2001). Each of these awards is worth two Victory Points which considering that the goal is to get ten points and each player starts with two, each award gets a player 25% closer to winning. Also if you have one or both of these awards, it means that another player doesn't have them. Typically it's not wise to try to get these awards in the early part of the game. If other players have been concentrating on increasing resource production, their ability to build faster will give them the advantage when competing for these awards.

A First Attempt at a Decision Making Algorithm

Considering the general strategies outlined in the previous section, we will make an initial attempt at designing an algorithm to pick what we should buy with our resources and if it's a road, settlement, or city, where to place it. This attempt won't be perfect, but looking at it's shortcomings will tell us something about how to design decision making systems for complex dynamic environments.

The first strategy outlined was the resource-production strategy. Recall that the point of this strategy was to build more settlements and cities in order to increase the frequency with which the player receives resources which in turn reduces the time it takes that player to build other things. By happy coincidence we have a way to measure how long it takes a player to build things. (It’s actually not a coincidence since we had the same strategic concerns during the initial placement phase.) We will use the metric here in a similar way. Before considering where to build next, we will measure our current building speed. Then we will pretend to place each possible piece that we could build next and measure our building speed again. Pieces that generate greater reductions in building speed will be preferred. If we consider how this simple algorithm will work right after the initial placement phase, we soon see a problem. Because each player only has one road from each settlement, and settlements must be placed at least two roads away from another settlement, the only pieces that a player can build next which affect building speed are cities. So our algorithm will decide which of the two cities is better to build first, and after that it will be useless. The solution is to look ahead a bit further, but in a focused way in order to keep our use of computational resources low. Right now we don’t really care about roads other than their ability to allow us to build settlements, so we should focus on the settlements. We need something similar to the legal and potential data structures described earlier so that we only consider legal settlements that are within our reach. We will call this new object a PlayerTracker since is contains strategic information about a player, as opposed to the Player object that contains mundane information. When a player places a road, that player’s PlayerTracker will look ahead by pretending to place new roads attached to that road and then recording new potential settlements (figure 4.3). Since the Settlers of Catan board is a graph, a standard graph traversal method can be used to look ahead as far as needed. For our purposes, two roads out is a good number. It is rare that a settlement more than two roads away is worth building before one that is closer. This kind of look-ahead is much cheaper than it may seem since we don't need to do it every time we make a decision. The PlayerTracker only needs to be updated when players put pieces on the board. This is due to the fact that board elements once placed, remain there for the rest of the game.

In addition to recording these new potential settlements, the PlayerTracker must record the possible roads that lead to them so that if our algorithm picks a settlement to build, we can know which roads to build first. Before moving on, I should point out that the PlayerTracker needs to be updated not only when that player builds a road but when any player builds a road or settlement. This is because another player's road or settlement may cut off a path to a future settlement. This update can be done by keeping track of which pieces support the building of others. Then when a new piece is placed that severs a support link, the supported piece is removed if nothing else is supporting it. This removal process is recursively applied to any pieces the removed one supported.



Possible settlements two roads away or less for the vertical stripe player are marked with circles.

Figure 4.3 Possible Future Settlements

Now that we have a way to consider possible settlements within reach, we can improve our algorithm. In addition to pretending to build potential cities and measuring their effect on our building speed, we will also measure the building speed difference of the potential settlements in our PlayerTracker. This allows us to compare the effectiveness of the resource producing building types, but it ignores an important factor: how long it will take to build.

We can now address the second general strategy of preferring to build things that can be built sooner. Again we can use our building speed algorithm to estimate how long it will take us to build something. But in this case rather than starting with no resources, we'll start with the resources in our hand and we'll only use the estimate for the item we're interested in building rather than the sum of all the estimates. We will refer to this estimate as the item's estimated time to build or its ETB. This will work well for cities and settlements which don't require roads to be built, but to estimate how long it will take to build something which requires multiple steps we need to modify our technique. What the building speed algorithm is really doing is estimating how long it will take to get a particular set of resources. By setting the target set of resources to match what is required to build a settlement, we get an estimate for how long it will take to build a settlement. If that settlement requires roads to be built, we can simply add the resources required to build those roads to our set of target resources. This estimate doesn't take the possibility of trading with other players into account, but it will still give us an idea of which things we can expect will take longer to build than others.

Finally we need to consider the importance of having the Longest Road and Largest Army awards. For this first attempt, let’s say that if the ETB for Longest Road or Largest Army is shorter than the ETB for any possible settlement or city then we’ll set that as our current goal and take the steps to achieve it. That leaves us with the problem of calculating the ETB for Longest Road and Largest Army.

To calculate the ETB for Longest Road, we need to figure out how many roads need to be built to take it and then use the building speed algorithm to get the ETB for that many roads. The simplest way to figure out how many roads are needed is by taking the length of the longest road on the board, subtracting the length of our longest road and adding one. This solution has two important shortcomings. One, it doesn’t tell us which roads we should build if we want to take the Longest Road award. Two, it won’t detect situations where a player can win the award by connecting two sets of roads. To address both of these problems, we need to simulate building roads on the board in order to see which ones contribute to lengthening the player’s longest road path. A brute force method for doing this would be to start at every place a player could build a road and do a depth-first traversal of the connected edges. At each edge we would pretend to build a road and then measure the player’s longest road length. Recall from chapter 2 that the algorithm for determining longest road length is a depth-first traversal of the player's connected roads starting from every node that the player's roads touch. If we consider that each depth-first traversal takes linear time in the number of edges and that there are approximately the same number of nodes as edges in all cases, we end up with a runtime of O(E^4), where the average value of E, the number of edges, is 8.5. This is too slow for our purposes. By using some domain knowledge and storage space, we can reduce this time significantly.

Since we're trying to increase the length of a player's longest road path, an obvious place to start building would be at the ends of their longest paths. We can get the coordinates for the ends as well as the lengths of the paths by modifying the algorithm that measures the longest road length so that it stores this information in a data structure. Then when we do our depth-first traversal from the end of a path, we can simply add one for every edge we traverse to the length of that path to get our new length. If the traversal finds the end of another one of our paths, we can add the length of that path to our total. The depth of the traversal is the minimum value of the length needed to win Longest Road minus the length of the path that we started from and the number of roads we have available to use. In pseudo code this is MINVAL((longestRoadLength+1)-pathData.length, player.availableRoads). During the depth-first traversal, we can store the coordinates of the traversed edges and return the shortest path that gains the Longest Road award. If no such path exists, a value to indicate that is returned instead. Once we have the number of roads necessary to achieve Longest Road and where to build them, we can calculate the ETB for these roads and decide if that is what we want to build next.

Calculating the ETB for taking the Largest Army award is fairly straightforward. First we determine how many knights we need. If no one has the Largest Army award, that number is three. Otherwise, we take the size of the largest army and add one. Then we determine how many knights we need to buy by subtracting the number of knights we have from the number we need. We then estimate how long it would take to buy that many Development Cards. For this estimate we are assuming that all cards in the Development deck are knights. This estimate also doesn't take into account the time needed to play the knight cards since a player can only play one Development Card on their turn. We could try to factor this into the estimate, but it probably wouldn't change the estimate in a significant way since a player usually will play knight cards as they're collecting resources to buy more cards.

Now that we can estimate how long it will take to build settlements in our vicinity, cities, enough roads to take Longest Road, and enough knights to take Largest Army, and we can calculate the effect a settlement or city has on our building speed, we can devise a simple method for deciding what to build. Following our strategy of preferring to build sooner than later, we will consider only the things that take the least amount of time to build. If there is a tie, pick the one with the greater effect on our ability to build quickly. Finally, following the strategy of not trying to gain Longest Road or Largest Army until later in the game, don't consider those possibilities until we've achieved at least four Victory Points. This seems to be a good cutoff because we will have built two more structures that improve our resource production, and we don't want to wait too long for risk of another player getting too far ahead in getting one of those awards.

Evaluation

Now that we have an algorithm for deciding what to build, we need a way to evaluate its behavior. To do that we need to be clear on what criteria are important to consider for a game-playing AI system. One of the criteria that were alluded to earlier was speed. The computer player must make a move in a timely manner. In this case it's reasonable for a player to take between one and five seconds to make a move. Taking too long to decide what to do can ruin the pacing of the game and frustrate players. It also breaks the illusion of playing against an intelligent opponent. In addition, we want this speed to be maintained when running many (40 to 60) of these computer opponents on a machine. This is so that we can support a good number of people playing the game at the same time, which in turn allows us to gather data quickly.

Making a move in a timely manner is no good unless that move makes sense to someone watching the game. If the computer opponent is doing things that an observer can't explain, the opponent will seem irrational. Of course people will do things that are sometimes inexplicable, but they have the ability to explain their actions. If our computer player could explain why it did something, that would definitely help, but for now we will concentrate on making obviously rational moves. In this case we have an advantage since not all of a player's information is publicly available. So if our computer opponent does something unexpected, the observer may attribute it to a piece of information that is hidden, like the opponent's resources.

In addition to making reasonable moves, we want the computer opponent to react to the player's actions. This may seem obvious since in this game only one person can win and the playing area is small enough that there is bound to be conflict, a decent player must have to react to the opponents' moves. But the algorithm we've just created doesn't take other players' actions into account. Later we'll see how this affects the computer opponent's ability to play and a possible way to address this problem. By reacting appropriately to the player's actions, the player's experience is enhanced. This is because it provides an opportunity to experiment with strategies that require a reaction on the part of the opponent.

Our fourth criterion is a result of making reasonable moves and reacting to other player's actions, and that is playing a challenging game. We want the computer to present a reasonable challenge for the player. If it is too easy to beat, the game is boring. If it's too hard, it can be frustrating. In the best possible world, the computer player would adjust its skill level to match the people it was playing against. But before we can do that we need to create a competent player.

Methods

Now we need methods for measuring the behavior of our computer opponent to see how well it meets the criteria. The first criterion of taking action in a timely manner can be easily measured. We simply take a look at the system clock before and after a decision is made to get a time measurement. Then we take the average of a large number of measurements. If the mean is within one to five seconds, then our algorithm meets the criterion.

The second criterion, making reasonable moves, is subjective and therefore difficult to measure quantitatively. Our goal is to get an idea of if and what situations the algorithm fails. Observing a number of games where we look for irrational actions will allow us to satisfy this goal.

The third criterion of being reactive to the players' actions is also difficult to measure, but in this case we know that the algorithm doesn't take the other player' actions into account. Whether or not people playing the game think that the computer is reacting to them is a separate issue that could be resolved with a survey.

For the fourth criteria, playing a challenging game, we can measure how often people win when playing against the computer. We will only measure games played to completion since it's hard to tell who will win looking at a game that isn't finished. This might bias our data if people always quit a game when they thought they were losing. From looking at a sample of games that weren't played to completion, there is no evidence that this occurs more often than when people quit when they were winning. Also, we won’t count games where a human player has taken over a computer player’s seat in the middle of a game and vice versa. If our computer player is playing at the same level as its human opponents, we should expect it to win 25% of the time due to the fact that it is a four-player game. Because the dynamic between the players change depending on the number of human players present, we will divide the results into three categories that correspond to having one, two, or three human players.

Results

The average time for a computer player to make a decision using the algorithm outlined above is 6.8ms with a standard deviation of 35ms. This measurement was taken on a 400MHz Sun Sparc Ultra 2 running four copies of the robot client, each one playing an average of 20 games. This result shows that our algorithm is well within our desired time limit.

For the most part our algorithm makes reasonable moves, but there are a number of situations where it doesn’t do what an experienced player might expect. One of these situations occurs when the computer player is attempting to take the Longest Road award. Since the function for estimating how much time it will take to gain the Longest Road award is looking for the minimum number of roads that when built will create the longest road, it will sometimes find a path that doesn't leave any room for expansion. Most people when given a choice will build roads in such a way that in addition to increasing the length of their longest road path will also provide an opportunity to keep lengthening their road after they take the Longest Road award. This is because another player will usually challenge them for Longest Road and if they don't leave room to expand they risk losing the award.

Our decision-making algorithm also fails to do the right thing in a related situation. If the computer player has the Longest Road award, it no longer considers building roads to extend the length of its longest road. It would be a good idea for it to do so because if someone else takes the award away it has to build at least two more roads to take it back and during that time the other player might win the game. It's better to build one more road now to keep a threatening player at bay. In the same vein, after the computer player has gained the Largest Army award, it won't consider buying more cards to increase the size of its army. On the other side of the coin when the computer player doesn't have the Longest Road or Largest Army awards, it fails to see when it's critical to try to take these awards away from a player that is about to win. This is apparent when the computer player builds a settlement when it could have built a road to take the Longest Road award from a player with nine points. Most people would prefer to build the road and extend the game.

The most apparent irrational behavior is when the computer player gets completely boxed in and has no place to build settlements or cities. The only option left that our algorithm considers is taking Largest Army and once it has done that it will just sit and collect resources. Even though this is a rare case, it is odd to see a player with a lot of resources doing nothing. Most people would use these resources to buy Development Cards hoping for enough Victory Points to win the game, or keep someone else from gaining the Largest Army award. Most of these behaviors that can be viewed as mistakes have to do with taking the other players' goals and actions into account. We will address this in the next version of the decision-making algorithm.

So how well does this initial attempt perform? To test this, we developed a computer player that would play the game fully except for trading with other players. This computer player used the decision-making algorithm described above to choose what item to build next and what resources it would trade for using ports and the bank. We encouraged people to play against multiple copies of this computer player in the following formats: 1 human versus 3 computer opponents, 2 humans versus 2 computer opponents, and 3 humans versus 1 computer opponent. After two weeks, we took a random sample of 4000 games for the 1 human versus 3 computer opponents case. The other cases did not have enough data to get a statistically accurate measure of the computer player's performance, but are presented for completeness (see table 4.3).

Individual Computer Player Data					
Computer Player 1					
No. Humans	Wins	Losses	Total	Computer Win %	Sampling Error
3	1	14	15	6.666667	±6.440612
2	15	130	145	10.34483	±2.529096
1	664	3336	4000	16.6	±0.588311
Computer Player 2					
No. Humans	Wins	Losses	Total	Computer Win %	Sampling Error
3	3	11	14	21.42857	±10.96642
2	17	152	169	10.05917	±2.313749
1	647	3353	4000	16.175	±0.582209
Computer Player 3					
No. Humans	Wins	Losses	Total	Computer Win %	Sampling Error
3	2	8	10	20	±12.64911
2	12	133	145	8.275862	±2.288044
1	650	3350	4000	16.25	±0.583296
Computer Player 4					
No. Humans	Wins	Losses	Total	Computer Win %	Sampling Error
3	2	12	14	14.28571	±9.352195
2	23	150	173	13.2948	±2.581311
1	670	3330	4000	16.75	±0.590432
Composite Computer Player Data					
No. Humans	Wins	Losses	Total	Computer Win %	Sampling Error
3	8	45	53	15.09434	±4.917422
2	67	250	317	21.13565	±2.293076
1	1976	2024	4000	49.4	±0.790512

Table 4.3 Initial Agent Attempt Performance

Individually, each computer player won about 16% of the games it played. When a person is playing against three computer opponents, the person wins a little over half the time. This seems to be a reasonable level of challenge. If the computer players were playing at the same level as a person, the person should only win one quarter of the time which might be frustrating for most people, especially ones learning to play the game. One criticism might be that we should strive to make the computer player play at an expert level and then introduce errors into its play in order to adjust the difficulty level. I believe this is a good goal, except that we want the errors that we introduce to look reasonable on the part of the human opponent, otherwise the illusion of playing against an intelligent opponent will be broken. The next incarnation of our computer opponent will address these issues.

Using a Plan to Guide Reactive Behavior

Our initial attempt at an algorithm for deciding what to build next seems to do a reasonable job at making moves that make sense. But as was pointed out in the previous section, it was not perfect. Where it fell short was mostly in not taking other player’s goals and actions into account. In this section we will discuss a way to address this shortcoming and in the process gain some other advantages to the previous method.

If you ask any player in the middle of a game of Settlers of Catan how they plan to get enough points to win the game, they can tell you pretty quickly. The description of their plan won't detail every move they're going to make, rather it will be in the form of a list of objectives which will gain the player Victory Points. This suggests that people have a rough plan in their heads when they are deciding what to do next. Further, experienced players are most likely forming plausible plans for how the other players will attempt to win the game. When deciding what to do next, a player looks at what can be done now or in the near future and sees how those actions further his plan and hinder his opponents' plans. We will use this idea to develop a better algorithm for deciding what our computer player will build next.

Here is a basic outline of how we want our new algorithm to work. First we need to make a plan for how to gain enough Victory Points to win the game from our current position. This plan will be at a level of detail such that it describes what sub-goals need to be achieved to gain the points, but not how those sub-goals are to be achieved. Using this rough plan we will determine the utility of each of our possible building options toward advancing this plan. We will also generate plans for the other players and determine the utility of each of our building options toward hindering their plans. Finally we must take into account the fact that the longer we have to wait to build something, the less desirable it is. To do this we will discount the total utility of a building option according to its estimated time to build. Finally, pick the building option with the highest utility and build it if we have the necessary resources, otherwise trade for them. This algorithm will be run every time we need to decide what to build. In this way it will be reactive to the environment since it will always be taking the state of the environment into account before making a decision. Even though it will be building a new plan every time the algorithm is executed, building this plan will be computationally cheap since we will be using a few high-level operators, and we won't try to build an optimal plan, just one that is reasonably good.

The basic approach for the plan-building part of our new algorithm will follow the guidelines used for making our first decision-making algorithm. There we used three heuristics: prefer to build things that minimize our building time, prefer to build things that take less time to build, and Longest Road and Largest Army are important to consider in the middle and final phases of the game. But instead of picking one thing to build, we will continue to pick things to build until we have ten or more Victory Points. Along the way we will simulate the important aspects of building each item. These are: how long each item takes to build, how it affects our building speed, what other building possibilities are opened up or removed, and how it affects our Victory Point total. Other aspects like how it might affect other players will be addressed in another part of the decision-making algorithm. The purpose of generating this plan is to get an idea of how long it will take a player to win the game. We will refer to this as the estimated time to win or ETW. Once we have that estimate, we can use it to generate a utility value for each possible place to build.

The method for calculating a player's ETW is as follows. Start by copying the player's possible settlement and possible city information so we can manipulate it without having to restore the original information when we're done. Then do the following until our player has at least nine points. Pick two cities that when built would decrease our building speed the most. Calculate the ETB for each city taking into account the change in building speed once the first city is built. We will call the sum of these ETBs the ETB for two cities. Next, pick a city and a settlement that when built would decrease our building speed the most. When considering where to build the settlement, only consider spots with the shortest ETB. If the number of available building pieces or building spots doesn't restrict the order of building, pick the order based on which ETB for building both is smallest. This estimate will be called the ETB for a city and a settlement. Next, pick two settlements that when built would decrease our building speed the most. Only consider settlement spots with the smallest ETBs. Calculate the

ETB for each settlement taking into account the change in building speed once the first settlement is built. Now that we have estimates for building two cities, a city and a settlement, and two settlements, we can decide which of these options to add to the plan. If the player has more than five points we will also calculate the ETBs for taking the Longest Road and Largest Army awards as long as we don't have them all ready. This will be done in the same way as it was in the first decision-making algorithm. Now we pick the smallest ETB and simulate the implementation of that plan step. If there is a tie, pick the one that reduces our building speed the most. To simulate the implementation of a plan step, we want to update a number of pieces of information that we care about. If the plan step is to take the Longest Road or Largest Army awards, we assume that we've succeeded and note that we have that award. If the plan step involves building a settlement or city, we update the player's building speed by updating what numbers that player has access to after the plan step has been implemented and then calculating a new estimate. We also want to keep track of what pieces are available for building. When a settlement is built, we use up a settlement piece. When a city is built, we use up a city piece and make a settlement piece available. Updating what spots are available for building is done in a local copy of that player's possible settlements and cities. If a settlement is built, we remove possible settlements that are adjacent to that spot and add a possible city for that spot. If a city is built, we remove that possible city spot. Finally, we add two points to this player's temporary score and add the ETB for this plan step to the running ETW total.

If the loop described above has exited and the player has nine Victory Points, then we do one more step, except in this case we don't consider building two items at a time, only one. Also, after we've picked something to build, we only need to update the score and the ETW total. Below is a pseudo code rendition of the method for calculating a player's ETW.

```
copy player’s possible settlement and possible city information
points = player’s current Victory Point total
etw = 0
while (points < 9) {
   pick two cities and calculate the ETB to build both
   pick a city and a settlement and calculate the ETB to build both
   pick two settlements and calculate the ETB to build
   both
   if ((points > 5) && (player doesn’t have LR)) {
      calculate ETB for Longest Road
   }
      if (points > 5) && (player doesn’t have LA)) {
         calculate ETB for Largest Army
         pick the plan step with the smallest ETB;
         tie goes to plan step with better building speed
   }
   // simulate implementation of choosen plan step
   if (plan step == take Longest Road) player has LR
   else if (plan step == take Largest Army) player has LA
   else {
      adjust building piece counts
      update building possibilities
      update player’s building speed
}
points = points + 2
etw = etw + ETB for plan step
}

if (points == 9) {
   pick a city and calculate its ETB
   pick a settlement and calculate its ETB
   if (player doesn’t have LR) get ETB for Longest Road
   if (player doesn’t have LA) get ETB for Largest Army
   pick the plan step with the smallest ETB; tie goes to plan step
   with better building speed
   etw = etw + ETB for plan step
}
```

Now that we have a way to estimate how long it will take a player to gain enough Victory Points to win, we can use this measure to generate a utility value for each of a player’s building options. There are two different kinds of value that we want to capture in this utility value. One is how does building this piece get the player closer to winning the game. The other is how does building this piece hinder the other players from winning before us. We will measure these values separately and then combine them as a weighted sum. This will allow us to change the behavior of the computer player by changing the weights.

To measure the effect of building something, we must first get a base line measurement. This is simply a matter of calculating the player's ETW from the current state. Then we will pretend to build the item being considered. When doing this we must simulate all of the effects that building that piece normally would have. In this case it is computationally cheaper to perform the action on the actual state of the game and then undo the move rather than perform the action on a copy that would be discarded. This simulation of building is different than the one done when we're estimating a player's ETW. In that case we only cared about a few specific aspects of the result of building something. For this calculation we want to take all of the effects of building something into account in order to get the best possible measurement. After doing this we calculate the player's ETW for the new state. By taking the percentage of difference between the two ETW estimates we get a measure of the utility of a building option toward furthering the player's goal of gaining 10 Victory Points. To measure the utility of this action toward the goal of slowing down other players, we perform a similar calculation. We start by calculating the ETW for each other player for the current state of the game. Then we do it again after simulating the building action and take the percentage of difference between the two estimates for each player. In this case we want to take the negative of the difference since we want to slow down the other players. Finally we combine these percentages in a weighted sum to get the total utility. Figure 4.4 and table 4.4 illustrate this process with an example.

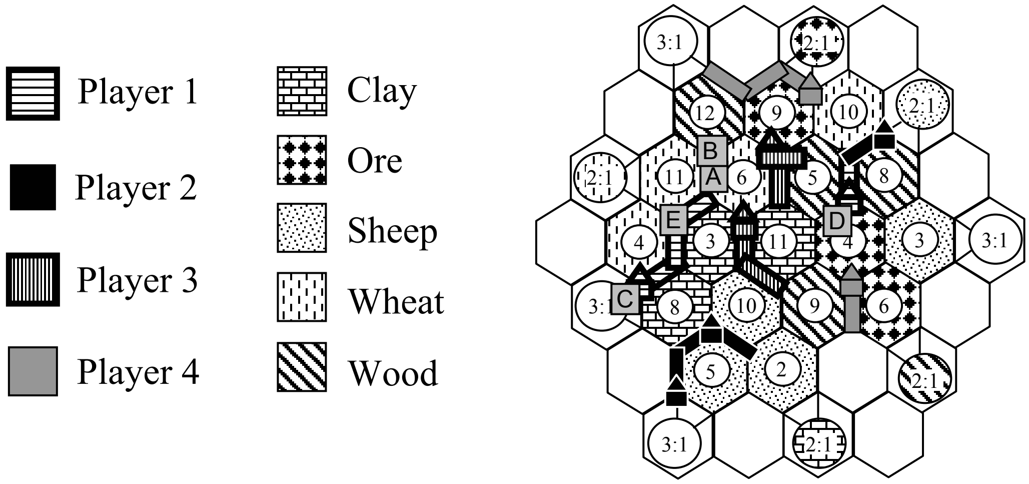


Figure 4.4 ETW Example

	Plan Steps	ETB
Before	Settlement at E; City at C	31
	City at D; City at E	22
	Longest Road	6
	Largest Army	21
	Total ETW	80
After	Settlement at E; City at C	31
	City at D; Settlement at B	21
	Longest Road	3
	City at B; City at E	15
	Total ETW	70

Table 4.4 ETW Estimates for Player 1

Player 1 is considering what to build next. In this situation, player 1 is looking at the value of placing a road at the edge marked A in figure 4.4. Table 4.4 shows player 1's plans before and after placing a road at A. Placing the road helps player 1 in multiple ways. It gets him closer to gaining the Longest Road award, and it opens up an opportunity to build a settlement which in turn changes the rest of the plan from wanting to take the Largest Army award to building two cities. Overall, building this road has the effect of decreasing player 1's estimated time to win by 12.5% giving that road a utility of 12.5. Building a road at A doesn't affect the plans of the other players and so no extra utility is added. The next example will show how interference with other players is taken into account.

Figure 4.5 shows the game board a couple turns later. Player 3 is considering what to build next and looks at what building a settlement at the spot marked A would do for her. Table 4.5 shows that building a settlement at A reduces player 3's estimated time to win by 19.7%. It also increases player 2's estimated time to win by 21.7%. This is because it takes away the opportunity to build a settlement at D. The total utility for building a settlement at A for player 3 is the sum of these two numbers, 41.4. In this example we added the different utilities without modification, but one could imagine multiplying each utility by a coefficient in an effort to change the behavior of the agent. For example, multiplying the utility of interfering with another player by a small coefficient should make the agent play less aggressively, while multiplying by a coefficient greater than one would make the agent go out of it's way to interfere with other players. Initial tests show this to be the case, but each extreme interferes with the agent's ability to win games. The configuration that results in the highest win percentage is simply multiplying each utility by one.

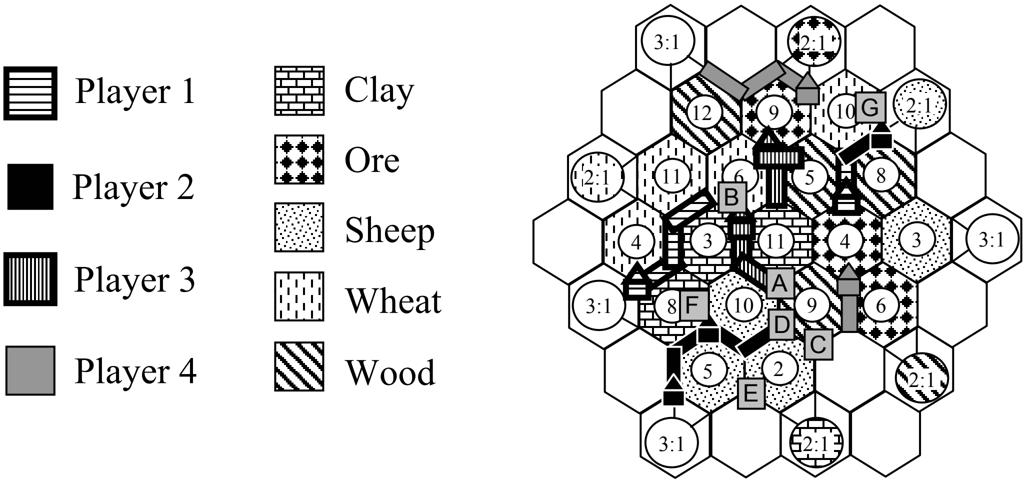


Figure 4.5 A Couple Turns Later

	Player 2	ETB	Player 3	ETB
Before	Settlement at D; Settlement at E City at F; City at G Longest Road City at D Total ETW	20 23 9 8 60	Settlement at A; City at B City at A; Settlement at C Longest Road City at C Total ETW	22 24 15 10 71
After	Settlement at E; Settlement at C City at F; City at C Longest Road City at G Total ETW	30 25 9 9 73	City at A; City at B Longest Road Settlement at C; City at C Total ETW	22 15 20 57

Table 4.5 ETW for Player 2 and Player 3

Using a utility based paradigm for choosing what to do makes it easier to incorporate a variety of factors. One of these factors is the composition of the Development Card deck. In the first decision- making algorithm we assumed that all of the cards in the deck were Knight cards, which are used to move the Robber and increase the size of the player's army. But only 14 of the 25 cards in the Development Card deck are Knights. The rest of the deck is composed of 5 Victory Point cards, and 6 other cards that allow the player to do various things like build two roads for free, or take all of one kind of resource from the other players. To get a more accurate measure of the utility of buying a Development Card we can break the deck into three categories, Knights, Victory Point cards, and the rest, and then find the utilities for each of these categories. We do this in the same way that we calculate the value of building something; first we take a base line measurement by calculating the player's ETW before buying the card. We then simulate giving that player a Knight card and recalculate the player's ETW. With these two numbers we can find a normalized utility by calculating the percentage difference between the two. Finally we multiply by the probability of drawing a Knight card, in this case its .56. We repeat this procedure for the Victory Point card, but instead of giving the player a Knight card, we give them a Victory Point. The final category of cards is too difficult to simulate efficiently, so we will simply find an appropriate utility for these cards through a number of trial and error experiments. To get the total utility for buying a Development Card, we take the sum of the three utilities.

Discounting Utility Over Time

There is one more factor that we haven't taken into account; the longer it takes to build something the less valuable it becomes. Discounting the utility of building something according to how long we expect it will take to build is the solution. We have already shown that we can estimate how long it will take to build something, so all we need is a discount function. Fortunately the field of microeconomics has proposed a solution [. The formula is as follows: $EU = U/(1+d)^t$ where EU is the expected utility, U is the original utility, d is the discount rate, and t is the time when we will benefit from the action. To find a good value for d, we can make an educated guess and then zero in on the best value through a series of experiments.

Experimental Results

How well does this new decision-making algorithm perform? We performed the same experiment with computer players that used the new algorithm. Table 4.6 shows the results of this experiment.

Individual Computer Player Data					
Computer Player 1					
No. Humans	Wins	Losses	Total	Computer Win %	Sampling Error
3	0	8	8	0	±0
2	16	97	113	14.15929	±3.279655
1	774	3226	4000	19.35	±0.624615
Computer Player 2					
No. Humans	Wins	Losses	Total	Computer Win %	Sampling Error
3	1	15	16	6.25	±6.051536
2	16	86	102	15.68627	±3.600883
1	743	3257	4000	18.575	±0.614912
Computer Player 3					
No. Humans	Wins	Losses	Total	Computer Win %	Sampling Error
3	2	13	15	13.33333	±8.777075
2	19	101	120	15.83333	±3.332465
1	722	3278	4000	18.05	±0.608111
Computer Player 4					
No. Humans	Wins	Losses	Total	Computer Win %	Sampling Error
3	2	10	12	16.66667	±10.75829
2	10	91	101	9.90099	±2.97193
1	726	3274	4000	18.15	±0.609421
Composite Computer Player Data					
No. Humans	Wins	Losses	Total	Computer Win %	Sampling Error
3	5	46	51	9.803922	±4.16398
2	61	158	219	27.85388	±3.029195
1	2223	1777	4000	55.575	±0.78564

Table 4.6 Revised Agent Performance

Looking at the case containing the most data, one person playing with three computer players, we see an improvement. The average win percentage for a computer player using the first algorithm is between 15.85% and 17.03%, while the win percentage using the new algorithm is between 17.92% and 19.15%. These results are from computer players that are using a discount factor d = 0.13 and a miscellaneous Development Card utility u = 3.0. These were the best values found through a method of gradually changing each value in separate experiments and looking at which settings produced the best win percentage. The quality of the play is very different using the new algorithm, most noticeably is the tendency of the new computer player to buy Development Cards. Reducing the value of the miscellaneous Development Card utility can curb this behavior, but it was surprising to see that buying cards can be a very effective strategy.

Good behaviors, bad behaviors

Recall that we designed this new algorithm with the goal of addressing the shortcomings of the previous algorithm. One of these was not being able to sense when a move could tip the balance of the game by pulling the leader behind and putting our player ahead. The new algorithm can sense these situations and will put a priority on making moves that take advantage of these situations. For example if there is an opportunity to build a settlement such that it cuts another player's longest road, the new algorithm will calculate how much that move will set the other player back and add it to the value that the settlement adds for our player. The previous algorithm had no way of detecting this situation. Similarly, the new algorithm can detect that it is advantageous to build a road even after our player has the Longest Road award because it makes it harder for the other players to take it away.

Another shortcoming that this algorithm addresses is that the original algorithm would decide to do nothing in some situations. Because we are assigning utility values to all legal moves, our algorithm will pick something to do even if every move has no value.

An advantage this algorithm has over the original on is that it can evaluate moves for their usefulness in achieving multiple goals. For example, building a road can be a step toward extending the longest road and building a settlement. The new algorithm calculates how useful a move is toward attaining the long-term goal of winning the game by simulating that move and then creating a new long-term plan. If a move advances multiple short-term goals, the new plan will reflect this by being much shorter than the original plan.

Another advantage this algorithm has is that it is has the potential for multilevel play. By changing the parameters for the utility for miscellaneous Development Cards and the discount factor, we can make the computer player play worse but still in a reasonable way. For example if we make the discount factor too high, the computer player won't hold out for things that take longer to build. This will result in fewer games won than if the discount factor were at it's optimum level, but the mistakes the player makes will be hard to detect. This is important for not breaking the illusion that the user is playing against an intelligent opponent (Scott 2002).

Even though there are a number of advantages to the new algorithm, there is still room for improvement. One area that both algorithms have trouble with is appreciating the value of open areas when considering where to place roads. This is mostly due to the fact that both algorithms use the same method for estimating how long it would take to win the Longest Road award. The new algorithm does a better job of maintaining the Longest Road award once it has it, but it will still evaluate a possible road that leads to a dead end as having the same value as a road that leads to an open area as long as they both result in at least one possible way to get Longest Road. To fix this we would need to add another utility measure for roads that takes the possibility of future expansion into account. A road with greater opportunity for future expansion would receive a higher utility measure. The somewhat difficult part would be integrating this utility with the current utility measure. This is because the current utility measure is centered on how much closer a move gets us to winning the game, while the potential expansion measure is centered on potentially maintaining the Longest Road award. Future work needs to be done on this topic before a clear solution can be developed.

Another place for improvement is in recognizing situations where multiple moves are required before an impact on the other players can be seen. The new decision-making algorithm can see the impact of a possible build on other players, but only the next possible build. It fails to see situations where two or more builds are required before having an impact on other players. In an attempt to address this problem, we modified the algorithm to do a two-step look ahead. It turns out that the average branching factor of the resulting tree is 13, which means that we had to run the algorithm 13 times more than the one-step look ahead version. This turned out to be too slow to be playable in real time, so no data was collected on its performance. One possible method for looking multiple steps ahead without having to traverse a large tree of possible moves is to consider sets of moves instead of single moves. These sets of moves would be ones that have strategic value, for example any set of moves that results in a Victory Point. Other possible sets of moves would be offensive ones blocking another player's access to a part of the island. When evaluating what to do next, an entire set of moves is simulated and a utility measure is taken. Then the entire set of moves is placed on a queue of things the computer player wants to build next. This is so that if our player has the resources to build more than once, we don't need to run the algorithm again. This is just a proposed solution and future work needs to be done before any conclusions about its effectiveness can be made.

Summary

This chapter discusses how to create a competent computer player for Settlers of Catan. The method employed is a technique of creating a rough plan to guide the agent's behavior. This method allowed us to meet the constraints imposed by our definition of the problem. These constraints included creating a player that was competent and challenging with a quick response in order to maintain human player interest, and a limited use of processing cycles in order to be viable in a commercial product application. The competency of our agent can be witnessed in its ability to win games against human players. As stated above our computer players win between 47% and 63% of the time when competing against a single human opponent. The response time of our computer player is between three and six seconds from making a decision to taking an action. This response time is fast enough that the Java Settlers website we established maintains a large number of human players. We were able to sustain this level of performance on a 400MHz Sun Sparc Ultra 2 running two computer player clients, each client playing between 20 and 30 games. This demonstrates that our technique works well when access to computing cycles is limited. Therefore our technique of creating a rough plan to guide a computer player's actions has been successful in creating an agent that is responsive and competent in a dynamic and complex domain while requiring minimal CPU processing power.

Take Aways

- Using a rough plan as a guide for creating a utility-based measure for making decisions in a dynamic environment with imperfect information is a useful technique for getting behavior that suffices as challenging and intelligent.
- Trying the simplest solution first and then seeing where it fails is a useful methodology for designing intelligent agents.

Chapter 5
Agent Implementation Part III: Negotiation and Trading

Negotiation and specifically trading is a large part of playing Settlers of Catan. Being able to successfully negotiate deals is crucial to being able to win consistently. But how do people typically negotiate while playing Settlers of Catan, and how can we facilitate that in an online version? Also, how can we create a computer player that make offers that are good enough for other players that they will take them, but not so good as to be to the detriment of our computer player? This section will endeavor to answer these questions.

How people negotiate in Settlers of Catan

The motivating factor for trading in Settlers of Catan is the fact that specific sets of resources are required to build while the distribution of resources is randomized. This results in players having some of the required resources to build something they want and having extra resources that they don't need right then. By trading these unwanted resources with other players for the ones that they need, they can build sooner than they could without trading. The following is a typical negotiation dialog:

Alice: Does anyone have clay?

Bob: What are you selling?

Alice: I have sheep and ore.

Bob: I'll give you one clay for two ore.

Alice: I only have one ore, how about I give you one sheep and one ore.

Bob: O.K.

Most trades have a ratio of one for one or two for one, but there are exceptions. The reason most trades have small ratios is because as the game progresses, players gain access to trade ports which allow them to trade three or two for one with the bank. A player can always trade four of one kind of resource for one from the bank. These alternatives to trading with other players mean that no one can charge an exorbitant price for a rare resource. Also, players are reluctant to give away too many resources since it may give the other player such a great advantage that it outweighs the value of the resources received in the trade. On the other hand, players are willing to pay a premium to get resources they can use immediately. Sometimes these resources can mean the difference between winning and losing a race to a spot on the board; in this case a player is usually willing to pay a high premium. In the late stages of the game, trading with the leading player or players happens very infrequently. This is because the risk of giving that player something that helps her end the game is too great.

In addition to regular exchanges of resources, there are other trades that happen among players during a game of Settlers. One of the more common ones is paying a player who is about to move the robber to not put it on one of their hexes. Sometimes this will turn into a bidding war among players who don't want to be robbed. Another thing that happens is people promising to exchange a resource that they don't have at the time the deal is made. This usually occurs because the player making the promise has access to a two to one port that he can use to get the resource, but only on his turn. Another promise that players sometimes include in a deal is to not use the resources to build on a particular spot on the board. If a player has the resources that another player needs but won't sell them because she's afraid they will be used in a way that interferes with her plan, this type of promise can make an offer more palatable to that player. Finally, another type of exchange that occurs, but not very often, is a three-way trade. This exchange involves three players, A, B, and C. Player A has what player B wants, but B doesn’t have what player A wants. But player C does have what player A wants and either A or B have something that C wants. What happens is that B exchanges with C to get what A wants, and then trades that with A to get what B wanted originally. A skilled player can spot when these potential exchanges can occur and profit from them even when they don’t have what the original player was asking for.

Analysis of Negotiation

In the book *Getting to Yes* (Fisher 1991), Roger Fisher and William Ury describe a method of negotiation they call “principled negotiation”. It’s useful here because they use a number of concepts that are good for describing what happens during the negotiation process. We can use these concepts to analyze the negotiation that occurs in Settlers of Catan and compare it to the type of negotiation that goes on in the online version Java Settlers.

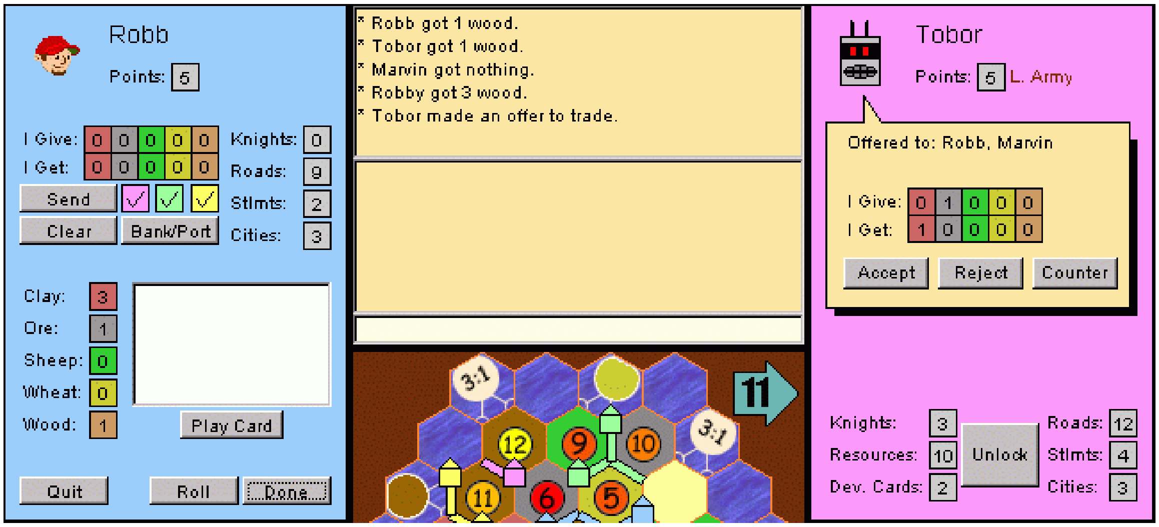
One of the principles of principled negotiation is that to be successful negotiator, one should focus on interests, not positions. Positions are statements of what a party wants out of a negotiation. If the negotiation is over a price, each parties position would be the price that they have currently put forth as what they want to pay or receive for the item in question. Interests are the goals that each party hopes to accomplish through a successful negotiation. These interests drive the kinds of positions each party takes. Thinking and talking about your own as well as the other parties interests makes it easier to generate solutions that are better for both parties. In the *Getting to Yes* terminology, these possible solutions are called options. If the negotiation is successful, the parties will have agreed to one of these options.

Another principle is that before going into a negotiation, one should know what he or she will do if an agreement is not reached. This is called the Best Alternative To a Negotiated Agreement or BATNA. Knowing your BATNA makes it easier to evaluate options that are put on the table. Considering what the other parties BATNA is will give you an idea of what you can realistically expect from the negotiation. Also coming up with a BATNA that depends less on the outcome of the negotiation will strengthen your position.

Using this terminology, we can describe a typical negotiation in Settlers of Catan. Before starting a negotiation, a player determines his interests and his BATNA. In this case an interest is something the player wants to build, preferably on this turn. The player may also have other interests like slowing down the lead player. The player's BATNA is what the player will do if he can't make a deal with someone. In Settlers of Catan, this means making a deal with the bank or a port, or waiting another round of turns. It might also mean changing what the player wants to build to something that takes less time and resources. After determining what he wants to build and what he'll do if he can't reach an agreement, the player will start to probe what the other player's interests are. He does this by asking if anyone is selling one or more types of resources. Players that are interested respond by asking what kinds of resources he's selling. They may get into a discussion of how the resources will be used in order to make sure that they don't have conflicting goals or to find out more about each other's interests. Eventually they will start proposing options in the form of specific amounts of resources to be exchanged. If they reach an agreement, the resources are exchanged; otherwise the process can start over with the player trying to satisfy a different interest.

How do people negotiate in Java Settlers?

In Java Settlers the negotiation process is basically the same, but there are differences due to the differences in negotiating face-to-face and online. In the online game, players have two modes of communication to use when negotiating; the chat interface and the trade interface. The chat interface is a standard chat mechanism where the user types what she wants to say in an input field and the responses from other players are displayed in a larger output area. The trade interface is unique to Java Settlers and allows players to make unambiguous offers as well as exchange resources (figure 5.1). When a player wants to make an offer or exchange resources, she clicks on the numbers in the colored squares to indicate what resources will be exchanged. Then she selects which players will receive the offer and then clicks the send button. The other players see the offer and can respond by clicking one of the response buttons to accept, decline, or respond with a counter offer. Players can also respond using the chat interface. If an offer is accepted, the resources are immediately exchanged.



Robb

Points: 5

I Give: 0 0 0 0 0 Knights: 0

I Get: 0 0 0 0 0 Roads: 9

Stlmts: 2

Cities: 3

Clay: 3

Ore: 1

Sheep: 0

Wheat: 0

Wood: 1

* Robb got 1 wood.
* Tobor got 1 wood.
* Marvin got nothing.
* Robby got 3 wood.
* Tobor made an offer to trade.

Tobor

Points: 5 L. Army

Offered to: Robb, Marvin

I Give: 0 1 0 0 0

I Get: 1 0 0 0 0

Knights: 3 Roads: 12

Resources: 10 Stlmts: 4

Dev. Cards: 2 Cities: 3

Figure 5.1 Java Settlers Trade Interface

For the most part, people will begin negotiation by making an offer using the trade interface. This would be considered strategically poor from the principled negotiation point of view since it is opening the discussion with a positional statement rather than a question or statement about interests. Sometimes a player will use the chat interface to ask if anyone is selling a type of resource, but this is not the norm. In response, people tend to use the accept, reject, and counter buttons over using the chat interface. When they do use the chat interface it is to say that they don't have what the player is asking for or that they are refusing to trade because that player is too close to winning.

The main design goal of the trade interface was to allow players to exchange resources using a minimum number of mouse clicks. In addition we wanted it to be useful as a tool for aiding negotiation. From observing a number of face-to-face and online games we have concluded that the style of negotiation is fundamentally different in each setting. Furthermore, the negotiation community would agree that the style of negotiation in the online version is strategically inferior to the style that is practiced when playing face-to-face. Our hypothesis is that changes to the interface could be made that would facilitate and even help people learn how to negotiate in a better way. Our evidence for this is an experiment we did where a small addition to the interface resulted in a significant change in negotiating behavior.

Before doing this experiment we observed that people online did not make counter offers very often as compared to people playing face-to-face. Since one of our goals was to create an environment that was as close to being face-to-face as possible we felt that the current interface was insufficient. With this interface, if a person wanted to make a counter offer they would use the trade interface in their player area. Our idea was that if we provided a "Counter" button with the "Accept" and "Reject" buttons that are shown with an offer from another player, we would see an increase in counter offers. When clicked, the "Counter" button would pop up a duplicate trade interface right below the current offer. This change may seem counter intuitive since we're just providing the same functionality that is already available to the user for an extra click of the mouse. But by doing this, we hoped to improve the interface by making all of the relevant options more obvious.

For each test condition we recorded data from 100 games where four people played to the end of the game without substitutions. The two conditions were the original interface without the counter offer button and with. Table 5.1 shows the results of the recorded data. Before the addition of the counter offer button, the average number of counter offers made per offer was 0.1. After the addition of the button, the average number of counter offers made per offer was 0.13. This is a 30% increase in the number of counter offers made, an indicator that our hypothesis was correct.

With this knowledge in hand we can speculate that changes can be made to the interface which would facilitate and encourage people playing the game to use the principled negotiation method. One idea would be to replace the trade interface with a few buttons that would be used to communicate messages about interests and options. We want to keep the interface simple, and allow users to create relevant messages with a few clicks of the mouse.

No counter offer button	
Average number of turns per game	71.42
Average number of offers per turn	0.54
Average number of counter offers per turn	0.05
Average number of rejects per turn	1.24
Average number of accepts per turn	0.15
Average number of counter offers per offer	**0.10**
Counter offer button	
Average number of turns per game	70.39
Average number of offers per turn	0.58
Average number of counter offers per turn	0.08
Average number of rejects per turn	1.33
Average number of accepts per turn	0.15
Average number of counter offers per offer	**0.13**

Table 5.1 Negotiation Behavior Data

In Settlers of Catan the most relevant messages about interests are “(Does anyone/Do you) want x?”, “(Is anyone/Are you) selling x?”, “I don’t have x.”, and ”I want x.” where x is a type of resource. The first two messages are statements that are usually posed by people who want to start a dialog, while the second two are statements in response to offers that aren’t satisfactory. With that in mind we can determine the best context in which to present buttons that generate these messages. The most logical place to find buttons to start a negotiation dialog is in the player’s personal information area where the trade interface is currently located. There would be two buttons to ask the two questions concerning buying or selling and one button to compose an offer. Even though it’s not recommended to start the negotiation with an offer, we don’t want to take that option away from the user. We don’t want to force the user to use the recommended method, just make it easy for them to do so. After clicking on one of these buttons to ask about interests, we need to prompt the user as to what kind or kinds of resources they want to talk about. This could be done with a set of five checkboxes, one for each resource type. Finally, we prompt the user to decide to whom the message should be addressed. Other players would see a speech balloon appear below that player’s face icon displaying the message along with some buttons to compose a response. These buttons would include ones for making the responses described above along with just a simple “yes” or “no”, and a button for responding with an offer. To measure the effectiveness of these changes, an experiment similar to the one described earlier could be done looking for evidence of people negotiating more effectively. This evidence would manifest itself as a reduction in the number of offers and an increase in the number of accepted offers. These changes and the associated experiment are left as future work.

An algorithm for trading

Trading is an integral part of playing Settlers of Catan, and so our computer players need to be able to trade in order to participate in all aspects of the game. The following will be a description of an initial attempt at an algorithm designed to generate plausible offers using ideas from the principled negotiation method. Part of this algorithm will also be used to evaluate offers made to our computer player to decide if it should accept, reject, or make a counter offer. We will evaluate this method in the same way we have evaluated previous behavior algorithms, by sampling a large number of games consisting of at least one computer player and one human opponent and measuring the win loss ratio.

The first step in our trading algorithm is to determine our interest; in this case it is what we want to build. This is done by using the decision-making algorithm described earlier. If we have the resources we need to build what we want, we can stop. Otherwise we need to determine our BATNA. This is done by looking at what resources we need to hold on to and what resources we can use for trading. If we have enough extra resources to trade with the bank or a port, then the best thing we can do if we can't trade with another player will be to trade with a bank or a port to get the kind of resource that is hardest for us to get. If we don't have enough resources to do this, our BATNA will be to simply wait another round of turns. After determining our BATNA we need to generate offers to consider proposing. This is breaking with the principled negotiation method of finding out interests before creating offers or options, but this is a first attempt at an algorithm for trading and so we want to make the simplest thing that works first and then see where it fails. Also the current player interface doesn't facilitate communicating interests easily with the computer players without the use of natural language generation and understanding. This will be addressed in the results section later on.

The first offers we will generate will be trading one resource for another, starting with giving the resource that is easiest for our player to get in exchange for a resource that is the most difficult to get. Before making this offer we want to make sure it is reasonable. This is because our computer player would be very annoying if it proposed every offer it generated. An offer is plausible if it is better than our BATNA and we think someone will take our offer. To determine if an offer is better than our BATNA we estimate how long it would take to build what we want given what we have now compared to how long t would take given the result of the offer. If the time is shorter after the offer, then the offer is better then our BATNA. In this case we are sure it will be better since we are trading a resource we don't need for one that we do. To determine if another player will take our offer we need to keep track of two things, what resources they have received recently and what offers they have rejected. Our tracking of the other players resources will be good but not perfect since we lose track after a person discards or has been robbed. If we lose track, we can still rely on what offers they have rejected as a clue to what resources they are willing to part with. For our algorithm, we will say that if a player rejects an offer, it means they aren't selling anything that was asked for in that offer. This is a very conservative approach, but we feel it is better to err on the conservative side and have the computer player make too few trade offers than to make too many.

To avoid making a lot of offers that would be rejected, we only want to make an offer if we think someone has what we are asking for and we think they will be willing to sell it. If there is at least one player who fits these criteria we do one final sanity check on the offer. For each player that is a prospective seller, we make a guess at what they want to build next and their BATNA. If our offer is better than their BATNA, then the offer is made, otherwise we generate a better offer and check it in the same way. Again, this method may throw out offers that the other player might have taken, but we prefer that the computer player make too few trade offers rather than too many.

After exhausting all possible offers where we give one unneeded resource for one needed, we then consider giving one needed resource for a different one that we needed. This makes sense if we trade a resource that's easy for us to get for one that is very hard get. Finally we consider trading two resources for one needed resource starting with combinations of unneeded resources and then needed. As long as an offer is better than our BATNA we are sure that the deal is not bad for us, at least in terms of resources gained and lost. But there may be reasons not to make an offer even if it's better than our BATNA and we think another player will take it.

If the player who takes our offer uses the resources to win the game, then the deal was not in our favor. The deal was also not in our favor if the player who took our offer uses the resources to beat us in a race to build on a spot where we wanted to. Therefore we need to do two more tests before making an offer. First we won't make an offer to anyone who is too close to winning. This is determined by estimating the ETW for each player. Any player under a predetermined threshold is considered too close to winning and poses too much of a risk to trade with. The other test uses our guess at what they want to build next. If it interferes with what we want to build, then we won't trade with that player. Both of these tests are things that human players are aware of when they are playing the game. If our computer player didn't look for these situations, it would appear unintelligent as it made a deal which gave the game away or allowed another player to block it's progress.

Counter offers

Counter offers are made in basically the same way except we want to make an offer that makes sense given the offer that was just made. This means that we want to make a counter offer that either asks for something different while still giving what they want, or give them something different that still achieves their goal while accepting what they are offering. To do this we perform the same procedure described above except we reject all possible offers that don't have something that is in common with the offer we are responding to. To illustrate what kinds of counter offers can be generated using this method, imagine that a player offers to give our player one clay for one ore. Two valid counter offers would be to give two wheat for their clay, or to give one ore for one wood. Making counter offers in this way signals that we either have what their asking for or want what they are offering. If both parties have what the other one wants then as long as they both make counter offers they are happy with, then they should close in on an offer that is good for both of them.

Tests

To test our algorithm, we performed the same data sampling technique used before. We let people play against one or more computer players and then recording the scores from games that were played to completion with no player substitutions. We then took a random sample of at most 4000 games from each category: one person with three computer players, two people with two computer players, and three people with one computer player for each computer player and calculated the percentage of games won by that computer player. We also took a random sample of all the games and measured the percentage of games won by any computer player. This composite data provides an idea of how often a person loses to a computer player.

Results

The data in table 5.2 shows that the initial attempt at a trading algorithm decreases the percentage of games won by the computer players. Without trading, the computer players won about 55% of the time when playing against one person. Using the trading algorithm described above they win about 51% of the time. This indicates that the deals that the computer players are agreeing to are on the whole not in their favor. As was pointed out earlier, the type of negotiation this algorithm performs deviates from the principled negotiation method since it makes offers before finding out about the other player's interests. Our method for generating counter offers and tracking what offers each player rejects are an attempt at gaining information about each player's interests without having an explicit dialog about them. A modified version of this trading algorithm that adheres closer to the principled negotiation method may prove to be better. To do this the player interface needs to be changed to facilitate explicit discussions of the players' interests. With a better interface we could better determine the other player's interests and BATNA, which in turn would allow our computer player to be a more effective negotiator.

Individual Computer Player Data					
Computer Player 1					
No. Humans	Wins	Losses	Total	Computer Win %	Sampling Error
3	3	33	36	8.333333	±4.606423
2	43	264	307	14.00651	±1.980745
1	686	3314	4000	17.15	±0.596003
Computer Player 2					
No. Humans	Wins	Losses	Total	Computer Win %	Sampling Error
3	3	23	26	11.53846	±6.265627
2	41	256	297	13.80471	±2.0016
1	701	3299	4000	17.525	±0.601119
Computer Player 3					
No. Humans	Wins	Losses	Total	Computer Win %	Sampling Error
3	3	30	33	9.090909	±5.004381
2	45	259	304	14.80263	±2.036788
1	688	3312	4000	17.2	±0.596691
Computer Player 4					
No. Humans	Wins	Losses	Total	Computer Win %	Sampling Error
3	5	35	40	12.5	±5.229125
2	36	256	292	12.32877	±1.923964
1	688	3312	4000	17.2	±0.596691
Composite Computer Player Data					
No. Humans	Wins	Losses	Total	Computer Win %	Sampling Error
3	14	121	135	10.37037	±2.623953
2	165	435	600	27.5	±1.822887
1	2059	1941	4000	51.475	±0.790225

Table 5.2 Negotiating Agent Performance

Summary

As we have seen, the ability to negotiate and trade is an important factor in playing Settlers of Catan. Our model for guiding our agents approach to negotiation was based on the ideas of principled negotiation. Specifically we used the idea of establishing a BATNA as a guide for evaluating offers made during a negotiation. This approach resulted in an agent who negotiates trades for resources in a way that makes sense to human observers. Through experimental observation, we found that the addition of our negotiation algorithm resulted in an agent that performed worse in terms of percentage of games won against a single human opponent as compared to the agent that doesn't negotiate. We believe this is due to incorrect assumptions our agent makes about the other players' interests and goals based on their behavior. If the agents could communicate more expressively about these interests, they would be able to gather more accurate information and therefore make better decisions during negotiation. We would also like to explore other aspects of the principled negotiation method and integrate them into our agent.

Currently the user interface for facilitating negotiation in Java Settlers works well for allowing people to communicate offers easily and unambiguously. It does not work as well in facilitating the use of principled negotiation. Our experiments show that small changes to the user interface can have large effects on user behavior. Specifically, providing context sensitive access to functions that are relevant to the task the user is performing can make it easier for the user to do what he would normally do in the analogous situation in the real world. This is important for artificial environments that are meant to reproduce ones in the real world. We believe with more research, the interface could be improved to allow users to negotiate using the principled negotiation method in an intuitive way. The Java Settlers platform could then be used as a teaching tool for people who wish to learn this method of negotiation.

Take Aways

- BATNA is a useful concept for designing an algorithm for negotiation. Other aspects of principled negotiation may also prove to be useful in agent design.
- The design of the user interface has a large effect on how people using the system negotiate. Providing the right options in context can help people negotiate better.

Chapter 6
Related Work

Introduction

The problem of creating a competent agent for playing Settlers of Catan is a planning problem in a multi-agent, imperfect information domain with uncertainty. In order to provide a background for work in the Settlers of Catan domain it is our intent in this chapter to discuss the current state of the art of game playing. We will look here at what work has been done in several types of game playing problems including both perfect and imperfect information situations as well as two player and multi-player environments. Our strategy for creating a competent agent for Settlers of Catan is a reactive planning approach using ideas borrowed from utility theory. We will therefore also examine what has been done with regard to reactive planning and approaches that use utility theory for decision making.

Related work in game playing

2-player turn-based perfect information games

To date traditional AI techniques have had the most success in two-player, turn-based, perfect information games. These include games such as chess, checkers, and Othello. The predominant technique used for deciding what move to make next in agents that play these games is the minimax algorithm (Epstein 1999; Schaeffer 2000). The intuition behind this algorithm is that before I make my move I want to consider what my opponent could possibly do in response. The best response by my opponent determines how good that move is for me. The name comes from the idea that I am trying to maximize the utility of my moves while my opponent is trying to minimize it (Luce 1985).

Agents which have been designed to play chess, checkers, and Othello using the minimax algorithm have been very successful, where success is measured by the skill level at which they play. In 1997, Deep Blue, a chess playing program running on specialized hardware beat the world champion chess player Gary Kasparov 2 games to 1 with 3 draws (Fogel 2002). That same year Logistello, an agent which played championship level Othello beat the world champion 6 games to 0 (Buro 1997). In 1990, Chinook beat the second best checkers player in the world (Schaeffer 2000).

In Settlers of Catan we are faced with a four-player environment in which not all information can be known. Minimax can not operate well in this domain because in order to get feedback for one of our moves we must consider the actions of three other players rather than just one. Additionally in Settlers of Catan players can make as many actions as they have resources for. This ability for opponents to make multiple moves combined with the unknown variable presented by the dice rolling makes the minimax search space grow very large very fast. This method is too slow for reasonable play and requires too much processing power to be viable for use in our context.

Multiplayer turn-based perfect information games

Multi-player turn-based perfect information games require the use of different techniques. In these games an additional layer of complexity is introduced with additional players. Minimax no longer works well as an algorithm since the intuitive notion of a single opponent no longer applies. If we were to try to consider all of the other agents in the environment as a single opponent the search space for the minimax procedure would become very large very quickly due to combinatorial explosion.

An example of a multiplayer turn-based perfect information game is Diplomacy (Calhamer 2000). In this game each player represents one of seven European powers just before the start of World War I. The object of the game is for one power to gain control over the majority of Europe by moving pieces representing armies and fleets. Combat is resolved in a simple way where the power with the most armies or fleets attacking or defending an area wins the battle. Movement is performed simultaneously by having players write down where they want their units to move before resolving combat and movement. During this time players are free to discuss plans and negotiate cooperative maneuvers.

In designing an intelligent agent to play Diplomacy, Sarit Kraus et al employed a distributed AI architecture (Kraus 1995). Their Diplomacy playing agent Diplomat consisted of a coordinated group of smaller agents representing the different parts of a government. The Prime Minister agent decides what course of action to take by looking at the suggestions and information provided by the other agents. The other agents provide cost benefit analysis of different courses of action for itself and the other players in the game. The intuition behind this method is that in a multi-player situation, not every player represents an immediate threat. Therefore one must consider the benefit of a course of action weighed against the risk of helping another player.

Sometimes this risk is minimal because the player a move helps is not in the lead. Also, there are moments when helping another player take down the leader is beneficial. This distributed AI approach lead them to create a competent agent that performed at least as well as the other people who participated in the game.

For the game-playing agent in Java Settlers, we did not use a distributed AI architecture in its design. This is not because we don't think that this approach won't work but because we adhered to the principle of Less is More throughout the development process. Our approach was to start with the simplest solution and see where it fell short. If we had discovered that breaking the agent into separate autonomous units was an improvement over what we had already built then we would have used that architecture.

Related work in reactive planning

Since my work could be considered a reactive planning technique we will describe some of the research that has been done in this area.

Pengi was made in response to the notion that you needed a planner to get planful behavior (Agre 1987). Pengi played a video game called Pengo where the player maneuvers a penguin to push blocks of ice and avoid killer bees. Pengi was a purely reactive system and did not reason about possible future states of the environment. Instead it used a set of rules that were indexed according to possible situations the agent could be in, like “when you are being chased, run away”. This research was the beginning of what became a large body of research in reactive planning. Using a purely reactive system architecture similar to the one used for Pengi for an agent in Java Settlers would probably not work as well as the design we used since there are many times in Settlers when one must consider future consequences to determine the best action to take in the moment. But the idea of leveraging the environment as much as possible to drive behavior has been used extensively in the Java Settlers agent.

Firby’s RAPs (Reactive Action Packages) system was created as a possible solution for the problem that symbolic planners were not useful on their own for an agent that had to act in a constantly changing environment (Firby 1989). The RAPs architecture consists of a library of hierarchically related RAPs, a working memory, a RAP interpreter, and a task agenda. During each cycle of execution the top task is selected from the task agenda, then its goal conditions are checked against the working memory. If they are not met, the RAP is executed. If it is a primitive action, this action is passed onto the hardware of the agent. Otherwise it is passed to the RAP interpreter, which breaks the

RAP down into smaller RAPS and then puts them on the agenda. Then the cycle repeats itself until no more tasks are on the agenda.

TRUCKWORLD was created as an environment to test the RAPs system (Firby 1989). In this environment, the RAPs agent drove a delivery truck in a simulated hostile war zone. The truck had to pick up and make deliveries while avoiding various dangers on the battlefield. The truck agent could only gather information from its sensors and therefore had incomplete information about the entire world state. Other agents in the world could effect changes in the world without the truck agent's knowledge. The RAPs architecture was later used for controlling a mobile robot with a gripper for tasks such as cleaning up an office space.

The idea of hierarchical plan representation used in RAPS was the inspiration to use a high-level representation in our agent to compose a overall plan for winning the game. We opted not to implement a RAPS system to control our Java Settlers agent because in addition to being a problem of planning, deciding what to build in Settlers of Catan is also a problem of optimization. Once a plan for how to win the game has been found, the steps in that plan have some flexibility in terms of their order dependency. The order in which the steps are carried out determines how long it takes to implement that plan, and because Settlers is a race this aspect of the plan is very important. The RAPS architecture lends itself to problem domains where tasks are measured as either completed or not completed. There is no simple way using the system to evaluate how well a job will be completed in the future given a possible action to be taken now.

Subsumption architecture

Brooks' subsumption architecture was created with the philosophy that one does not and should not explicitly represent the world in the agent (Brooks 1986). Rather, the agent should be built as a set of behaviors that are linked directly to sensors that respond to stimuli in the environment. These behaviors are arranged in a hierarchy with the lowest being the most basic kind of behavior like moving forward, to higher behaviors like seeking an object. High-level behaviors interact with low-level behaviors by activating or inhibiting them. The representation of the behavior is not important; in fact the simpler it is the better. Brooks used simple finite state machines in his insect-like robots. These robots demonstrated that the subsumption mechanism could be used to build functional robots that act robustly in highly dynamic environments. The ideas behind the subsumption architecture formed the basis for the area of research now known as behavior based robotics.

Again, the notion that the behavior of the agent should be grounded in the environment is illustrated here. This research also demonstrates the Less is More principle being applied to robot design.

Behavior networks

One of the architectures inspired by the subsumption architecture was Maes' behavior networks (Maes 1990). Maes designed the behavior network architecture with the goal of creating an action-selection architecture capable of the following:

- Prefer actions which contribute to multiple goals
- Take advantage of opportunities
- Look ahead to anticipate and avoid hazardous situations
- Act in a reactive and fast manner

The behavior network architecture is composed of a set of competence modules that consist of a tuple (c, a, d, α) where c is a list of preconditions that must be true before the module can become active, a and d are add and delete lists of the classical form, and α is an activation level. What a competence module does is not specified, it may enable an actuator, perform a calculation, or just enable other modules. Modules are linked according to their preconditions, and add and delete lists. Modules that add propositions are connected to modules that require those propositions with a successor link. A predecessor link is connected in the opposite direction. Finally conflicter links are made between two modules where the delete propositions match the preconditions for the other module. These links control the flow of spreading activation which enable the module with the most activation energy to be activated. Opportunistic vs. steadfast behavior and thoughtful vs. reactive behavior can be controlled by changing the values of global variables which control how much activation energy is in the entire system and how sensitive each module is to it.

Maes demonstrated the performance of her system in a simple simulated domain, but others have used it to control agents in the RoboCup robotic soccer competition (Dorer 1999). This has many of the same design goals as our system: preferring actions that contribute to multiple goals, opportunism, anticipation of problematic situations, and acting in a timely manner. One of the drawbacks to using this technique is that there is no way to easily manipulate numerical variables because of it’s use of first order predicate calculus as a representation language.

Related work in the use of utility theory

Utility theory was created in the domain of Economics with the goal of producing mathematical models that would predict the behavior of rational agents such as companies, countries, or even people.

Utility theory states that an agent must have preferences over different possible outcomes of different courses of action in order to make a decision on what to do (Luce 1985). These preferences are captured in a quality called utility represented by a single real number associated with each possible outcome. The function that maps states of the world to these numbers is called a utility function, denoted $U(S)$. A nondeterministic action A from the current state will have possible outcome states $Result_i(A)$, where i ranges over the possible outcomes. Before taking action A, the agent assigns a probability $P(Result_i(A)|Do(A),E)$ to each outcome, where $Do(A)$ is the proposition that the action was taken and E is the available evidence about the state of the world. With this we can calculate the expected utility $EU(A|E)$ of taking an action A from a state given the current evidence E:

$$EU(A \mid E) = \sum_i P(Result_i(A) \mid E, Do(A))U(Result_i(A))$$

The principle that a rational agent will always choose the action that will result in the highest expected utility is called the principle of maximum expected utility (MEU) (Howard 1977). This principle encapsulates the intuitive notion that a rational agent will always do what it thinks is best for itself. This principle is the basis for most AI techniques that make use of utility theory.

In Java Settlers, our agents calculate only the utility of each action taken from the current state. The agent then discounts each utility based on how long it predicts it will take to perform that action. This is different than calculating the expected utility according to the above equation which relies on using the probability of the outcome given the action. Determining the probability of a particular outcome is difficult in an environment with many self-interested agents, since there are so many possible things those agents could do to effect the future. We feel that the use of discounted utility over time as a measure of expected utility is a satisfactory measure for our purposes.

Value iteration

Value iteration is a dynamic programming technique used to find the utility of taking any action from a particular state in a state space (Howard 1960) this is also know as a policy. After this policy has been determined, an agent can simply follow it using the current state of the environment as input to the policy and taking the action that leads to the next state with the highest expected utility. This technique only works for state spaces where the utility function on histories of states (U_h) has the property of separability. This means that there exists a function such that

$$U_h([s_0, s_1, ..., s_n]) = f(s_0, U_h([s_1, ..., s_n]))$$

The value iteration algorithm is based on the following equation which uses a reward function $R(i)$ and a model of the environment M_{ij}^{a} that gives the probability of reaching state *j* if action a is taken in state *i*:

$$U(i) = R(i) + \max_a \sum_j M_{ij}^{a} U(j)$$

The second term of the right-hand side of the equation makes sure the agent follows the policy of maximum expected utility. Because the utility of state *i* is expressed as a function of its successors, one can use dynamic programming to find *U*(*i*) for any state. The cost of solving this decision problem is no more than $O(n|A||S|)$ where *n* is the number of steps, |*A*| is the number of possible actions at each state, and |*S*| is the number of possible states. This method works well in environments where the state space of the environment doesn’t change often. For example a room with no moving entities could be divided into a number of states connected by movement actions. A policy could then be calculated for the best way to get around the room. If new obstacles are introduced, the policy must be recalculated to take the changes in the state space into account. If the state space is small enough, this can be computed in real-time.

This technique is related to our work on Java Settlers in that it uses utility to determine what action to take next. Our approach differs in that we don’t try to calculate the utilities for all of the states from where the agent is now to the goal state. Rather we only calculate the utility of the states that can immediately follow the current state and we use a rough plan to estimate the series of states that lead from that state to the goal state.

Q-Learning

Q-Learning is a form of temporal difference learning used to learn the optimal policy for what action to take given a state in a state space (Watkins 1989). It is one of the most popular reinforcement learning techniques (Kaelbling 1996). One of the useful qualities of Q-Learning is that unlike value iteration, a probabilistic model is not required in order for the technique to work. The Q-Learning algorithm learns an action-value function $Q(a,i)$ where a is an action that can be taken in state *i*. These values are called Q-values. Q-values are related to utility values by the following equation:

$$U(i) = \max_{a} Q(a,i)$$

Intuitively this means that the utility of being in state *i* is the same as the utility of the best action we could take in that state. These values are learned by taking actions in the environment than then recording the reward feedback *R*(*i*) from the results of the action. The equation that captures this notion is:

$$Q(a,i) \leftarrow Q(a,i) + \alpha(R(i) + \max_{a'} Q(a', j) - Q(a,i))$$

which is calculated after each transition from state *i* to state *j*. In this equation, α is the learning rate parameter, which controls the amount of change that can occur to a value on an iteration.

This technique is useful for learning optimal policies for acting in state spaces when no transition model is available. When using this technique, the number of iterations required before it converges on a policy grows quickly with the size of the state space (Kaelbling 1996). It has been successfully combined with neural networks where the Q-Learning technique is used to learn the best settings for the weights in the network. The most famous demonstration of this technique is the program TD-Gammon which learned to play Backgammon at a very competitive level using only raw board information as input to the network (Tesauro 1994).

Our Java Settlers agents do not make use of any machine learning techniques. In future work, trying a similar technique to the one used in TD-Gammon to make decisions in Java Settlers may prove to be an interesting experiment. This is because there are similarities between the environments but there are also important differences: four players as opposed to two; multiple moves can be made on a turn; and the two main tasks a player must perform, trading and building, are intertwined but are very different. Would a single network suffice to control both behaviors, or would two networks be required. There are many issues other than these that could be explored using the Java Settlers environment as a test bed.

Summary

This has been a brief overview of some of the ideas that have been developed for solving the problem of how to make an agent that makes rational decisions in the face of a complex dynamic environment. Which technique is best depends a lot on what the agent is expected to do and the specifics of its environment.

Take Aways

- If the environment is very dynamic, a reactive approach is probably called for.
- Utility theory has inspired a number of algorithms for decision-making and is a good starting point for thinking about how an intelligent agent should make decisions.
- Using the environment as much a possible to guide behavior has proven to be a useful technique.
- Less is More: simple approaches can result in complex behavior that appears intelligent.

Chapter 7
Conclusion and Future Work

Our original problem was to create an agent that would be able to make decisions within a reasonable time frame while maintaining compelling and sophisticated behavior using limited processing power. We arrived at this problem by choosing a real world task. We found this to be a compelling approach because it eliminated the temptation of creating a problem to fit a predetermined solution. In addition, our hope was to end up with an artifact that would be useful to people outside the research community as well as a tool for research. We feel that we have succeeded in achieving our original goals, and in the process of doing this we have created a tool for studying methods for planning in adversarial, incomplete information environments. In this chapter we shall summarize some of the advantages and successes of our solution in addition to realms in which further study could be fruitful.

Our solution utilizes a decision-making strategy that involves the creation of a rough plan to guide reactive behavior. In addressing the negotiation aspect of the game we used strategies adapted from business negotiation literature. In particular the concept of finding a BATNA for both parties and using it to guide the value of possible offers on the table was found to be very useful. The process that led us to this solution was to begin with the simplest idea first, implement it, and then see where it fails. We then used observations derived from that initial attempt to improve upon it, and continued this process until we arrived at a solution that was able to satisfy our original goals.

Decision making

There are a number of areas in which our solution to the decision problem performs well. One of these is sensing whether an action will affect other agents in the environment and whether it is a positive or negative effect and to what degree. Our agent uses this ability to sense when a move will take a leading player down, or when a move may inadvertently help another player. It also uses this ability to see the value of maintenance actions that don't gain the agent anything except making it harder for other agents to do something that would upset that goal. One example of this behavior is when an agent builds roads after gaining the Longest Road award. These aspects of our decision-making algorithm are useful for many other tasks involving multiple self-interested agents whether for cooperation or competition.

Another area where our solution performs well is in evaluating moves for their usefulness in achieving multiple goals. By simulating a move and then creating a new long-term plan the agent can assess which moves contribute to the plan in more than one way. This aspect of our solution is useful in domains where conserving resources that are required for taking actions is important.

Our solution also makes it easy to tune for players of different skill levels. By changing parameters that affect the utility calculation we can make the computer player play worse but still in a reasonable way. For example if we increase the discount factor for an action above the optimal level, our agent will neglect beneficial actions that may take longer to achieve. This will result in fewer games won than if the discount factor were at its optimum level, but the mistakes the agent makes will be hard to detect because it will still play in a reasonable way. This is important in order to maintain the illusion that the user is playing against an intelligent opponent.

One area where our solution falls short is in recognizing situations where multiple moves are required before an impact on another player can be seen. A possible solution to this issue is to do a two-step look ahead, but for some domains this may be computationally too expensive. Another possible method for looking multiple steps ahead without having to traverse a large tree of possible moves is to consider sets of moves instead of single moves. These sets of moves would be ones that have strategic value when performed in a series. When evaluating what to do next, an entire set of moves is simulated and a utility measure is taken. Then the entire set of moves is placed on a queue of things the computer player wants to do next. This is just a proposed solution and future work needs to be done before any conclusions about its effectiveness can be made.

Trading

Our data shows that our initial attempt at a trading algorithm decreases the percentage of games won by the computer players by about 4%. This indicates that the deals that our agents are agreeing to are on the whole not in their favor. As was pointed out earlier, the type of negotiation this algorithm performs deviates from the principled negotiation method since it makes offers before finding out about the other player's interests. A modified version of this trading algorithm that adheres closer to the principled negotiation method may prove to be better. To do this the player interface needs to be changed to facilitate explicit discussions of the players' interests. With a better interface we could better determine the other player's interests and BATNA, which in turn would allow our computer player to be a more effective negotiator. Our negotiation method does fulfill our goal of creating an agent that performs negotiation in a reasonable manner. This has been determined through observation and feedback from users of the system.

Future work on negotiation

We would like to put forth Java Settlers as an artifact that may have some value as a tool to study negotiation. This tool is not a substitute for data collected from student role-playing negotiation scenarios. It is however unique in that it draws thousands of people from around the world who participate repeatedly. We can study negotiations between humans as well as between humans and machine controlled agents. What are the different dynamics between these different scenarios? Do people treat machines differently? If so, how? Java Settlers provides a rich environment for practicing negotiation while making it tractable for the researcher to represent what is being negotiated.

Future work on natural language

Java Settlers provides a good setting for doing research in natural language understanding and generation. This work could be used to understand conversations that happen over chat services. The language in these conversations is very truncated compared to conversations that happen face-to-face or over the phone. Words are often abbreviated, and knowing the context of the message is usually important for understanding it. Since in Java Settlers most messages are about the game, the researcher can leverage this context when designing the natural language system.

Comments

Something that people engage in while playing on Java Settlers is kibitzing. This often takes the form of comments on rolls of the dice, or where someone placed a piece. The framework for the agents in Java Settlers provides representation that can be useful for generating these comments at appropriate moments. This would be one way of creating an agent with more compelling, humanlike behavior. This information can also be used to aid in understanding messages.

Trading

Currently, the only way to communicate trade offers with the computer players is to use the trading interface. People also use this interface to communicate trade offers with each other, but they sometimes use messages to augment the offers. This usually occurs when rejecting an offer, since the only feedback the interface gives is a rejection message. Natural language could be used to increase the expressiveness of the agents by providing additional feedback. It could also be used before making an offer to gather information on what other players are willing to buy or sell.

Arguing/convincing

An important aspect of the game that the agents do not participate in currently is that of drawing attention to the player they think is in the lead. This is important for playing the game skillfully because if the players agree on who is in the lead, they can cooperate to slow that player down. This also means that an important skill is defending oneself when accused of being in the lead by arguing against it. Both of these skills have been studied by AI researchers in other domains, but not with a large group of test subjects. Java Settlers could serve as a platform for furthering this research because it provides a large group of willing participants.

Develop tools for use as teaching aid

The Java Settlers system was designed so that we could easily create experiments for testing hypothesis on decision-making algorithms, negotiation methods, and interface design issues. Currently any skilled Java programmer could make small changes to try a different decision-making algorithm, or record data on a specific behavior. With some more development, Java Settlers could be made easier to modify which would make it useful as a teaching aid for a course in AI.

The Java Settlers system could also be used as a teaching aid for a class in strategic thinking or negotiation. The game provides many opportunities for strategic decision-making, negotiation, and diplomacy. One could imagine a curriculum where students were taught techniques for negotiation and then told to play a game on the Java Settlers system. The teacher could then pick a recorded game from the database and show an example of the negotiation technique being successfully used by one of the students. The issues of resource management, cooperation with competitive agents, and strategic planning are all found in many business situations. And the aspects of the game could be changed to another setting that was closer to a modern business scenario and still not change the game mechanics to suit a more professional audience.

Other domains

We believe that our decision-making algorithm will work for agents in other complex domains. These would be other games that require the non-player characters to act in a strategic planful manner in the face of a changing hostile environment. Examples of this would be turn-based strategy games, or adventure games were the non-player characters are autonomous. This technique could also prove useful for agents in simulated environments used for education. An example of this would be a simulation where the player had to successfully start and expand a franchise business. Our techniques could be used to control other agents competing in the same market. The player could then use the system to learn and practice techniques for strategic planning and negotiation.

Summary

Our main goal was to create an agent that would be able to make decisions in a complex adversarial environment within a reasonable time frame while maintaining compelling and sophisticated behavior using limited processing power. We feel that we have succeeded in achieving this goal, and in the process we have created a tool for studying methods for planning in adversarial environments using incomplete information. The heart of our agent is a decision-making strategy that utilizes a rough plan to guide reactive behavior. This solution has three important and useful properties. First, our solution can signal when a potential action might affect other agents in the environment, whether it will have a positive or negative effect, and roughly to what degree. Secondly, our solution performs well in evaluating moves for their usefulness in achieving multiple goals. Thirdly, it can make a reasonable decision in a short amount of time using limited processing resources.

Our agent also can negotiate using a strategy based in ideas found in the business negotiation literature. Even though there is room for improvement in this area, we feel we have made a good start by laying the foundation for more research on this subject.

To do this research we have created the Java Settlers system. It is a robust, stable service capable of supporting hundreds of simultaneous users and capable of recording data on any aspect of behavior of the people or agents using the system. By conservative estimates, Java Settlers currently receives over 10,000 visitors a month and has a thriving competition ladder created by fans of the site. We feel that this, in addition to the fact that half of all games played are solo games where one person plays against three agents, is evidence that we have created an easy to use system with an intuitive user interface and agents which exhibit compelling intelligent behavior.

We also feel that our agent design is a good example of the idea that less is more. By this we mean that when designing an agent for a real-time system, one should focus on getting it to perform 80% of the desired behaviors competently since this can be done using relatively simple methods and suffices for most situations. This is an important concept to keep in mind when designing agents to operate in real-time environments such as entertainment software or simulations used for training and education.

Take Aways

- Using a rough plan as a guide for creating a utility-based measure for making decisions in a dynamic environment with imperfect information is a useful technique for getting behavior that suffices as challenging and intelligent.
- Trying the simplest solution first and then seeing where it fails is a useful methodology for designing intelligent agents.
- Less is More: simple approaches can result in complex behavior that appears intelligent and suffices for most situations.

Appendix A

Java Settlers Coordinate System

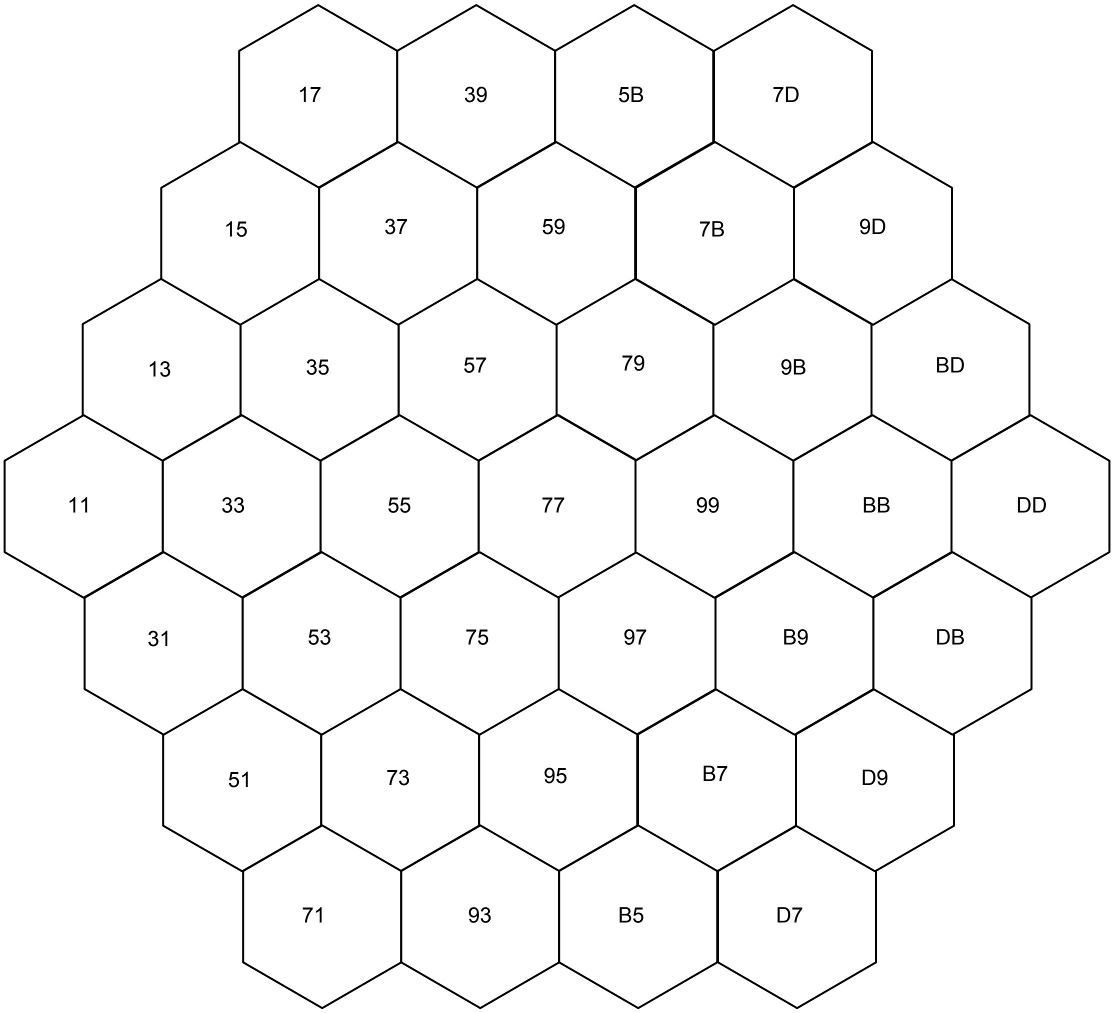


Figure A.1 Java Settlers Trade Interface

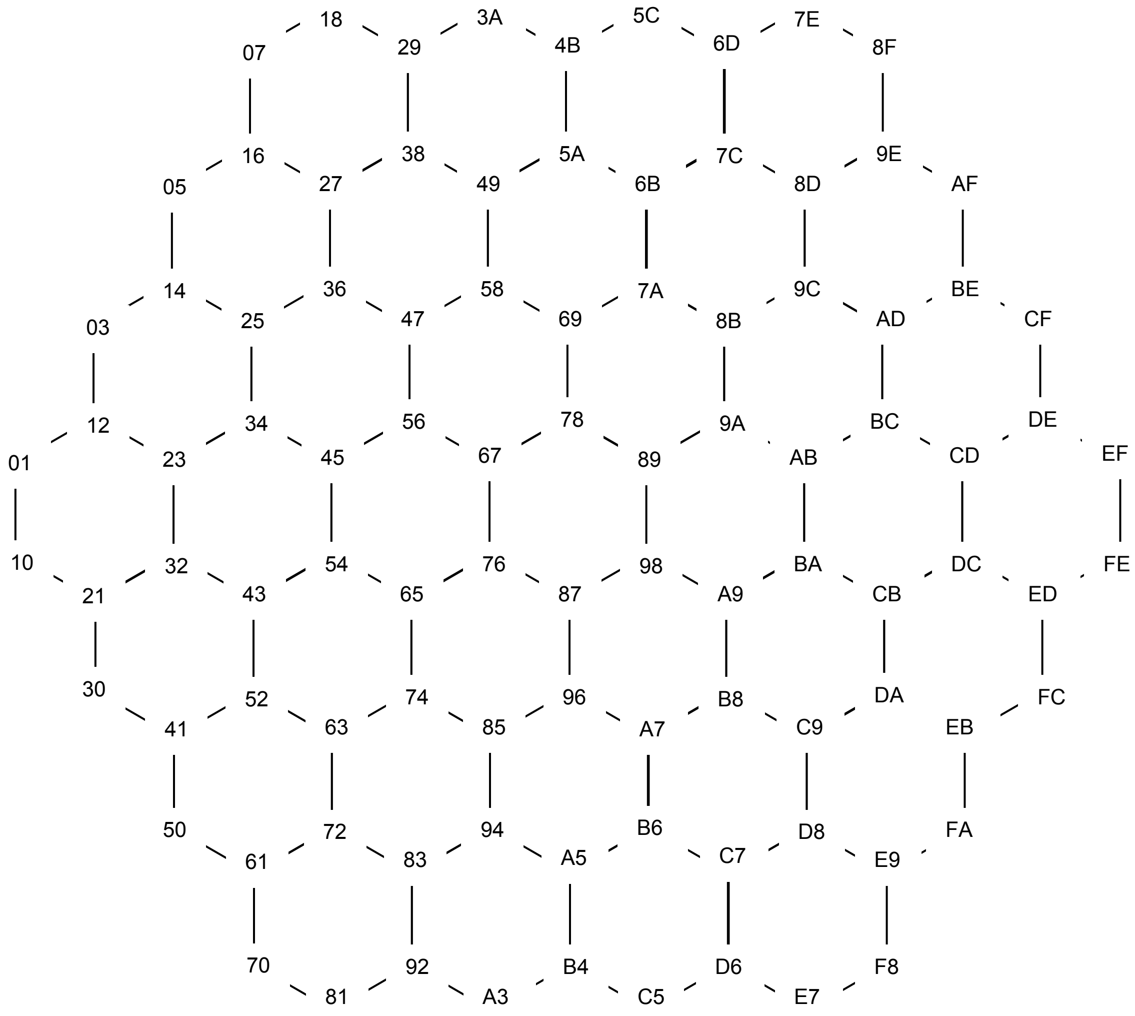


Figure A.2 Node Coordinates

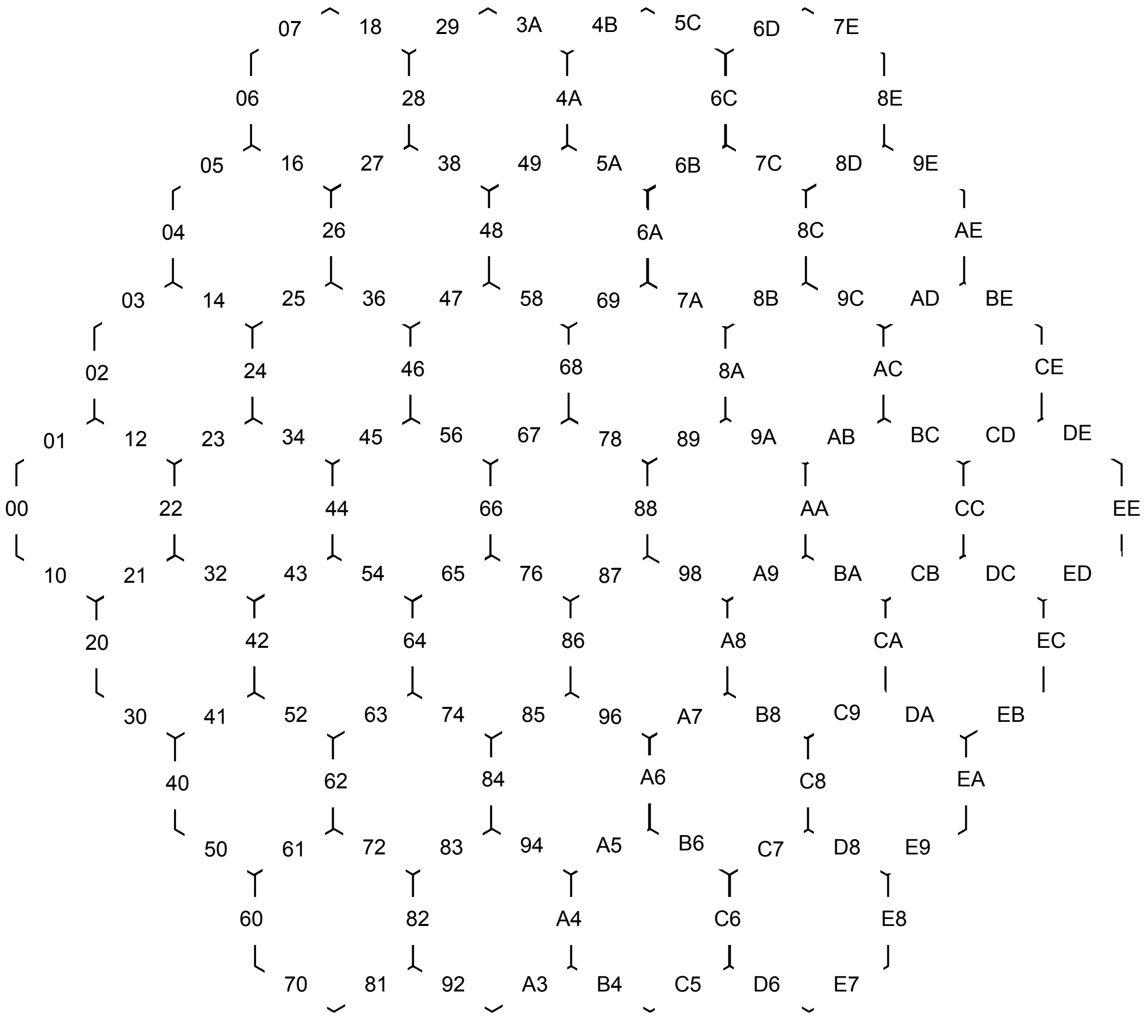

07 18 29 3A 4B 5C 6D 7E
06 28 4A 6C 8E
05 16 27 38 49 5A 6B 7C 8D 9E
04 26 48 6A 8C AE
03 14 25 36 47 58 69 7A 8B 9C AD BE
02 24 46 68 8A AC CE
01 12 23 34 45 56 67 78 89 9A AB BC CD DE
00 22 44 66 88 AA CC EE
10 21 32 43 54 65 76 87 98 A9 BA CB DC ED
20 42 64 86 A8 CA EC
30 41 52 63 74 85 96 A7 B8 C9 DA EB
40 62 84 A6 C8 EA
50 61 72 83 94 A5 B6 C7 D8 E9
60 82 A4 C6 E8
70 81 92 A3 B4 C5 D6 E7

Figure A.3 Edge Coordinates

To compute the coordinates of an adjacent element given a set of coordinates, find the picture of the element you are starting with and then add the coordinate modifier of the adjacent element to get the coordinates of that element.

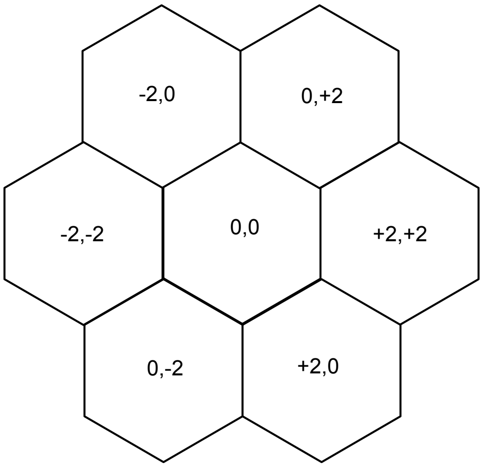


Figure A.4 Adjacent Hex Coordinates to a Hex

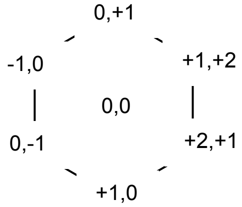


Figure A.5 Adjacent Node Coordinates to a Hex

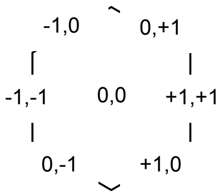


Figure A.6 Adjacent Edge Coordinates to a Hex

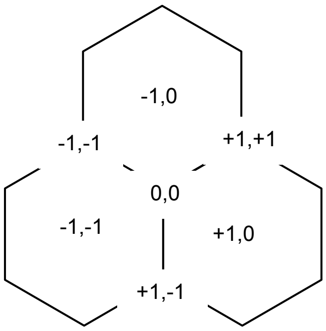


Figure A.7 Adjacent Hexes and Nodes an [Even, Odd] Node

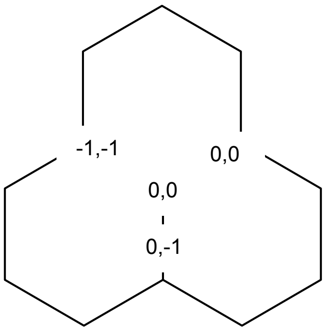


Figure A.8 Adjacent Edges to an [Even, Odd] Node

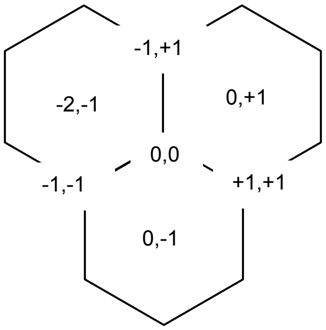


Figure A.9 Adjacent Hexes and Nodes to an [Odd, Even] Node

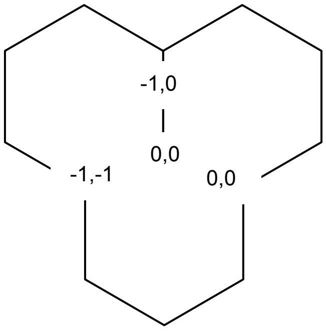


Figure A.10 Adjacent Edges to an [Odd, Even] Node

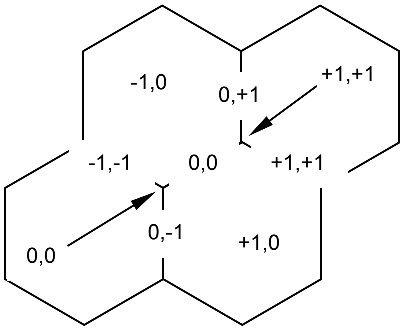


Figure A.11 Adjacent Hexes, Nodes, and Edges to an [Even, Odd] Edge

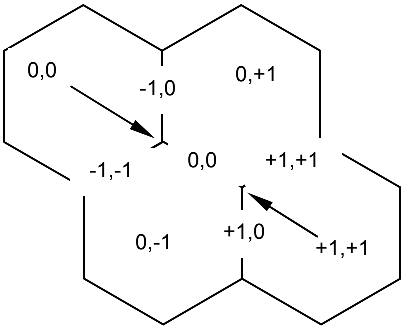


Figure A.12 Adjacent Hexes, Nodes, and Edges to an [Odd, Even] Edge

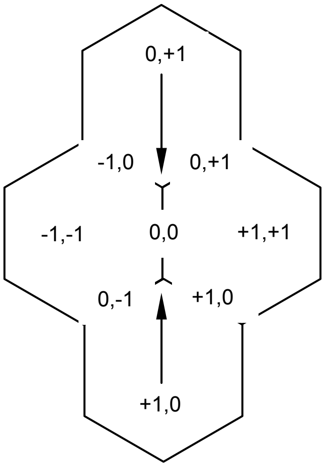


Figure A.13 Adjacent Hexes, Nodes, and Edges to an [Even, Even] Edge

Appendix B

Board Layout for Settlers of Catan

These are the steps for setting up a board for Settlers of Catan. The same algorithm is used in Java Settlers when a game is started.

1. Shuffle the land hexes and lay them out as shown in figure B.1.
2. Shuffle the port hexes and lay them out as shown in figure B.1.
3. Place water hexes between the port hexes.
4. Place the number markers on the land hexes in alphabetical order (in the board game the number markers have letters on them to indicate the order in which they should be placed) starting on an outside corner and continuing counterclockwise. When you come to the desert hex, do not place a number marker there.

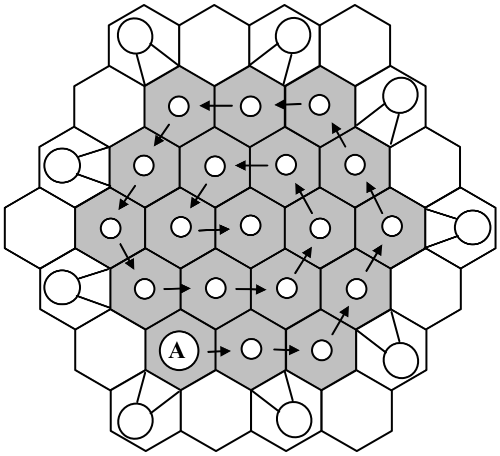


Figure B.1 Board Layout

References

Agre, P., Chapman, D. (1987). Pengi: An implementation of a theory of activity. Sixth National Conference on Artificial Intelligence, AAAI Press.

Aquino, G. (2002). Online Game All-Stars. PC World.

Bates, J., Loyall, B., Reilly, S., (1991). Broad Agents. Symposium on Integrated Intelligent Architectures, Stanford University.

Brooks, R. (1986). "*A robust layered control system for a mobile robot*." IEEE Journal of Robotics and Automation **RA-2**(1): 14-23.

Burke, R., Hammond, K., Young, B. (1997). "The FindMe Approach to Assisted Browsing." IEEE Expert **12**(4): 32-40.

Buro, M. (1997). "The Othello match of the year, Takeshi Murakami vs Logistello." ICCA Journal **20**(3): 189-193.

Calhamer, A. (2000). Diplomacy, Hasbro/Avalon Hill.

Christian, B. (1996). My Chat Applet.

Croxton, D. (2001). Settlers ofCatan. Moves**:** 5-12.

Dorer, K. (1999). Extrended Behavior Networks for the Magma Freiberg Team. RoboCup-99 Team Descriptions for the Simulation League, Linköping University Press**:** 79-83.

Epstein, S. (1999). Game Playing: Next Moves. Sixteenth National Conference on Artificial Intelligence, The MIT Press.

Firby, J. (1989). *Adaptive Execution in Complex Dynamic Domains*, Yale University.

Fisher, R., Ury, W. (1991). Getting to Yes. New York, NY, Penguin Books.

Fogel, D. (2002). Blondie24 Playing at the Edge of AI. San Francisco, CA, Morgan Kaufmann Publishers.

Howard, R. A. (1960). Dynamic Programming and Markov Processes. Cambridge, MA, MIT Press.

Howard, R. A. (1977). Risk Preference. Readings in Decision Analysis. R. A. Howard, Matheson, J. E. Menlo Park, CA, Decision Analysis Group, SRI International**:** 429-465.

Kaelbling, L., Littman M., Moore, A. (1996). "Reinforcement Learning: A Survey." Journal of Artificial Intelligence Research **4**: 237-285.

Kraus, S. (1995). "Designing and Building a Negotiating Automated Agent." Computational Intelligence **11**(1): 132-171.

Luce, D., Raffia, H. (1985). Games and Decisions. Mineola, NY, Dover Publications.

Maes, P. (1990). "How to do the Right Thing." Connection Science Journal, Special Issue on Hybrid Systems **1**.

Schaeffer, J. (2000). The Games Computers (and People) Play. AAAI/IAAI, Academic Press.

Schaeffer, J. (2000). One Jump Ahead. New York, NY, Springer-Verlag.

Scott, B. (2002). The Illusion of Intelligence. AI Game Programming Wisdom. S. Rabin. Hingham, MA, Charles River Media Inc.**:** 16-20.

Smed, J., Kaukoranta , T., Hakonen, H. (2002). **A Review on Networking and Multiplayer Computer Games**, Turku Centre for Computer Science.

Smith, T. (1996). English Rules for The Settlers of Catan, Mayfair Games.

Tesauro, G. (1994). “TD-gammon, a self-teaching backgammon program, achieves master-level play.” Neural Computation **6**(2): 215-219.

Tozour, P. (2002). The Evolution of Game AI. AI Game Programming Wisdom. S. Rabin. Hingham, MA, Charles River Media Inc.**:** 3-15.

Watkins, C. J. (1989). Learning with Delayed Rewards. Cambridge, UK, Cambridge University.