CAL STATE FULLERTON

Department of Computer Science

CPSC 597 PROJECT DEFINITION

To the graduate student:
1. Complete a project proposal, following the department guidelines.
2. Have this form signed by your advisor and reviewer.
3. Submit it with the proposal attached, to the Department of Computer Science.

Please print or type.

Student Name: ANONYMOUS Student ID: [redacted]

Address: [redacted] Cambridge Rd Dana Point 92629
(Street / City / Zip Code)

Home Phone: ([redacted]) [redacted]-[redacted] Work Phone: _____

Units: 3 Semester: Spring

Are you a Classified graduate student?

[X] Yes

[ ] No

Is this a group project?

[ ] Yes

[X] No

Proposal Date: 12/20/03

Completion Deadline: 5/30/04

Mismatch with the title

Tentative Project Title: Reinforcement Path Finding Agents in Dynamic Environments

We recommend that this proposal be approved:

Faculty Advisor _____
(Printed name / Signature / Date)

Faculty Reviewer _____
(Printed name / Signature / Date)

Mismatch with the project title on the cover page

Reinforcement Path Finding

by

ANONYMOUS

May 30, 2003

Advisor: Dr. Chang-Hyun Jo

1. Introduction:

The problem of path finding is one that spans to many domains in computer science. The need for an efficient path finding mechanism can be found in web routing, game playing, and street navigation, to name a few areas. As a result of this need for efficient path finding methods, many heuristic based algorithms have been developed to find the most optimal path, with the hopes of accomplishing this with the minimal amount of computation effort and computation time. Of the many heuristic based algorithms used for path finding, there are a few algorithms that have provided a foundation for new path finding algorithms. These algorithms are Dijkstra's algorithm [Morris 1998], Best First Search [Luger 2002], and the A* Algorithm [Patel 2001]. Many newer algorithms that have been proposed typically use one of the aforementioned algorithms as a base, and then modify the algorithm to enhance the algorithm or improve upon any deficiencies the algorithm may have had.

However, these algorithms cannot be deemed "intelligent" in the true sense of the word because the agents using these algorithms for path finding rely on the heuristic of the algorithm to provide the means of intelligence for the agent. However, no actual intelligence is present, in that, the agent doesn't actually understand what environment its in, doesn't remember where it has been previously, nor does the agent understand why it is traversing the path is has found. As a result, many CPU cycles are wasted as a result of the path finding agent needing to re-compute the path every time a new path needs to be traversed.

Some attempts have been made to remedy the deficiencies of heuristic based algorithms, but still, each algorithm possesses deficiencies in solving the problem of path

finding. For example, the D* algorithm [Stentz 1994] is essentially the A* algorithm, that has been modified to adapt to dynamic environments. While this algorithm can handle dynamic environments, it fails to remember the places or paths it had traversed previously. Another example is the Incremental A* [Koenig 2001], which provides a method for the path finding agent to retain knowledge about the paths it has traversed, and use that knowledge to help generate future paths with less computation than the original A* algorithm. However, the Incremental A* algorithm has the deficiency of requiring complete knowledge of the environment.

Adding machine learning methodologies to the problem of path finding and current path finding algorithms can provide the solution to the problem of path finding. Machine learning has been shown to excel in many problem domains where traditional algorithms have failed to provide adequate solutions. Current research in reinforcement learning shows promising results from experiments that show a machine is able to learn concepts and environments through various reinforcement learning methodologies [Dietterich 1997].

At the time of writing, no path finding agent has combined the methods of machine learning with heuristic based algorithms to create an "intelligent" path finding agent. Creating an intelligent path finding agent in a 3-D virtual environment provides an excellent opportunity to engineer a new and unique path finding method, while also exercising multiple computer science disciplines in a single, cohesive project.

Need to compare with literatures which adopt intelligent path finding that do not employ machine learning. Also explain why machine learning will be better than other techniques used in the existing work related intelligent path finding which are studied already.
Give references for this case too.

2. Project Objectives:

The disciplines this project will bring together are: Artificial Intelligence, Computer Graphics, File I/O, Software Engineering, Data Structures and Algorithms, and Compiler Theory. Together, these various areas of computer science will all take part in creating an intelligent path finding agent, and a rich 3-D virtual environment for the agent to explore. This project will force creativity on the part of the developer in terms of engineering an optimal path finding solution, and utilize learned knowledge from a multitude of computer science areas learned throughout the graduate program.

Artificial Intelligence methods will be used to give the path finding agent intelligence while exploring the environment. The methodologies used for machine learning, specifically Reinforcement learning, will be applied to the problem of path finding to develop a more intelligent and more efficient algorithm.

Computer graphics is a necessary means of providing a 3-D virtual environment for the path finding agent to explore. Computer graphics is also necessary to create a realistic environment, such as different terrain and collision detection. Varying the type of terrain that exists in the virtual environment will provide a richer environment for the agent by enforcing different costs to traverse different types of terrain. Specifically, the Graphics Engine will be responsible for rendering the 3-D environment, handling collision detection, lighting, and converting screen coordinates to world coordinates and vice versa.

File I/O provides the necessary means of dealing with the many files needed for the project. The types of files that will be needed for the project are 3-D model files (representing the environment and interacting agents) and intelligence scripts (used to

store learned knowledge between sessions). The File I/O module will also be responsible for writing out path finding statistics and agent statistics, which will be used to compare the performance of the path finding algorithm against current path finding algorithms.

Efficient data structures and algorithms will be needed to ensure that the path finding agent operates in a real time environment. Hence, the goal with this restriction is to create a path finding agent that can determine the most optimal path efficiently, in terms of time, computation, and intelligence. For example, computing a path in real time by re-calculating the path it has already traversed would not be efficient in terms of intelligence because no intelligence would have been used to compute the path.

Compiler theory will be used to generate a compiler to generate and read the original AI script language, that will be used to store the agent's memory between run time sessions. The AI script language must be stored in a human readable format on disk when not in use by the agent, thereby, allowing hand modifications to be done to the script if necessary. This format will also allow the developer to carefully monitor the learning ability of the agent. However, the AI script language will need to be parsed and compiled into a more efficient format for the agent to be able to store in memory and use than a human readable format would provide. The reason a scripting language will be created is to allow the agent to achieve a higher level of intelligence by allowing the agent to learn for an unbounded amount of time. If the agent had to start with no intelligence at the beginning of every run time session, depending on how long the session was, the agent would only be able to achieve a certain level of intelligence. Furthermore, depending on the intelligence of the interacting agent's in the environment,

whether they're human or other agents, each session could potentially cause new learning to occur that would be forgotten if its memory was not saved.

3. Project Activities:

- AI engine:
  - Design, document, implement, and test the AI Engine that will be used for path finding

  - Design, document, implement and test:
    - modified path finding algorithm using the A* algorithm as a base, but modifying the algorithm to use prior traversals to calculate new paths, and handle dynamic environments
    - Reinforcement learning algorithm
    - Case Based Reasoning algorithm
    - Rote Learning algorithm
    - AI script data structure
    - Path Finding Agent data structure

- Graphics Engine
  - Design, document, implement, and test the Graphics Engine

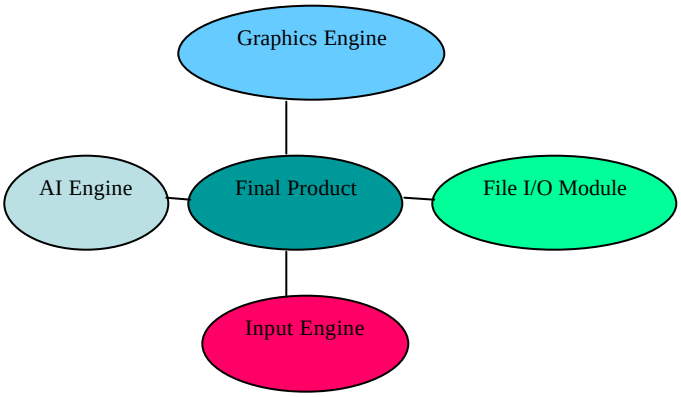
- File I/O:
  - Design, document, implement, and test the File I/O Module

- AI Scripting Language Compiler:
  - Design, document, implement and test the AI script compiler
    - reading in AI script language and compiling to a more efficient format
    - storing the data in a data structure in memory
    - writing the agent's memory to a file in the AI script language

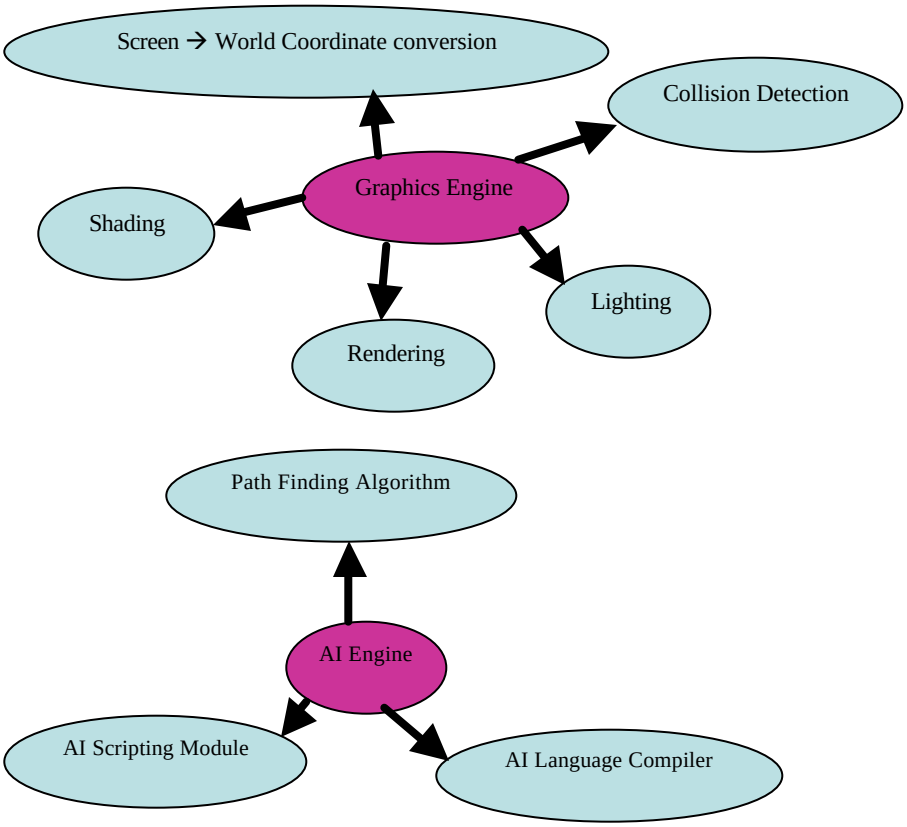
The project can be broken up into 5 separate development phases:

(1) Graphics Engine, (2) AI Engine, (3) File I/O Module, (4) Input Engine, and

(5) Final Product. A logical overview of the development phases is shown below:



The graphics engine and AI engine are the heart of the project. The graphics engine is responsible for rendering the models to the screen for visual feedback of the environment, as well as handling the graphical objects that comprise the environment (e.g. Collision detection, movement costs, traversable terrain, etc.). This part of the development phase consists of designing, documenting, implementing, and testing the classes that will comprise the graphics engine. The graphics engine will essentially provide a project specific wrapper to the OpenGL 1.2 routines. The AI engine is the other driving force of the project, in that, it is responsible for providing the agents in the environment with intelligence. The AI engine development phase will consist of designing, documenting, implementing, and testing the classes that will comprise the AI engine. The classes designed will handle the calculation of the most optimal paths in the environment, as well as handle the methods for allowing the agent to learn its environment. A logical view of the graphics and AI engine is shown below:



The third development phase consists of designing, documenting, implementing, and testing the File I/O module. The File I/O module is responsible for reading and writing all forms of persistent data that are required for the operation of the project. This includes reading the MD2 model format for objects in the environment, and reading and writing the AI scripting language that comprises the agent's cumulative knowledge about its environment.

The fourth development stage consists of designing, documenting, implementing, and testing the input engine. The input engine is comprised of the classes that handle both keyboard and mouse input. Input is designed to manipulate attributes of the environment and agents in real time. User input is also designed to allow the user to interact with the agents in a supervised learning mode, to help aid their learning process.

The input engine will consist of classes that provide a project specific wrapper to the DirectX 9 DirectInput API.

Lastly, the final development stage consists of joining the separate engines and modules to form a cohesive product. This involves creating, designing, and documenting extra boundary test cases to uncover any remaining errors that were formed by combining the modules. This stage also consists of training the agents to see how well they interact in the environment on their own, with other agents, and with human interaction. All tests will be documented with an objective measuring standard to compare the results of the reinforcement path finding algorithm against existing heuristic based, path finding algorithms.

4. Development Environment:

- **Target system the software is designed to run on**:
  - Intel Pentium III/4 1Ghz+ or AMD Athlon 1Ghz+
    512 MB RAM
    at least 32MB 3-D Graphics Card, supporting OpenGL 1.2 standard

- **Target Operating System**:
  - Windows 2000 or higher

- **System the project will be developed on**:
  - AMD Athlon Thunderbird 1 Ghz
    Geforce 2 Ultra with 64MB DDR RAM
    512 MB Ram
    Windows XP sp1

- **Extra test systems to analyze performance**:
  - Pentium-III 600 Mhz
    256MB Ram
    Riva TNT2 Ultra with 32MB RAM
    Windows 2000 sp3

- Pentium-4 2Ghz
  512MB Ram
  Geforce2 mx 440 with 32MB Ram
  Windows 2000 sp1

- **Development Languages**:
  - C/C++
  - OpenGL 1.2
  - DirectX 9
  - original AI Scripting language

- **Development tools**:
  - Visual C++
  - lex
  - yacc

5. Project Results:

The following items will be delivered to the advisor for demonstration:

1. Requirements documentation
2. Specifications documentation
2. Detailed design documentation
3. Preliminary and detailed design diagrams
4. Source code
5. Binary image to be run / demonstrated for the advisor
6. Test Cases
7. Project Testing Results
8. User's manual
9. Final Report

6. Project Schedule:

2004	February				March					April				May					Summary	
Tasks:	1	2	3	4	1	2	3	4	5	1	2	3	4	1	2	3	4	5	Hours	Percent
Requirements	20	10	10																40	12%
Specifications		10	15	10															35	10%
Preliminary Design				10	20	15													45	13%
Detailed Design						5	20	20	20										65	19%
Graphics Engine										20	15								35	10%
AI Engine											10	15							25	7%
File I/O Module												10	5						15	4%
Input Engine													5						5	1%
Integratation / Testing											5	8	10						23	7%
Write User's Manual														10	10	4			24	7%
Write Final Report															10	10	5		25	7%
Demonstration																		2	2	1%
Hours	20	20	25	20	20	20	20	20	20	20	30	33	20	10	20	14	5	2	339	100%

7. References:

[Dietterich 1997] Dietterich, Thomas G. 1997. Machine-Learning Research: Four Current Directions, AI Magazine, 15 (3), 97-136.

[Luger 2002] Luger, George. 2002. Artificial Intelligence: Structures and Strategies for Complex Problem Solving. Pearson Education Limited, Edinburgh Gate, Harlow, Essex CM20 2JE.

[Koenig 2001] Koenig, Sven. 2001. Glossary of Fast Replanning and Greedy On-Line Planning, http://www.cc.gatech.edu/fac/Sven.Koenig/greedyonline/glossary.html#LifelongPlanning A*.

[Morris 1998] Morris, John. 1998. Data Structures and Algorithms: Dijkstra's Algorithm, http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/dijkstra.html.

[Patel 2001] Patel, Amit. 2001. Amit's A* Pages, http://theory.stanford.edu/~amitp/GameProgramming/.

[Stentz 1994] Stentz, Anthony. 1994. Optimal and Efficient Planning for Partially-Known Environments, Proceedings IEEE International Conference on Robotics and Automation, San Diego, California, (May), 3310-3317.

8. Appendix:

Two appendices are included to give the reader an encompassing breadth of knowledge if he/she desires to research the topics of machine learning and path finding in further depth. Appendix A covers the topic of machine learning (report 1), while Appendix B covers the topic of path finding (report 2).

8.1: Appendix A

Abstract

In recent years, research in the area of machine learning has increased due to the applicability of the methods to a wide range of areas. Machine learning algorithms are now found to be more efficient than standard algorithms for solving many problems in many different areas. Many of the methods used for machine learning are based on, or patterned after, the way humans learn. Current areas of research in which studies are being conducted are: 1) Improving classification accuracy, 2) Scaling learning algorithms, and 3) Reinforcement learning. This paper summarizes past, current, and future research that is being conducted in the area of machine learning.

1. Introduction

The quest to make machines simulate human intelligence has always been a desire of mankind. Up until recently, making machines act and think like humans has only been an element of science fiction, in which humanlike robots are seen interacting with humans as if the robots were real humans. Recently, there has been a surge in research done in the area of machine learning. Machine learning is the process of trying to create a machine with the ability to learn. Currently the main categories of research being done in the field of machine learning are: 1) Improving classification accuracy through the use

of ensembles of classifiers, 2) Scaling up learning algorithms, and 3) Reinforcement learning [Dietterich 1997].

2. Objectives

The subject of machine learning is a fascinating topic to me because of the applications that it entails. For example, creating a robot that can sense its surroundings and interact with its environment was once only a possibility in a science fiction movie. However, today these types of robots are actually being developed. Also, the idea of a machine acting and thinking like a human is interesting because the machines are no longer just "dumb" terminals that process the set of instructions they were programmed to do. Instead, they begin to think and interact with the user, not only making the interaction between the computer and the user more productive, but more enjoyable and entertaining as well. Implementing these types of robots is already being done, as demonstrated during the "9-11" incident, in which mechanized robots were able to navigate through the rubble in search of survivors. Currently, machine learning is applicable to so many areas because many areas are still using algorithms to solve the problems. My goal is to discover the different methods used in machine learning and apply this knowledge to an area that currently isn't using machine learning in its implementation.

3. Summary

3.1 Machine Learning

The reason why there has been a surge of research in the area of machine learning is because computer scientists have found that machine learning can be applied to many

fields to solve problems that were too difficult before, or to create more efficient solutions. For example, "machine learning algorithms are already the method of choice for developing software" that require face recognition [Mitchell 1997]. Machine learning has also been found to be applicable to other real world situations. For instance, machine learning is now being used to help detect credit card fraud by using data mining to make a decision based on past transactions. Another example of its use in current applications is in medicine, in which, machines are able to decide which treatment a patient would respond to best [Mitchell 1997]. Machine learning also gives people an insight into how humans learn because as we attempt to teach a computer, we search for similar methods that we use to teach other people, as well as conduct further research on methods of teaching people [Dyer 2003]. Many methods of teaching that mimic the human learning process have been implemented in machines.

3.2 Methods of Machine Learning

One method of learning is called rote learning. Rote learning involves a one to one mapping between the inputs and the correct outputs. Essentially, rote learning is learning by memorization. Another method is through induction. Induction involves using examples to reach a general conclusion. An example of induction learning would be if several round objects of different colors were given as input, and all of the objects had the property "bouncing." A machine using induction learning would make the general assumption that round objects that bounce can be classified as a "ball." Clustering is another method, in which the machine tries to group similar inputs or objects together. Analogy learning is another methodology that involves determining the relationship between 2 different representations. An example of this would be if a

machine was given a new input, the machine would try to use old inputs to make conclusions about the new input based on similarities between the new input and an old input. Discovery learning is a method that is a form of unsupervised learning. In discovery learning, no specific goal is given and the machine learns through exploration. Similar to discovery learning, reinforcement learning is a method that gives a machine positive or negative feedback based on the actions the machine takes [Dyer 2003].

3.3 Classification

The first main direction research is heading in is in the process of classification. Training a machine to correctly classify objects is accomplished by seeing how well a classifier can be created to predict the true function, how fast learning takes place, the speed in which the classifier can predict the true function, and the space requirements required for the learning algorithm [Dietterich 1997]. A classifier is a hypothesis that is formulated from the learning algorithm. This classifier, or hypothesis, attempts to estimate a true function *f*. Typically, in order to formulate a classifier, some type of training is needed for the machine. There are various types of training methods, such as, supervised learning, unsupervised learning, and reinforcement learning.

Supervised learning involves giving the machine a subset of input and output from a function *f*. While the true function *f* may not be known by the "teacher," the output for the corresponding input will be known for a particular subset of cases. Thus, this method is referred to as supervised learning since an agent must act as a "teacher" by providing the input and telling the machine what the correct output should be. With each set of input and output, the machine begins to formulate a classifier that can be used to predict the true function *f*. Once a classifier is formulated, the machine can then use the

classifier to predict new output that may not be known. In this manner, researchers that are trying to find an unknown function *f* can use the machine to predict a function [Dietterich 1997].

Current research in this area involves creating an ensemble of classifiers. By creating an ensemble of classifiers, a decision can be made based on weighted or unweighted instances of the classifiers it has formulated through training. Research is currently being done on methods to find a "good" ensemble. The ultimate goal is to form a large set of independent classifiers to create a balanced, weighted ensemble of classifiers. The reason independent classifiers are desired is because if multiple classifiers are the same and they all happen to be formulated improperly, or all of the classifiers yield incorrect output of a true function *f*, then there will be a larger weighting of incorrect responses, thereby ultimately skewing the decision made. However, if the ensemble consists of independent classifiers, then more factors can ultimately decide the output. Multiple classifiers are desired over a single classifier because they allow more factors to come into play and use weighting to predict the output, as opposed to a single classifier that has no other factors to influence the decision if it predicts the wrong output [Dietterich 1997].

There are 4 main methods currently being investigated in finding a good ensemble of classifiers. They are: 1) Sub-sampling the training examples, 2) Hand picking input to create the desired subsets, 3) Manipulating the output targets, and 4) Randomly generating new classifiers. Method 2 has had real world success. Various images of volcanoes were given to a machine, with the goal of correctly predicting which mountains were actually volcanoes. The images given were hand picked by a group of

scientists to form the desired ensemble of classifiers for the machine. Once the machine learned how to identify volcanoes through supervised learning, it was able to identify volcanoes as well as human experts [Dietterich 1997].

Recent research is being conducted that adds another method to create an ensemble of classifiers. A study by Ware, Frank, Holmes, Hall, and Witten was conducted that created an easy to use GUI interface, allowing users to visually build the classifiers. By having a user build the classifier, this has a distinct advantage over a machine because background knowledge about the problem can be used to build a better ensemble of classifiers. The GUI was limited to two dimensions because it was thought that users would be able to identify patterns easier in 2-D. Users were also limited to looking at 2 attributes at a time because of the same reason. In the first test, a user with no prior knowledge about the problem domain was able to produce a tree with "53 nodes, that achieved an accuracy of 85.8% in correctly classifying Kiwi fruit vines" (less nodes are better because the machine won't have to search as many nodes to determine the output). Using the same test data with a "decision tree inducer, [it] produced a tree containing 93 nodes with an accuracy of 83.2%." With such productive results using someone with no prior knowledge about the domain, further experiments were conducted with experts in a domain. Classifiers constructed by experts in the domain were able to produce decision trees that surpassed the accuracy yielded by automated learning algorithms. However, this method has its downfalls. For one, it is limited to only two dimensions. Using more dimensions for manual classifier construction is far too "tedious to be worthwhile" [Frank et al. 2001]. However, further research is being conducted to allow a "symbiotic relationship" to be formed that "combines the skills of a human user

and a machine learning algorithm" [Frank et al. 2001]. In this fashion, a user will be able to use his/her background knowledge about the domain and use the machine learning algorithm to take over when manual classifier construction is not feasible.

3.4 Scaling Algorithms

The second direction of research that machine learning is heading in is how to scale up the machine learning algorithms to handle billions of training examples. Current learning algorithms have been effective with millions of training examples, but they may not scale up well if billions of training examples need to be done on the machine [Dietterich 1997]. Some fields involving this area of research are: database mining, information search/retrieval, and object/pattern recognition (Example: recognizing Chinese/Japanese characters). Many algorithms have not scaled well with a large increase in the number of training examples. Thus, 4 current methods for handling larger sets of training data are: 1) Sampling subsets of the training data, 2) Creating new data structures that are efficient enough to hold the relevant data in RAM as opposed to disk, 3) Using ensembles of decision trees with multiple processors to speed up data processing, and 4) Breaking up the training data into ranged categories (Example: 1-5, 6-10, etc.).

3.5 Reinforcement Learning

Reinforcement learning is the third key direction of machine learning. Reinforcement learning is classified as a form of unsupervised learning. With reinforcement learning, a machine is given a reward or punishment based on the action the machine takes, but the machine will never be told what the correct action is [Doyle 2003]. Learning occurs when the machine interprets the feedback from its actions and

makes new decisions based on the feedback it received [Luger 2002]. Work began in the area of reinforcement learning with Arthur Samuel in 1859, when he developed a checkers program. In 1992, Gerry Tesauro created TD-GAMMON, which is a machine that learned how to play backgammon and currently is almost as good as the best human players [Tesauro 1995].

One method of implementing reinforcement learning is to use dynamic programming. Dynamic programming is used to find the optimal policy, which translates to the most optimal reward gain. Due to the fact that dynamic programming is infeasible in many situations, 3 newer algorithms are under development. They are policy iteration, value iteration, and modified policy iteration. However, while these algorithms are applicable to more situations than dynamic programming, they all require performing backups of every state. Hence, their running time scales with the number of states, making them impractical for problem sets with large amounts of states. Another problem with these algorithms is that a complete model of the system is required, which in many cases, may not be known. For example, if a robot is interacting in an unknown environment, it is impossible to provide a complete model of the system to the machine. Hence, newer studies are investigating the possibilities of stochastic approximation backups, value-function approximation, and mode-free learning as ways to overcome this hurdle [Dietterich 1997].

A real world example of how reinforcement learning techniques were able to surpass the performance of normal algorithms was in a study of elevator control. The elevator had to decide which direction to go after stopping at a floor, and when approaching a floor, whether to stop at it or skip it. The goal was to minimize the time

people waited for the elevator. Q-learning, a technique for reinforcement learning where the optimal value is chosen while adding in a small chance of randomness for other actions, was used. When the Q-learning algorithm competed against standard algorithms for elevator control, the elevator using reinforcement learning performed the best [Dietterich 1997].

One problem with current reinforcement learning algorithms, and machine leaning algorithms as a whole, is that current learning algorithms don't work well with very large spaces [Ito et al. 2001]. Another problem is the method of exploration used for reinforcement learning. Exploration is necessary for a machine using reinforcement learning algorithms in order for the machine to learn. However, current methods use a random exploration method, and thus, future research involves creating more intelligent exploration methods rather than random exploration. Another problem is that currently, finding the optimal policy is the goal of reinforcement learning. However, this isn't always applicable to all cases, as in the case of an ongoing system. With an ongoing system, there isn't an optimal policy since the system is ongoing. Instead, the "optimal policy" should be the policy that achieves the greatest average reward per given unit time [Seri and Tadepalli 2002].

Current research that has taken on this idea of measuring reward per unit of time, instead of trying to find the optimal policy, has been done by Sandeep Seri and Prasad Tadepalli in their Hierarchical Average-reward reinforcement learning model. They have demonstrated the effectiveness of using task hierarchies "as a way to tame the complexity of reinforcement learning." Using task hierarchies involves a parent in the tree assigning tasks to its children. Since their studies deal with a continuous system, the parent node

has a recurrent or never-ending task. Each child receives a reward upon accomplishing its goal. As each action is accomplished by each child node, policies are created for that particular node. The "optimal policy" using task hierarchies involves finding the "maximum gain among all hierarchical policies" (the parent node and all of its children). However, the performance of hierarchical average reward reinforcement learning is slower than current model based methods, and thus further research is being investigated in the area of using a hierarchical learning model [Seri and Tadepalli 2002].

Another problem with reinforcement learning algorithms is that oftentimes a great deal of time must pass before learning occurs. Typically, the speed of learning is dependent on the "number of states [being the] bottleneck" [Ito et al. 2001]. Coarse graining of perceptions has been one solution to reduce the number of states that need to be investigated during the learning process, and has proved to be a feasible solution to the slow learning process. However, there is a tradeoff between using coarse graining of perceptions and complete perception. By using coarse graining of perception, the learning process is sped up, but results in a degradation of overall performance. Furthermore, "bad habits" that are formed early on during the coarse graining oftentimes make it "difficult to rectify bad habits acquired at the early stage of learning" [Ito et al. 2001]. While complete perception gives better overall performance and a lesser chance of bad habits being formed during the early stages, the downfall is the amount of time it takes for learning to occur. To give the reader an idea how many states have to be investigated with even a few number of agents, the following table is shown:

n* \ *m	**3**	**5**	**7**	**9**
1	9	25	49	81
2	81	625	2401	6561

3	729	15,625	117,649	531,441
4	6561	390,625	5,764,801	43,046,721

In the table above, *n* refers to the number of hunters while *m* refers to the lattice size. A hunter is exactly that in this scenario, a hunter in search of prey. *m* refers to the lattice size and is the size of the grid the hunter can move on. As shown, even with a small amount of hunters and a small lattice size, the amount of possible states the hunter's have to investigate (which includes keeping track of other hunter positions and the prey position) increases at an exponential rate. To also give the reader an idea of how much time it takes for 4 hunters to learn on a machine, it took 8 hours of CPU time using a Pentium III 640-MHz PC with 2 GB of memory. Thus, it is easily evident that learning in real time is not yet possible, even with a small amount of hunters. Thus, the solution is the use coarse graining of perception [Ito et al. 2001].

Coarse graining of perception involves "regarding several nearby (but distinct) states as the same," which effectively reduces the number of states to be explored [Ito et al. 2001]. As an example with 3 hunters and a lattice size of 7 (n = 3, m= 7), a complete perception of 117,649 was reduced to only 49 states by using coarse graining of perception. However, the problem of performance degradation still exists if coarse graining perception is used throughout the learning process. Hence, the researchers Akira Ito and Mitsuru Kanabuchi proposed a method that incorporates both coarse grained and complete perception models to form the learning model. The key problem of using both perception models was finding the point in time to switch from coarse grained perception to complete perception. If a switch is made too late, the possibility of bad habits being formed is the problem. If the switch is made too early, the learning process will still be relatively long. Using residual entropy was proposed as the method to calculate the

switch point. Residual entropy involves tracking "how much freedom is left for [a] state *S*" [Ito et al. 2001]. As learning occurs, the residual entropy approaches zero. Therefore, experiments were conducted by measuring the residual entropy of both the coarse graining perception method and the complete perception method, to conclude a respectable switching time.

Another proposed learning methodology for reinforcement learning is that of Inverse Reinforcement Learning, proposed by Andrew Ng and Stuart Russell. Normal reinforcement learning algorithms try to find the optimal policy based on the positive and negative rewards the machine receives according to its actions. Inverse reinforcement learning attempts to extract the reward function by observing optimal behavior from an "expert" [Ng and Russell 2000]. This type of learning is more like "apprenticeship learning to acquire the skilled behavior" because the machine observes the behavior and then attempts to create a function based on that behavior, much like an apprentice would observe a professional and attempt to learn by watching [Ng and Russell 2000]. A problem with inverse reinforcement learning is that of degeneracy. In the process of apprenticeship, many policies will be created in an attempt to find the optimal policy. Thus, there must be a way of picking out the optimal policy from the rest of the sub optimal policies. Current research in this methodology works for moderate sized discrete and continuous domains. However, one of the continuing problems with inverse reinforcement learning algorithms is that they don't scale well for larger domains. Thus, future research is currently being conducted to see if the method proposed by Andrew Ng and Stuart Russell can scale well. Another hurdle to overcome with this method is that the current experiments have been tested with relatively little or no noise. Noise is often

a problem for learning algorithms because they impede the machines ability to distinguish between noise and valid data.

4. Conclusion

Further research and development on algorithms for machine learning will hopefully lead developers one step closer to the goal of making a human like machine. In the meantime, machine learning has already been proven to be an effective method at solving current tasks or making human like decisions, such as identifying credit card fraud. No doubt, as technology progresses, more powerful hardware will allow computer scientists to perform more real time machine learning and more complex algorithms.

5. References

[Dietterich 1997] Dietterich, Thomas G. 1997. Machine-Learning Research: Four Current Directions, AI Magazine, 15 (3), 97-136.

[Doyle 2003] Doyle, Patrick. 2003. Learning, Stanford University, http://www.cs.dartmouth.edu/%7Ebrd/Teaching/AI/Lectures/Summaries/learning.html.

[Dyer 2003] Dyer, C. 2003. CS 540 Lecture Notes, University of Wisconsin - Madison, http://www.cs.wisc.edu/%7Edyer/cs540/notes/learning.html.

[Frank et al. 2001] Frank, Eibe, Holmes, G., Ware, M., Witten, I. 2001. Interactive Machine Learning: Letting Users Build Classifiers, International Journal of Human-Computer Studies, 55 (3), 281-292.

[Ito et al. 2001] Ito, Akira, and Mitsuru Kanabuchi. 2001. Speeding Up Multiagent Reinforcement Learning by Coarse-Graining of Perception: The Hunter Game, Electronics and Communications in Japan, 84 (12), 37-45.

[Luger 2002] Luger, George F. 2002. Artificial Intelligence: Structures and Strategies for Complex Problem Solving. Pearson Education Limited, Edinburgh Gate, Harlow, Essex CM20 2JE.

[Mitchell 1997] Mitchell, Tom. 1997. Does Machine Learning Really Work?, AI Magazine, 18 (3), 11-20.

[Ng and Russell 2000] Ng, Andrew Y. and Russell, Stuart. 2000. Algorithms for Inverse Reinforcement Learning, Proceedings of the Seventeenth International Conference on

Machine Learning, Stanford University, (July), 663-670.

[Seri and Tadepalli 2002] Seri, Sandeep and Tadepalli, Prasad. 2002. Model-based Hierarchical Average-reward Reinforcement Learning, International Conference on Machine Learning, Sydney, Australia, (July), 562-569.

[Tesauro 1995] Tesauro, Gerald. 1995. Temporal Different Learning and TD-Gammon, Communications of the ACM, 38 (3), 58-68.

8.2: Appendix B

Abstract

The problem of path finding can cause an extended computation time if the algorithm used to find an optimal path is inefficient. Even if the algorithm is efficient, lengthy computation times may still occur due to a large search space. Machine learning provides a solution to this necessary re-computation by providing mechanisms that enable a machine to learn. Through a combination of rote learning, reinforcement learning, and case based reasoning, a path finding agent will be able to not only form an optimal reward policy for an unknown environment, but also formulate and remember the most optimal paths to achieve the optimal reward policy. This method will create a seemingly intelligent agent that is also efficient in path finding due to its ability to learn the environment it is in, as opposed to relying on an algorithm for its intelligence.

1. Introduction

The problem of path finding is a common problem that arises in many fields, such as robotics, games, or web routing. The need for efficient path finding is necessary in order to find an optimal or shortest path, while also ensuring minimal computation time. Many of the path finding algorithms use a heuristic to compute a path on an ad hoc basis. This can be both time and computation efficient because the entire search space does not have to be examined in order to find the optimal path. What follows is a discussion of

the major path finding algorithms that have been used in the past or are being used currently for path finding problems. While most of the algorithms are quite efficient in finding an optimal path, even in situations that involve real time computation, each algorithm has its disadvantages. The algorithms discussed below all share the common disadvantage of needing to compute a path every time a new goal node needs to be reached. Hence, while the algorithm may be efficient, it is wasting CPU cycles. Furthermore, the path finding agent cannot be considered intelligent if it must constantly search for paths, most importantly, to places the agent has previously visited.

2. Objectives

The problem of path finding is a commonly encountered problem that pertains to a wide array of subjects. Up to date, the majority of the methods that have been used for path finding are based on heuristics. As a result, unnecessary computation burdens the CPU every time a new path needs to be found. Furthermore, path computation is necessary even when the same path needs to be traversed again because the path finding agent often retains no memory of paths it has traversed. Another downside to many of these algorithms is that they require a complete knowledge of the environment. In many cases, it is impossible to possess complete information about the environment. Hence, this paper seeks to uncover the path finding mechanisms that have been used in the past, as well as the path finding mechanisms that are currently in use. Topics that are not directly related to the topic of path finding, such as machine learning and reinforcement learning, are covered in this paper because of their potential relevancy to a more efficient and intelligent means of path finding. This paper concludes by proposing a method of

path finding in unknown environments, that is intended to outperform current heuristic based algorithms.

3. Summary

3.1 Overview

There are a multitude of heuristic path finding algorithms that have been developed over the years. Essentially, there are two ways to find a path: Path finding and step-taking [Patel 2001]. Path finding involves attempting to find the shortest path in its entirety before the path is even traversed. Only when the path has been calculated will the agent traverse the path. Step taking differs from path-finding, in that, the agent will decide which step to take based on the next best step. The entire path is not calculated in its entirety, but rather, each step is calculated one at a time based on the current location. The advantage of step taking over path finding is that no computation spikes occur because the CPU is not stressed each time a path needs to be found. The disadvantage of step taking is that the agent might choose a local maximum and lead itself into a corner because it did not take into consideration a global maximum.

3.2 Dijkstra's Algorithm

Dijkstra's algorithm is the simplest path finding algorithm. Dijkstra's algorithm works as follows: Starting from the source node, the cost of traversing a path to all neighboring nodes is calculated. The node with the shortest path is added to the shortest path list. For all nodes that are in the shortest path list, each connecting node(s) are calculated to determine the cumulative cost of traversing the path to the node. Ultimately, the shortest path to all nodes from a single source node will be found [Morris 1998].

Dijkstra's algorithm uses a graph G, that represents all of the paths from one vertex to another. The graph G can be presented by the following notation:

G = (V, E)

where G refers to the entire graph
V refers to a set of vertices
E refers to a set of edges

Pseudocode for Dijkstra's algorithm is shown below [Morris 1998]:

```
Dijkstra_shortest_path( Graph g, Node s )
    initialise_single_source( g, s )
    S := { 0 }        /* Make S empty */
    Q := Vertices( g )
    while not Empty(Q)
        u := ExtractCheapest( Q );
        AddNode( S, u ); /* Add u to S */
        for each vertex v in Adjacent( u )
            relax( u, v, w )

initialize_single_source( Graph g, Node source )
   for each vertex v in Vertices( g )
       g.shortest_distance[v] := infinity
       g.predecessor[v] := nil
   g.shortest_distance[source] := 0;

relax( Node u, Node v, double w[][] )
    if d[v] > d[u] + w[u,v] then
       d[v] := d[u] + w[u,v]
       pi[v] := u
```

- g.shortest_distance[] = array of shortest distances
- g.predecessor[] = array of predecessor nodes
- S = set of vertices already visited
- Q = set of vertices remaining to be visited
- d[v] = current shortest path
- d[u] + w[u,v] = current path distance being calculated

What occurs in the above pseudocode is that the algorithm first calls the function initialize_single_source(). initialize_single_source() initializes every vertex in the graph to have a shortest distance of infinity and no predecessor node. A predecessor node can be thought of as a parent to the node, in that, it will be the node that comes before that node in a shortest path. Hence, the source node will never have a predecessor. The

shortest distances are initialized to infinity so that when shorter routes are found, they can replace the infinite values. Typically, some type of marker or character is used to represent the infinite value. Finally, the shortest distance for the source node is set to 0 because this is the node the algorithm will start searching from.

The above pseudocode keeps 2 lists, S and Q. S refers to the set of vertices that have already been visited by the algorithm. Q maintains a list of vertices that still need to be visited or processed. The algorithm first sets the set S to be the empty set, since no vertices have been visited yet. Q is then initialized with all of the vertices in the graph G. For each vertex, the cost to traverse that edge is calculated, and then the cheapest edge cost is picked and added to S. For each node *w* that is adjacent to the node *x* that was just added, the algorithm checks whether the distance through this new node *w* is shorter than the current distance to a neighboring node *z*. If it is, then that new shorter distance replaces the older longer distance. If it isn't, the older distance is kept. Dijkstra's algorithm is more comprehendible through an example.

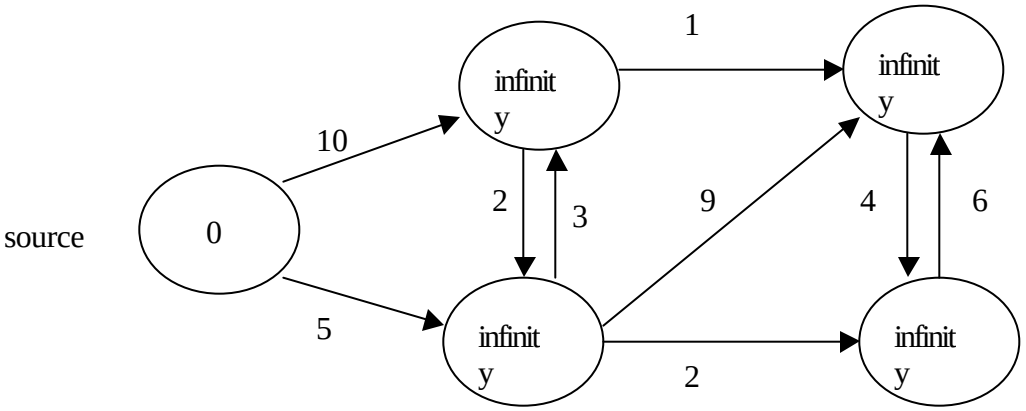


Fig. 1

As shown in figure 1, all of the nodes are initially set to a value of infinity, except for the source node. The number of each edge refers to the cost of traversing that edge.

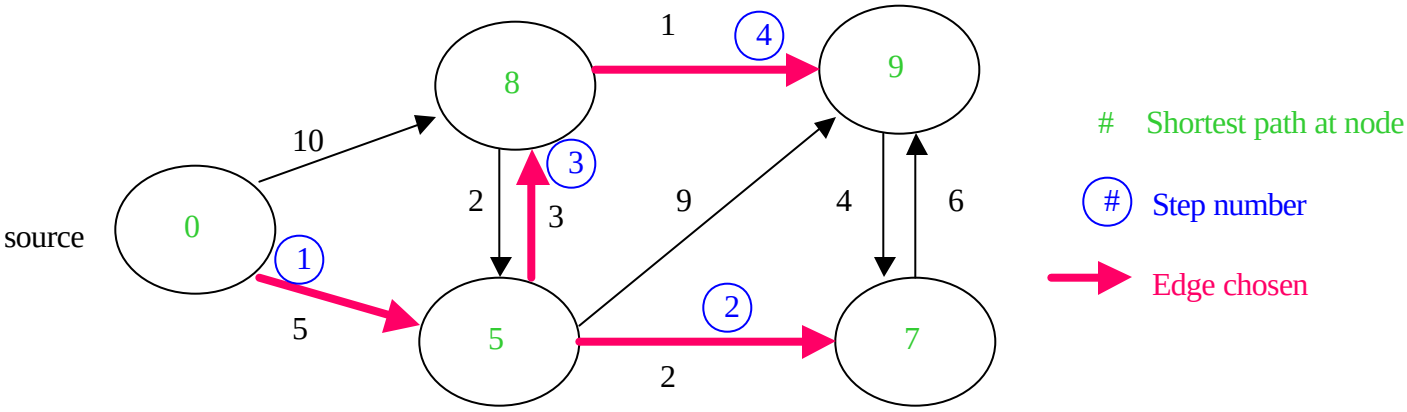


Fig. 2

As shown in figure 2, starting from the source node, the edge with the least cost is chosen at each decision. The cost to reach a node is the sum of all the predecessor nodes taken to arrive at that node.

3.3 Best First Search

Another algorithm for finding the shortest path is Best First Search. Best First Search attempts to find the shortest path by considering the estimates of the best partial solution next. Best first search is often known as being part of a more general category of search strategies, referred to as "hill-climbing strategies" [Luger 2002]. In a hill-climbing strategy, the current state is used to search and evaluate its children. The best child that resulted from the evaluation is then selected for further expansion, disregarding the parent and any siblings that didn't evaluate to the most optimal value amongst all siblings. However, one problem with hill-climbing strategies is that they may often result in failing paths because they fail to see the long term goal or reward. They can also be tricked into going to a potentially optimal local value, thereby, being fooled into going for a failing goal, as opposed to looking ahead to find the globally optimal result. However, Best First search resolves this problem by providing a method of backtracking, to recover from failing paths. Typically, a priority queue is used to implement the Best

First Search Algorithm, since a priority queue will keep the best states at the front of the queue. During each iteration, the "open list" is sorted based on the evaluation of the heuristic used to calculate a value for each node, thus, forming the priority queue.

Pseudocode for a best first search implementation is shown below [Luger 2002]:

```
function best_first_search()

begin
        open := [Start];
        closed := [];
        while open != [] do
                begin
                        X = leftmost state from open
                        if X = goal then return path from State to X
                        else begin
                                generate children of X;
                                for each child of X do
                                case
                                        child not on open or closed:
                                        begin
                                                assign child a heuristic value;
                                                add child to open
                                        end;

                                        child already on open:
                                                if child reached by a shorter path then
                                                        give state on open shorter path

                                        child already on closed:
                                                if child was reached by a shorter path then
                                                begin
                                                        remove state from closed;
                                                        add child to open;
                                                end;
                                        end;
                                end;
                                put X on closed;
                                re-order states on open by heuristic value (best is leftmost)
                        end;
```

```
                return FAIL
end;
```

As shown by the pseudocode, best first search begins by removing the first element from the open list. If this node is not the current goal, then the algorithm will generate the descendents of that node. If the descendent is not on the open or closed list, it is assigned a heuristic value and then added to the open list. If the child is already in the open list, then its path distance is compared with the old value to see if it is shorter. If it is shorter, the node's path distance is updated with the shorter value. If the node is already on the closed list and a shorter path is found, then that node is removed from the closed list and added to the open list. The algorithm then re-evaluates the nodes on open, giving each node a heuristic value, and then the open list is sorted based on heuristic value, with the node with the best heuristic value being the leftmost of the list. In this manner, the "best" node will be chosen during each iteration of the algorithm, and ultimately the shortest path will be found while avoiding a search of the entire state space.

3.4 A* Algorithm

Currently, the most used heuristic based algorithm for path finding is the A* algorithm (pronounced A star) [Pinter 2001]. The A* algorithm is used more than Dijkstra's algorithm or best first search because it is faster than both Dijkstra's algorithm and Best First Search. A* uses a combination of the distance from the starting point to an intermediary node, and the estimated distance from the intermediary node to the goal node, as a way to find the shortest path. This heuristic can be represented by the following equation:

$$F(p) = G(p) + H(p)$$

where G(p) is the cost from the start state to an intermediary node p and
H(p) is the cost from the intermediary node p to the goal node and
F(p) is the sum of G(p) and H(p)

The advantage of the A* algorithm is that the heuristic used to estimate the distance from an intermediary node *p* can be adjusted depending on the type of accuracy desired for the path. For instance, the algorithm can be adjusted to overestimate the cost of H(p), which results in faster computation time, but a less accurate shortest path, or the cost of H(p) can be severely underestimated, thereby causing more of the search space to be searched and a more accurate shortest path, but a longer computation time.

The steps to perform the algorithm are as follows [Lester 2003]:

1. Begin at the starting state, which is initially in the "open list".

2. Look at all adjacent nodes to the starting node and add all of these nodes to the "open list", provided they can be traversed (e.g. nodes are not a wall, water, etc.). Mark that the parent node to these nodes is the starting node, and calculate the F(p) value for all of these adjacent nodes.

3. Add the starting node to the "closed list". Choose the neighboring node with the lowest F(p) value to be the current node.

4. For the new current node, find the F(p) values of all of the adjacent nodes to this current node, provided they are not on the "closed list" and they are valid nodes to be traversed. If the nodes are not traversable or are already on the closed list, ignore the nodes. Otherwise, add the nodes to the "open list".

5. Add the current node to the closed list and remove the current node from the open list. Choose the node in the open list with the lowest F(p) to be the current node. Repeat steps 4 and 5 until the goal node is reached.

When adding an adjacent node to the "open list", if the node already exists in the open list, then the G(p) value must be recalculated to see if the F(p) value using the current node to get to the adjacent node is shorter than the previously calculated F(p) value for the adjacent node. If it is lower, the parent should be changed to the current node, and the F(p) values should b be recalculated for the adjacent node.

The above process should cease when the goal node is found, or the "open list" is empty. If the goal is found, the optimal path may be found by tracing back from the goal node to the starting node by using the parent pointers for each node. If the open list is empty and the goal node was not reached, this means that no path exists to the goal node. Note that in order to calculate G(p) and H(p), and thus F(p), some heuristic needs to be used to estimate the distance from a node *p* to the goal node. Also, each node must be assigned some value in order to allow the heuristic to estimate a value. One method is to assign each node a value, based on the cost of the amount of horizontal, vertical and diagonal moves to get to that node [Lester 2003]. For example, diagonal moves may cost more than a single horizontal and vertical move, thereby, making the cost of diagonals more costly. As a simple example, a vertical or horizontal move could have a of 1, while a diagonal move could have a cost of 3. G(p) for a node would be the G(p) cost of that node, plus the G(p) cost of its parent node.

H(p) can be seen as the approximation of the distance from the current node *p* to the goal node. This value can be over or underestimated depending on the accuracy needed for the shortest path found. One method to calculate the value of H(p) for a node is the "Manhattan method" [Lester 2003]. It is based on how a path would be traversed in the city of Manhattan. Since Manhattan consists of many blocks of buildings, only vertical and horizontal directions are viable (diagonals aren't allowed). Hence the H(p) value evaluates the sum of all horizontal and vertical blocks moved, ignoring diagonals.

A big advantage of the A* algorithm is that it is at least equally as good in terms of performance as Dijkstra's, in that it is "guaranteed to find the best path from the origin to the destination, if one exists" [Pinter 2001].

Pseudocode for the implementation of the A* algorithm is shown below (adapted from [Tanimoto 1995]):

```
Algorithm A*
Begin
        Input the start node S and set GOAL to goal node;
        OPEN := {S};
        CLOSED := null;
        G[S] := 0;
        PRED[S] := null
        found := false;

        while OPEN != empty and found = false do
        begin
                X := node on OPEN with smallest F(p) value;

                remove X from OPEN and put X into CLOSED;

                if X is a goal node then
                        found := true
                else
                begin
                        generate the set ADJACENT of adjacent nodes to X;
                        for each Y in successors do
                                if Y is not already in OPEN or in CLOSED then
                                begin
                                        G[Y] := G[X] + distance(X, Y);
                                        F[Y] := G[Y] + h(Y);
                                        PRED[Y] := X;
                                        Insert Y on OPEN;
                                end;
                                else
                                begin
                                        Z := PRED[Y];
                                        temp := F[Y] – G[Z] – distance(Z, Y) + G[X] +
                                        distance(X, Y);
                                        if temp < F[Y] then
                                        begin
                                                G[Y] := G[Y] – F[Y] + temp;
                                                F[Y] := temp;
                                                PRED[Y] := X;
                                                If Y is on CLOSED then
                                                        Insert Y on OPEN and remove Y
                                                        from CLOSED
                                        end;
```

```
                    end;
            end;
    end;
```

Again, the A* algorithm is easier to understand by observing the algorithm in action.

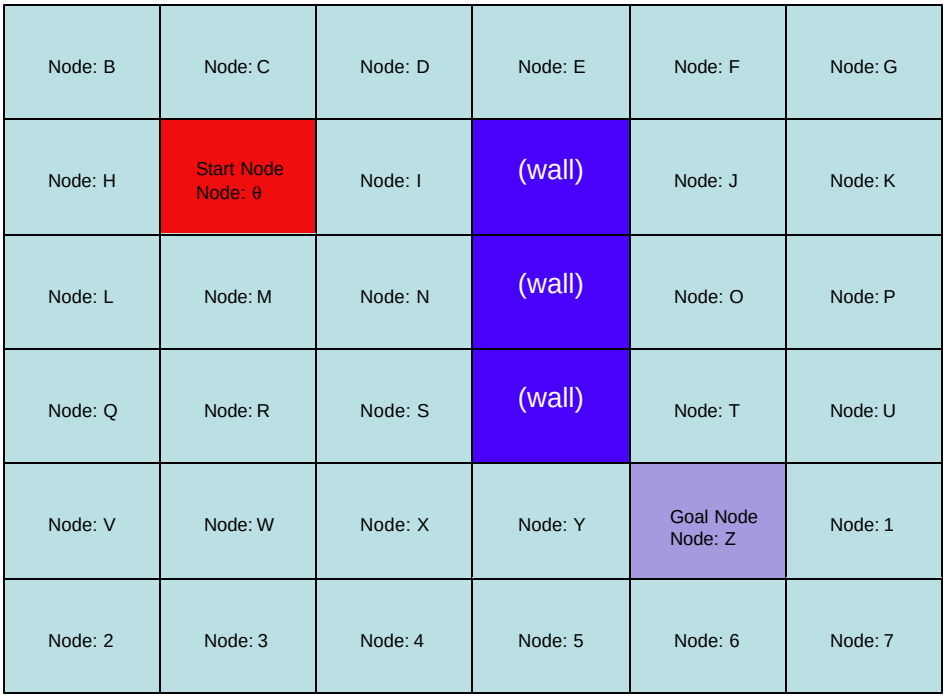


Fig. 3

Node: B G(p) = 14 H(p) = 80 F(p) = 94 Parent = θ	Node: C G(p) = 10 H(p) = 70 F(p) = 80 Parent = θ	Node: D G(p) = 14 H(p) = 60 F(p) = 74 Parent = θ	Node: E	Node: F	Node: G
Node: H G(p) = 10 H(p) = 70 F(p) = 80 Parent = θ	Start Node Node: θ	Node: I G(p) = 10 H(p) = 50 F(p) = 60 Parent = θ	(wall)	Node: J	Node: K
Node: L G(p) = 14 H(p) = 60 F(p) = 74 Parent = θ	Node: M G(p) = 10 H(p) = 50 F(p) = 60 Parent = θ	Node: N G(p) = 14 H(p) = 40 F(p) = 54 Parent = θ	(wall)	Node: O	Node: P
Node: Q	Node: R	Node: S	(wall)	Node: T	Node: U
Node: V	Node: W	Node: X	Node: Y	Goal Node Node: Z	Node: 1
Node: 2	Node: 3	Node: 4	Node: 5	Node: 6	Node: 7

Horizontal / Vertical cost: 10
Diagonal Cost: 14
H(p) uses Manhattan method

Open_List = [B, C, D, H, I, L, M, N]
Closed_List = [θ]

Fig. 4

Node: B G(p) = 14 H(p) = 80 F(p) = 94 Parent = θ	Node: C G(p) = 10 H(p) = 70 F(p) = 80 Parent = θ	Node: D G(p) = 14 H(p) = 60 F(p) = 74 Parent = θ	Node: E	Node: F	Node: G
Node: H G(p) = 10 H(p) = 70 F(p) = 80 Parent = θ	Start Node Node: θ	Node: I G(p) = 10 H(p) = 50 F(p) = 60 Parent = θ	(wall)	Node: J	Node: K
Node: L G(p) = 14 H(p) = 60 F(p) = 74 Parent = θ	Node: M G(p) = 10 H(p) = 50 F(p) = 60 Parent = θ	Node: N G(p) = 14 H(p) = 40 F(p) = 54 Parent = θ	(wall)	Node: O	Node: P
Node: Q	Node: R G(p) = 28 H(p) = 40 F(p) = 68 Parent = N	Node: S G(p) = 24 H(p) = 30 F(p) = 54 Parent = N	(wall)	Node: T	Node: U
Node: V	Node: W	Node: X	Node: Y	Goal Node Node: Z	Node: 1
Node: 2	Node: 3	Node: 4	Node: 5	Node: 6	Node: 7

Horizontal / Vertical cost: 10
Diagonal Cost: 14
H(p) uses Manhattan method

Open_List = [B, C, D, H, I, L, M, R, S]
Closed_List = [θ, N]

Fig. 5

Node: B G(p) = 14 H(p) = 80 F(p) = 94 Parent = θ	Node: C G(p) = 10 H(p) = 70 F(p) = 80 Parent = θ	Node: D G(p) = 14 H(p) = 60 F(p) = 74 Parent = θ	Node: E	Node: F	Node: G
Node: H G(p) = 10 H(p) = 70 F(p) = 80 Parent = θ	Start Node Node: θ	Node: I G(p) = 10 H(p) = 50 F(p) = 60 Parent = θ	(wall)	Node: J	Node: K
Node: L G(p) = 14 H(p) = 60 F(p) = 74 Parent = θ	Node: M G(p) = 10 H(p) = 50 F(p) = 60 Parent = θ	Node: N G(p) = 14 H(p) = 40 F(p) = 54 Parent = θ	(wall)	Node: O	Node: P
Node: Q	Node: R G(p) = 28 H(p) = 40 F(p) = 68 Parent = N	Node: S G(p) = 24 H(p) = 30 F(p) = 54 Parent = N	(wall)	Node: T	Node: U
Node: V	Node: W G(p) = 38 H(p) = 30 F(p) = 68 Parent = S	Node: X G(p) = 34 H(p) = 20 F(p) = 54 Parent = S	Node: Y G(p) = 38 H(p) = 10 F(p) = 48 Parent = S	Goal Node Node: Z	Node: 1
Node: 2	Node: 3	Node: 4	Node: 5	Node: 6	Node: 7

Horizontal / Vertical cost: 10
Diagonal Cost: 14
H(p) uses Manhattan method

Open_List = [B, C, D, H, I, L, M, R, S, W, X, Y]
Closed_List = [θ, N, S]

Fig. 6

Node: B G(p) = 14 H(p) = 80 F(p) = 94 Parent = θ	Node: C G(p) = 10 H(p) = 70 F(p) = 80 Parent = θ	Node: D G(p) = 14 H(p) = 60 F(p) = 74 Parent = θ	Node: E	Node: F	Node: G
Node: H G(p) = 10 H(p) = 70 F(p) = 80 Parent = θ	Start Node Node: θ	Node: I G(p) = 10 H(p) = 50 F(p) = 60 Parent = θ	(wall)	Node: J	Node: K
Node: L G(p) = 14 H(p) = 60 F(p) = 74 Parent = θ	Node: M G(p) = 10 H(p) = 50 F(p) = 60 Parent = θ	Node: N G(p) = 14 H(p) = 40 F(p) = 54 Parent = θ	(wall)	Node: O	Node: P
Node: Q	Node: R G(p) = 28 H(p) = 40 F(p) = 68 Parent = N	Node: S G(p) = 24 H(p) = 30 F(p) = 54 Parent = N	(wall)	Node: T G(p) = 52 H(p) = 10 F(p) = 62 Parent = Y	Node: U
Node: V	Node: W G(p) = 38 H(p) = 30 F(p) = 68 Parent = S	Node: X G(p) = 34 H(p) = 20 F(p) = 54 Parent = S	Node: Y G(p) = 38 H(p) = 10 F(p) = 48 Parent = S	* Goal Node * Node: Z G(p) = 48 H(p) = 0 F(p) = 48 Parent = Y	Node: 1
Node: 2	Node: 3	Node: 4 G(p) = 52 H(p) = 30 F(p) = 82 Parent = Y	Node: 5 G(p) = 48 H(p) = 20 F(p) = 68 Parent = Y	Node: 6 G(p) = 52 H(p) = 10 F(p) = 62 Parent = Y	Node: 7

Horizontal / Vertical cost: 10
Diagonal Cost: 14
H(p) uses Manhattan method

Open_List = [B, C, D, H, I, L, M, R, S, W, X, T, Z, 4, 5, 6]
Closed_List = [θ, N, S, Y, Z]

Fig. 7

Figures 3 through 7 show the A* in action. Red nodes shown are nodes that are currently in the closed list. Teal nodes are nodes that are either in the Open_List or have not been visited yet. Blue nodes are obstacles, in this case, a wall. As demonstrated by each successive figure, neighboring nodes are examined for their F(p) values. The node with the lowest F(p) value is chosen to be the current node and the previous current node is added to the closed list. Ultimately when the path is found, tracing the path is a simple matter of starting from the goal node and traversing the parent of each node.

3.5 Incremental A*

Due to the fact that every algorithm has its advantages and disadvantages, there have been many modifications to the A* algorithm that attempt to fix any deficiencies the algorithm has. One method has been the "Incremental A*" method. Researchers have dubbed the incremental A* as "Life long planning" [Koenig 2001]. Lifelong planning performs better than the standard A* algorithm in environments that are constantly changing the paths. Incremental A* improves upon the standard A* by combining the standard A* heuristic used for calculating paths, and knowledge from previous searches, to speed up the current search. The first search performed by the Incremental A* algorithm performs equally as well as the standard A* because the algorithm can only use the same heuristic that A* uses. No prior path knowledge will have been formulated on the first run to be able to speed up the processing from prior knowledge. However, any subsequence approaches will most likely perform better than the standard A*, and at the worst, as well as the standard A*.

3.6 Hierarchical A*

Another hybridization of the A* algorithm is known as Hierarchical A* [Patel 2001]. One of the main problems with path finding algorithms is that it often takes more than double the amount of time to calculate the shortest path for search spaces that are double the size. This is akin to a circle's radius doubling, but quadrupling the actual area of the circle (e.g. radius = 2, area = 4*pi; radius = 4, area = 16*pi). Hierarchical A* attempts to break up the search space into varying degrees of levels. This approach involves starting at a very broad view of the search space, and then narrowing the search space as needed, but only after the search space has been pruned considerably. For example, an analogy can be made to finding a location on a map. If the goal of a person that did not reside in the United States is to find a destination that resides in a city in the United States, a search would first consist of locating a world map and pinpointing the United States on the map. Then, a map of the state would be consulted to find the general location of the city. Next, a city map would be used to locate the exact street. If the city map wasn't detailed enough, perhaps a street map would have to be used. Similarly, by using a hierarchy with the A* algorithm, this allows the agent to prune the search space, thereby, making the search more computation efficient.

One method of creating a hierarchy could be to use varying resolution. For example, a map could be partitioned into 9 squares at a coarse resolution, and then move to 18 squares at a finer resolution, and then ultimately to 36 squares at the finest resolution to determine the final path.

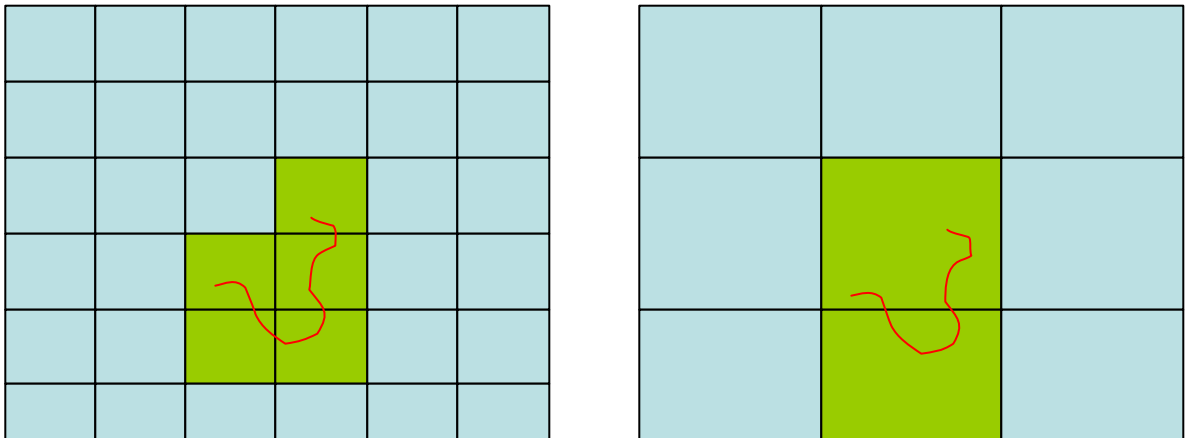


Fig. 8

In Figure 8, it is evident that the search space is reduced dramatically by pruning the nodes that become irrelevant to the finer grained search. At the coarse grained view, the search space is reduced to 2 coarse grained nodes, while in the finer grained view, the search space is reduced from 36 nodes to only 5 nodes.

A necessity of the algorithms described above is a complete knowledge of the entire environment. For example, with Dijkstra's algorithm, in order to calculate and find the shortest path, all of the nodes and their respective cost to traverse the nodes must be known prior to calculating the shortest path. Similarly, the A* algorithm requires that the entire search space be known, with the traversable and non-traversable nodes marked. However, this may be impractical in many cases, such as situations in which the agent possesses no knowledge about the environment. One example of a case in which this would be true is a robot. If a robot is placed in a new environment, it does not have any knowledge about its environment. Hence, if the robot relies on a heuristic based algorithm that requires full knowledge of the environment, the path finding mechanism of the robot will fail.

3.7 D* Algorithm

One method that deals with path finding when all of the information of the environment is not known is the Dynamic A* algorithm, which is also known as D*. Developed by Anthony Stentz, the D* algorithm is essentially the A* algorithm modified to handle a dynamic environment. The problem with the A* algorithm by itself is that it assumes that the path finding agent has complete knowledge of the environment before attempting to find a path to traverse [Stentz 1994]. However, this is often an unrealistic possibility, and thus, the D* algorithm allows an agent with sensors to find an optimal path with little or no knowledge of the environment.

Similar to A*, D* maintains an Open_List and a Closed_List. The Open_List contains nodes that have not been visited yet or are being examined, while the Closed_List holds states that have already been visited. Each node is also given a tag, that marks whether the node has not been visited (tag = NEW), if it is currently in the open state (tag = OPEN), or if it is in the closed state (tag = CLOSED). The D* algorithm uses 2 main functions: Process-State() and Modify-Cost(). Process-State() is repeatedly called to find the shortest path from the starting node to the goal node. Modify-Cost() updates the cost of traversing nodes as new information is found (e.g. obstacles), that make the current optimal path impossible to traverse.

Pseudocode for the Process-State() and Modify-Cost() functions appears below [Stentz 1994]:

```
1       Function Process-State()
2       {
3               X = min-state()
4
5               If X = null then
6                       return -1;
7
8               k_old = Get-KMin();
```

```
9               DELETE(X);
10
11              if k_old < h(X) then
12              {
13                      for each neighbor Y of X:
14                      {
15                              if h(Y) <= k_old and h(X) > h(Y) + c(Y, X) then
16                              {
17                                      b(X) = Y;
18                                      h(X) = h(Y) + c(Y, X)
19                              }
20                      }
21              }
22
23              if k_old = h(X) then
24              {
25                      for each neighbor Y of X:
26                      {
27                              if (t(Y) = NEW) or
28                                 (b(Y) = X and h(Y) != h(X) + c(X, Y)) or
29                                 (b(Y) != X and h(Y) > h(X) + c(X, Y)) then
30                                 {
31                                      b(Y) = X;
32                                      INSERT(Y, h(X) + c(X, Y));
33                                 }
34                              else
35                              {
36                                      for each neighbor Y of X:
37                                      {
38                                          if t(Y) = NEW or
39                                          (b(Y) = X and h(Y) != h(X) + c(X,Y)) then
40                                          {
41                                              b(Y) = X;
42                                              INSERT(Y, h(X) + c(X,Y))
43                                          }
44                                        else if b(Y) != X and h(Y) > h(X) + c(X,Y) then
45                                              INSERT(X, h(X))
46                                        else if b(Y) != X and h(X) > h(Y) + c(Y,X) and
47                                                t(Y) = CLOSED and h(Y) > k_old then
48                                        {
49                                              INSERT(Y, h(Y))
50                                        }
51                              }
52                      }
53              }
54
```

```
55              return Get-KMin();
56      }

1       Function Modify-Cost(X, Y, cval)
2       {
3               c(X, Y) = cval;
4
5               if t(X) = CLOSED then INSERT(X, h(X))
6
7               return Get-KMin();
8       }
```

Stentz's implementation of D* differs a little from the A* implementation described above. The function b(X) = Y refers to the predecessor node of the node X. The predecessor nodes are used to formulate the optimal path by backtracking from the goal state, if it has been reached, to the start state. Rather than use the node with the minimum F(p) value, as in the A* algorithm, Stentz chose to retrieve the node with the minimum key value. The k() function either classifies a node X in the OPEN list as a RAISE state or a LOWER state. RAISE states are used to propagate information about cost increases as they are found (e.g. obstacles are found), while LOWER states are used to propagate information about cost decreases as they found.

As shown by the pseudocode, Process-State() begins by assigning X the state with the minimum key value from the OPEN list. The function Get-KMin() returns the minimum key value prior to any nodes being removed from the OPEN list. Thus, on the first call of Process-State, k_old will be the null value since no states will have been removed from the OPEN list. The function DELETE(X) performs the duty of removing the state X from the OPEN list, and adding it to the CLOSED list.

Lines 11-21 are very similar to the original A* algorithm. The algorithm first compares the k_old value with the heuristic value of the current node X (estimated value

from the current node X to the goal state). If the k_old value is less than the h(X) value (heuristic value), this means that one of the neighboring nodes offers a better path to the current state. Hence, the code calculates the heuristic values of the neighboring nodes of the current node X, and updates the predecessor node of X to be the node that results in a lower cost by going through that node.

In lines 25-33, if k_old is equal to h(X), then the D* algorithm performs similar tasks to the A* algorithm. For all neighboring nodes Y of the current node X, if the node has not been visited yet (tag = NEW), if the node Y does not have its predecessor equal to the node X, or if the h(Y) value does not equal the heuristic value of reaching the neighboring node Y through X, then the algorithm updates the predecessor of the neighboring node Y to be X. The algorithm then updates the OPEN list by inserting the neighboring nodes Y into the OPEN list.

Finally, in lines 34-50, if none of the above conditions were met, the neighboring nodes Y of the current node X are examined. If the neighboring nodes have not been visited (tag = NEW) or h(Y) does not equal the heuristic value of reaching the node Y through node X, then the predecessor of the neighboring node Y is updated to be node X, and the neighboring node is inserted into the OPEN list. The nodes on the CLOSED list are examined in a special case. If a neighboring node Y is already on the CLOSED list, but the cost to reach the current node X is less by traversing a neighboring node Y to reach X, the neighboring node Y is inserted back into the OPEN list.

In testing the D* algorithm with a robot for path finding, the optimal path was roughly “twice the cost of [the] omniscient optimal [path]” [Stentz 1994]. This extra cost is due to a lack of prior knowledge of the environment when formulating the path. This

is akin to planning a route to a destination, only to find out a road required to reach the destination is blocked due to construction and an alternate route must be planned from that point. Hence, this is essentially a comparison of D* in an unknown environment vs A* in a known environment, which in effect is comparing apples to oranges. Comparing the A* algorithm, that possesses knowledge of the total environment, of course provides the A* algorithm with a more optimal path and advantage over the D* algorithm, that has incomplete knowledge of the environment.

Hence, further testing was done to compare the D* algorithm with the optimal replanning algorithm. The optimal replanner algorithm first computes the optimal path from the start state to the goal state. At each obstruction or error in the path, the optimal replanner then recalculates the optimal path from that point in the environment. The results of the studies are best represented as a table:

Algorithm	**1,000**	**10,000**	**100,000**	**1,000,000**
Replanner	427 msec	14.45 sec	10.86 min	50.82 min
D*	261 msec	1.69 sec	10.93 sec	16.83 sec
Speed-Up	1.67	10.14	56.30	229.30

The top row in the table lists the state size. Hence, 1000 refers to 1000 states to search in the space, 10000 refers to 10000 states in the space, etc. It is evident that the D* algorithm improves its performance over the Optimal Replanner algorithm as the state space grows [Stentz 1994].

3.8 CD* Algorithm

Further research has been done by Stentz for path finding. In his most recent work, he investigates a method he deems CD* (constrained D*), which is used for globally constrained problems. A local constraint for a path finding agent is a constraint that can be evaluated at a single step in the solution. One example of a local constraint is

to avoid all obstacles in the path. Hence, at each step, it can be determined whether an obstacle is blocking a particular path or not. A global constraint differs from a local constraint because it is a constraint that applies to the entire solution. An example of a global constraint is when a robot must reach its goal before exhausting its battery. Hence, this is referred to as a global constraint because this constraint applies to the entire path solution [Stentz 2002]. Also, the determination of a step cannot control whether the robot will reach its destination without exhausting its battery, rather, the robot must look at the path altogether to determine this. The D* algorithm has been proven to operate more efficiently than the A* algorithm in environments where the total environment is unknown or complete knowledge of the environment is missing. However, the D* algorithm does not offer an efficient way to optimize globally constrained problems because of the necessity to re-compute paths along the way. This necessary re-computation cannot guarantee that the robot will reach its destination before expiring its battery.

CD* works by incorporating the global constraint in the function itself. This constraint is multiplied by a weight (yielding a weight *w*), which then is adjusted using a binary search algorithm [Stentz 2002]. For each iteration of the CD* algorithm, the weight *w* is adjusted until an optimal solution is found that satisfies the global constraint. Also at each iteration, the algorithm attempts to reduce the weight *w* while still satisfying the global constraint. If at any point this global constraint is violated, then the CD* algorithm will steer the solution away from the states that violated the constraint. The algorithm repeats itself until the optimal path that does not violate the global constraint is found.

3.9 Intelligent Route Finding

Despite the improvement in path finding algorithms, one thing remains in common with all the path finding algorithms described above: "These algorithms are wasteful in terms of computation when applied to the route finding task." [Liu 1996]. The reason is that the agent that use heuristics as their method of path finding don't actually learn the paths that are found. Instead, the agent simply relies on the algorithm to compute an efficient, optimal path any time it needs to move from a node to a new goal node.

Work in intelligent route finding has been conducted by Bing Liu. In his research, he was faced with the problem of designing an intelligent path finder that not only finds the optimal path, but also takes into consideration routes that may not be suitable for human drivers, as well as driver feedback from past routes driven [Liu 1996]. Hence, a driver may decide that he/she does not like a particular road, and the path finder must take this into consideration when delivering an optimal path to the driver. Case Based reasoning, in which the agent formulates new cases based on prior cases was one method of dealing with intelligent path finding that was considered. However, the problem with case based reasoning is that the more cases that are retained, the slower the performance will be in finding an optimal path. With case based reasoning, there is always a trade off between the number of cases verses the performance desired. Hence, for the problem of finding routes in a city, it was deemed that case based reasoning would be inefficient due to the vast number of cases that would need to be stored for the entire city.

A Unique system was designed to avoid case based reasoning. The system designed uses a "disk and memory design" to find routes in the city [Liu 1996]. The system consists of storing MJN's (Major Junction Network's) in the system's memory, while keeping DMR's (Detailed Major Road's) on disk. MJN's represent the major coarse grained view of the city, while the DMR's represent the detailed streets in a fine grained view. The reason DMR's are kept on disk is because they are a lot bigger than MJN's. How the system works is, during a route driven, the driver may inform the system whether he/she does not like to drive that particular road. If a user expresses satisfaction of a road traversed (by not expressing dissatisfaction), then the road cost is lowered in the memory and disk. If the user expresses dissatisfaction with a minor road, then the road cost is raised on disk. When a detailed view is needed, such as needing to find a route in a detailed road view, then that particular part of the DMR is read in. Hence, this avoids using case based reasoning to remember all of the routes for a user. Instead, the cost of traversing roads that are desired by the driver are decreased so that they are chosen in the future, while the cost of traversing nodes that are disliked are increased, so that they aren't chosen or are chosen less frequently. By keeping only the major road networks in memory, this helps keeps the memory usage down and only calls in the necessary nodes when a detailed view is needed to finish the detailed path finding. For example, only traversing the detailed roads is necessary when approaching the destination, otherwise freeways or major highways, which can be viewed in a less detailed view, will be sufficient enough for the first part of the route.

3.10 Hierarchies as a Means of Reducing Computation

As used in Bing Liu's research, one of the ways to face the ever growing complexity of problems is to break the problem down into hierarchies [Seri and Tadepalli 2002]. In Liu's research, he uses a combination of a less detailed view of the city and a detailed view of the city to allow a quicker processing time and less memory space. Further research has been conducted in using hierarchies to help reduce computation time.

One of the main problems with reinforcement learning algorithms is that oftentimes a great deal of time must pass before learning occurs. Typically, the speed of learning is dependent on the "number of states [being the] bottleneck" [Ito et al. 2001]. Coarse graining of perceptions has been one solution to reduce the number of states that need to be investigated during the learning process, and has proved to be a feasible solution to the slow learning process. However, there is a tradeoff between using coarse graining of perceptions and complete perception. By using coarse graining of perception, the learning process is sped up, but results in a degradation of overall performance. Furthermore, "bad habits" that are formed early on during the coarse graining oftentimes make it "difficult to rectify bad habits acquired at the early stage of learning" [Ito et al. 2001]. While complete perception gives better overall performance and a lesser chance of bad habits being formed during the early stages, the downfall is the amount of time it takes for learning to occur. To give the reader an idea how many states have to be investigated with even a few number of agents, the following table is shown:

n* \ *m	**3**	**5**	**7**	**9**
1	9	25	49	81
2	81	625	2401	6561
3	729	15,625	117,649	531,441
4	6561	390,625	5,764,801	43,046,721

In the table above, *n* refers to the number of hunters while *m* refers to the lattice size. A hunter is exactly that in this scenario, a hunter in search of prey. *m* refers to the lattice size and is the size of the grid the hunter can move on. As shown, even with a small amount of hunters and a small lattice size, the amount of possible states the hunter's have to investigate (which includes keeping track of other hunter positions and the prey position) increases at an exponential rate. To also give the reader an idea of how much time it takes for 4 hunters to learn on a machine, it took 8 hours of CPU time using a Pentium III 640-MHz PC with 2 GB of memory. Thus, it is easily evident that learning in real time is not yet possible, even with a small amount of hunters. Thus, the solution is the use coarse graining of perception [Ito et al. 2001].

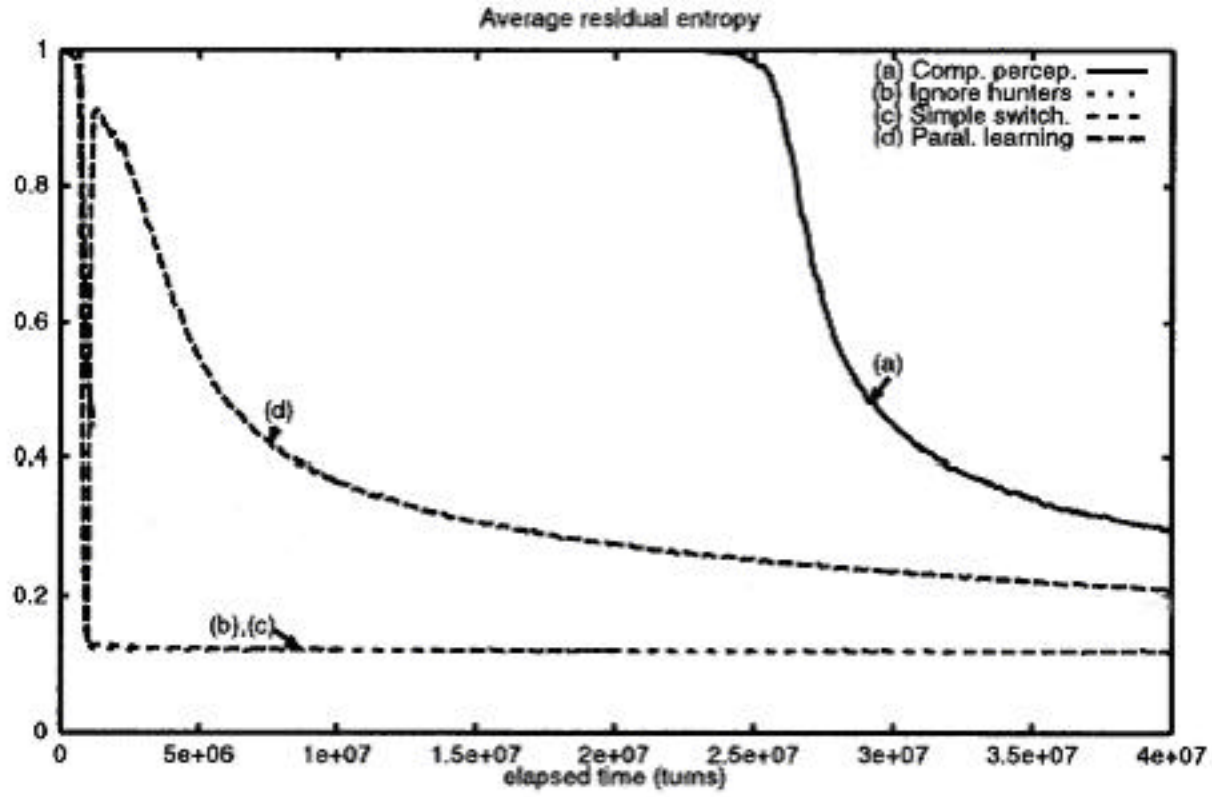
Coarse graining of perception involves "regarding several nearby (but distinct) states as the same," which effectively reduces the number of states to be explored [Ito et al. 2001]. As an example with 3 hunters and a lattice size of 7 (n = 3, m= 7), a complete perception of 117,649 was reduced to only 49 states by using coarse graining of perception. However, the problem of performance degradation still exists if coarse graining perception is used throughout the learning process. Hence, the researchers Akira Ito and Mitsuru Kanabuchi proposed a method that incorporates both coarse grained and complete perception models to form the learning model. The key problem of using both perception models was finding the point in time to switch from coarse grained perception to complete perception. If a switch is made too late, the possibility of bad habits being formed is the problem. If the switch is made too early, the learning process will still be relatively long. Using residual entropy was proposed as the method to calculate the switch point. Residual entropy involves tracking "how much freedom is left for [a] state

S" [Ito et al. 2001]. As learning occurs, the residual entropy approaches zero. Therefore, experiments were conducted by measuring the residual entropy of both the coarse graining perception method and the complete perception method, to conclude a respectable switching time.

Residual entropy can be defined as follows:

$$\text{For each state S,}$$
$$I(S) = -(1 / \log 5) \sum_{a} p(a, S) \log p(a, S)$$

This equation describes an index that computes how much uncertainty is left for a particular state S. For an unlearned state, the residual entropy is log 5 / log 5, which equals 1. This means that there are 5 possible actions with equal probabilities for each state. However, as learning proceeds, the residual entropy approaches zero, as only 1 of the actions then is preferred over the other 4. The results obtained by using residual entropy to decide the switching point from coarse grained perception to complete perception are best represented using a graph:



The (a) line shows how learning occurs with complete perception. As shown by the graph, it takes quite a long time before learning actually occurs, however, once learning does occur with complete perception, it steadily learns. Learning is represented on the graph by a reduction of residual entropy. Learning occurs as the residual entropy approaches 0. Hence, as the lines approach 0, more learning occurs. While the simple switch seems to be good [lines (b) and (c)], the flatlining of the curve means that no more learning is occurring. Hence, learning begins to occur until the switch point in time, at which no more learning occurs. As evidenced by line (d), learning occurs through the timeline by using a combination of coarse graining and complete perception. This is evidenced by line (d) constantly approaching 0. It is also evident that line (d) learns much faster much faster than using solely a complete perception learning model, since residual entropy for line (d) is reduced earlier than line (a) [Ito et al. 2001].

4. Conclusion

What is missing from current research and methods of efficient path finding is an intelligent path finder that is able to find goals and the rewards associated with these goals on its own, as well as explore the unknown environment, determining the most optimal paths as it discovers more about its environment. Many of the standard heuristic based searches for optimal paths do not work well in dynamic environments [Linden 1992]. For example, Dijkstra's algorithm and the A* algorithm require complete knowledge of the environment before a path can be formulated. The D* algorithm comes close to solving the problem of changing environments by using an algorithm that can plan an optimal path as new information is discovered. However, even though the D* algorithm provides an improved efficiency in a dynamic environment over the original A* algorithm, this agent is still not intelligent, in that, any time a new goal state is desired, the agent has to recalculate a path even if has already traversed that path before. Even if this computation does not require a long time, the computation is unnecessary, provided the agent is able to learn paths and learn its environment. The Incremental A* algorithm comes close to being "intelligent", in that, it uses previous traversals to help formulate new paths. However, it also requires complete knowledge of the entire search space since it uses a modified A* algorithm.

For dynamic environments or environments where complete knowledge of the environment is not known prior to exploration or path finding, exploration of the environment is key [Seri and Tadepalli 2002]. Without exploration, the agent will not discover rewards in the environment, let alone the obstacles or information about the

environment. Hence, motivating the agent to effectively explore the area is necessary for agents in an unknown environment.

Using reinforcement learning as a means of exploring the environment is one feasible method for exploration. However, there are many problems with reinforcement learning that must be tackled. One of the problems is the computation time for learning to occur. Another problem with using reinforcement learning is the means of finding the optimal reward policy in the environment.

Hence, it is the author's desire to create an agent that can effectively and efficiently explore an environment that it has no prior knowledge about. Through exploration, the agent will determine the most optimal reward policy of the environment and simultaneously determine the most optimal paths to reach the goal states to receive the optimal rewards.

Many factors can be obstacles for the agents. For example, if a road is suddenly blocked, alternate routes must be found. If a human is unknowledgeable about the environment, the person will either have to randomly explore alternate routes around the blocked street, or determine an alternate route using prior knowledge of the environment. The means of deciding routes is what the author intends to base his research and implementation of the intelligent path finding agent on. Similar to how humans learn routes, exploration will be used for the agent to find optimal paths. A system of reinforcement learning will be implemented, in which rewards will be given for exploration, while negative rewards will be given for negative aspects of exploration, such as bumping into walls, entering hazardous zones, etc. Interaction with the environment will also result in positive and negative rewards associated with goals.

Hence, obtaining certain items in the environment will result in positive rewards given to the agent, while obtaining other items might result in a negative reward. Through this form of reinforcement learning, the agent will ultimately form the optimal reward policy. Using the agent's knowledge of the environment, paths will be created to optimally reach the goal states and retained for future use. Much as humans learn their environment, the path finding agent will learn and remember its environment to automatically traverse the most optimal path in its attempt to reach a goal. At no point will the agent have to recalculate the path towards a goal, unless a new obstruction is added into the environment.

The approach used will combine the concepts of hierarchies to reduce computation time, case based reasoning to formulate new paths, and rote learning (memorization) to traverse previously found optimal paths. However, there is a tradeoff with case based reasoning, in that, the more cases that are retained, the slower processing time is. Since the intended path finding agent must act in a fast paced, real time 3-D virtual environment, determining the most optimal path is a must for the agent since other hostile agents with similar goals will be present in the environment. Taking too long to calculate a path will result in the death of the agent. Thus, to avoid an abundant amount of cases, only the cases that result in achieving a positive reward goal state will be kept in the agent's memory.

Assessment of obstructions of a current memorized path will occur anytime an obstruction is present. An obstruction can be another hostile agent or a permanent obstacle, such as a crumbled wall that now blocks the path. If obstructions are permanent, path splicing will be used to temporarily navigate around the obstacle. When

the agent is deemed to be in a safe environment or the CPU is not under heavy use by the agent, the most optimal path will be re-calculated for that path. Non-permanent obstructions such as moving objects or hostile agents will not force the current case that holds the optimal path to be replaced. The A* algorithm will be used to compute the optimal path since it has been proven to be the most efficient and effective heuristic based algorithm if complete knowledge of the environment is known. Hence, the A* algorithm will only be used to compute paths after a reasonable percentage of the environment has been explored.

5. References

[Ito et al. 2001] Ito, Akira, and Mitsuru Kanabuchi. 2001. Speeding Up Multiagent Reinforcement Learning by Coarse-Graining of Perception: The Hunter Game, Electronics and Communications in Japan, 84 (12), 37-45.

[Koenig 2001] Koenig, Sven. 2001. Glossary of Fast Replanning and Greedy On-Line Planning, http://www.cc.gatech.edu/fac/Sven.Koenig/greedyonline/glossary.html#LifelongPlanning A*.

[Lester 2003] Lester, Patrick. 2003. A* Pathfinding for Beginners, http://www.policyalmanac.org/games/aStarTutorial.htm.

[Linden 1992] Linden, Alexander. 1992. On Discontinuous Q-Functions in Reinforcement Learning, 16th German Conference on Artificial Intelligence, Bonn, Germany (GWAI 92), (August), 199-209.

[Liu 1996] Liu, Bing. 1996. Intelligent Route Finding: Combining Knowledge, Cases, and An Efficient Search Algorithm, 12th European Conference on Artificial Intelligence, Budapest, Hungary, (August), 380-384.

[Luger 2002] Luger, George. 2002. Artificial Intelligence: Structures and Strategies for Complex Problem Solving. Pearson Education Limited, Edinburgh Gate, Harlow, Essex CM20 2JE.

[Morris 1998] Morris, John. 1998. Data Structures and Algorithms: Dijkstra's Algorithm, http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/dijkstra.html.

[Patel 2001] Patel, Amit. 2001. Amit's A* Pages, http://theory.stanford.edu/~amitp/GameProgramming/.

[Pinter 2001] Pinter, Marco. 2001. Toward More Realistic Pathfinding, http://www.gamasutra.com/features/20010314/pinter_01.htm.

[Seri and Tadepalli 2002] Seri, Sandeep and Tadepalli, Prasad. 2002. Model-based Hierarchical Average-reward Reinforcement Learning, International Conference on Machine Learning, Sydney, Australia, (July), 562-569.

[Stentz 1994] Stentz, Anthony. 1994. Optimal and Efficient Planning for Partially-Known Environments, Proceedings IEEE International Conference on Robotics and Automation, San Diego, California, (May), 3310-3317.

[Stentz 2002] Stentz, Anthony. 2002. CD*: A Real-time Resolution Optimal Re-planner for Globally Constrained Problems, Proceedings of the National Conference on Artificial Intelligence (AAAI-02), Edmonton, Alberta, (July), 605-611.

[Tanimoto 1995] Tanimoto, Steven. 1995. The Elements of Artificial Intelligence Using Common Lisp, 2nd Edition. Computer Science Press, New York, New York.