

Search Algorithms in Intelligent Agents

Hemant M. Joshi

Joshua A. McAdams

{hmjoshi, jamcadams}@ualr.edu

Abstract

In a distributed system, the task of finding relevant information and services in a timely manner is complex. When using traditional searching techniques across heterogeneous systems and multiple data formats, this task becomes nearly impossible. This paper will examine intelligent search algorithms that can be used to perform fast and reliable distributed searches. Some of the algorithms examined will be the Generate-and-Test, Best-First, A, and Means-Ends searches. After reviewing some intelligent distributed search techniques, an example of how these techniques can be used within an emerging distributed system called the "Semantic Web" will be presented. In this system, RDF (Resource Definition Framework) metadata in the form of URIs (Uniform Resource Indicator) and XML are used to give semantic meaning to data and services. This metadata is connected via a domain-common vocabulary, or ontology, defined using OWL (Web Ontology Language). Machines can use this vocabulary to intelligently connect and derive meaning from related data. Also included is an example of semantic search. In closing, the reader will find comparison of various algorithms from space and time domain point of view.*

Keywords: A, Asynchronous Dynamic Programming, Beam, Best-First, Bidirectional, Breadth-First, Constraint, Depth-First, Depth-First Iterative Deepening, Generate and Test, Hill-Climbing, Iterative Deepening A*, Learning Real-Time A*, Means-Ends, OWL, RDF, Real-Time Multi-agent, Search*

1. Introduction

The domain of intelligent search algorithms for general-purpose application can be classified into two primary categories: Path Finding and Constraint Satisfaction. These divisions arise based on the type of problem that is being solved.

Path finding problems are focused on finding the path from some initial state to some final state. When solving this type of problem, the start and end points of the search might be known in advance. Finding an efficient, and possibly optimal, path between the start and end state is the goal.

Constraint satisfaction problems are concerned with taking some initial state and converting it to a final state that conforms to some predefined constraints. With these problems, the path is not often as important as the fact that a solution can be found.

There are other domain specific categorizations of search algorithms that can be derived. However, most problems will fall into one of the two previously mentioned divisions.

Search algorithms needed for intelligent agents should be able to take advantage of the distributed nature of the system and are recommended to be of asynchronous nature.

The search algorithms for intelligent agents will focus on following features

1. The localized ability to handle and distribute search responsibilities. It involves breaking the given search task into parts (if possible) and then distributing them in a way that it can effectively produce the search results.
2. Even though search algorithms have to be realistic, they need not always be real time interactive. The search algorithm may take a little more time to produce meaningful results. The response time of these algorithms can always be tuned to achieve better system performance.
3. Localized intelligent search agents are completely aware of the environment and preferences of the user in order to perform the search efficiently.
4. Cooperatively the search problem is broken among various agents and is collectively solved if it is a suitable for the given problem. Synchronization between different agents is handled by the main agent that initiated the search request.
5. All the search agents apart from knowing their share of the problem also possess global knowledge of the problem to yield better results.

The most commonly used problem category in intelligent search algorithms is the Constraint satisfaction algorithms. CSP is a problem to find a consistent value assignment of variables that take their values from finite, discrete domains. A constraint is defined by a predicate. Graph coloring is considered a Constraint satisfaction problem because the constraint is that no two adjacent nodes can have same color. Filtering algorithm can be used to remove those values that do not match two given constraints and end result would be the one that matches the given set of constraints. Similar to filtering algorithm, Hyper-Resolution-Based Consistency Algorithm can be used that determines the values as "no-good" if they do not satisfy the given constraint. This algorithm considers all constraints at any given node. Asynchronous backtracking algorithm is also a constraint satisfaction problem solution with only one difference from Hyper-Resolution-Based Consistency Algorithm i.e. unlike

Hyper-Resolution-Based Consistency Algorithm; it will consider only the current constraint at any given time. So the entire set of constraints to create new “no-good” entries need not be considered.

Intelligent agents can be classified according to their search capabilities and strategies [1]. If the search requirements are simple, a simple learning agent (SLA) that can learn from its experience and infer additional relevant rules can be considered. For the situation of searching from various sources, Simple learning agent with selector (SLAS) can be employed that can learn from its experience, infer additional relevant rules and also select the right possible set. Another scenario for searching where the search is not only from various sources but the results from the various sources need to be combined into a consolidated result, which is Simple agent with combiner (SLAC). SLAC can learn from its experience, infer additional relevant rules and also combine various rules and sources to yield the result.

Like most of other agent related issues, planning forms the most important aspect for searching [2]. A possible algorithm for learning can be devised. Also such an algorithm can be appended with conditions for clause generation. It involves selection of class of values with its relative ness and frequency of occurrence to match the criterion for the goal state. Relative ness of information even though a subjective entity can be formulated to get closer to human like searching. Relative ness of the information can be achieved by two factors, namely, support and confidence of the dataset. These may be affected by the entropy (randomness measure) of the information. An algorithm for learning when the goal state is well defined involves two important design issues. One, relevance to the goal state (i.e. Measure of the adjacency, closeness) and second, relevance to the goodness criterion (i.e. Minimum distance or time)

2. Algorithm Analysis and Examples

In the following sections, many search algorithms will be presented. They range from traditional search methods, to heuristically driven searches, through asynchronous dynamic programming methods.

After reviewing some algorithms, an example of searching the Semantic Web will be given. It will illustrate the combining of distributed searching with semantic meaning.

It should be noted that each search problem is unique, and therefore has a particular search algorithm that would be considered the best solution. There is no search algorithm that satisfies all requirements for all problem sets.

Also, problem spaces for searches are typically thought of as being represented by directed graphs. In these graphs, nodes represent search state. Paths represent the operators that are applied to change state. To simplify the algorithms that must perform searches, it is often more

convenient to logically and programmatically represent a problem space as a tree [3]. This usually decreases the complexity of a search at the cost of duplicating some nodes on the tree that were linked numerous times in the graph. (See Figure 1) Examples in this paper will be mixed, with some problem spaces represented as trees and others as graphs.

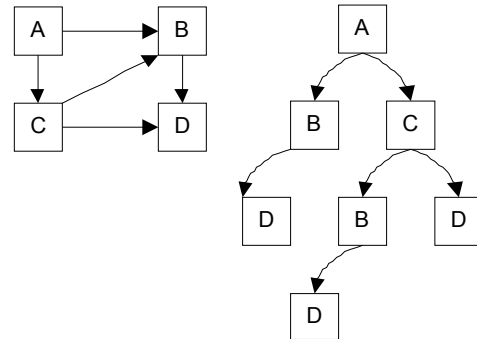


Figure 1 – Graph and Tree

2.1. Traditional Search Algorithms

There are many traditional search algorithms for searching graph and tree data structures. These algorithms are the foundation for more advanced search algorithms created for use with intelligent agents. Traditional algorithms are not designed to run in a distributed fashion. Likewise, these algorithms do not perform intelligent decision-making. They are included in this paper for the sake of completeness and for a basis on which to build an understanding of search algorithms intended for intelligent agents.

The algorithms examined will be the Breadth-First, Depth-First, Depth-First Iterative Deepening, and Bidirectional. They are presented in their purest form in order to highlight the theory behind them. No common modifications of these algorithms will be covered.

2.1.1. Breadth-First

The Breadth-First search [3] is a path-finding algorithm that is capable of always finding a solution, if one exists. The solution that is found is always the optimal solution. This task is accomplished in a very memory intensive manner. Each node in the search tree is expanded in a breadth-wise at each level. All expanded nodes are retained until the search is complete.

The memory intensive nature of this algorithm prevents widespread application, as even a moderate-sized problem space can quickly fill the memory of modern computers.

The search tree for the Breadth-First search is illustrated in Figure 2. Each node is expanded in the order

it is generated. The number inside each node notes the order.

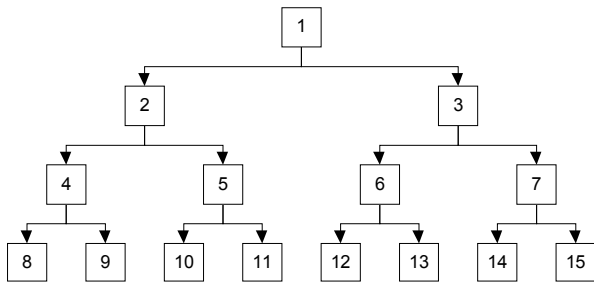


Figure 2 – Breadth-First Search

2.1.2. Depth-First

The Depth-First search [3], [4] attempts to derive a path from some initial state to one of possibly many target goal states. It does this by checking to see if the node that it currently is on is the solution. If not, it generates all immediate successors to the current node. After generation, it travels down each successor in order, recursively searching each branch. This algorithm is equivalent to a common backtracking algorithm in that a path is expanded until an endpoint is found. Once an endpoint is found, the path is reversed, until it can be expanded again down a different branch.

The Depth-First search is not as memory intensive as the Breadth-First search. After the lowest-level node in each branch is traveled, it is freed from memory. The memory savings help with the efficiency of the algorithm. However, the Depth-First search is not always preferable to the Breadth-First search. The reason for this is that the first path traveled is not necessarily the shortest. This allows for the Depth-First search to return a sub-optimal solution.

Also, the assumption that an answer will be provided if it exists is weak. Though problem spaces are logically represented as trees, they could still physically be graphs. For this reason, it is customary for an arbitrary cut-off point to be used to limit the depth of a Depth-First search. This prevents the search from infinitely looping should a cycle occur. The cost of this arbitrary cut-off is never finding a solution if that state exists deeper than the maximum search depth.

An illustration of the search tree for a depth first search can be found in Figure 3. For each node, the number on the left-hand side is the order in which the node was generated, while the number on the right-hand side is the order in which the node was traveled. It can be seen in this illustration that the child nodes are expanded sooner than the peer nodes.

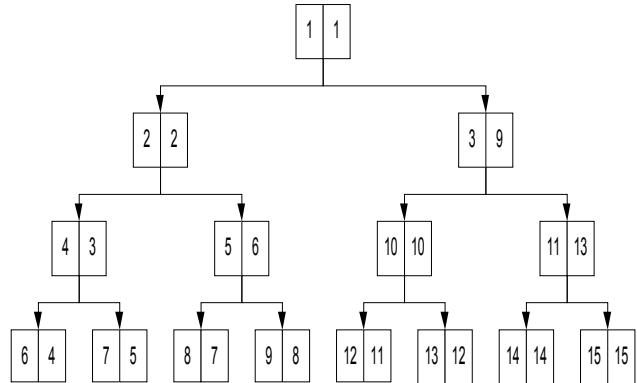


Figure 3 – Depth-First Search

2.1.3. Depth-First, Iterative Deepening

Depth-First, Iterative Deepening search [3] was created as an attempt to combine the ability of the Breadth-First search to always find an optimal solution with the lower memory overhead of the Depth-First search. This is accomplished by performing multiple rounds of Depth-First searches. Each round allows for an increasing maximum depth. This iterative deepening is repeated until a result is found.

For instance, the search will start by performing a Depth-First search for a depth of one. This simply examines the root node. Next, a Depth-First search will be performed to a depth of two. This examines the root node and its immediate successors. If that search comes up empty-handed, the search is continued to an iteratively increasing depth.

This search does provide the desired results stated above. It always finds an optimal solution, if one exists. Also, it has the same memory overhead as the Depth-First search. The downside to this search is that all previous searches are repeated with each iteration. This adds extra time to the search process; however, this time is considered nominal since the search tree expands exponentially upon each iteration. The exponential expansion causes the search to always spend more time in the lower-level nodes than the intermediate nodes.

An illustration of the Depth-First, Iterative Deepening search can be seen in Figure 4. As with the Depth-First search, for each node, the number on the left-hand side is the order in which the node was generated, while the number on the right-hand side is the order in which the node was traveled. The first four iterations of searching are displayed.

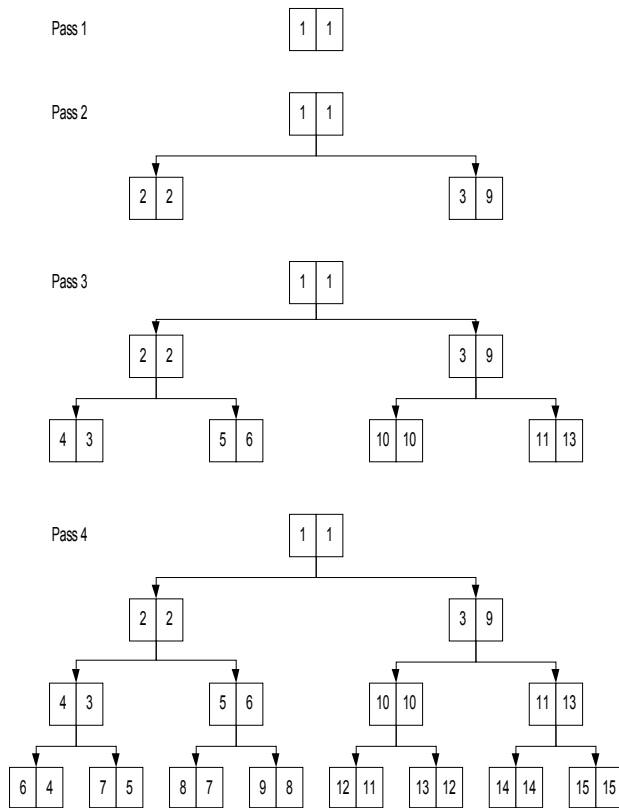


Figure 4 – Depth First, Iterative Deepening Search

2.1.4. Bi-directional

All of the traditional algorithms covered so far run single-threaded from initial state until they find the goal state. Often in search algorithms, the value of the goal state is not as important as the path that it took to get there. In almost all path-finding problems the goal state is known well in advance. In these problems, if any state is variable, it is the initial state.

When both the initial and goal states are known in advance, a bidirectional algorithm [3] can be used to speed up the search process. With this algorithm, a search is initiated from each end of problem space. A solution is found when the searches meet. The type of search algorithms implemented at each end of the problem space is unimportant, as long as one of the searches is breadth-first in nature. Having one search span the breadth of the problem space guarantees that the searches will meet if there is a solution.

Since the Bidirectional search works at both ends of the problem space, the time required to perform the search can be significantly reduced. However, the requirement that one of the searches be Breadth-First can cause the algorithm to use large amounts of memory. Also, if the goal state is not known in advance, this search is impossible to perform.

2.2. Heuristic Search Algorithms

Heuristic search [5], [6] algorithms typically take the form of traditional algorithms, modified to make intelligent decisions when choosing which path to travel first. The heuristic is a 'rule of thumb' that is used to steer the algorithm in a direction that seems more likely for the given problem. These algorithms are useful in intelligent agent systems, especially when compared to traditional algorithms.

However, heuristic algorithms are still limited in that they are designed to run in synchronous order. This does not mean that the searches cannot run with some degree of parallelism. The limitation is that global synchronization is needed for each step in the algorithm to assure successful operation.

In the following sections, two very basic heuristic algorithms will be covered: Generate & Test and Hill-Climbing. Then, the Best-First algorithm will be discussed, along with its derivatives: A*, Iterative-Deepening A* and Beam. Finally, more abstract methods will be reviewed, including Means-Ends and Constraint searches.

2.2.1. Generate and Test

Generate and Test algorithms [5], [6] are brute-force algorithms that simply generate a possible solution and test to see if it is correct. If the solution is not correct, a new possible solution is generated and the cycle repeats. This algorithm benefits from being relatively simple to implement; however, its time complexity is higher than any other algorithms reviewed in this paper.

One benefit of this algorithm is that the memory footprint is typically only as large as the depth down the search tree to the solution. However, this is only the case if the algorithm is designed in such a way as not to retain any knowledge of past attempts. If no history of failed guesses is retained, care must be taken to ensure that the algorithm does not repeatedly generate past attempts, or an infinite loop could form.

This algorithm can be modified to never test obviously bad solutions and to lean toward the generation of more common solutions first. However, the potential always exists for this algorithm to take an unreasonable amount of time to complete.

An example of a Generate and Test algorithm is a brute-force combination lock cracker. This type of problem is illustrated in Figure 5, assuming a combination of two numeric digits.

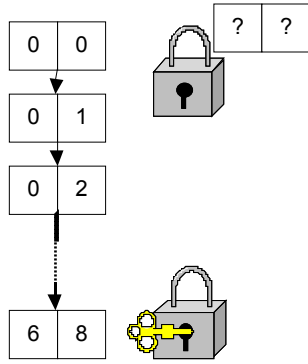


Figure 5 – Generate and Test Search: Lock Cracker

2.2.2. Hill-Climbing

Hill-Climbing [3] is a very basic form of search algorithm in which only the local state is considered when making a decision of which node to expand next. When a node is entered, all of its successor nodes have a heuristic function applied to them. The successor node with the most desirable result is chosen for traversal.

One obvious downside to this algorithm is that it does not maintain any state. This can lead to sub-optimal routes being taken, as the local maximum is not always the global maximum. Also, there is potential for the algorithm to get stuck in a loop, since it does not remember where it has been. The only way to resolve this issue is to keep a list of visited nodes. This; however, creates potential for a large amount of memory overhead.

Figure 6 illustrates the Hill-Climbing search starting at node A and trying to reach node E. In case 1, the solution is found; however, it is not the optimal solution. In case 2, the search gets stuck in a loop. In all cases, the numeric values on the paths indicate heuristic value, with lower values assumed to be better. These searches are shown as graphs to illustrate looping.

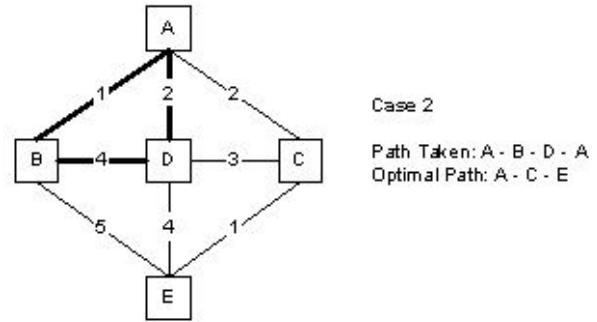
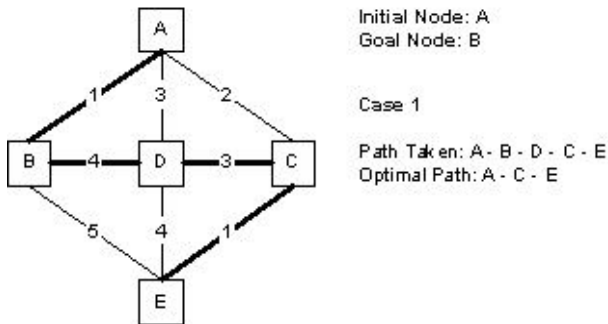


Figure 6 – Hill-Climbing Search

2.2.3. Best-First

The Best-First search [3], [5], [7] is a heuristic algorithm that serves as a combination of the Breadth-First and Depth-First searches. This algorithm has the potential to change its search from one of breadth to depth, and visa-versa, at any step during execution. This is possible because the search generates each node's successors when the node is processed. At that time a heuristic weight that is used to indicate the best node is assigned. This weight is used to select the next node to process, regardless of that node's position in the search tree.

Because of this weighted selection, the next node in the search could be one of the current node's successors, an adjacent node, or even higher-level node in the search tree. A priority queue is used to cache the generated nodes waiting to be processed. The priority used in the queue is the heuristic value representing distance-to-goal.

Due to the need to keep track of all processed nodes, the memory requirements of this algorithm approach that of the Breadth-First algorithm. The gamble is that the heuristic function will be reliable enough to find a solution before memory limits are met.

An illustration of the Best-First algorithm can be found in Figure 7. In this example, the numeric value to the top-left is the order in which the node was generated. The numeric value to the top-right is the order in which the node was processed. The alphabetic value at the bottom of each node is an example of a heuristic value, evaluated in alphabetical order.

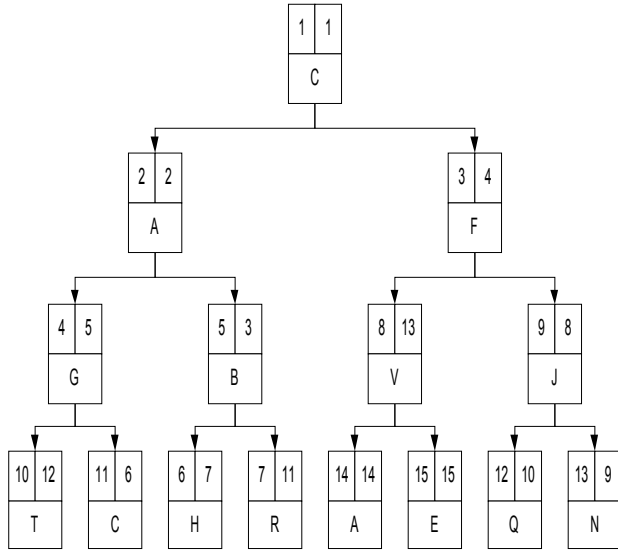


Figure 7 –Best-First Search Algorithm

Best-First algorithms are often referred to as Greedy algorithms. This is because they quickly attack the most desirable path, as soon as its heuristic weight becomes the most desirable.

2.2.4. A*

The A* search [3], [8], [9] is a modified version of the Best-First search. An A* search selects the next node to expand using a two-part heuristic estimate that considers the cost of past travel added to the estimated cost of reaching the goal state from that node. This is more involved than the Best-First search, which only considers the likelihood that the next node traveled will be closest to the goal. The general formula that defines this algorithm is often stated as:

$$f(n) = g(n) + h(n)$$

In this formula, $g(n)$ represents the cost that was incurred in reaching a given node; n . $h(n)$ represents the heuristically estimated cost of reaching the goal state from that node. This makes $f(n)$ equal the estimated total cost to reach the final goal state if node n is expanded.

A* uses an OPEN and CLOSED list of nodes in order to perform a search. The OPEN list contains nodes that have been generated, but not traveled. The value of $f(n)$ is computed for each node in the OPEN list in order to choose which node to expand next. A priority queue weighted on the value of $f(n)$ can be used to select the next node to expand. As a node is expanded, it is put on the CLOSED list and its immediate successors are put on the OPEN list.

The speed of execution of the A* search is highly dependent on the accuracy of the heuristic algorithm that is used to compute $h(n)$. If this function is not well designed, the A* algorithm can use large amounts of time and memory as it searches through numerous dead-ends,

keeping track of each node that it has visited, as well as, their immediate successors. One strong point of the A* search is that if the heuristic function never over-estimates the value of $h(n)$, then the first solution found will always be the optimal solution.

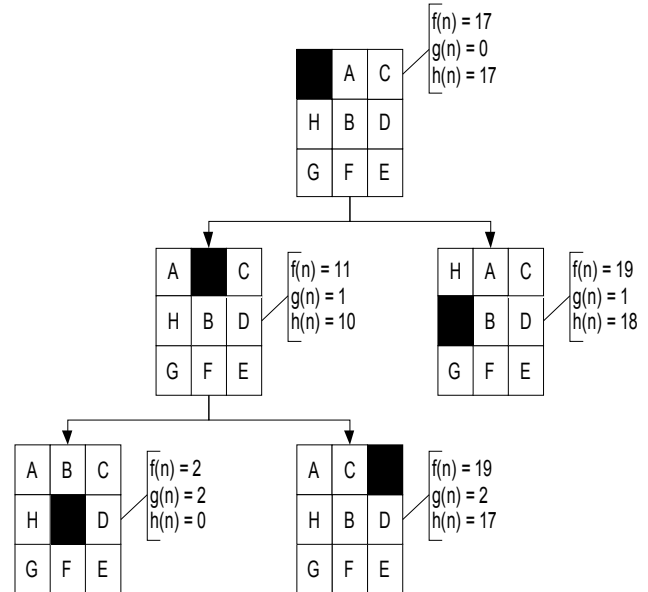


Figure 8 – A* Search: 8-Squares

An illustration of the A* algorithm can be seen in Figure 8. In this illustration, the algorithm is attempting to solve the 8-puzzle. The heuristic function used to calculate $h(n)$ is Nilsson's Sequence. This is a proven heuristic function for estimating the distance an 8-puzzle is from being completed. It would be noted that $h(n) = 0$ when the solution is discovered.

2.2.5. Iterative Deepening A*

Using the same reasoning as the Depth-First, Iterative-Deepening search [3], the Iterative-Deepening A* has potential to reduce the overall memory requirement of the A* search. In addition, the iterative deepening mechanism allows for an optimal solution to always be found if it exists. The idea behind this algorithm is to set an arbitrary cut-off depth at which the A* search stops searching. If a search is performed to some depth and no solution is found, the depth is extended and the search is executed again.

As expected, this search has the potential to take more time than the traditional A* search; however, the exponential nature of the search tree makes this time insignificant.

2.2.6. Beam

The Beam search [3], [10] is yet another modification of the A* search. This search only retains information for the n best nodes. At each iteration, nodes that do not seem

promising for leading to a search solution are pruned from the list of generated nodes that are waiting to be expanded. This pruning reduces the memory footprint of the A* search. However, the optimal solution could exist on a seemingly sub-optimal path and therefore be discarded. Likewise, there is potential to never find a solution, even if one exists, since that solution could have only been accessible by traveling down a pruned path.

2.2.7. Means-Ends

Means-Ends [11], [12], [13], [14] searches are a breed of algorithm that uses forward and backward planning in order to reduce the difference between the current and goal states. This class of search works by taking a goal and breaking it down into sub-goals. These sub-goals are attacked in the order of most effect within the current goal or sub-goal. The effect being distance moved from initial state to the goal state. The overall effect desired is to reduce the difference between the initial and goal states to zero.

There are three major types of goals used in many Means-Ends algorithms. They are:

1. Transform a state into a set of states
2. Reduce a difference possessed by a state
3. Apply an operator to the state to reduce the difference

These goals are illustrated in Figure 9. The example given is the process of solving the Tower of Hanoi problem. It can be seen that the Means-Ends analysis breaks the overall goal up into smaller goals, which lead to the final solution.

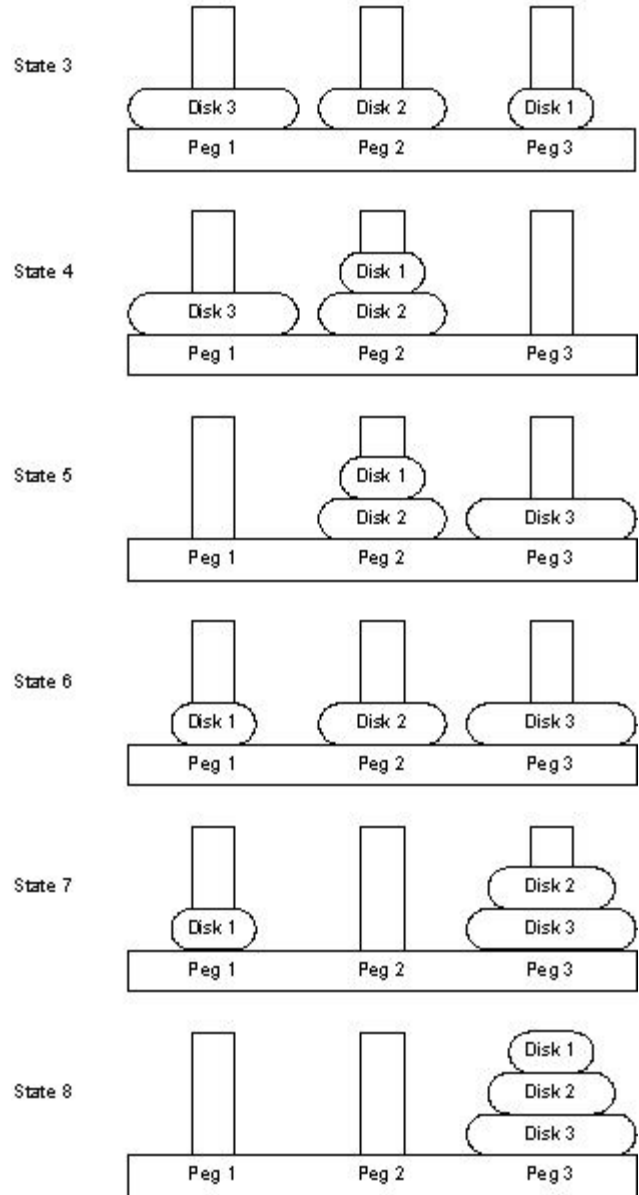
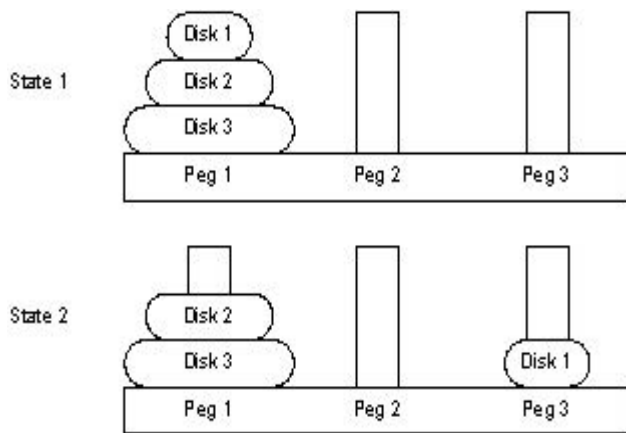


Figure 9 – Means-Ends Search: Towers of Hanoi

One problem that can occur with Means-Ends algorithms involves the narrow focus taken when reaching a goal. Sub-goals have little or no knowledge of the other goals pending at any given time. For this reason, it is possible for a sub-goal to undo a necessary precondition of a super-goal and send the algorithm into a loop or invalidate it entirely.

Means-Ends searches are a common form of heuristic algorithm that has stood the test of time. Early implementations included the General Problem Solver (GPS), FDS, and STRIPS. Today, Means-Ends analysis is still used to create effective searches in the fields of distributed computing and artificial intelligence.

2.2.8. Constraint

A Constraint search [5] does not refer to any specific search algorithm, but to a layer of complexity added to existing algorithms that limit the possible solution set. For instance, a search could be executed to find a computer system for a consumer to purchase. If the consumer has a limited budget and the CPU, monitor, keyboard, mouse and printer all had to be purchased separately; multiple searches might have to be performed in order to piece together the best system. Heuristics and acquired knowledge can be combined in order to produce the desired results during each iteration.

2.3. Asynchronous Search Algorithms

None of the algorithms examined to this point directly lend themselves to execution in a distributed system. All have required a degree of synchronization that lessens the benefits of a loosely coupled distributed environment. Recently, there have been attempts to rectify this and create a set of algorithms that can perform well in a parallel asynchronous fashion. This allows for each individual program unit to operate locally without knowledge of the global problem, yet still benefit the search as a whole.

The algorithms covered will include Asynchronous Dynamic Programming, Learning Real Time A*, Real Time A* and Real-Time Multi-agent.

2.3.1. Asynchronous Dynamic Programming

Asynchronous Dynamic Programming, ADP, is the most basic form of asynchronous path-finding algorithms. Its simplicity prevents it from being applicable except in very small and controlled environments. However, it serves as a starting point for other asynchronous path-finding solutions that will be examined in the following sections.

The idea behind this model is to asynchronously determine the minimum distance from each node to the goal state. This distance, represented by $h^*(n)$ is calculated using the function:

$$f^*(j) = k(j, n) + h^*(j) \text{ for each successor } j$$

$$h^*(n) = \min f^*(j)$$

This reads that each node, n , calculates its distance from the goal state, $h^*(n)$. It does this by, in turn, asking all successor nodes, j , how far they are from the goal state, $h^*(j)$. Node n adds this value to its distance from node j , $k(j, n)$, for each succor node. The minimum value is selected as the distance from n to the goal state.

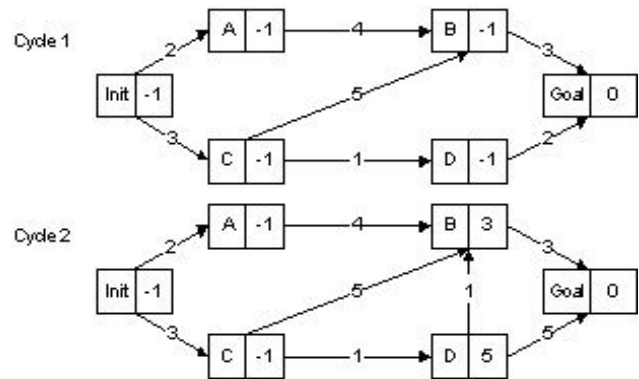
For this system to work, all goal states must return a distance of zero. Also, since each node will be executing asynchronously special care must be taken on the heuristic values exchanged between nodes. For example, when a node asks a successor for its distance from the goal, it will not get a valid value back until that successor is able to calculate its distance from goal. For

this reason, all nodes are seeded with a grossly large or invalid value, possibly infinity, until they can begin to make calculations that are more accurate.

The system works because each node continually polls its peers for their distance from the goal. As the nodes execute, accurate values propagate from the goal state to the initial search state until a valid search path is found.

In Figure 10, the propagation of heuristic distance is illustrated. Node labels are displayed on the left side of each node and the current value of $h^*(n)$ is displayed on the right. Heuristic weights are assigned to each path. They are the result of $k(i, n)$. A detail description follows.

1. Cycle 1 - All nodes are set to a default invalid value, except for the goal node, which is set to zero.
2. Cycle 2 - Nodes B and D estimate their heuristic values by polling the goal state. At the time of estimation, node B actually has an invalid value, so node D is unable to take advantage of the path from D to B.
3. Cycle 3
 - a. Nodes A and C, are able to make their first distance estimations. This is because nodes B and D provided them with valid values.
 - b. Node D now has different distance estimation, as it was able to find a shorter path to the goal by traveling through B. This estimation puts it out of sync with node C.
4. Cycle 4 - The source node has found a path to the goal. However, it is not the optimal path.
5. Cycle 5 - The optimal path surfaces when the new heuristic values finally propagate down from node D through node B.



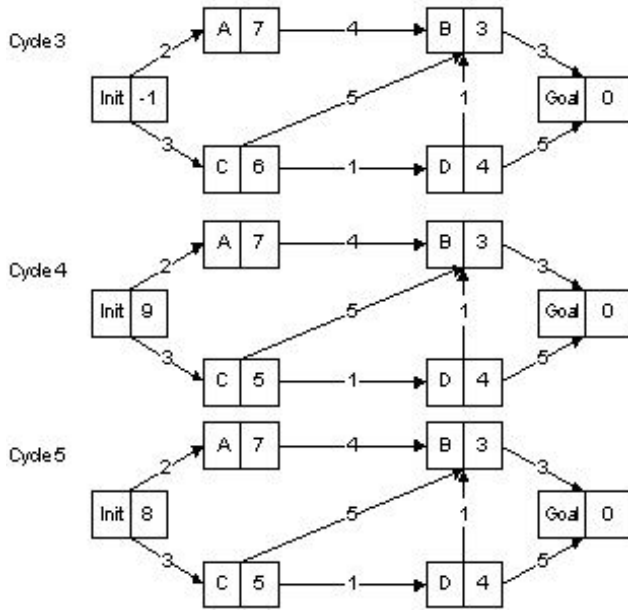


Figure 10 – Asynchronous Dynamic Programming

2.3.2. Learning Real-Time A*

The Learning Real-Time A*, LRTA*, is a variation of Dynamic Programming that does not require a search process to be running on every node in the problem space. Instead, nodes are activated in an A* search fashion. This style of search is performed by starting at the initial node and applying the A* heuristic to each possible successor. The lowest heuristic valued successor is then expanded and the initial node is left executing an update program, like those from ADP. This cycle continues, with an ever-expanding trail of dynamic asynchronous heuristic values being updated until a solution for the search is reached or the search returns failure.

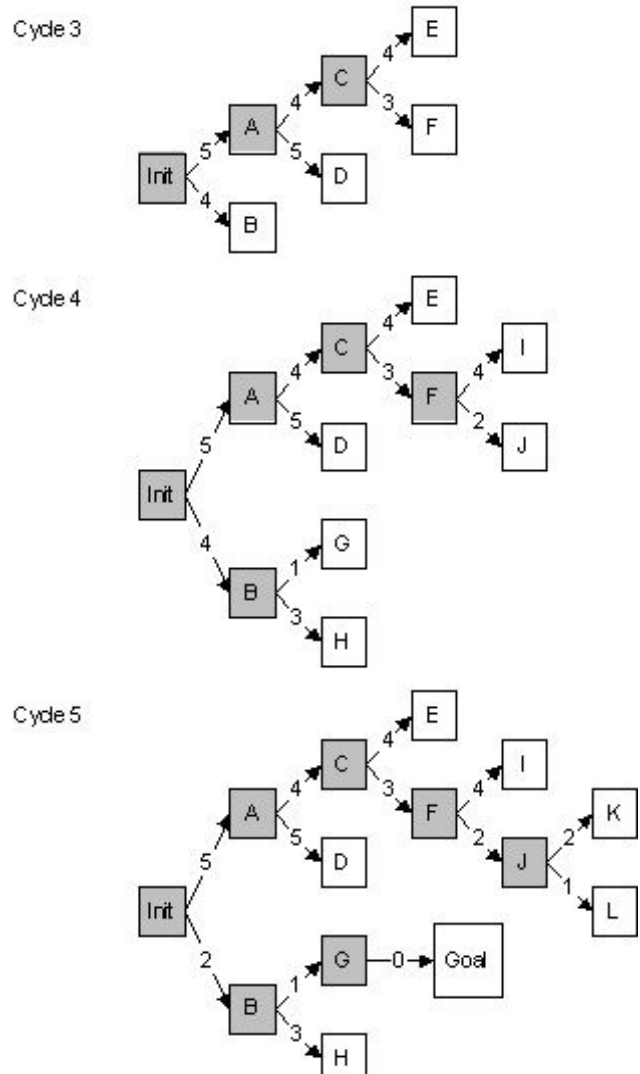
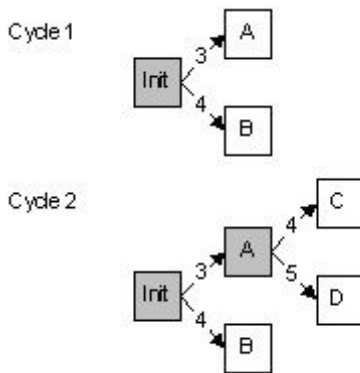


Figure 11 – Learning Real-Time A*

An example of this type of search is detailed in Figure 11. In the example, the A* style of search is used to travel down the best path. However, after the path's heuristic value is updated asynchronously, the initiating node realizes that there is possibly a better path, which it then expands. This example is explained in detail below.

1. Cycle 1 - The initial state evaluates $f(j)$ for each successor, it will choose to expand node A. The initial node will repeatedly check values of $f(j)$ asynchronously.
2. Cycle 2 - Node A performs heuristic evaluation on nodes C and D. Node C will be the next to expand. Node A will now recalculate heuristics asynchronously.
3. Cycle 3
 - a. Node A realizes that its own heuristic estimate is bad, based on the estimates of C and D, so it updates its weight to 5.

- b. Node C performs a heuristic evaluation of nodes E and F. Node C will now run asynchronously. Node F will be the next node expanded.
 - c. The initial node notices that the heuristic weight of node B is now desirable, so it initiates node B.
4. Cycle 4
- a. Node F evaluates nodes I and J, node J will be the next to be expanded. Node F will run asynchronously.
 - b. Node B evaluates nodes G and H. Node G will be the next to be expanded. Node B will run asynchronously.
5. Cycle 5
- a. Node J evaluates nodes K and L.
 - b. Node B updates its heuristic value.
 - c. Node G finds that its next successor is the goal state.

A variant of the LRTA* search, called the Real-Time A*, RTA*, search uses the same basic algorithm as above. The difference is that instead of accepting the best heuristic value for updating path weights, the second best value is used. This change is intended to make the search learn better by increasing the number of overall asynchronous processes. Unfortunately, this also causes the heuristic value of $h(n)$ to possibly be greater than the actual path distance. This breaks the ability of the A* algorithm to find the optimal path first.

2.3.3. Real-Time Multi-agent

Any of the asynchronous searches detailed above can be executed with a single agent or with multiple agents. The existence of multiple agents requires of inter-agent cooperation. This is most often done by having all agents operate in parallel in the same problem space while attempting to obtain the same goal. Each agent runs its own version of the search, while all agents share information that would benefit each other. When the first agents search is complete, all other stop searching, since the current task has been accomplished. When a real-time asynchronous search is performed with multiple agents, it is considered to be a Real-Time Multi-agent search.

2.4. Semantic Search Algorithm

Apart from the distributed path finding and similar search algorithms, attention should be paid to meaningful information searching using intelligent agents. In the semantic world of intelligent agents used for searching, the stress will have to be more on the information semantics or in other words the meaningfulness of the information. Consider an example situation [16].

"Tell me what wines I should buy to serve with each course of the following menu. And, by the way, I don't like Sauterne."

It would be great deal of difficulty if one has to do a semantic search for a particular type of a wine excluding the unwanted ones. On the same lines, assignment of software agent can be considered as the task of "making a coherent set of travel arrangements". To be able to successfully search for this sort of expression, search efforts have to make beyond mere keywords. Due focus should be on the semantics of the resources described on the Internet. Off course additional interpretation of the data is also necessary and done in a separate layer.

According to [16], "The OWL Web Ontology Language is a language for defining and instantiating Web Ontologies. Ontology is a term borrowed from philosophy that refers to the science of describing the kinds of entities in the world and how they are related. OWL ontology may include descriptions of classes, properties and their instances. Given such ontology, the OWL formal semantics specifies how to derive its logical consequences, i.e. facts not literally present in the ontology, but entailed by the semantics. These entailments may be based on a single document or multiple distributed documents that have been combined using defined OWL mechanisms." The semantic information is represented using RDF. The Resource Description Framework (RDF) is a general-purpose language for representing information in the Web [17].

Implementing an intelligent web hunting agent [18] is a reality and can be explored to find out the application of semantic searching into real life practice. A practical life situation is explained below and the semantic search process is mapped to the human like search capabilities. Consider a situation when John has a tooth ache. He needs to maintain his daily routine, but also need to schedule an appointment with the dentist in his area approved by John's insurance company, at the earliest. Now first thing to notice is that this is a constraint based search. The success state is the earliest appointment with a dentist in John's area supported by his insurance company. Here are the steps, semantic searching should follow:

1. John's agent contacts his insurance company website and looks for the doctors in John's area supported. This is Resource Description Format that John's agent is able to read and interprets to deduce the top 3 results of the doctors with their relevant information.
2. After learning John's problem and matching the type of doctor (dentist) with the type of a problem (tooth ache), agent will try and check the reports of patient feedback ratings as well as ADA (American Dental Association) credentials for the considered dataset of dentists.
3. The next constraint is to match the earliest appointment. In this case John's agent after understanding John's schedule and nature of problem

(severe, average, mild etc.) will contact individual agents of the considered dentists to request an appointment.

4. Upon receiving the favorable results from one of the agents, the appointment is confirmed at the desired dentist and that is marked in John's calendar as well to remind him about it.
5. John's case history and other patient related information is also transacted at due course in time and the bill is also paid later.

In the above solution, first three stages will make extensive use of search algorithms that are helpful. Also to note that constraints used in the above situation are as follows:

- The doctor should be a dentist.
- Dentist should be in the vicinity of the place where John lives.
- Dentist should be in the provider list of John's insurance company.
- Dentist(s) has required credentials and patient feedback ratings as well as experience.
- Dentist is free at the time John is trying to seek an appointment.

This situation can also be extrapolated to fit another scenario like "Fixing an entertainment schedule John and his date tonight at Broadway Theater." The knowledge query can be made to retrieve meaningful and relevant information according to given set of constraints using Knowledge Query and Manipulation Language (KQML) [19]. KQML supports and allows agents to communicate with one another and share knowledge with one another.

3. Comparison of Algorithms

3.1. General Information

For the agents to be effective the issues that are considered for design should be finalized and different search algorithms can be compared according to these features acting as a metric. Some of the features to compare these algorithms with are as follows:

1. Latency of the network affecting search process
2. Searching for semantically correct information
3. Relevance of the information
4. Order of complexity of an algorithm
5. Suitable for Text, graphics, audio, video (multimedia) contents
6. Does the algorithm consider aging of information and outdated/obsolete information
7. Distributiveness of an algorithm

The relevance of the information and aged information pose great importance for effective search results. Intelligent agents will play a pivotal role in linking metadata of the information to the actual information content. The order of complexity of these algorithms will

be less if the algorithms can be distributive. Apart from the features mentioned above, for real life applications, there are two important criteria. A search algorithm is optimal if it is guaranteed to find the best solution from a set of possible solutions. An algorithm is complete if it will always find a solution if one exists [20]. The *time complexity* defines how fast an algorithm performs and scales. The *space complexity* describes how much memory the algorithm requires to perform the search. In the following section, some of the search algorithms, mentioned in this paper, are evaluated based on these two parameters.

Breadth first search is a complete algorithm with exponential time and space complexity. (Order of complexity $O(2^n)$) It examines each one step away from the initial step, then each state two steps away, and so on.

Depth-first search has lower memory requirements than *breadth-first search*, but it is neither complete (it can get stuck in loops) nor optimal. In *depth-first search*, a single path is chosen and followed until a solution is found, or a dead end (a leaf node) is reached. The algorithm then backs up and continues down another path.

Iterated-deepening search uses a modified version of *depth-first search* to limit the depth of search. It does this with a control loop that increases the depth with every iteration. The approach combines the best of *breadth-first* and *best-first search*. It is complete and optimal, and is has much lower memory requirements than unlimited *depth-first search*. The only problem that can be foreseen with this algorithm is its use in distributed agent world. This algorithm is not geared towards being used by task sharing, distributed approach etc.

Heuristic search algorithms use information about the problem to help direct the path through the search space. This information is used to select which nodes to expand. *Best-first search* always expands the node that appears to be closest to the solution. *A* search* uses the known cost combined with an estimate of the distance from the state to the goal to choose a node to expand. *A* search* is complete and optimal, and thus has memory requirements comparable to *depth-first search*.

Constraint satisfaction search uses information about valid states to help limit or constrain the range of the search. This algorithm can be designed to work in distributed environment where each processor can work on the given constraint and the final result or search solution is found by combining these constraints together.

Means-ends analysis solves problems by detecting differences between states and then trying to reduce those differences. Memory requirements can be high for this analysis but with distributed task sharing, memory requirements can be minimized. It is a complete algorithm and it will find a search solution if one exists.

Genetic search uses a biological metaphor that relies on problem solutions being represented by binary

strings called chromosomes. The chromosomes are manipulated by genetic operators such as crossover and mutation. *Genetic search* performs parallel search, where the population size represents the degree of parallelism. The various genetic operators can force widespread exploration of the search space (crossover) or relatively localized hill-climbing behavior (mutation). High Memory requirements in genetic search are taken care of by its parallelism features. Genetic algorithms are used for complex interpolated search requirements.

3.2. Order of Complexity of Algorithms

When determining which search algorithm is appropriate for a problem space, it is necessary to derive and compare general attributes of each algorithm [7]. These attributes create a basis for decision making. Each of the algorithms examined in this paper posses strong and weak points. It is a function of the problem space to weigh the trade-offs between algorithms and find which search algorithm provides the best solution.

Common attributes that are used in search algorithm analysis include system memory requirements, search time expectations, the ability to find a solution if one exists, and the ability to find the optimal solution from a set of valid solutions.

Memory requirements are an increasingly important attribute of search algorithms. When a search is performed within a measurable finite space, required memory is easy to calculate and provide. However, when a search is performed in a dynamic and un-measurable space, such as the Internet, the memory of even the largest super-computer can quickly be consumed while caching data needed to successfully complete a search.

Search time is a slightly more fluid requirement. With proper usage of agents and/or off-line searches, it is possible for a query to take hours, or even days, without adversely affecting the user. However, there still exists times where fast, user-interactive, queries are necessary. Though search time can often be worked around, it should still be considered when selecting a search algorithm.

It would seem obvious that the ability to find a solution, if one exists, is a necessary requirement for a search algorithm. However, there are cases where a quick failed search is more desirable than an extended successful search. Take for instance an overloaded node of a distributed operating system looking to migrate off one of its processes. If it can quickly find an available node that will accept its extra process, then the process will be migrated. However, if the search for an available node takes too long to complete, the original node possibly could have executed the process locally faster. Also, a close node that was unable to accept the process at the beginning of the search might now be available for process migration. In this case, repeatedly executing a fast incomplete search is more desirable than executing an exhaustive complete search.

Some searches can have many possible results that will satisfy the criteria of the search. Of the result set, only portions of the results are optimal. Optimal can have many different meanings. For example, it can mean the shortest path or cheapest end point. When searching, there are times where any answer will do and other times where only the best answer will suffice. Knowing whether the best answer is needed or not can affect the selection of search algorithms.

Algorithm	Time	Memory	Complete	Optimal
Breadth-First	$O(b^d)$	$O(b^d)$	Yes	Yes
Depth-First	$O(b^d)$	$O(d)$	No	No
Depth-First Iterative-Deepening	$O(b^d)$	$O(d)$	Yes	Yes
Bi-directional	$O(b^{d/2})$	$O(b^{d/2})$	Yes	Yes
Hill-Climbing	$O(b^d)$	$O(1) - O(b^d)$	No	No
Best-First	$O(b^d)$	$O(b^d)$	Yes	No
A*	$O(b^d)$	$O(b^d)$	Yes	Yes
Iterative-Deepening-A*	$O(b^d)$	$O(d)$	Yes	Yes
Beam Search	$O(n^d)$	$O(n^d)$	No	No
Means-Ends	$O(b^d)$	$O(b^d)$	No	No
Generate and Test	$O(((bd!) / ((bd-d)!)) / 2)$	$O(d)$	Yes	No
Asynchronous Dynamic	$O(b^d)$	$O(1)$	Yes	Yes
Learning Real-Time A*	$O(b^d)$	$O(b^d)$	Yes	Yes
Real-Time A*	$O(b^d)$	$O(b^d)$	Yes	No

Table 1 – Search Comparison

Listed in Table 1 are the attributes of the algorithms that have been reviewed in this paper.

In Table 1, the 'Algorithm' column refers to the search algorithm in question. The 'Time' column is the order of complexity search time used by algorithm, expressed as a function. The 'Memory' column is the order of complexity memory requirements of algorithm, also expressed as a function. The 'Complete' column is a Boolean indicator of whether or not the search algorithm is exhaustive. This means that if a solution exists, it will be found. The 'Optimal' column is a Boolean indicator of whether or not the solution found will always be the optimal solution.

The variables used in the memory and time estimates are defined as follows:

b – Branching factor of search tree

d – Solution depth within search tree

n – Subset of b for which algorithm will actually process

It can be seen that the time estimates for all of the searches are similar. The three exceptions are the Bidirectional, Beam, and Generate and Test searches.

The reason that the Bidirectional search has a lesser time estimate is because it is simultaneously working from both ends of the problem looking for a common intermediate node. There are searches working from start to finish, as well as, finish to start. This causes the bidirectional search to be limited to only problems where the final solution is known in advance, severely limiting the scope of application for the algorithm.

The Beam search has a time estimate of $O(n^d)$, as opposed to the more common $O(b^d)$. This is because the Beam search is a modified A* search that examines on the best n branches at any node. This does speed up processing, but at the cost of assuming that a sub-optimal node will never need to be traveled to reach the goal state. If this is the case, the solution to the search will never be found.

The Generate and Test search has a time complexity equal to half of the permutation of b to d . This is because every ordered set of nodes that are generated is pulled from a pool of choices the size of the permutation of b to d . Only half of this pool is considered for statistical fairness.

It should also be noted that even though the brute-force searches listed at the top of Table 1 have similar run-time estimates as the heuristic algorithms listed at the bottom, the true average runtimes can vary greatly. With a well chosen heuristic function, the A*-style of algorithm can be empirically shown to find a solution faster on average than a brute-force algorithm that blindly chooses its search path.

The memory requirements of the search algorithms are more distributed than the time estimates. For the most part, a search algorithm will approach a problem breadth-first or depth-first.

The breadth-first approaches consume more memory, typically $O(b^d)$, as they must retain record of every node they expand in the search tree. However, they also tend to provide a complete and optimal solution, except for a few special cases.

The depth-first approaches usually only consume linear amounts of memory, $O(d)$, as they only need to remember the path down the current branch that they are searching. However, to prevent running into loops, they often have to have an arbitrary cut-off depth that can lead to incomplete searches. Of course, there are exceptions, such as the Depth-First Iterative-Deepening that provide average time and below average memory requirements for a complete and optimal search.

This is also a short-sighted approach that is used by the Hill-Climbing algorithm. This approach only knows where it is at, but has no memory of where it has been. This leads to a constant memory usage, $O(1)$. However, this search is often incomplete and suboptimal, and occasionally continues looping for infinity. The only

way to repair this searches shortcomings is by retaining a list of visited nodes, which moves the memory requirements from $O(1)$ to $O(b^d)$, while still retaining the incomplete and suboptimal properties.

Another peculiarity in terms of memory usage is the ADP search. This search used $O(1)$ memory, since every node contains a process and each process only needs to hold a small amount of data.

The complete and optimal properties of search algorithms are straighter forward to understand than their time and memory estimates. It can be said from Table 1 that an optimal search is always complete, but that a complete search does not always find the optimal solution. As discussed above, the choice of an incomplete search algorithm is not always a bad decision, just as the choice of a suboptimal solution is often suitable.

Obviously, no single search algorithm will suffice for all searches that must be performed by intelligent agents in a distributed system. The choice of search is highly problem-specific and should be approached as such.

4. Conclusion

This paper closely looks at various intelligent search algorithms. Time and space complexity of any algorithm are very important factors to distinguish different search algorithms. It should also be noted that different search problems have different suitable search algorithms as solution and there is no one algorithm that can solve all search problems. While making decision on which search algorithm is best to solve any given problem, complexity of the problem itself should be studied carefully. Distributed nature of the system could also be a deciding factor to select an algorithm. For more meaningful search, semantic search algorithms should be important means to reach goal state. Semantic ontology and related development is very important to the development of Semantic Search algorithms.

5. References

- [1] Alexander M. Meystel and James S. Albus, "Intelligent Systems", pp 182.
- [2] Alexander M. Meystel and James S. Albus, "Intelligent Systems", pp 526,527.
- [3] Korf, R.E. (1987). Search. *Encyclopedia of artificial intelligence* (Vol. 2, pp. 994). John Wiley & Sons: New York
- [4] Kumar, V. (1987). Search, depth first. *Encyclopedia of Artificial Intelligence* (Vol. 2, pp. 1004-1005). John Wiley & Sons: New York

- [5] *Abou-Assaleh*, T. (February 1999). Intelligent search algorithms. *Hello world! An magazine for computer science students*. Retrieved October 2003 from http://www.cosc.brocku.ca/~cspress>HelloWorld/1999/02-feb/search_algorithms.html
- [6] Schmidt, C. Artificial intelligence and search. Retrieved December 2003 from http://www.rci.rutgers.edu/~cfs/472_html/AI_SEARCH/GenTest1.html.
- [7] Deterministic search: informed search algorithms. Retrieved December 2003 from <http://www.owl.net.rice.edu/~comp440/lectures/lec3.6pp.pdf>.
- [8] (1987). Raphael, B. A*. *Encyclopedia of artificial intelligence* (Vol. 1, pp. 1). John Wiley & Sons: New York
- [9] Heyes-Jones, J. A* algorithm tutorial. Retrieved December 2003 from <http://www.geocities.com/jheyesjones/astar.html>.
- [10] (1987). Bisiani, R. Beam Search. *Encyclopedia of artificial intelligence* (Vol. 1, pp. 56-57). John Wiley & Sons: New York
- [11] (1987). Ernst, G. Means-Ends Analysis. *Encyclopedia of artificial intelligence* (Vol. 1, pp. 578-584). John Wiley & Sons: New York
- [12] Chong, R., & Phillips, J., & Wray, R. Forward and backward planning. *A survey of cognitive and agent architectures*. Retrieved December 2003 from <http://ai.eecs.umich.edu/cogarch0/common/prop/plan.htm>.
- [13] Parmar, A. Problems with means-ends analysis and strips planning format. *Some ideas about means-ends analysis*. Retrieved December 2003 from http://www-formal.stanford.edu/aarati/draft_papers/means_ends_analysis/node2.html.
- [14] Tower of Hanoi. Retrieved December 2003 from <http://www.mazeworks.com/hanoi/>.
- [15] Ishida, T., & Yokoo, M. (2000). Multiagent systems: a modern approach to distributed artificial intelligence. pp 166-196.
- [16] OWL semantic search. Retrieved Dec 2003 from <http://www.w3.org/TR/2003/CR-owl-guide-20030818/>
- [17] Resource Description Framework, Retrieved December 2003 from <http://www.w3.org/TR/rdf-syntax-grammar/>
- [18] Web Hunting: *Design of Intelligent Web Search Agent*, G. Michael Youngblood, ACM Crossroads, student magazine, May 1999, Retrieved December 2003 from <http://www.acm.org/crossroads/xrds5-4/webhunting.html>
- [19] Knowledge Query and Manipulation Language, KQML Retrieved December 2003 from <http://www.cs.umbc.edu/kqml/>
- [20] Joseph and Jennifer Bigus, “*Constructing intelligent agents using Java*”, Second Edition pp 65.