Design of Java Application Programmer's Interface (API) for Heuristic Tree Search Algorithms

Prepared for submittal to the Midwest Computer Conference

Patrick Clemins

February 7, 2000

Table of Contents

Heuristic Tree Searches Defined 1
Current Implementations 5
The API Exposed and Explained 6
Implementation 9
Use of the API 10
Real-World Applications 11
Bibliography 12

Heuristic Tree Searches Defined

Artificial intelligence (AI) has been an ongoing research area since the advent of computers. There are many different aspects to AI including speech synthesis, image recognition, and problem solving. There are also many different algorithms that can be used to solve these problems. Some are complex and attempt to use some kind of intelligence to solve the problem, while others search all possible states until the solution is reached. This set of all possible configurations of the problem is called the solution space. The solution space needs to be represented by some data structure, so that it can be searched in an organized fashion. A tree is one common way to represent the solution space of a problem.

A tree (Figure 1) is a hierarchical data structure. Each circle represents one state of the problem and is referred to as a node. The top node, the one from which all others are descended, is called the root (node 1). The children of a node are all nodes descended from it. The parent of a node is the node from which it is descended. For example, node 2 is a child of node 1 and node 1 is the parent of node 2. A leaf is any node without any children (node 2).

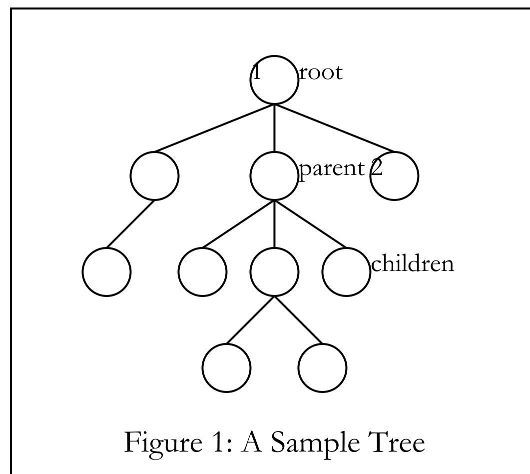

Figure 1: A Sample Tree

There are two different strategies for searching a solution space. One of these strategies is data-driven. A data-driven strategy starts at the root and transverses the branches of the tree until it finds the solution. The other is a goal-driven strategy that starts at the solution and then tries to find a path back to the starting point. All four algorithms considered here are data-driven strategies.

There are four different search algorithms that will be implemented in the API. These four algorithms are depth-first, breadth-first, best-first and the A* (A-star) algorithm. The A* algorithm is actually a best-first algorithm with some improvements to make it more efficient. These algorithms were chosen because they are the most commonly used tree search algorithms in current literature. The four algorithms are outlined below with examples.

The depth-first search starts at the root and examines the root's leftmost child to see if it is the solution. If it is not the solution, the search continues to examine that node's leftmost child. This continues until a leaf is encountered. A leaf is a node without any children. After a leaf is examined to see if it is the solution, the search goes back to the closest parent node with children that have not been examined yet. This continues until all nodes are searched or the solution is found. Figure 2 shows how a depth first search would progress through a sample tree.

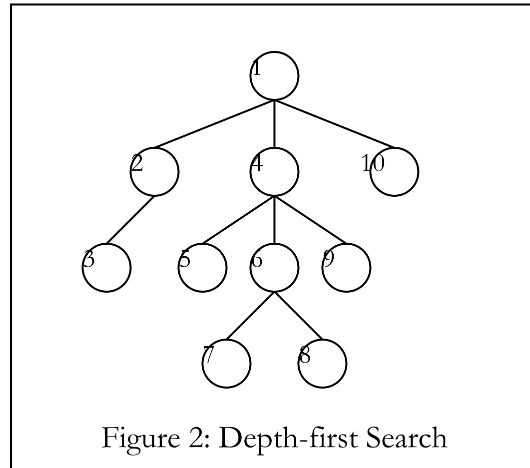

Figure 2: Depth-first Search

The breadth-first search starts at the root and examines its leftmost child to see if it is the solution. If not, the search continues to examine each child of the root until all of the children are examined. After all of the children are examined, the search examines each of the children's children. This continues until the

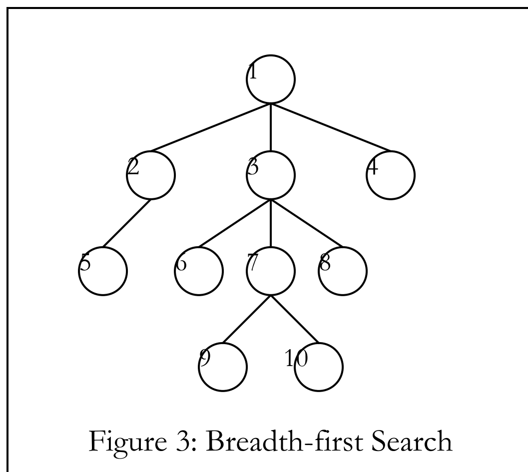

Figure 3: Breadth-first Search

solution is found or all nodes are searched. Figure 3 shows how a breadth-first search would progress through a sample tree.

The two searching algorithms discussed thus far use no intelligence to determine the best path to the solution. Instead, they follow a systematic pattern, searching the entire tree. For this reason, these algorithms are sometimes called brute-force algorithms. These algorithms are simple to implement. However, they typically perform a lot of comparisons before they find the solution. In contrast, heuristic algorithms try to choose a particular path of the tree to search along. The next two algorithms discussed use a heuristic to choose a particular path to follow. A heuristic is a formula used to approximate how close a given state is to the solution. They maintain a list of open nodes and a list of closed nodes. Open nodes are nodes that have not been expanded yet. Once a node is expanded and examined, it is moved to the closed list. The open list is usually ordered based on the value the heuristic function returns.

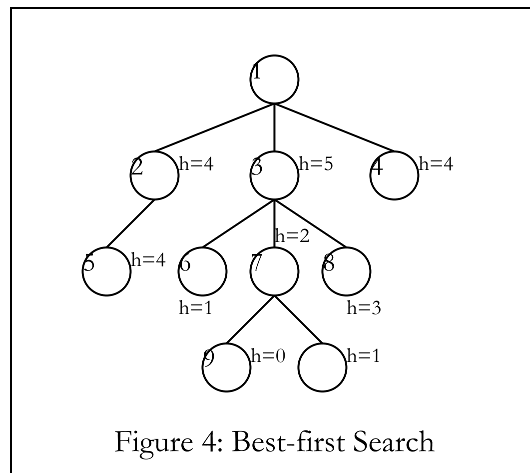

Figure 4: Best-first Search

The best-first search starts out by placing the root in the open list. As with the rest of the algorithms, once the solution is found, the search stops. With this search, if the solution has not been found and the open list is empty, then the search failed. For each iteration of the search, the node with the best ranking (based on the heuristic) is removed from the open list. In this example, the lower the heuristic value, the better the ranking. Then, the highest ranked node's children are examined to see if any of them is the solution. If not, the heuristic function is calculated for each child and the children are added to the open list if they are not already in one of the two

lists. Nodes are removed from the open list in this fashion until the list is empty or the solution is found. Notice the three nodes with a heuristic value of four near the top of the tree. There is no absolute rule governing which node should be searched first when they both have the same heuristic value, but a generally accepted tie-breaker is which ever node was added to the open list first. Figure 4 shows how a best-first search would transverse a sample tree. The number in the node represents the order in which they were examined, not the order in which they were expanded. For example, although node 8 is examined, it would never be expanded (i.e. taken off the open list) because node 7 has a lower heuristic value.

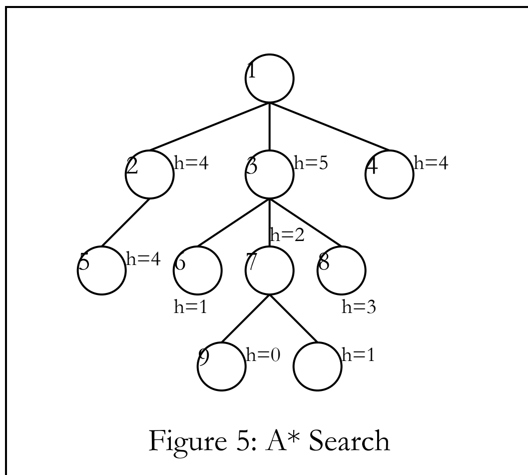

Figure 5: A* Search

The A* algorithm is a modified best-first search. The only change to the best-first search is the way the next node is picked from the open list. Instead of picking the next node based solely on the heuristic function, the value used to determine the next node is calculated by adding the heuristic function and the distance from the node to the root. For instance, in the example A* search in Figure 5, the second node would have a value of 5 (4 + 1) while its child (node 5) would have a value of 6 (4 + 2). So, even though both nodes return the same value from the heuristic function, node 2 is more desirable because it is higher up in the tree hierarchy. The A* algorithm is modified from the best-first search so that it can find not only the solution, but also the shortest path to the solution. That is why the distance from the root is factored into each node's assigned value.

In the small example tree used to demonstrate the algorithms, the best-first and A* algorithms follow the same path, but as the solution space gets larger, this would not be the

case. The difference in this example is the order in which the nodes are expanded. The best-first algorithm would expand node 5 and remove it from the open list. However, the A* would not expand node 5 because node 3 would rank equal to node 5 and win the usual tiebreaker of being added to the open list first.

Current Implementations

The above four algorithms have been implemented numerous times in almost every programming language in existence. Significant research efforts have been focused on different heuristic algorithms and how to implement them more efficiently. However, each implementation has always been application specific. No one has ever set out to create a common framework that can be used in different applications to ease development. Discussed below are two examples of what has been done in the past.

Mark Watson has a website dedicated to discussing a Java implementation of search algorithms. On his site, he has two applets that show how a depth-first search and a breadth-first search transverse a graph. He talks about creating a Java framework, but his framework consists of an abstract search class. This abstract class is then inherited to create the depth-first search and breadth-first search algorithm classes. However, the two classes he creates from the search class are application specific. He presents some good ideas on using an object-oriented implementation of the algorithms to create a framework, but he does not create one.

Kim Tracy and Peter Bouthoorn discuss how to implement tree search algorithm in an object-oriented fashion in their book, Object-Oriented: Artificial Intelligence Using C++. They explain and discuss the code for implementing all of the algorithms described above, but each example is written for a specific problem. No mention is made of the idea of

creating a generalized framework, but they do provide an interesting discussion on the efficiency of the implementations.

It should be noted that the goal of this project is not to create a new algorithm or even to improve on an existing algorithm, but instead to provide easy access to these algorithms. It is intended that people with very little knowledge about how these algorithms work will be able to use them to solve various AI problems.

The API Exposed and Explained

The API consists of various objects and interfaces to aid in the solving of problems that can be represented in a tree or graph data structure. Each search implemented in the API is represented by two methods, both named solve. The difference between the two methods is their parameters. These methods are static. That is, they are class methods and do not operate on a specific instance of the algorithm.

One solve method's parameters are the starting state of the problem and the goal state of the problem. It returns an array of puzzle states that show each step on the way from start to solution. These two parameters passed to the solve method must implement the Solvable interface. This interface consists of two functions, generateChildren and equal. These are required so that the algorithm can create the child nodes and determine whether the solution is met or not by each node. To use a heuristic search algorithm, the two parameters must meet the HSolvable interface that adds the heuristic method, Hfunc, to the original two methods of the Solvable interface.

The other solve method's parameters are a pjc.ai.tree.Tree and an object. This solve method searches the given tree for a node containing the given object. This method returns the same thing as the other solve method, an array of puzzle states that show each step on

the way from start to solution. This version of solve is added to give programmers more control over how the solution space is constructed.

There are two additional elements in the API. These are the tree object and the node object. Although the programmer does not need to use these objects directly, they are used by the search algorithms to construct, manipulate and transverse the tree. They are also available for those who might wish to add an algorithm to the API. The entire API will be packaged in the pjc.ai.tree package. Below is the entire API with all of its methods explained.

<u>Solvable Interface</u>

Object[ ] generateChildren ( )
    Generates an array of possible next states for the object

boolean equals (Object o)
    Returns true if o is logically equal to the object

<u>HSolvable Interface</u> (sub-interface of Solvable)

Object[ ] generateChildren ( )
    Generates an array of possible next states for the object

boolean equals (Object o)
    Returns true if o is logically equal to the object

int Hfunc (Object o)
    Returns an integer specifying how logically close the object is to o

<u>BreadthFirstSearch Object</u>

static Object[ ] solve (Solvable start, Solvable goal)
    Returns the solution path (in order, start is array element 0), using a breadth-first tree search, from start to goal

static Object[ ] solve (Tree t, Object o)
    Returns the solution path from the root of tree t to the node which contains object o

DepthFirstSearch Object

static Object[ ] solve (Solvable start, Solvable goal)
Returns the solution path (in order, start is array element 0), using a depth-first tree search, from start to goal

static Object[ ] solve (Tree t, Object o)
Returns the solution path from the root of tree t to the node which contains object o

BestFirstSearch Object

static Object[ ] solve (HSolvable start, HSolvable goal)
Returns the solution path (in order, start is array element 0), using a best-first tree search, from start to goal

static Object[ ] solve (Tree t, Object o)
Returns the solution path from the root of tree t to the node which contains object o

AStarSearch Object

static Object[ ] solve (HSolvable start, HSolvable goal)
Returns the solution path (in order, start is array element 0), using the A* tree search, from start to goal

static Object[ ] solve (Tree t, Object o)
Returns the solution path from the root of tree t to the node which contains object o

Tree Object

void Tree ( )
Null Constructor. Creates a Tree with null root

void Tree (Node n)
Constructor. Creates a new Tree with root n

void addNode (Node n, Node p)
Adds node n to the tree with p as its parent

Node getRoot ( )
Returns the root of the tree

void removeNode (Node n)
Removes node n from the tree

void setRoot (Node n)
Sets the root of the tree

Node Object

void Node ( )
Null Constructor. Creates a new node with null contents

void Node (Object o)
Constructor. Creates a new node with contents o

void addChildren (Node[ ] n)
Add the nodes in n to the list of children for the node

Node [ ] getChildren ( )
Returns an array of all of the children of the node

Object getContents ( )
Returns the contents of the node

Node getParent ( )
Returns the parent of the node so that transversal up the tree is possible

void setChildren (Node[ ] n)
Sets the children of the node to n. This removes all children previously assigned to the node

void setContents (Object o)
Sets the contents of the node equal to o

Implementation

The search algorithms will be implemented as discussed in the first section of the report. A java.util.Hashtable will be used to maintain the open and closed lists for the heuristic searches. A hash table will be used so that a value can be associated with each node in the hash table. This value is necessary for the heuristic algorithms that require the nodes to be picked based on their heuristic value. The pjc.ai.tree.Tree object will be used by the search objects to structure the solution space. The trees will be constructed of pjc.ai.tree.Node objects so that the API as described above can be used to manipulate the solution space and solve the problem.

The tree object will be constructed using a single node as the root. The rest of the tree will be linked to the root within the nodes themselves. So, the only property of the tree

will be the root node. The addNode and removeNode methods will actually just serve as wrappers for node methods to accomplish the appropriate action.

The node object will be the most complicated object in the API. Its properties will consist of an object that will hold the contents of the node. There will also be a single node variable that will store the parent of the node. Finally, an array of nodes will hold all of the children nodes of the node.

Use of the API

The API can be used in many different ways. Some of these are discussed below. The API could be used to solve puzzles in different ways and then determine which algorithm is the most efficient. The way to do this would be to define a puzzle object that implements the HSolvable or Solvable interface. Then, the programmer would select which search to use and simply call the solve method of that search object. The programmer would have to define the start and end puzzle states once, but then could rapidly call the search methods to solve the problem in different ways.

The other way the API could be used is to define another search object that implements a different tree search algorithm. The programmer could use the tree and node objects to construct a solution space and then write the algorithm using the API methods to manipulate and search the tree.

One last way the API could be used is to search a predefined tree. Notice that there are two solve methods for each search. The search method variant that takes a tree and an object as parameters is designed for those programmers who wish to construct their own tree and then search it. This method of use allows the programmer to control how the solution space is created instead of using the API to create the solution space.

Real-World Applications

This API has many applications for use. One use for this API is education. This API could be a good learning tool as students can easily compare real-world examples of which algorithms work better under what conditions. All the students would have to do is describe the problem in terms of the Solvable interface and the check the system time before and after the solve method is called to determine the total search time. Also, challenging students to expand the API would be a good project in a computer science or computer engineering classes.

Another use is Java-based gaming. Simple AI algorithms are used often in gaming for computation, such as determining the next move in games like chess or checkers. These algorithms are also used for character path finding. Although Java is not used for complex three-dimensional graphics because of speed issues, many websites have multiplayer Java-enabled games. Instead of writing customized AI code for different games, a programmer could simply describe the game in terms of the Solvable interface and then use this API for computer players. As graphic accelerators become more powerful and move the graphic processing load off of the CPU, game developers have been able to add more complex physics and/or AI to games. This API can make the addition of these features easier.

The shear amount of documentation, written and electronic, on heuristic tree searches demonstrates a need for a quick way to implement these algorithms. This API could be used for any of the various heuristic search demonstration web sites or for authors who wish to concentrate more on describing the problem than describing the algorithms used to solve the problem. One advantage of object-oriented programming is how it tends to lead to reusable code. This API attempts to exploit this fact by implementing a generalized form of the search algorithm that can be incorporated into various programs.

Bibliography

8 puzzle. Geert-Jan van Opdorp. <http://www.aie.nl/~geert/java/public/EightPuzzle.html>.

Artificial Intelligence Search Techniques in Java. Mark Watson. 1998 <http://www.markwatson.com/opencontent/aisearch>.

Bolc, Leonard and Jerzy Cytowski. *Search Methods for Artificial Intelligence*. London: Academic Press, 1992.

Java Game Programming Resources and Source Code. Altnet Inc. <http://www.altnetinc.com/training/gdc99/source.htm>.

Nilsson, Nils J. *Problem-Solving Methods in Artificial Intelligence*. New York: McGraw-Hill, 1971.

Pearl, Judea. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Reading, MA: Addison-Wesley, 1984.

Tracy, Kim W. and Peter Bouthoorn. *Object-Oriented Artificial Intelligence Using C++*. New York: Computer Science Press, 1997.

Tzeng, Chun-Hung. *A Theory of Heuristic Information in Game-Tree Search*. Berlin: Springer-Verlag, 1988.