Artificial Intelligence: Navigating Polygonal Obstacles Using Searching Techniques

Lashanda Dukes (ECSU), Justin Deloatch (ECSU)
Ronald E. McNair Post baccalaureate Achievement Program - Elizabeth City State University
1704 Weeksville Road Elizabeth City, North Carolina 27909

Abstract - **The project uses artificial intelligence searching techniques to find a path around polygonal obstacles on a plane. The solution is based on both non-informed and informed algorithms. The algorithms are compared and contrasted. Each of these algorithms will work on the problem represented in terms of states and transitions between them. The algorithms then find a path to a goal state by choosing one segment at a time. Java programming will be used to implement the algorithms and present the solution in a graphical user interface.**

I. Introduction

What is Artificial Intelligence?- Intelligence is the capacity to learn and solve problems [1, 2]. Intelligence is also the ability to solve novel problems, to act rationally, to act like humans, and to acquire knowledge, learn from experience. Modern Artificial Intelligence models how ideal agents should act [1]. Artificial Intelligence is the science and engineering of making intelligent machines. It is related to the similar task of using computers to understand human intelligence, but Artificial Intelligence does not have to confine itself to methods that are biologically observable [3]. There are several successes in Artificial Intelligence such as: computer vision, robotics, natural language processing, expert systems, speech understanding systems, planning and scheduling systems, learning systems, and games. Computer vision has the capability of face recognition. In Robotics, vehicles have become autonomous. In natural language processing, machines are capable of machine translation. Expert systems can carry out medical diagnosis in a narrow domain. Speech understanding systems can recognize a thousand words in continuous speech. Planning and scheduling systems were employed in scheduling experiments in the Hubble Telescope. Game programs can play up to the world champion levels in chess. Artificial Intelligence has Logics of Knowledge: automated theorem provers and formal verification. Automated Theorem Provers encode a problem in logic and they try to solve it in the theorem. Formal verification involves the usage of huge circuits and gates and the user can try to verify if it is going to do what you want it to do [4].

Artificial Intelligence was founded after World War II, after several individuals began to work on intelligent machines. It is cited that the English mathematician, Alan Turing, was the first to work with Artificial Intelligence. In 1950, he wrote an article titled “Computing Machinery and Intelligence” which stated the vision of Artificial Intelligence. Within the article, Turing discussed heuristics, game playing, and learning from experience. Turing also came up with the idea of the Turing Test in 1950. The Turing Test is the process of a computer being interrogated by a human via teletype. To pass the test, the human must not be able to tell if there is a computer or human at the other end, otherwise it fails the test. Another form of the Turing Test is the Chinese Room Argument. The Chinese Room Argument is when a person is in a room and doesn’t know Chinese but has a huge volume of Chinese literature. A translator sends in literature in Chinese and the person in the room outputs the English version of what is in the text. Although the person in the room does not know Chinese, the translator assumes that the person in the room knows Chinese [4]. In 1956, at a conference in Dartmouth, the term Artificial Intelligence was adopted. In the 1990s advances in Artificial Intelligence came about such as: machine learning and data mining, intelligent tutoring, case-based reasoning, multi-agent planning and scheduling, uncertain reasoning, natural language understanding and translation, vision, virtual reality, games, and other topics [4].

Branches of Artificial Intelligence- Artificial intelligence has several branches, although some of them have yet to be identified. They include: consumer marketing, identification technology, intrusion detection, predicting the stock market, machine translation logical Artificial Intelligence, search, pattern recognition, representation, inference, common sense knowledge and reasoning, learning from experience, planning, epistemology, ontology, heuristics, and genetic programming,. Consumer marketing is when a consumer uses a credit card, ATM card, or store card while shopping and the consumer’s information is recorded digitally. Most companies gather information weekly and search for changes in consumer behavior, hot areas in marketing, and tracking responses to new products. Identification technologies are in the process of using ID cards and biometric identification. ID cards can be ATM cards, debit cards, or credit cards. Some forms of biometric identification are using your face, eyes, fingerprints, or voice pattern as a signature for identification. Intrusion detection is a system of learning the signature of an authorized user by recording the user’s commands. Predicting the stock market is a process of using learning algorithms to learn a predictive model from historical data. Machine translation is the use of automated translation to translate words and their meanings in other languages for people who do not share a

common language [1]. Logical Artificial Intelligence is what a program already knows about the world by using a representation of mathematical logical language. Search is the process of examining a large number of possibilities. Pattern recognition is trying to find a pattern in an observation. Representation is facts being represented in a type of form. Inference is drawing a conclusion of a fact. Common sense knowledge and reasoning is the area in which Artificial Intelligence is farthest from human-level, in spite the fact that it has been an active research area since the 1950s [3]. Learning from experience is based on the ability to represent information. Planning is a strategy to achieve a goal or a sequence of actions. Epistemology is the study of different kinds of knowledge's in order to solve problems that occur in the world. Ontology is the study of things that exist. Heuristics is a way of trying to discover an idea imbedded into a program. Genetic programming is a technique for getting programs to solve a task by mating random Lisp programs and selecting fittest in millions of generations [3].

Applications of Artificial Intelligence

Artificial Intelligence also consists of several applications, such as: game playing, speech recognition, understanding natural language, computer vision, expert systems, and heuristic classification. Game playing is a process of using machines to play games to beat a world champion, playing over 200 million positions per second. Speech recognition is a technique for computers to use speech to input information into the computer instead of using a keyboard. Understanding natural language is the computer understanding what text is input into the computer is about. Computer vision is used for three-dimensional information that is not just a set of two-dimensional views. Expert systems are processes of having a computer expert embody their knowledge in a computer program for carrying tasks. Heuristic classification is an expert system that puts information into fixed categories using several information sources [1].

II. Review of Literature

A. State Space Search

To solve a problem using Uniformed Search, the problem is represented in terms of: State space, Start state, Goal states and Operators [1, 4].

States: The state space is the search agent's model of the world and is usually a set of discrete states. For example in driving, the states in the model could be towns/cities. The start state is where the agent searching begins. The goal state(s) are defined as desirable state for an agent. There may be many states which satisfy the goal. For example drive to a town with a ski-resort or just one specific town. A goal is a desirable state for an agent. There can be many states or one state that satisfies the goal state [1].

Operators: These are legal actions which the agent can take to move from one state to another.

The entire problem can be formulated as a tuple $[S, s, O, G]$ where

- S is the (complicity specified) set of states
- s is the start state
- O is the set of state transition operators
- G is the set of goal states

Search solution consists of [1]:

- A unique goal sate, G.
- A sequence of operators which transform S into a goal state G (This is the sequence of actions the agent would take to maximize the success function).
- For now we are interested in any path from S to G.

B. A State Space and a Search Tree are different

A Search Tree represents how the algorithm explores the state space as it solves a search problem. Different search trees can be used for the same problem. Search trees continue to grow until the goal is found or the algorithm runs out of states to explore [1].

C. Types of Search Algorithms

The two kinds of search are non-informed or informed search based [1]. Blind or uniformed searches do not follow any specific information for a problem. An example of an uninformed search would be the 8-puzzle (sliding-tile) problem where the numbers will slide only if the number is next to the blank; no other information on how far the puzzle is from the goal. The puzzle starts off with the numbers arrange out of order. The goal is to get the numbers in the correct order by using the blank space. Examples of a blind or uninformed search are the breadth-first and depth-first searches. Informed search algorithms use problem specific information to speed up the search process. In the 8-puzzle problem, each search evaluates how far each number is from the goal configuration.

D. Uninformed/Blind Search

An Uninformed search algorithm does not use any specific problem domain information. It searches by exploring states and neighboring states. The next state to choose from neighboring states is either the first or last to be encountered. It is like searching for a route on a map without using any information about direction. It is inefficient in time and computing memory but its power is in its generality. It can be applied to any search problem.

E. Algorithm for Breadth-First Search

Breadth-First means that the nodes are explored level by level. In other words, node's neighbors are all explored before proceeding further down the tree. The algorithm below performs a Breadth-First Search of a search space. It dynamically generates a search tree as it expands the nodes. It doesn't use any specific problem domain information [1].

```
Initialize: Let Q = {s}
While Q is not empty
        Pull Q1, the first element in Q
        if Q1 is a goal
                report (success) and quit
        else
                Child_nodes = expand(Q1)  (That is next states from Q1)
                eliminate child_nodes which represent loops
                put remaining child_nodes at the back of Q
        end
Continue
```

•Open nodes are stored in a queue Q of nodes.

•Convention is that expanded nodes are ordered "left to right".

F. *Algorithm for Depth-First Search*

Depth-First means that a node's branches are fully explored before other branches. The algorithm below performs a Depth-First Search of the search space. It dynamically generates a search tree as it expands the nodes. It does not use any specific problem domain information [1].

```
Initialize: Let Q = {s}
While Q is not empty
        Pull Q1, the first element in Q
        if Q1 is a goal
                report (success) and quit
        else
                Child_nodes = expand(Q1) (That is next states from Q1)
                eliminate child_nodes which represent loops
                put remaining child_nodes at the front of Q
        end
Continue
```

•Open nodes are stored in a stack Q of nodes.

•Convention is that expanded nodes are ordered "left to right".

G. *Pseudocode for Uninformed Search*

The algorithm below implements the Breadth-First and Depth-First search algorithms [7]. The queue or stack of states is implemented using the data structure OPEN. To aid in the elimination of child nodes which represent loops, another data structure CLOSED is used.

Algorithm 1:

```
Step 1: Initialize
Set OPEN = {s}, CLOSED = {}.

Step 2: Fail
If OPEN = {}, Terminate with failure.

Step 3: Select
Select a state, n, from OPEN and save n in
CLOSED.

Step 4: Terminate
If n is in G, terminate with success.

Step 5: Expand
Generate the successors of n using operators
O.
For each successor, m, insert m in OPEN only
if m is not in [OPEN or CLOSED].

Step 6: Loop
Go to Step 2.
```

H. *Informed State Space Search*

Each Informed search algorithm uses an evaluation function that gives a measure about which node to expand [2]. They are also referred to as Best-First Search algorithms. The different search Strategies are as below.

- Uniform-cost search: Minimizes the path cost so far.
- Greedy search: Minimize the estimated path cost.
- A* Algorithm: Minimize the total path cost.

A* is complete, optimal, and optimally efficient among all optimal search algorithms, but A* usually runs out of space long before it runs out of time. The Uniform-cost search algorithm is slow since it does not utilize estimates on how far states are from the goal. The Greedy search algorithm is faster than the Uniform-cost but is not complete (does not guarantee to find the solution).

Using heuristic is a "rule-of-thumb based on domain-dependent knowledge to help solve a problem. You would use a heuristic function of a state where: h(node) = estimated cost of cheapest path from the state for that node to a goal state G. This helps with the search because we can use knowledge (in the form of h(node)) to reduce search time and generally, will explore more promising nodes before less promising ones. When combining heuristic and path cost, it can be done with fcost(node) = g(s to node) + h(node to G) where fcost(N) = estimated cost from s to G via N, g(N) = path cost from s to N(exact) and h(N) = estimate of path cost from N to G.

If a heuristic is "admissible" and accurate it will quickly zero in on the goal but in general it is difficult to come up with a good heuristic [2]. The estimate of the distance is called a heuristic and the closer the heuristic is to the real (unknown) cost, the more effective it will be.

I. *Algorithm for the A*Search Algorithm*

Initialize : Let $Q = \{s\}$
While Q is not empty
  Pull $Q1$, the first element in Q (the "best" state from Q)
  if $Q1$ is a goal
  report(success) and quit
  else
    Child_nodes = expand($Q1$) (That is next states from $Q1$)
    eliminate child_nodes which represent loops
    put remaining child_nodes in Q
    Sort Q according to ucost = pathcost(s to node) + h(node)
  End
Continue

J. *Pseudocode for A* Search*

Algorithm 2:

```
Step 1: Initialize
Set OPEN = {s}, CLOSED = {}, g(s) = 0, f(s) = h(s).

Step 2: Fail
If OPEN = {}, Terminate with failure.

Step 3: Select
Select the minimum cost state, n, from OPEN. Save n
in CLOSED.

Step 4: Terminate
If n ∈ G, terminate with success, and return f(n)

Step 5: Expand
For each successor, m, of n
If m ∉ [OPEN ∪ CLOSED]
Set g(m) = g(n) + c(n,m)
Set f(m) = g(m) + h(m)
Insert m in OPEN
If m ∈ [OPEN ∪ CLOSED]
Set g(m) = min {g(m), g(n) + C(n,m)}
Set f(m) = g(m) + h(m)
If f(m) has decreased and m ∈ CLOSED, move m to OPEN

Step 6: Loop
Go to step 2
```

Note: Suppose the problem has only heuristic information but no path costs, the algorithm then uses the heuristic as the path cost.

K. *The 8-Puzzle Problem*

The 8-puzzle is a sliding-tile puzzle where tiles slide if they are next to a blank tile.

Start state

1	3	5
2	7	4
6		8

Goal State

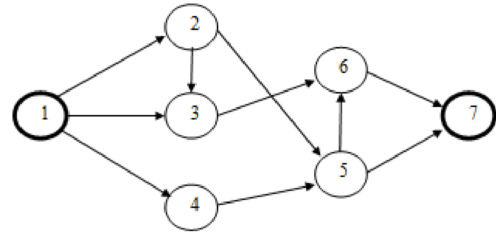
1	2	3
4	5	6
7	8	

The formulation of the search state space is as follows [4]:

- State description (S): Locate of each of the eight titles (and the blank).
- Start state (s): The starting configuration (given)
- Operators (0): Four operators, for moving the blank left, right, up or down
- Goals (G): One or more goal configuration (given)
- Heuristic function (h): The number of tiles in the wrong position, OR the sum of the distances of the tiles from their goal position.

L. *Application of Algorithms to an example*

Consider the graph below with states and transitions between them. The start state is 1 and the goal state is 7. Using Algorithm 1, with OPEN being a queue and stack, gives results that follow.



Breadth-First Search

OPEN (Queue)	CLOSED
1	1
2 3 4	1 2
3 4 5	1 2 3
4 5 6	1 2 3 4
5 6	1 2 3 4 5
6 7	1 2 3 4 5 6
7	1 2 3 4 5 6 7

Depth-First Search

OPEN (Stack)	CLOSED
1	1
2 3 4	1 4
2 3 5	1 4 5
2 3 6 7	1 4 5 7

The path from the start state to the goal can be obtained by keeping track of a predecessor state for each state added to OPEN. The results below tracks predecessor states.

Breadth-First Search

OPEN (Queue)	CLOSED
1(-)	1(-)
2(1) 3(1) 4(1)	1(-) 2(1)
3(1) 4(1) 5(2)	1(-) 2(1) 3(1)
4(1) 5(2) 6(3)	1(-) 2(1) 3(1) 4(1)
5(2) 6(3)	1(-) 2(1) 3(1) 4(1) 5(2)
6(3) 7(5)	1(-) 2(1) 3(1) 4(1) 5(2) 6(3)
7(5)	1(-) 2(1) 3(1) 4(1) 5(2) 6(3) 7(5)

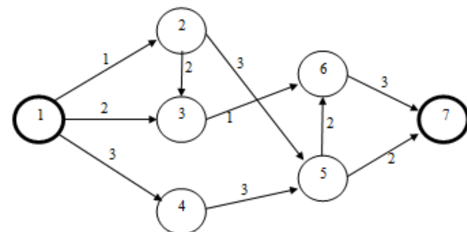
Path to goal: 1-2-5-7

Depth-First Search

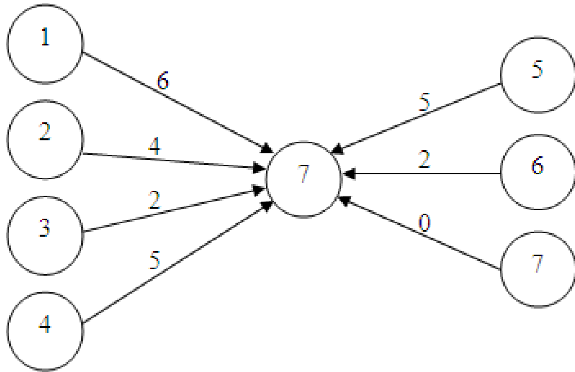
OPEN (Stack)	CLOSED
1(-)	1(-)
2(1) 3(1) 4(1)	1(-) 4(1)
2(1) 3(1) 5(4)	1(-) 4(1) 5(4)
2(1) 3(1) 6(5) 7(5)	1(-) 4(1) 5(4) 7(5)

Path to goal: 1-4-5-7

Consider the graph below with states and transitions between them and heuristic values for each state as indicated on each state.



Admissible Heuristics: A heuristic is admissible if its value is less that the real cost of the state to the goal. Consider the graph above with the costs given. Examples of admissible heuristics are as below. The implementation uses OPEN as a priority queue. The state with the lowest heuristic value is chosen first from OPEN.



Using Algorithm 2, gives the results below. These results exhibit a different search path from uninformed search paths. Its cost is shorter or equals the others.

Best-First Search

	OPEN (Priority Queue)	CLOSED
State	1	1
Predecessor	–	–
Heuristic	6	6
State	3 2 4	1 3
Predecessor	1 1 1	– 1
Heuristic	2 4 5	6 2
State	6 2 4	1 3 6
Predecessor	3 1 1	– 1 3
Heuristic	2 4 5	6 2 2
State	7 2 4	1 3 6 7
Predecessor	6 1 1	– 1 3 6
Heuristic	0 4 5	6 2 2 0

Path to goal: 1-3-6-7

III. Statement of the Problem

Consider the problem of finding the shortest path between two points on a plane that has convex polygonal obstacles as shown in the Figure below. This is an idealization of the problem that a robot has to solve to navigate its way around a crowded environment [2].

a) Suppose the state space consists of all positions in the plane. How many states are there? How many paths are there to the goal?

b) Explain briefly why the shortest path from one polygon vertex to any other in the scene must consist of straight-line segments joining some of the vertices of the polygons. Define a good state space now. How large is this state space?

c) Define the necessary functions to implement the search problem, including a successor function that takes a vertex as input and returns the set of vertices that can be reached in a straight line from the given vertex. (Do not forget the neighbors on the same polygon.) Use the straight-line distance for the heuristic function.

d) Apply one or more of the algorithms to solve a range of problems in the domain, and comment on their performance.

IV. Methodology

To answer the questions above, the obstacles, start and end points should be transformed into a state search space.

The state space consists of corners, starting points, and endpoints that are encoded into x, y positions. The number of state spaces is dictated by the number of corners in the obstacles. The path are drawn from one corner to the next, avoiding the interior of the obstacles.

The shortest path is to go straight to the corner instead of going around it. The shortest path consists of the segments joining corners. The state space should consist of the x, y values of the corner and the position of the goal. This state space of eight polygons has thirty-five states.

The state space implemented as `Coordinate` class includes the coordinate position, the goal, and predecessor. This class includes methods to access, the predecessor, test equality, compute the distance from the goal, and to test if the state itself is the goal. A class called `CoordinateSuccessorFunction` has a method called getSuccessors. This method will take the current state as input and return a list of its successors as an arraylist data structure. Its header is as follows `public ArrayList getSuccessors(Coordinate currentState).` An example of how it works is how to go from the start state to its successors. The if-statement below accomplishes it:

```
if(currentState.equals(cord0))
{
  //Successors of cord0.
  list.add(cord1); list.add(cord2); list.add(cord6);
  list.add(cord7);
}
```

Other states are treated in a similar fashion. The heuristic function to speed up the search process is in the Coordinate state class and is implemented as:

```
        public int distFromGoal()
        {
          int distance = 0;
         distance+=Math.sqrt(Math.pow(goal.x-
position.x,2)+Math.pow(goal.y
         position.y, 2));
         return distance;
       }
```

A. *Java Implementation of a Search Algorithm*

Assume that the state class is called State. The following Java code implements the uniform search algorithm.

```
import java.util.ArrayList;
import java.util.Stack;
import java.util.State;
//Search algorithm

State s = new State();// Initialize s with the
starting state.
```

```
Stack open = new Stack(); //1: Initializes open and
closed.

ArrayList closed = new ArrayList();


while (true)
{
 //2: Failure
 if (open.empty())
 {
   System.out.println ("Failed to Search");
   return;
 }
  //3: Select a state from open and put it in
closed.
  State state = (state) open.pop();
  closed.add(state);

  //4: Terminate with failure.
  if(state.isGoal()) //State is a goal
  {
    System.out.println("Success: Goal found");
    System.out.println(state); //calls toString()
  }

  //5: Expand a node.
  ArrayList successors = Successor
class.getSuccessors(state);

  for (int i=0; i<successors.size(); i++)
  {
    State succState = (state)successors.get(i);
    if (/*succState not in open or closed*/)
      open.push(succState);
  }
 }

 private boolean inStack(Stack s, Object o)
 {
   int i=0;
   while(i<s.size())
   {
     Object so = s.get(i);
     if(so.compareTo(o)==o)
       return true;
       i++;
   }
   return false;
 }

 class SuccessorClass
 {
  //Generating successor states.
   public static ArrayList getSuccessors(State
state)
    {
      ArrayList list = new ArrayList();

      if(state is 1)
      {
        list.add(newState(2));
        list.add(newState(3));
        list.add(newState(4));
      }

      else if(state is 2)
        list.add(newState(3));
      return list;
    }//get successors

 }//successor class
```

B. Implementation of the State Space Search Algorithm in Java

The following code is the implementation of the state space search algorithm. The program uses each step in the algorithm to reach the goal coordinates. The program also traces the path used to get to the goal. The program can find the goal by using a stack, queue, or priority queue. Different paths can be generated by each of the different data structures above.

```
import java.util.ArrayList;
import java.util.Stack;
import java.awt.Point;
import java.util.Vector;
import java.util.PriorityQueue;

class EmptyQueueException extends RuntimeException
{
       public EmptyQueueException()  { }
}

class Queue extends Vector
{
    public Object enqueue (Object element)
        {
                addElement (element);
                return element;
        }

    public Object dequeue ()
        {
                int len = size();
                if (len == 0)
                        throw new
EmptyQueueException();
                Object obj = elementAt(0);
                removeElementAt (0);
                return obj;
        }

        public boolean empty()
        {
                return isEmpty();
        }

        public Object peek()
        {
                int len = size();
                if (len == 0)
                        throw new
EmptyQueueException();
                return lastElement();
        }

        public int search(Object o)
        {
                return indexOf(o);
        }
}
class Coordinate implements Comparable
{
  private Point position;
  private Point goal = new Point(164,79);
  private Coordinate predecessor;

  public Coordinate(int x, int y)
  {
    position = new Point(x,y);
    predecessor = null;
  }

  public Coordinate getPredecessor()
  {
    return predecessor;
```

```
  }

  public void setPredecessor(Coordinate pre)
  {
    predecessor = pre;
  }

  public boolean equals(Object obj)
  {
    Coordinate cord = (Coordinate) obj;
    return position.equals(cord.position);
  }

  public String toString()
  {
    String string = "";
    string = String.format("[%d, %d]", position.x,
position.y);
    return string;
  }

  public boolean isGoal()
  {
    return (position.equals(goal));
  }

  public int distFromGoal()
  {
    int distance = 0;
    distance += Math.sqrt(Math.pow(goal.x -
position.x, 2)+Math.pow(goal.y -
    position.y,2));
    return distance;
  }

  public int compareTo(Object o)
  {
    double dist1 = distFromGoal();
    Coordinate compCoordinate = (Coordinate)o;
    double dist2 = compCoordinate.distFromGoal();

    if (dist1 > dist2) return 1;
    else if (dist1 < dist2) return -1;
    return 0;
  }
}

class CoordinateSuccessorFunction
{
   private Coordinate cord0 = new Coordinate(16,
22);  //start
   private Coordinate cord1 = new Coordinate(25,
28);
   private Coordinate cord2 = new Coordinate(25, 7);
   private Coordinate cord3 = new Coordinate(85,
28);
   private Coordinate cord4 = new Coordinate(85, 7);
   private Coordinate cord5 = new Coordinate(42,
83);
   private Coordinate cord6 = new Coordinate(24,
67);
   private Coordinate cord7 = new Coordinate(25,
45);
   private Coordinate cord8 = new Coordinate(45,
39);
   private Coordinate cord9 = new Coordinate(55,
62);
   private Coordinate cord10 = new Coordinate(69,
65);
   private Coordinate cord11 = new Coordinate(60,
34);
   private Coordinate cord12 = new Coordinate(77,
34);
   private Coordinate cord13 = new Coordinate(77,
81);
   private Coordinate cord14 = new Coordinate(91,
83);
   private Coordinate cord15 = new Coordinate(103,
70);
   private Coordinate cord16 = new Coordinate(80,
58);
   private Coordinate cord17 = new Coordinate(90,
45);
   private Coordinate cord18 = new Coordinate(95,
12);
   private Coordinate cord19 = new Coordinate(110,
26);
   private Coordinate cord20 = new Coordinate(103,
80);
   private Coordinate cord21 = new Coordinate(138,
80);
   private Coordinate cord22 = new Coordinate(103,
35);
   private Coordinate cord23 = new Coordinate(138,
35);
   private Coordinate cord24 = new Coordinate(125,
22);
   private Coordinate cord25 = new Coordinate(140,
29);
   private Coordinate cord26 = new Coordinate(153,
22);
   private Coordinate cord27 = new Coordinate(153,
6);
   private Coordinate cord28 = new Coordinate(140,
3);
   private Coordinate cord29 = new Coordinate(125,
6);
   private Coordinate cord30 = new Coordinate(145,
74);
   private Coordinate cord31 = new Coordinate(154,
80);
   private Coordinate cord32 = new Coordinate(159,
72);
   private Coordinate cord33 = new Coordinate(155,
29);
   private Coordinate cord34 = new Coordinate(164,
79); //goal

   public ArrayList getSuccessors(Coordinate
currentState)
   {
       ArrayList list = new ArrayList();

          if(currentState.equals(cord0))
          {
            //Successors of cord0.
            list.add(cord1);  list.add(cord2);
list.add(cord6);
            list.add(cord7);
          }

          else if(currentState.equals(cord1))
          {
            //Successors of cord1.
            list.add(cord0);  list.add(cord2);
list.add(cord3);
            list.add(cord6);  list.add(cord7);
list.add(cord8);
               list.add(cord11);  list.add(cord12);
           }

           else if(currentState.equals(cord2))
           {
             //Successors of cord2.
               list.add(cord0);  list.add(cord1);
list.add(cord4);
               list.add(cord6);  list.add(cord7);
list.add(cord28);
               list.add(cord29);
            }
```

```
else if(currentState.equals(cord3))
{
  //Successors of cord3.
   list.add(cord1);  list.add(cord4);
list.add(cord10);
   list.add(cord11); list.add(cord12);
list.add(cord16);
   list.add(cord17); list.add(cord18);
 }

else if(currentState.equals(cord4))
{
  //Successors of cord4.
   list.add(cord2);  list.add(cord3);
list.add(cord17);
   list.add(cord18); list.add(cord24);
list.add(cord25);
   list.add(cord29);
 }

 else if(currentState.equals(cord5))
 {
   //Successors of cord5.
    list.add(cord6);  list.add(cord9);
list.add(cord10);
    list.add(cord13); list.add(cord14);
list.add(cord16);
  }

   else if(currentState.equals(cord6))
   {
    //Successors of cord6.
     list.add(cord0);  list.add(cord1);
list.add(cord2);
     list.add(cord5);  list.add(cord7);
    }

    else if(currentState.equals(cord7))
   {
     //Successors of cord7.
      list.add(cord0);  list.add(cord1);
list.add(cord2);
      list.add(cord6);  list.add(cord8);
list.add(cord11);
    }

  else if(currentState.equals(cord8))
     {
    //Successors of cord8.
       list.add(cord1);
list.add(cord7);  list.add(cord9);
       list.add(cord10);
list.add(cord11); list.add(cord13);
      }

      else
if(currentState.equals(cord9))
      {
        //Successors of cord9.
        list.add(cord5);
list.add(cord8);  list.add(cord10);
        list.add(cord11);
list.add(cord13);
       }

      else
if(currentState.equals(cord10))
      {
        //Successors of cord10.
        list.add(cord3);
list.add(cord5);  list.add(cord8);
        list.add(cord9);
list.add(cord11); list.add(cord12);
        list.add(cord13);
list.add(cord16); list.add(cord17);
                    list.add(cord18);
list.add(cord22);
                   }

                  else
if(currentState.equals(cord11))
                  {
                    //Successors of cord11.
                    list.add(cord1);
list.add(cord3);  list.add(cord7);
                    list.add(cord8);
list.add(cord9);  list.add(cord10);
                    list.add(cord12);
                   }

                  else
if(currentState.equals(cord12))
                  {
                    //Successors of cord12.
                    list.add(cord1);
list.add(cord3);  list.add(cord10);
                    list.add(cord11);
list.add(cord13); list.add(cord15);
                    list.add(cord16);
list.add(cord17);
                   }

                  else
if(currentState.equals(cord13))
                   {
                     //Successors of cord13.
                     list.add(cord5);
list.add(cord8);  list.add(cord9);
                       list.add(cord10);
list.add(cord12); list.add(cord14);
                       list.add(cord16);
                     }

                  else
if(currentState.equals(cord14))
                  {
                   //Successors of cord14.
                      list.add(cord5);
list.add(cord13); list.add(cord15);
                      list.add(cord20);
list.add(cord21); list.add(cord31);
                   }

                   else
if(currentState.equals(cord15))
                   {
                     //Successors of cord15.
                       list.add(cord12);
list.add(cord14); list.add(cord16);
                       list.add(cord17);
list.add(cord20); list.add(cord22);
                       }

                  else
if(currentState.equals(cord16))
                  {
                      //Successors of cord16.
                      list.add(cord3);
list.add(cord5);  list.add(cord10);
                      list.add(cord12);
list.add(cord13); list.add(cord15);
                      list.add(cord17);
list.add(cord19); list.add(cord22);
                      list.add(cord24);
                  }

                  else
if(currentState.equals(cord17))
                  {
                      //Successors of cord17.
```

```
                    list.add(cord3);
list.add(cord4);  list.add(cord10);
                    list.add(cord12);
list.add(cord15); list.add(cord16);
                    list.add(cord18);
list.add(cord19); list.add(cord22);
                    list.add(cord24);
                  }

                  else
if(currentState.equals(cord18))
                  {
                    //Successors of cord18.
                    list.add(cord3);
list.add(cord4);  list.add(cord10);
                    list.add(cord16);
list.add(cord17); list.add(cord19);
                    list.add(cord23);
list.add(cord24); list.add(cord25);
                    list.add(cord28);
list.add(cord29);
                 }

                  else
if(currentState.equals(cord19))
                  {
                    //Successors of cord19.
                    list.add(cord16);
list.add(cord17); list.add(cord18);
                    list.add(cord22);
list.add(cord23); list.add(cord24);
                    list.add(cord25);
list.add(cord29);
                  }

                  else
if(currentState.equals(cord20))
                  {
                    //Successors of cord20.
                    list.add(cord14);
list.add(cord15); list.add(cord21);
                    list.add(cord22);
list.add(cord31);
                  }

                  else
if(currentState.equals(cord21))
                  {
                    //Successors of cord21.
                    list.add(cord14);
list.add(cord20); list.add(cord23);
                    list.add(cord25);
list.add(cord26); list.add(cord30);
                    list.add(cord31);
list.add(cord33);
                  }

                  else
if(currentState.equals(cord22))
                  {
                    //Successors of cord22.
                    list.add(cord10);
list.add(cord15); list.add(cord16);
                    list.add(cord17);
list.add(cord19); list.add(cord20);
                    list.add(cord23);
list.add(cord24); list.add(cord25);
                    list.add(cord29);
list.add(cord33);
                  }

                  else
if(currentState.equals(cord23))
                  {
                    //Successors of cord23.
                    list.add(cord18);
list.add(cord19); list.add(cord21);
                    list.add(cord22);
list.add(cord24); list.add(cord25);
                    list.add(cord26);
list.add(cord30); list.add(cord33);
                  }

                  else
if(currentState.equals(cord24))
                  {
                    //Successors of cord24.
                    list.add(cord4);
list.add(cord16); list.add(cord18);
                    list.add(cord19);
list.add(cord22); list.add(cord23);
                    list.add(cord25);
list.add(cord29);
                  }

                  else
if(currentState.equals(cord25))
                  {
                    //Successors of cord25.
                    list.add(cord4);
list.add(cord18); list.add(cord19);
                    list.add(cord21);
list.add(cord22); list.add(cord23);
                    list.add(cord24);
list.add(cord26); list.add(cord30);
                    list.add(cord33);
                  }

                  else
if(currentState.equals(cord26))
                  {
                    //Successors of cord26.
                    list.add(cord21);
list.add(cord23); list.add(cord25);
                    list.add(cord27);
list.add(cord30); list.add(cord33);
                 }

                 else
if(currentState.equals(cord27))
                 {
                    //Successors of cord27.
                    list.add(cord26);
list.add(cord28); list.add(cord33);
                    list.add(cord34);
                  }

                  else
if(currentState.equals(cord28))
                  {
                     //Successors of cord28.
                     list.add(cord2);
list.add(cord4);  list.add(cord18);
                     list.add(cord27);
list.add(cord29);
                   }

                   else
if(currentState.equals(cord29))
                   {
                    //Successors of cord29.
                    list.add(cord2);
list.add(cord4);  list.add(cord18);
                    list.add(cord19);
list.add(cord22); list.add(cord24);
                    list.add(cord28);
                   }

                   else
if(currentState.equals(cord30))
                   {
```

```
                    //Successors of cord30.
                    list.add(cord21);
list.add(cord23); list.add(cord25);
                    list.add(cord26);
list.add(cord31); list.add(cord33);
               }

               else
if(currentState.equals(cord31))
               {
                    //Successors of cord31.
                    list.add(cord14);
list.add(cord20); list.add(cord21);
                    list.add(cord30);
list.add(cord32); list.add(cord34);
               }

               else
if(currentState.equals(cord32))
               {
                  //Successors of cord32.
                    list.add(cord31);
list.add(cord33); list.add(cord34);
               }

               else
if(currentState.equals(cord33))
               {
                  //Successors of cord33.
                    list.add(cord21);
list.add(cord22); list.add(cord23);
                    list.add(cord25);
list.add(cord26); list.add(cord27);
                    list.add(cord30);
list.add(cord32); list.add(cord34);
               }

               else
if(currentState.equals(cord34))
               {
                    //Successors of cord34.
                    list.add(cord27);
list.add(cord31);  list.add(cord32);
                    list.add(cord33);
                   }
                     return list;
    }
 }

////////////////////////////////////////////////////
// The class Robot does a search of the coordinates
to find the goal.   //
// Written by La'Shanda Dukes and Justin Deloatch.
//
// The class uses data structures open and closed to
maintain the       //
//    states in the algorithm. Each state selected
from open is         //
//    expanded and saved in closed. The path is
maintained is           //
//    another data structure.
//
////////////////////////////////////////////////////

public class Robot {
       public static void main(String args[])
       {
              //Initializes to open and closed.
              //Stack open = new Stack();
              Queue open = new Queue();
              //PriorityQueue open = new
PriorityQueue();
        ArrayList closed = new ArrayList();

        //The starting position is added to open.
        Coordinate start = new Coordinate(16,22);
        //open.push(start);
        open.enqueue(start);
        //open.add(start);

               while (true)
               {
                  // Terminate with failure.
                  if (open.empty())
                  //if (open.isEmpty())
                  {
                    System.out.println ("Failed to
Search");
                    break;
               }

              //Expand an open node and add it to
closed.
                 //Coordinate state =
(Coordinate)open.pop();
                 Coordinate state =
(Coordinate)open.dequeue();
                 //Coordinate state =
(Coordinate)open.poll();
                 closed.add(state);

             //System.out.println("Extracting from
open: " +state);

                //If the goal is found, terminate
with success.
                if(state.isGoal()) //State is a goal
                {
                  System.out.println("Success: Goal
found");
                  System.out.println(state); //calls
toString()
                  break;
                }

            //The node is expanded.
            CoordinateSuccessorFunction succFcn =
            CoordinateSuccessorFunction();
               ArrayList successors =
succFcn.getSuccessors(state);

               for (int i=0; i<successors.size();
i++)
               {
                //Get successor of the current node.
                  Coordinate succState =
(Coordinate)successors.get(i);
                //System.out.println("\nSucessor:" +
i + " is distance " +
             succState.distFromGoal() + " from
goal.");

             if(!open.contains(succState) &&
!closed.contains(succState))
                {
               //open.push(succState);
                    open.enqueue(succState);
                  // open.add(succState);

succState.setPredecessor(state);
                  //System.out.println("Adding to
open: "+succState);
               }
             } // end of for loop
             //System.out.println();
           } //end of while loop

         //Sets up the path to the goal and prints
it.
           ArrayList path = new ArrayList();
         int size = closed.size();
```

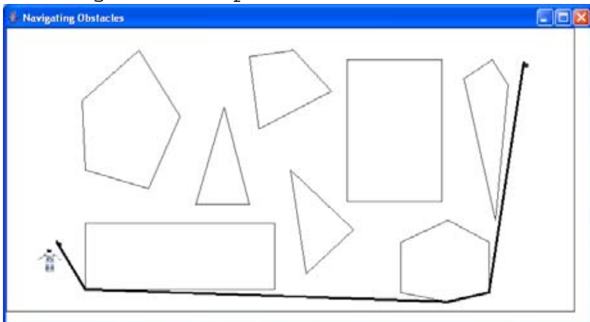
```
            Coordinate predecessor =
(Coordinate)closed.get(size - 1);
             System.out.println("Search path");

            while (predecessor != null)
             {
             path.add(0,predecessor);
               predecessor =
predecessor.getPredecessor();
            }//End of while loop

            for (int i=0; i<path.size(); i++)
                System.out.print(path.get(i) +" ");
              System.out.println();

        }//end of main

  }//end of Coordinate class
```

V. Results

A. Breadth-First Search

The search space was coded using (x,y) coordinates with the starting coordinate as (16,22) and the goal as (164,79). A search path of four line segments and length 218 was generated. This is the second longest path in all three algorithms.

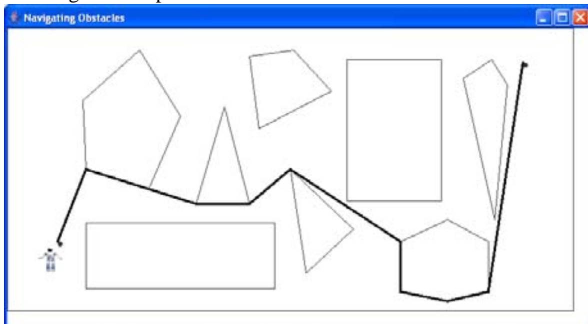
```
Success: Goal found
[164, 79]
Search path
[16, 22] [25, 7] [140, 3] [153, 6] [164, 79]
The length of the path is: 218
```



B. Depth-First Search

A search path of the four line segments and the length 252 was generated. This is the longest path in all three algorithms.

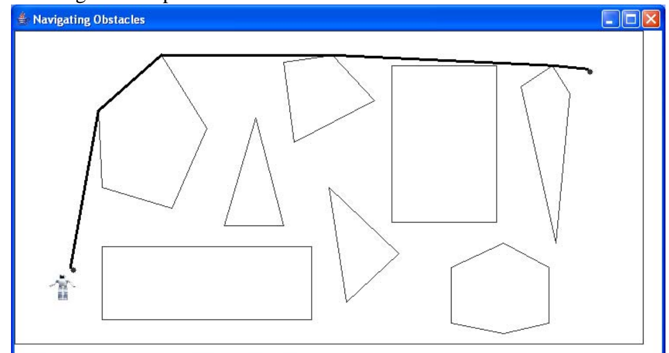
Success: Goal found
[164, 79]
Search path
[16, 22] [25, 45] [60, 34] [77, 34] [90, 45] [125, 22] [125, 6] [140, 3] [153, 6] [164, 79]
The length of the path is: 252



C. Best-First Search

A search path of the five line segments and the length 191 was generated. This is the shortest path in all three algorithms.

Success: Goal found
[164, 79]
Search path
[16, 22] [24, 67] [42, 83] [91, 83] [154, 80] [164, 79]
The length of the path is: 191



VI. Suggestions for Further Research

There are several ways on which this research can be extended. The obstacles considered are polygonal in nature. If they can have curvatures, then a polygonal boundary around such an obstacle can be used on the algorithm. This research can be extended to assume that the surface on which the polygons are is not flat, but can have valleys and hills. Another dimension is to allow the obstacles to be in motion. This involves bringing in time as an added feature.

VII. Bibliography

[1] ICS 171 Lecture Notes: Introduction to Artificial Intelligence, Stephen Bay, University of California, Irvine.

[2] Artificial Intelligence: A Modern Approach, Stuart Russell and Peter Norvig, Prentice Hall, 1995.

[3] Applications and Branches of AI: http://www-formal.stanford.edu/jmc/whatisai/node3.html

[4] [4] Artificial Intelligence Lecture Notes, Professor P. Dasgupta, National Programme on Technology Enhanced Learning (NPTEL) Courses, Indian Institute of Technology http://nptel.iitm.ac.in/

[5] Introduction to Computer Science using Java by Bradley Kjell, Central Connecticut State University, http://www.cs.iastate.edu/~honavar/JavaNotes/csjava.html

[6] Principles of Artificial Intelligence, N.J. Nilsson, Springer-Verlag.