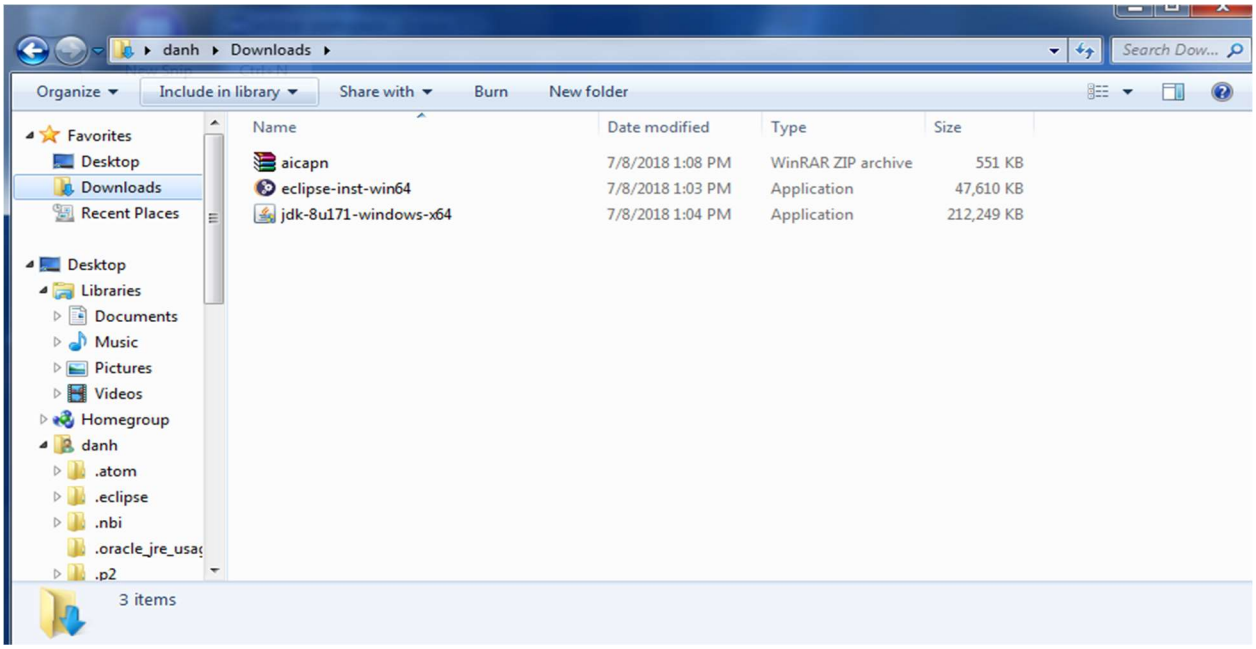
AI Cap'n
CPSC 481 Project
Summer 2018

Instructor
Dr. Paul Salvador Inventado
Phone: 657-278-3821
Email: pinventado@fullerton.edu
Office: CS 534

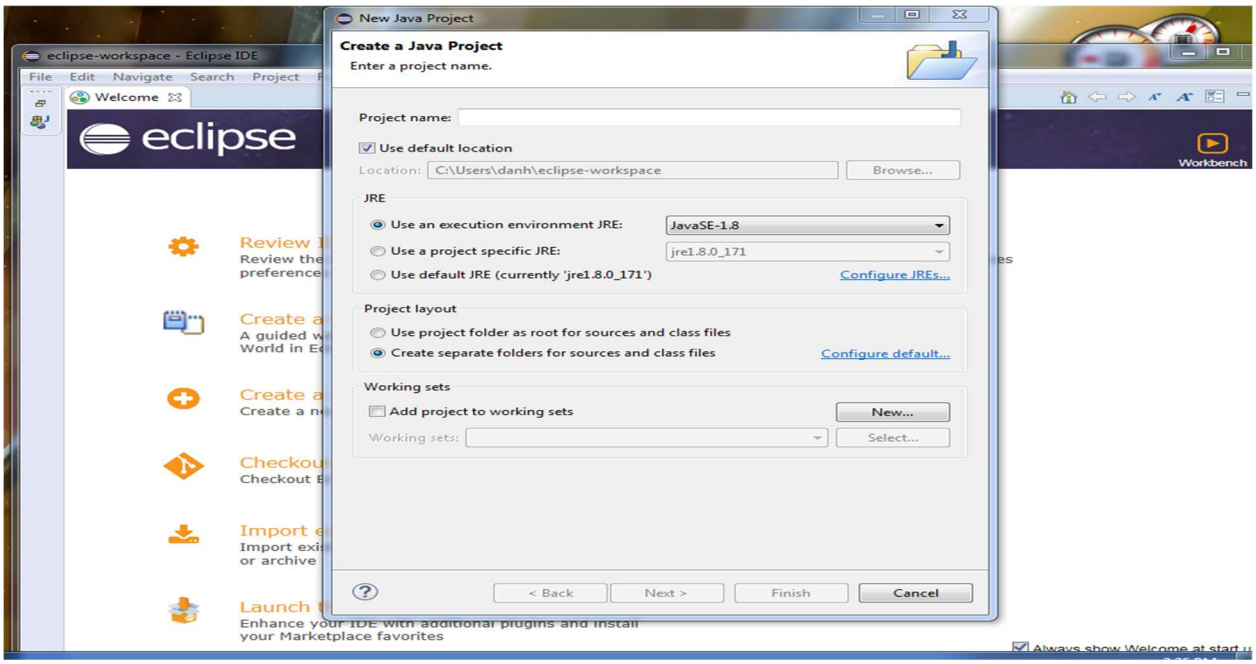
Agent : Za Warudo
By Steven Kha and Danh Pham

Milestone 0: Setting Up the environment with eclipse

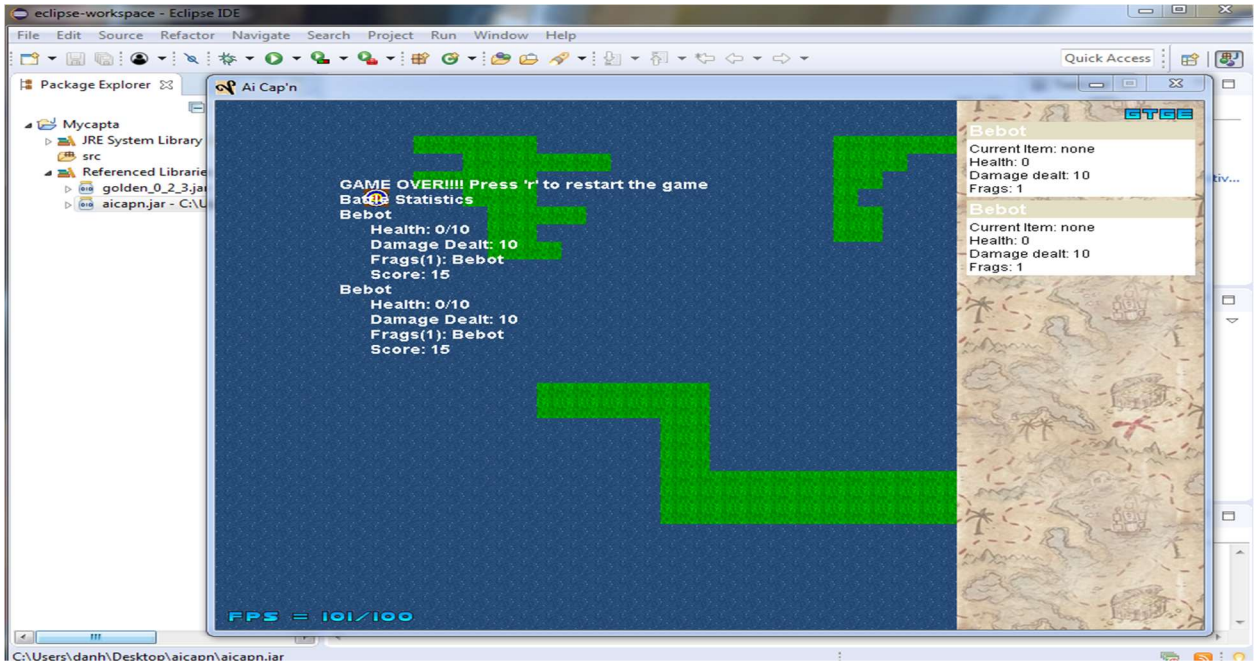
Step 1: download all files need JDK, EC, and Capn



Step 2: Exacting and Importing aicapn to Eclipse



Step 3: run the programing to check if it works or not


eclipse-workspace - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Package Explorer
Mycapta
JRE System Library
src
Referenced Librarie
golden_0_2_3.jar
aicapn.jar - C:\U
Ai Cap'n
GAME OVER!!!! Press 'r' to restart the game
Battle Statistics
Bebot
Health: 0/10
Damage Dealt: 10
Frags(1): Bebot
Score: 15
Bebot
Health: 0/10
Damage Dealt: 10
Frags(1): Bebot
Score: 15
FPS = 101/100
Bebot
Current Item: none
Health: 0
Damage dealt: 10
Frags: 1
Bebot
Current Item: none
Health: 0
Damage dealt: 10
Frags: 1
C:\Users\danh\Desktop\aicapn\aicapn.jar

Introduction

Ai Cap'n is a video game project where a boat character sails to its destination on an ocean map while battling other boats along the way and avoiding any hazards without any player input. In this project, our team will program our own boat against others to see whose boat is the last one standing. The program that helps our boat act on it own involves search algorithm such as breadth first search, depth first search, best first search and or more.

Heuristics

The purpose of our **informed search** is programming our boat-bot to find its destination and the path to reach it. We first chose to implement a Breadth First Search. Breadth will make the boat search for spaces surrounding its current location and then spaces surrounding those spaces. It will keep finding surrounding spaces until the destination is found. Afterwards, we backtracked the spaces that lead to the destination and created the destination path for the ship.

Breadth First Search :

```
function breadth_first_search;

begin
   open := [Start];                                                    % initialize
   closed := [ ];
   while open ≠ [ ] do                                                 % states remain
      begin
         remove leftmost state from open, call it X;
            if X is a goal then return SUCCESS                        % goal found
               else begin
                  generate children of X;
                  put X on closed;
                  discard children of X if already on open or closed;  % loop check
                  put remaining children on right end of open          % queue
               end
      end
   return FAIL                                                         % no states left
end.
```

However, there is a better function to find the informed the search than breadth first search. A quicker search algorithm involves the **best first search**. While breadth did find and create a destination path, it requires too much computational time. Instead of checking every space surrounding our boat's current location, best first search first evaluates the heuristic value of each space. Since we want to find spaces that are closer to the destination, our program checks each for spaces with smallest heuristic value. However, this requires us to create a heuristic function to assign the heuristic values.

Best First Search:

```
function best_first_search;

begin
   open := [Start];                                                    % initialize
   closed := [ ];
   while open ≠ [ ] do                                                 % states remain
      begin
         remove the leftmost state from open, call it X;
         if X = goal then return the path from Start to X
         else begin
                  generate children of X;
                  for each child of X do
                  case
                        the child is not on open or closed:
                           begin
                                 assign the child a heuristic value;
                                 add the child to open
                           end;
                        the child is already on open:
                           if the child was reached by a shorter path
                           then give the state on open the shorter path
                        the child is already on closed:
                           if the child was reached by a shorter path then
                              begin
                                 remove the state from closed;
                                 add the child to open
                              end;
                  end;                                                 % case
                  put X on closed;
                  re-order states on open by heuristic merit (best leftmost)
               end;
return FAIL                                                            % open is empty
end.
```

The **heuristic function** we decided to use is the **distance formula**. Using the distance formula, we can check a space's distance to the destination. First the heuristic calculate the distance the next space is form the destination. Next, it calculates the number of turns that have passed before this space was seen. If the number of turns is too great, the algorithm may backtrack to pick a space with a larger distance but smaller turn cost. This keeps repeating until the destination is reached.

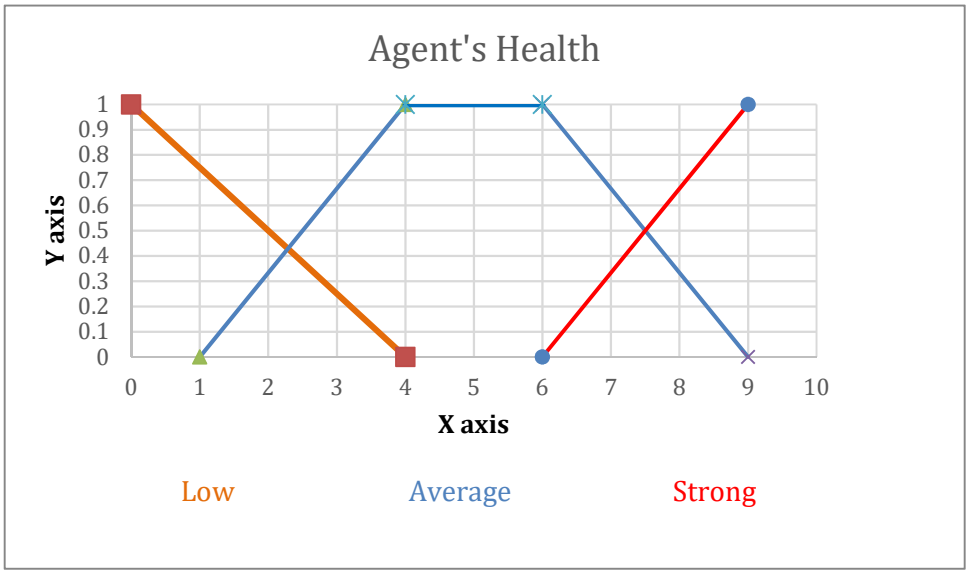
Advantage of using the distance formula is assigning heuristic values to spaces. The search is determined by these values which narrow down the list of spaces to check. The computational time to backtrack and form a path is considerably saved .

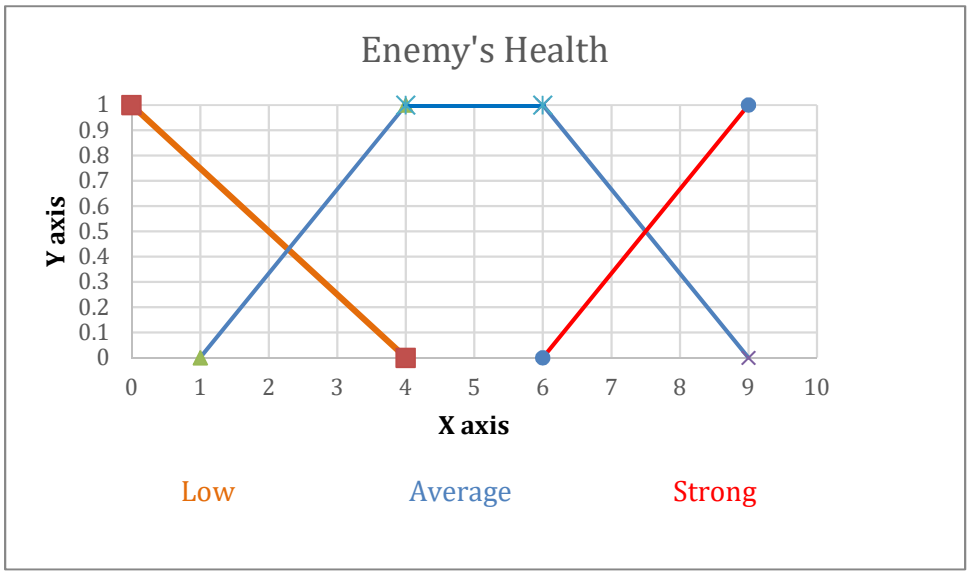
Disadvantages of using this heuristic is the algorithm may be harder to implement. The heuristic estimate may be also lower than the expected number of moves to reach the destination. However, it is very close. I the end it, the **distance formula saves more computational time**.

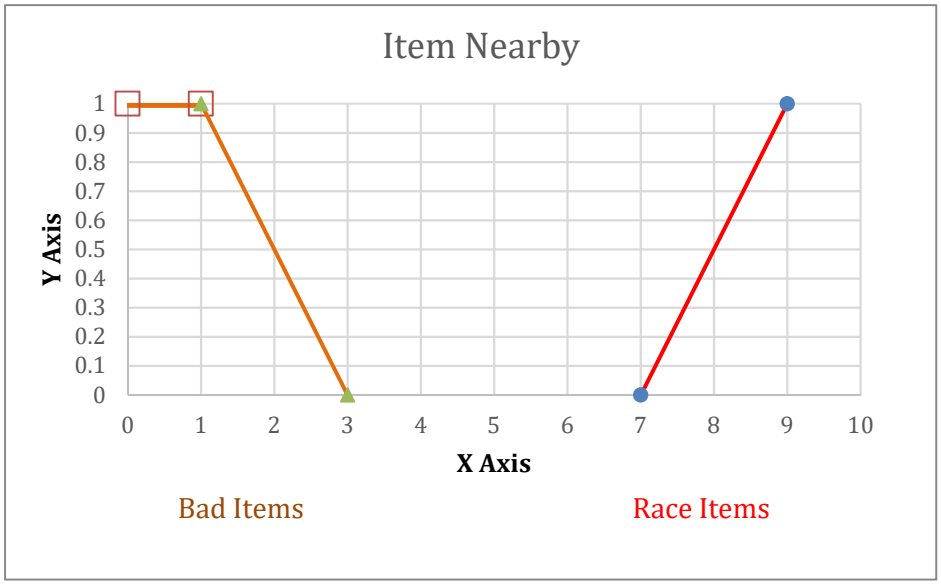
Decision Making

The AI technique we used for our ship's decision making was the fuzzy logic. For our logic we decided to have three major decisions for our ship: evade, collect items, and attack ships. Our four **inputs** include **opponent health**, our **agent's health**, and **items' location**. Please refer to the graph below:

JFuzzylogic graph theory:




Enemy's Health
Y axis
X axis
Low
Average
Strong


Item Nearby
Y Axis
X Axis
Bad Items
Race Items


Fire
Y Axis
X Axis
Run
Run for Items
Kill

The agent and opponent's health share similar coordinates. Low health is represented from (0,1) to (4,0). Average spans from (1, 0), (4,1), (6,1), and (9,0).

The item nearby logic is categorized into bad items and race items. Bad items are from (0, 1) (1, 1) (3,0) and race items span from (7,0) (9,1).

Lastly, the fire logic is split into three. Run: (0,0) (5,1) (10,0), run for items: (10,0) (15,1) (20,0), and kill: (20,0) (25,1) (30,0).

The **output** for each three are used by the integer types. Evade is represented by integer 0-9, collect item represented 10 - 19, and attack ships is above 20.

Combining these data, we generated the four rules:

RULE 1 : IF opponent_heath IS low OR item_nearby IS bad_items THEN fire IS kill;

RULE 2 : IF opponent_heath IS average AND agent_heath IS strong THEN fire IS kill;

RULE 3 : IF NOT(opponent_heath IS low) AND agent_heath IS average THEN fire IS run_for_item;

RULE 4 : IF opponent_heath IS strong AND NOT(agent_heath IS strong) THEN fire IS run;

Rule 1 and 2 tells the ship when to hunt for others. In rule 1 believe the conditions when the opponent health is low or there are no valuable items nearby is a good prerequisite to engage combat. We also believe if the opponent's health is average and our ship health is strong is also a good condition to kill

Rule 3 is our ship's logic to run for items. If the opponent's health is not low and our ship's health is average, then the ship decides to run for items.

Lastly Rule 4 is our ship's logic to simply evade. If our opponent health's is strong and our ship is weak then our ship decides to evade.