Sabareesh Kknan Subramani
Sina Nejati
nejat001@cougars.csusm.edu
subra001@cougars.csusm.edu
571 – Artificial Intelligence
Prof. Guillen
Team Project – Design Document

Introduction:

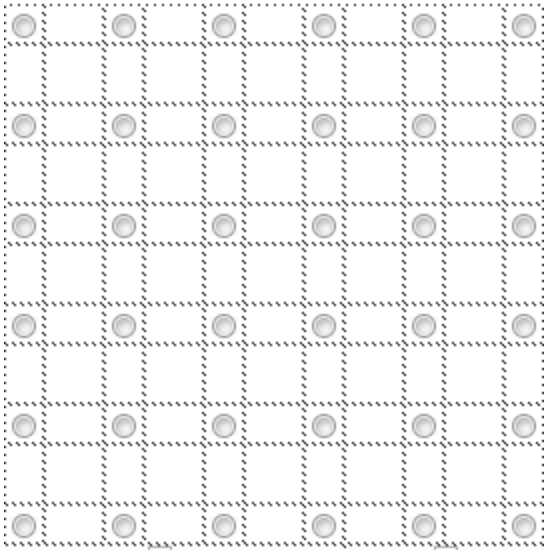
We chose the Dots and boxes for our team project. Dots and boxes is a turn based game. Starting with an empty grid of dots, players take turns, adding a single horizontal or vertical line between two un-joined adjacent dots. A player who completes the fourth side of a 1×1 box earns one point and takes another turn. (The points are typically recorded by placing in the box an identifying mark of the player, such as an initial). The game ends when no more lines can be placed. The winner of the game is the player with the most points.

Implementation:

We used JavaScript for the implementation of this project. The AI of this game starts the game with two basic rules: avoid filling the third side to any squares, fill the fourth side of a square if possible. To implement we start by scanning the grid.

Grid:

We defined the grid with an array of the length of 60. If the bit for the line is 1 it means that the line has been drawn by player. The value of the line is 2 it means the line has been drawn by AI. The overview of the grid is shown in the figure 1-1.



Mappers:

This function draws a line either horizontal or vertical with the respect to the radio buttons that are selected by the user. It looks to the ID of the radio button numbers to find which line it has to draw.

Boxes:

The ownership of boxes is stored in CBOX array which has a length of 25. If the value is 1 then the box belongs to the user and if it's 2 it belongs to the agent. We will use this variable in our utility function to know which state is more promising for our minimax algorithm. Each time a box is filled it returns true which will keep the turn for the player or AI who has completed the fourth side of the box.

AGENTS:

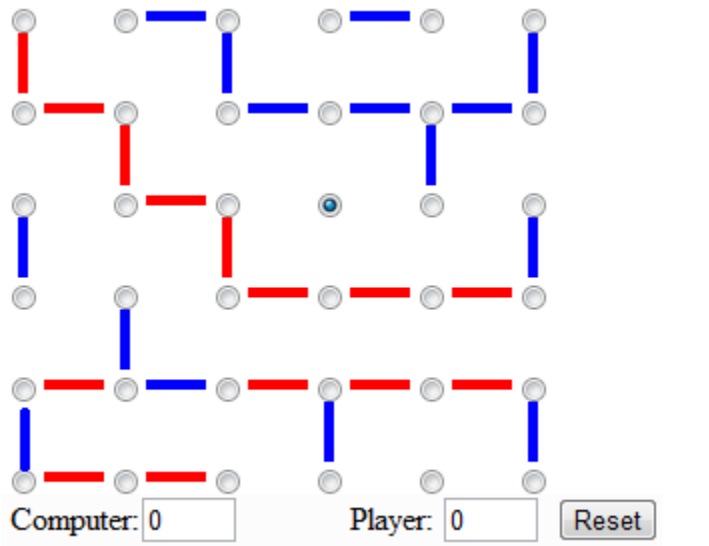
We designed two agents for our AI. Since the state space of this program consists of 60! Nodes, therefore starting the search from the beginning of the program would not be possible. If we decide to do a minimax search at a limited depth from the beginning of the game the utility function would only pass up 0 since at the start of the game it is unlikely that any player fills any boxes. To overcome this problem we use our first agent. The AI of this agent starts the game with two basic rules: avoid filling the third side to any squares, fill the fourth side of a square if possible. To implement we start by scanning the grid.

Choice():
(Simple Agent)

This function scans the grid for the state of each line, after obtaining the state of each line by checking the condition of all the boxes and all the lines corresponding to that boxes it will assign a number to each line. The choice function will chose the line with the value of 3, which means the three sides of a square has been filled. It will avoid lines with the assigned value of 1, which mean by adding a line the other player will get a chance to complete a square. Other than this it will randomly chose the lines that do not meet these conditions. If the value of the line is 0 then it will fill the box by changing value in the CBOX. Keeping the turn the agent is called again.

Agent():
(Minimax)

If the choice does not pass up any valid values then the state of the game has reached a point where we can conduct a search. The variable "SEA" conserves this state. If the SEA is set to true the agent for the minimax algorithm is called.



The agent starts by the first line choosing the first line with the value of 0, doing so the turn will negate. We then call our first agent which shall simulate the effects of running the agent on this state. We store the outcome of this new grid in **Grid2**.

Pruning:

The search space is not yet optimal we continue the pruning the expanded nodes in **Grid2[]** by checking the lines that were drawn while simulating the agent . All these lines will be ignored when looking for the next line to drawn. A downside of this technique would be the incorrect result that we pass to **Space[]** (which we store the number of the lines that can be drawn by each move in), to overcome this problem we perform another search starting from the last one storing the number of lines associated with each move to **Space2[].** Comparing these two Space arrays we will go on to chose the one with the less value. By doing that we are performing a minimax at the depth of 2, looking only one step ahead but pruning the lines that will cause us to land in the similar end state.

Design:

Percepts:

Each instance of the array Grid which represents the lines drawn, boxes filled and the turn of the player to play.

Sensors:

We use radio buttons to read the input from the users.

Actuators:

Our agents are the actuators that will alter the state by adding lines, filling the boxes and changing the turn if necessary.

Initial State:

Empty grid, all lines set to 0 which is an array with the length of 60. Turn set to player.

Successor function:

Uses the update() function to draw a line , and also uses box to find the turn of the next player.

Goal test:

No lines left to draw.

Path cost function:

The utility function on the MIN will be passed up to the node , therefore the evaluation of the MIN grid is our path cost.

Heuristic function:

Is the one used by the simple agent to know which line to draw? It will use the choice() function to evaluate.

Utility function:

Number of AI boxes minus the number of boxes which the ownership has been set to the user.

The game can be accessed from the website : www.seedspirit.com/dots