CS 4701 - Foundations of Artificial Intelligence Practicum (Fall 2014)

Scientific Visualizations of A.I. Algorithms and Methods

Project Team Members:
Jie Lin (jl987)
Zichuan Zhou (zz278)
Kent Huang (kh395)

Table of Contents

I. Abstract:

In lecture, we encountered several artificial intelligence algorithms and methods including various search algorithms, machine learning algorithms, and adversarial search algorithms. Much of what students typically encounter in class are pseudo-codes, a very abstract description of the algorithm, which typically do not aid students to further gain a big picture of the algorithms.

Although students can try to trace out the pseudocode of a small example dataset, they are typically limited to a very small and manageable data. For algorithms that has a large size input with millions of data points, it is a challenging task to visualize how the algorithm will deal with the data. Thus, large size examples with millions of data points can be hard to trace. Our main motivation for this project will be to implement several of these algorithms and build web-based visualizations of them in order to further enhance the understandings of the techniques behind the algorithms and to see how the algorithms treat the large data inputs.

II. Problem Statement

The algorithms that we choose to implement and visualize come from class lectures and from the textbook "Artificial Intelligence: A Modern Approach." We chose the following list of algorithms to implement. Each project team member will implement several A.I. algorithms listed above.

Pathfinding algorithms:

- A* search:
- Breadth-First Search
- Depth-First Search

Adversarial Search:

- Minimax search

Machine Learning:

- K-Means Clustering
- Perceptron algorithm

III. Problem Approach

The main tools that we will be using to build a web-based visualization are *Javascript* and *D3.js*. Javascript is a well-developed language that uses the DOM model and has many pre-written functionalities. Additionally, Javascript has great external libraries such as D3.js to

create visual effects on the screen, calculating data and perform data processing on web pages using the web browser.

Search Algorithms

Introduction

The A* search, best-first search, breadth-first search, and depth-first search will be implemented with a grid of x-y coordinates where each square represents a node of the graph. The main components of the grid consist of **start state**, **goal state**, and **walls**. The start state represents the starting point for the algorithm. The goal state represents the destination that the search algorithms need to find. Walls serve as "obstacles" which an agent cannot traverse. To run the algorithm, the user will select a start state, a goal state, and a type of search. Once that's done, a path will be displayed to the user showing the correct path to the goal state.

Description of the Search Algorithms

1. A Star Search:

The pseudocode for the A* star search is the following. The heuristic used is manhattan distance.

```
search: function(graph, start, end, options):
	initialize graph;
	start = user selection in the interface;
	queue = new priority_queue();
	queue.enqueue(start);
	while (queue.size > 0) {
		curr_node = queue.dequeue();
		if (curr_node == goal state) {
			return path;
		}
		curr_node.visited = true;
		foreach(curr_node's neighbors) {
			if (!neighbor.visited || curr_score < neighbor.score) {
				neighbor.visited = true;
				update heuristic score for this neighbor
				queue.enqueue(neighbor);
			}
		}
	}
	// path not found:
	return []
```

2. Breadth First Search:

The pseudocode for the breadth first search is the following. Instead of a priority queue, bfs uses a normal FIFO queue with no heuristics.

```
search: function(graph, start, end, options):
	initialize graph;
	start = user selection in the interface;
	queue = new queue();
	queue.enqueue(start);
	while (queue.size > 0) {
		curr_node = queue.dequeue();
		if (curr_node == goal state) {
			return path;
		}
		curr_node.visited = true;
		foreach(curr_node's neighbors) {
			if (!neighbor.visited || curr_score < neighbor.score) {
				neighbor.visited = true;
				queue.enqueue(neighbor);
			}
		}
	}
	// path not found:
	return []
```

3. Depth First Search:

The pseudocode for the breadth first search is the following. The main difference between this search algorithm and the previous two is that dfs uses a stack instead of a queue.

```
search: function(graph, start, end, options):
	initialize graph;
	start = user selection in the interface;
	stack = new stack();
	stack.push(start);
	while (stack.size > 0) {
		curr_node = stack.pop();
		if (curr_node == goal state) {
			return path;
		}
		curr_node.visited = true;
		foreach(curr_node's neighbors) {
			if (!neighbor.visited || curr_score < neighbor.score) {
				neighbor.visited = true;
```

```
                    stack.push(neighbor);
                }
            }
        }
        // path not found:
        return []
```

Description of the User Interface

The main user interface for the search algorithms looks like the following. On the left side of the panel, the user can select which type of algorithm that will be run. Beneath the algorithm selections are the typical benchmarks: **execution time**, **path length**, and **number of iterations**. The execution time is the time in milliseconds that the algorithm took to find the path from the start state to the goal state. The path length is the number of squares that separates the start state from the goal state. Paths cannot be diagonal. The number of iterations represent the number of times that the *while* loop inside the algorithm is executed.

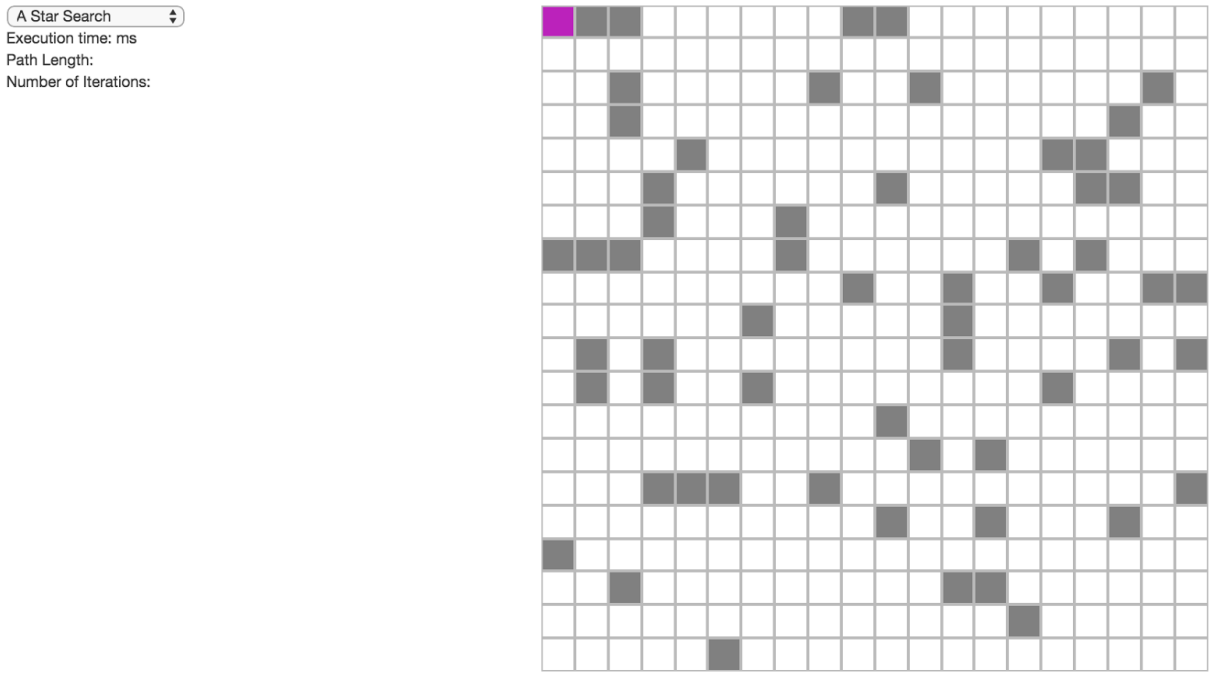


Fig.1. Left Panel contains the algorithm selections and benchmark status. Right Panel displays the path finding visualization for the selected algorithm.

The following displays a close-up snapshot of the left panel of the search algorithm visualization. The default selection is A Star Search.

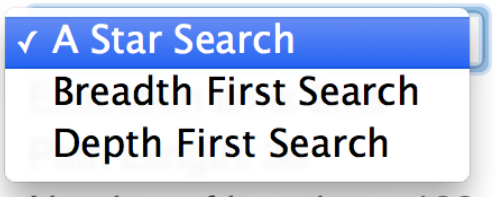


Fig. 2. Different types of algorithms users can select

The following displays the path found by the search algorithm.

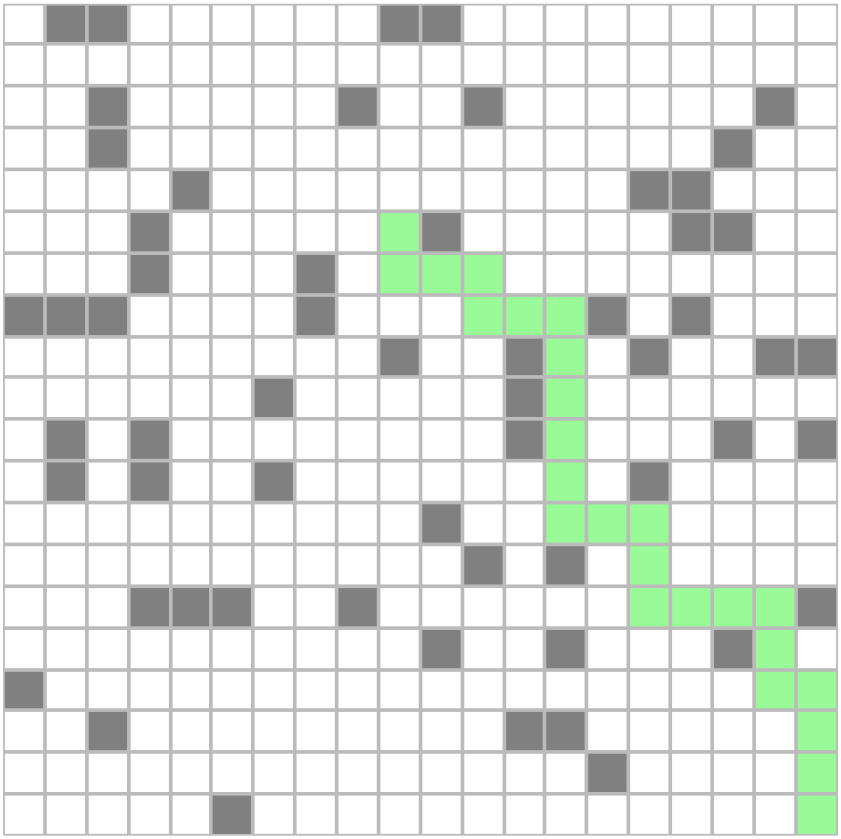


Fig. 3. Algorithm displays the path traversed.

Evaluation Measure

The A* search, breadth-first search, and depth-first search will be measured by the efficiency of the algorithm based on the execution time, the path length returned by the code, and the number of iterations that the algorithm took. From running the algorithms many times

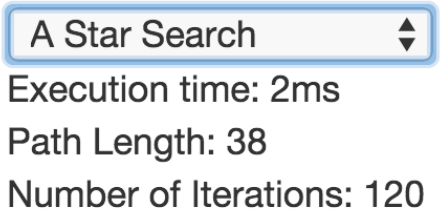


Fig. 4. Evaluation Technique

Minimax with Tic-Tac-Toe

Introduction

We implemented an A.I. that plays the game of Tic-Tac-Toe. According to the minimax algorithm, the A.I. player will construct all the possible game states. From these possible game states, the A.I. player will select the best possible move given the human player moves. Typically for games such as chess or checkers, the number of such game states is very large and it's not feasible to compute all possible game states. However, it is possible for the Tic-Tac-Toe A.I. to enumerate all the different positions. Hence, it's possible for the A.I. to play Tic-Tac-Toe perfectly (i.e., no loss possible).

Description of the Minimax Algorithm

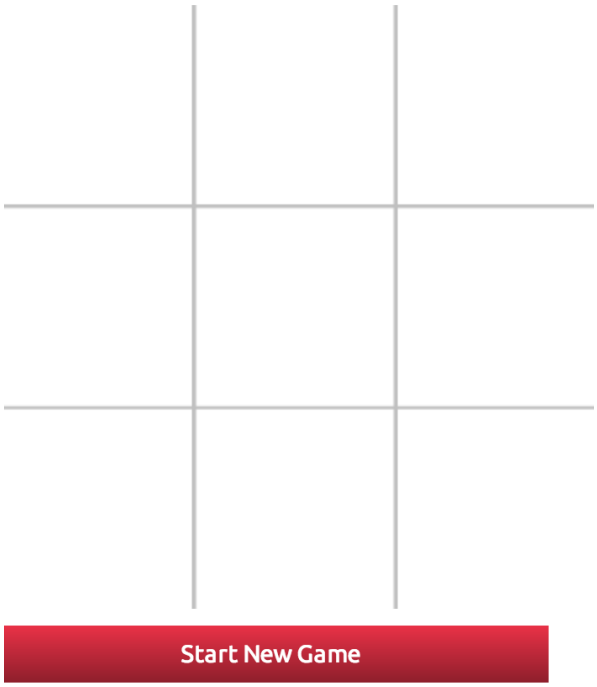
The pseudo-code for the minimax algorithm is the following (in this demo, minimax is simplified to negamax). There are two key methods. The first one is next_move, which helps the computer determine the optimal move. The second one is negamax, which is a helper method to calculate the optimal move by enumerating all possible results. The idea of the negamax algorithm is similar to minimax, except that we simply negate the other player's score to obtain that of ourselves. Note we did not do any pruning in this algorithm, since the cases are easy to enumerate given the small scale of the game.

```
next_move():
	best_score = NEGATIVE_INFINITY
	temp = 0

	number_of_results = 0
	for(i=0;i<100;i++}{
		if(cell_is_empty){
			cell[i] = computer
		}
		temp = -negamax(the_other_side)
		if(temp_score > best.score) {
				best.move = i;
				best.score = temp_score;
		}
```

```
        }

        draw_move()
        check_winner

negamax(side):
        // Result is determined
        if (check_winner):
                reward(winner) = 1
                number_of_results++
        if (check_tie):
                reward(both) = 0
                number_of_results++
        // Result is undetermined
        max_score = NEGATIVE_INFINITY
        temp = 0
        for(i=0;i<10;i++){
                if(cell_is_empty){
                        cell[i] = side
                }
                temp = -negamax(the_other_side)
                cell[i] = empty

                if(temp>max){
                        max=temp
                }
        }
}
return max
```

Description of the User Interface

The initial state of the game consists of a blank board. The user clicks “start new game” to begin a new game with the A.I. During each iteration, the player is allowed to select one of the nine squares on the game board to place his/her move. The initial state looks like the following (see Fig. 5). The game ends when either player has three of his pieces in either a horizontal, vertical, or diagonal position (see Fig. 7). The game is declared a tie when none of the players reached the above condition.



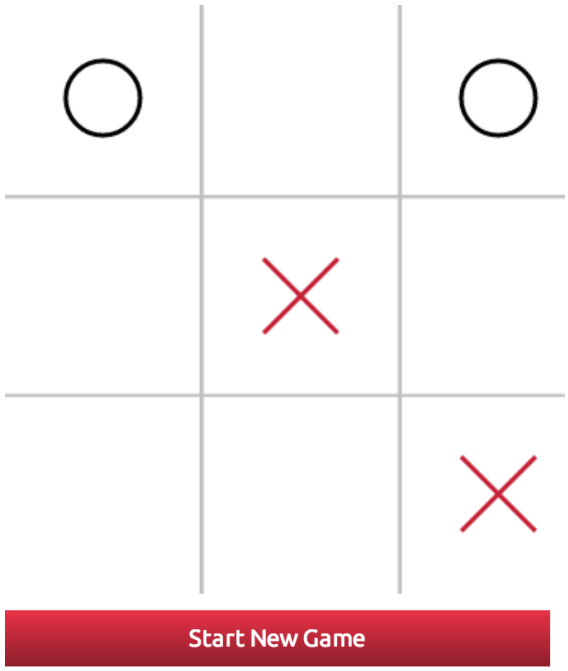


Fig. 5. Initial State of the Board. **Fig. 6. In-Game Board State**

When the player loses, the game board state displays the following.

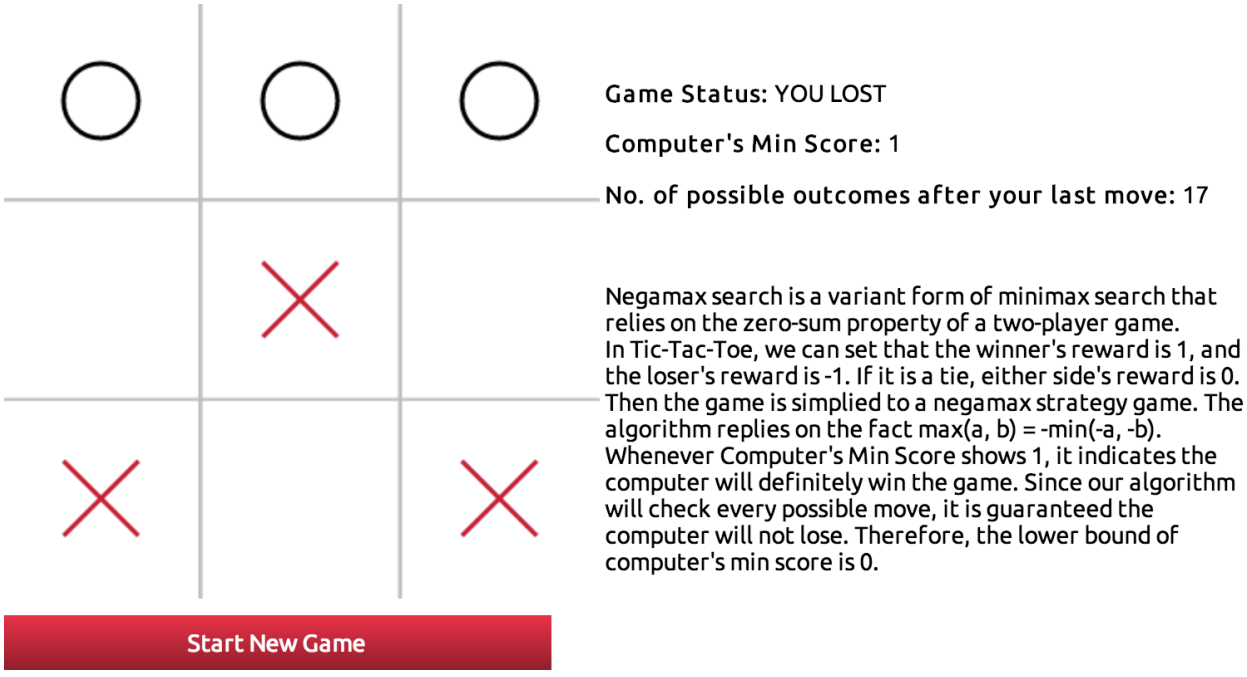


Fig. 7. Final Board State with a win by the A.I. player.

Evaluation Measure

We played the game numerous times and we saw that most of the time the A.I. was able to beat us. Occasionally if we play with care, we were able to tie the game. On the other hand, we can never beat the A.I. This shows that the A.I. is able to perfectly play the game of Tic-Tac-Toe.

K-Means Clustering

Introduction

The K-Means clustering algorithm is an unsupervised learning algorithm which, when given a set of points, tries to find *k* number of centroids. We will be visualizing this algorithm using a grid of x-y coordinates in 2 dimension where a point on the graph represents an example to be clustered. The main components of the grid consists of the **centroid** and a generated **list of x-y coordinate points**. To run the algorithm, an user selects the number of points to be generated and the number of clusters to classify the points.

Description of the K-Means Clustering Algorithm

The pseudocode for the K-means clustering algorithm is the following. The distance measure used is the Euclidean distance. In the below pseudocode, the *data_points* represent the initial data generated by the random generator. The *k* represents the number of clusters. The K-means algorithm will keep updating the centroids of each cluster until convergence.

```
k-means(data_points, k):
	select k random points {s1, …, sk} as seeds
	while (!clustering convergence of the data points):
		foreach data_point pi,
			assign pi to the cluster cj such that distance between pi and cj is
			minimized.
		foreach cluster cj,
			update the centroid of the the cluster.
	return centroids
```

Description of the User Interface

The main user interface consists of the left panel which allows the user to select the number of data points and the number of clusters. The “number of iterations” field allows the user to see how many iterations the k-means algorithm has gone through. This will be an important factor in determining the quality of the clustering.

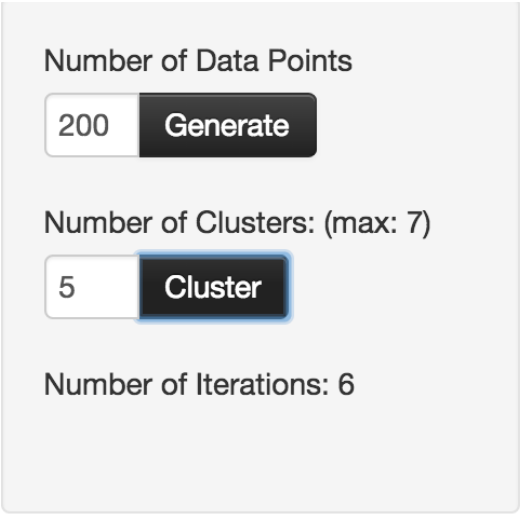


Fig. 8. Users can select the number of data points and the number of clusters

Additionally when the clustering algorithm converges, we can see the each point is connected by a color to a given centroid. The points connected by the same color form a cluster.

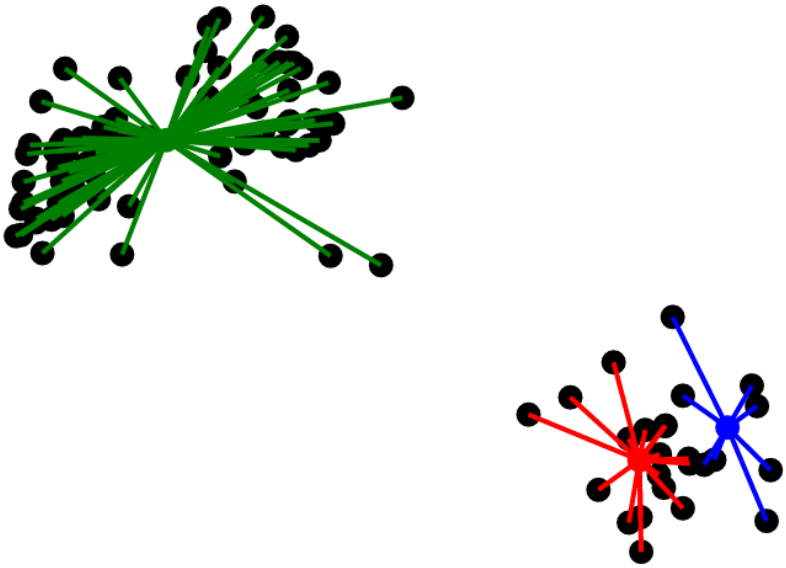


Fig. 9. Clusters determined by K-means algorithm where the number of clusters is 3.

Evaluation Measure

The main evaluation measure for the K-means is the number of iterations taken for the algorithm to converge. We saw that for large number of data points, the time taken to converge is much greater. Additionally for datasets where the points are scattered loosely, the clustering algorithm was not able to separate the data in a small number of iterations.

Perceptron (linear classification)

Introduction

The Perceptron is a supervised learning algorithm and a linear classifier. During its training phase, the perceptron iterates through each data point and updates its weight vector, which defines its slope, every time it incorrectly classifies a point. After the perceptron is trained, a test point can then be classified to one class or the other. The perceptron is a strict classifier, meaning that it can only classify data that is linearly separable. On data that is not linearly separable, the perceptron algorithm may not converge since it will continually make classify points incorrectly.

Description of the Perceptron Algorithm

The following pseudocode shows the Perceptron algorithm. Each iteration of the Perceptron algorithm updates the misclassified point.

```
Perceptron(data_points, max_iterations):
       weights: initialize the weights to zero vector
       for (i = 1 to max_iterations):
           for (i = 1 to data_points.length):
              if (perceptron makes mistake):
                    update weights()
       return weights
```

Description of the User Interface

Training points are generated at random using a normal generator. On click of the Update but, the perceptron completes one iteration over a single test point, and if there was an error in the classification of the point, the perceptron will be updated. Otherwise the perceptron will remain the same. The user is also able to input a test point, and depending on the trained perceptron, the point will be classified as either positive (blue) or as negative (red).

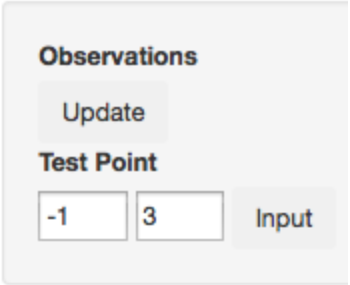


Fig. 10. User can click on the update button to perform the update rule for Perceptron.

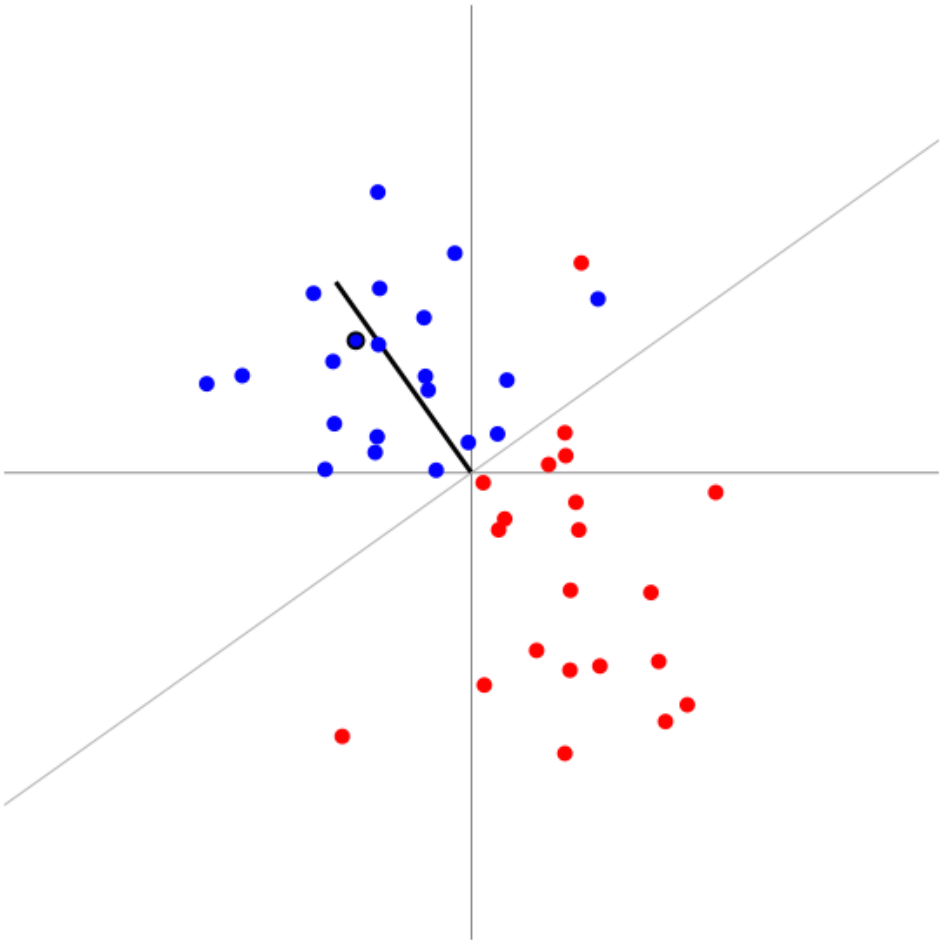


Fig. 11. The user can select a misclassified point for the Perceptron to correct.

Evaluation Measure

The main evaluation measure for the perceptron algorithm is whether or not the perceptron will be able to separate training data that is linearly separable, and whether or not the perceptron can classify a test point correctly. In our case, the generated data points are mostly not linearly separable, so we have to click on "update" button several times in order get a minimum number of misclassifications or errors.

VI. Conclusion

The most interesting and rewarding part of this project was implementing the various algorithms and seeing them in action. Surprisingly, we spend more time trying to setup the visualizations than actually implementing the algorithms themselves.

VII. Future Work

In the future, we would like to expand on our list of implementations of algorithms to include artificial neural networks, support vector machines, k-nearest neighbors. We are also interested in sharing this with others on the web.

VIII. References

(Note: We referenced the K-means and Perceptron algorithm from Professor David Mimno's course CS 3300 - Data-Driven Web Applications Spring 2014. We further enhance some of the K-means and Perceptron algorithms.)

1. **http://www.cs.cornell.edu/courses/CS4701/2014fa/projects-prop_v4.pdf (project guideline)**
2. **http://www.cs.cornell.edu/courses/CS4701/2012fa/projects-sugg-b_v1.html (project suggestions)**
3. **http://en.wikipedia.org/wiki/A*_search_algorithm (A star algorithm)**
4. **http://www.cs.cornell.edu/courses/cs4700/2014fa/slides/CS4700-Games1_v5.pdf (Adversarial Search)**
5. **http://www.cs.cornell.edu/Courses/cs4780/2014fa/lecture/06-perceptron.pdf (Perceptron algorithm)**
6. **http://www.cs.cornell.edu/Courses/cs4780/2014fa/lecture/22-clustering2.pdf (K-means clustering)**
7. **http://en.wikipedia.org/wiki/Tic-tac-toe#Number_of_possible_games (tit-tac-toe)**
8. **http://www.briangrinstead.com/blog/astar-search-algorithm-in-javascript (A star search algorithm)**
9. **http://www.neverstopbuilding.com/minimax (Tic-Tac-Toe)**