Queen's University

Solving Game Playing Problems: Pathfinding and Alpha-Beta Pruning

Group 13

Echo Gao [10149157]

Tianyu Hua [20142378]

Shirley Xiao [10145624]

Feiyi Yang [10151502]

Wanyu Zhang [10141510]

CISC 352

Professor Sidney Givigi

Friday, March 15, 2019

Table of Contents

Introduction

This technical document will be discussing two problems: Pathfinding and Alpha-Beta Pruning. The section of each problem will go through the data structure(s) used, the code and the purpose of each function, and the discussion of results from testing.

Pathfinding problem is a problem commonly encountered in the game. The goal of pathfinding is to find the shortest path in the least amount of time between two points on a map while avoiding obstacles in the way. The map is often broken down into grids to be explored. The grid breakdown can have high resolutions with more grids of smaller sizes, or low resolutions with fewer grids of larger sizes. In this assignment, pathfinding was performed using A* algorithm and greedy search algorithm. The performance was measured on graphs with dimensions 16 to 1024 (i.e. graphs of 16*16 to 1024*1024) with one start node and one goal node. A* algorithm and greedy search are both informed search algorithm where the nodes that are most likely to be the least expensive gets expanded using heuristics. The difference lies in A* algorithm, which takes into consideration both the cost so far to reach the next node as well as the approximated cost of that node to the goal, while greedy search only concerns the approximated cost. This makes greedy search very efficient, however greedy search often cannot find the most optimal path (i.e. the path of the shortest length). More computation is involved in A*, however, it can guarantee that the optimal path can be found. The heuristics function chosen for the two algorithms are very important – the more informed the function is, the more efficient the process will become. Two distance functions are used for the two modes of movement involved – when the only movement in four directions: up, down, left and right, are allowed, Manhattan distance is chosen; when movement in the diagonals are allowed, Chebyshev is chosen. Both functions are the most informed in this case.

The Alpha-Beta Pruning is an adversarial search algorithm that is commonly used for two-player computer playing games. The main idea of this algorithm is to keep track of the values of alpha and beta that represents the minimizing score of maximizing player and vice versa, where alpha is set to -infinity and beta is set to infinity - this means that both players will both be starting at the worst possible score. As the algorithm uses the minimax algorithm recursively, it will update the alpha and beta values to find the best possible score for the minimizing and maximizing player. Once the minimax algorithm finds a node that is worse than the previous node examined, the subtree or leaf node will be cut off, and there will be no further evaluation for that branch. The efficiency of Alpha-Beta Pruning depends on the order in which children are visited.

1. Pathfinding

Purpose

The purpose of pathfinding is finding the shortest path from the starting point to the goal. This can be achieved by using search algorithms, *Greedy* and *A**. Each algorithm has its advantages and disadvantages. This section of the document will test the path determined by *Greedy* and *A** search algorithms with one or more of the following heuristics: *Euclidean*, *Manhattan*, *Chebyshev* distance algorithm.

Data Structure

In order to find a path from the start node to the goal, it is not sufficient enough to record the visited nodes only. Instead, every visited node is stored with its preceding node. Every node in the grid is identified using coordinates with row number specifying x coordinates and column number specifying y coordinates. The following figure shows an example of a 3 by 3 grid with labelled coordinates. The possible input grids for this problem are of size m by n, where 8 ≤ m, n ≤ 1024. They are stored as a list of strings separated by rows.

(0,0)	(0,1)	(0,2)
(1,0)	(1,1)	(1,2)
(2,0)	(2,1)	(2,2)

Figure 1. An example of 3 by 3 grid with labeled coordinate.

Tuple

The coordinates of each node are stored in a tuple in the form of (x, y), where 0 ≤ x, y ≤ 1024-1. A tuple is chosen to be the structure used for coordinates due to its immutable property. In the algorithm of pathfinding, the information of the visited node will be stored in a dictionary, which means that the coordinate of a node will become a key in the dictionary. The property of dictionary restricted that its keys must be immutable data types. Thus, a tuple is used instead of a list to represent the coordinates of the node.

Dictionary

In A* algorithm, two dictionaries are used, *cost_so_far* and *came_from*. The *cost_so_far* dictionary stores the cost for each node. The cost for the node is defined as the sum of the cost of each move from the start to reach the concerning node. In this assignment, the cost of a move in any direction is one. The key of this dictionary is a tuple of the coordinate for the spanned node, and the value of the dictionary is the cost of that particular node as an integer. The dictionary is only updated when a new node is being added, or an existing node is found to have a lower cost than what's already recorded in the dictionary. An example of an element in this dictionary may be {(0,1): 1}.

The *came_from* dictionary is used for both the A* and the Greedy algorithm. This dictionary stores spanned nodes as its keys and the preceding nodes as its corresponding value. However, there is a slight difference in the use of this dictionary between the A* and the Greedy algorithm. The A* algorithm modifies this dictionary only when a key is newly added or when a key is found to have a lower cost. On the other hand, the Greedy algorithm appends any spanned node into the dictionary as long as it does not exist already. For example, if the path goes from (1, 1) to (1, 2), then the corresponding element in this dictionary will be {(1, 2): (1, 1)}.

Priority Queue

The use of the priority queue is critical in the pathfinding problem. This assignment used a priority queue called *frontier* to store the spanned nodes. Each element of the queue is a tuple of the node's coordinates. The priority value of each element is computed differently depending on the algorithm used, and the structure of the priority queue ensures that the highest priority element is examined first every time the queue is visited.

Technique[1]

Every input grid is stored as a list of strings such that each row of the grid is a string in the list. The start node and the goal were first identified using the *findSG* function contained in the *Graph* class, which loops through the list of string and identifies the index where the character is "S" or "G". The *Graph* class also contains a *neighbor* function, which searches all the neighboring nodes (up, down, left, right) for the provided node and store them into a list. The *diagNeighbor* function in the same class calls the *neighbor* function and then add the diagonal neighbors to the list.

```
class Graph
```

This function is used to find all neighboring node of the concerning node (in directions: up, down, left, right), and return a list of neighbors
@param *current*: a tuple representing the coordinate of concerning node

```
    def neighbor(self,current):
        initialize a list for neighboring nodes

        only open space("_") and goal("G") are accepted to be valid
        neighbors

        set the range of search such that it does not exceed the
        dimension of the grid

        check the coordinates that neighbors current
        (up, down,left, right)
            if it is acceptable
                append this coordinate into the list

        return the neighbor list
```

This function is used to find all neighboring node of the provided node (including diagonals), and return a list of neighbors
@param *current*: a tuple representing the coordinate of concerning node

```
    def diagNeighbor(self, current):
        define the accepting neighbor as "G" and "_"

        find a list of up, down, left, right neighbors using
        neighbor(current)

        set the range of search to be within the dimension of grid

        check the coordinate that neighbors current
        (upper-left, upper-right, lower-left, lower-right)
            if it is acceptable
                append this coordinate to the neighbor list

        return the neighbor list
```

Two heuristic functions are considered in this assignment. In the first part of the problem, a move can only be made in the directions of up, down, left, and right. In such a case, the heuristic function is based on the Manhattan distance since it is the most informed heuristic. When a move is allowed to be taken diagonally, the Chebyshev distance becomes the most informed heuristic and thus is selected to be used in such problems.

This function takes two node coordinates and find the Manhattan distance between them, returns this value
@param *goal*: tuple represents the coordinate of goal node
@param *nextG*: tuple represents the coordinate of concerning node

```
def manhattan(goal, nextG):
      goal and nextG are in format of (x,y) coordinates

      find the absolute value of difference between goal x and nextG x

      find the absolute value of difference between goal y and nextG y

      sum the x and y difference

      return sum of differences
```

This function takes two node coordinates and find the Chebyshev distance between them, returns this value
@param *goal*: tuple represents the coordinate of goal node
@param *nextG*: tuple represents the coordinate of concerning node

```
def chebyshev(goal, nextG):
      goal and nextG are in format of (x,y) coordinates

      find the absolute value of difference between goal x and nextG x

      find the absolute value of difference between goal y and nextG y

      find the higher value in x and y differences

      return the higher difference
```

The function *aStar* finds the path from start to goal for the provided grid using A* algorithm. The best solution (lowest cost, and in this case shortest) is guaranteed with A* algorithm. A* uses two dictionaries and a priority queue to keep track of the spanned nodes. The dictionary *cost_so_far* records the cost of reaching the concerning node from the start. The dictionary *came_from* records the coordinates of concerning node and its preceding node. The priority queue stores the next possible move based on priority value. For every node expanded, A* considers the cost so far and the heuristics value, which forms the priority value for that node. All three of these data sets are updated only when the concerning node had not been visited or it exhibits a lower cost than what's already recorded.

This function find the path using A* algorithm
@param *thisGraph*: the input grid as a graph object
@param *mode*: indicates whether diagonal moves are allowed or not

```
def aStar(thisGraph, mode):
      obtain the grid from the graph object

      find the start and goal index

      initialize a priority queue(frontier)
      put the start index into the queue with priority value 0

      initialize two dictionaries (cost_so_far and came_from)
      put start with cost 0 into cost_so_far
      put start with preceding node as None into came_from
      set flag found at False

      while frontier is not empty:
            pop the first element in the queue as current node

            if current node is the goal:
                  found = True
                  break from while

            if mode is 0, no diagonal moves:

                  for nextG in the neighbor list of current node:
                        current cost increment by 1

                        if nextG is not visited or if (it has been visited
                        and current cost is lower than the cost of its last
                        visit)
                              set the nextG cost to be current cost(update
                              or append)

                              priority value for this node is current cost
                              plus manhattan heuristics value

                              add this node to frontier

                              current is nextG

            elif mode is 1, allow diagonal moves:

                  use diagNeighbor to get all neighbors including diagonal
                  neighbors

                  same computation as mode == 0

                  priority value is current cost plus chebyshev heuristics
                  value

      if found, there is a solution to this grid:
            call makePath to return the result #formats the result graph
      else:
            result is the original grid, no solution found

      return result
```

The greedy function uses the Greedy algorithm to solve for a path, which is very similar to the A*. The only difference is that greed function does not consider the cost so far for each concerning node. Thus, greedy returns the first solution it finds and does not guarantee it is the best one. The following pseudo-code only specifies the part that is different from the *aStar* function.

This function finds the path using the Greedy algorithm, only the part that differs from *aStar* is shown below.
@param *thisGraph*: the input grid as a graph object
@param *mode*: indicates whether diagonal moves are allowed or not

```
def greedy(thisGraph, mode):
      same as aStar, cost_so_far dictionary is not needed

      while frontier is not empty:
            same as aStar

            for nextG in the neighbor list of current node:

                  if nextG is not in came_from: # if nextG is not visited
                        priority value is only defined as heuristic values

                        add nextG into frontier with its priority value

                        add (nextG):(current) into came_from

      return result
```

The *aStar* and *greedy* functions find the path from start to goal. However, this information is embedded in the dictionary *came_from* since it stores all spanned nodes with their preceding nodes. Hence it needs to be processed to generate the correct sequence that contains the nodes on the path only. This is done by the *makePath* function.

This function uses the generate the path from start to goal and mark it on the grid
@param *came_from*: the dictionary contains all spanned nodes with their preceding nodes
@param *start*: a tuple containing the coordinate of the start node
@param *goal*: a tuple containing the coordinate of the goal node
@param *grid*: the input grid in the format of list of strings

```
def makePath(came_from, start, goal, grid):
      set current to goal, start at the goal

      initialize a list called path, to contain tuples of coordinates
      put current into the list

      while current is not start and the node current came from is not
      start:
            find current as a key in came_from
            current is now the value of that key
            #current now becomes where the old current came from

            use string slicing to mark the node on grid "P"

            add current to the path

      return the modified grid, which has all node on the path marked "P"
```

Results

The A* algorithm and greedy algorithm are run on graphs size of [16, 32, 64, 128, 256, 512, 1024], with Manhattan distance as heuristics function when it is only allowed to move in up, down, left and right (mode 0), and Chebyshev distance as heuristics function when it is allowed to move in all directions including the diagonals (mode 1). The process is repeated at each size for 50 times. Since the graph is randomly generated, it cannot be guaranteed that all graphs can be solved, but from the data we collected, the smallest repetition number is 15, so the averaged result will be decent enough.

The two tables below summarize the average time (in milliseconds) to solve the graph at a given size and the number of steps taken to reach the goal for A* and greedy algorithms under two modes. Comparing vertically, for the same graph size, it takes mode 1 less time to find the path, and the length of the path is shorter in comparison to mode 0 overall for both algorithms. The exception to note is that for the Greedy algorithm, more time is required to solve smaller graphs (size = 16 and 32) at mode 0 than at mode 1. Comparing horizontally, greedy algorithm on average finds longer paths than A* in all cases and requires less time than A* with one exception when size = 32 at mode 1. The Greedy algorithm performs exceptionally well when the rule allows movement in the diagonals – on average it only takes 21.58 ms to find a path to the goal on a graph with size 1024. There is a 15.98%

((greed– A*) / A*) increase in step count from the optimal path, but the time required to reach the goal is only 2.38% of the time required by A* for the same graph. When diagonal movement is not allowed, the Greedy algorithm achieved a 54.06% increase in step count from optimal while requiring only 22.24% of the time required by A* to achieve the optimal.

Table 1. Average time to find path and path length on graphs sizes of 16 to 1024 where diagonal moves are not allowed (mode 0).

	A*		Greedy	
Size	Time (ms)	Step	Time (ms)	Step
16	1.11602	13.03571	0.00000	14.10714
32	3.21845	32.03571	0.90320	35.10714
64	7.82422	64.45000	3.34444	69.25000
128	30.09901	157.73333	8.51218	186.80000
256	106.47999	276.04545	28.03983	355.40909
512	585.37575	721.29630	163.73097	1014.85185
1024	2027.40245	1292.47826	450.83997	1991.26087

Table 2. Average time to find path and path length on graphs sizes of 16 to 1024 where diagonal moves are allowed (mode 1).

	A*		Greedy	
Size	Time (ms)	Step	Time (ms)	Step
16	0.31246	8.20000	0.31245	9.04000
32	0.96066	17.56000	1.40569	19.12000
64	4.51647	29.26531	0.63773	33.48980
128	19.22548	65.38000	0.86490	76.08000
256	47.68745	108.44000	4.39962	126.24000
512	222.68797	245.40000	9.63959	293.28000
1024	907.02314	510.74000	21.57769	592.36000

The general performance of the four algorithms is represented as the average amount of time (in milliseconds) required to find one step on the path to the goal. It can be seen from the graph below, the larger the graph is, the slower they are at finding the path – this is because with a larger graph more nodes will be explored even though they are not on the path to the goal. A linear trend is observed in the increase in time/step as graph size increases. A*, in general, finds the path at a slower rate than the greedy algorithm. For A* algorithms, when given more directions allowed to move to, the rate decreases. The opposite trend is observed in the Greedy algorithm. For both algorithms, more nodes will need to be examined when more directions are allowed. For A*, this means more probable paths to keep track of, and more costs associated with these paths needs to be stored and accessed for each iteration. Even though the path may be shorter, the number of nodes explored does not necessarily become less. For the greedy algorithm, unlike A*, it does not need to calculate the cost of the path explored so far and make comparisons based on that, and the heuristics value alone is used to add to the priority queue. While more nodes get examined, it does not add a lot of burdens. For each node it examines, the computation involved is very simple. Since the path is likely to be shorter now, movements in more directions are allowed. The greedy algorithm terminates as soon as a path to the goal is found and is able to succeed sooner. While for A*, more comparison needs to be made. Therefore, decreasing the rate, showing as an increase in time/step.

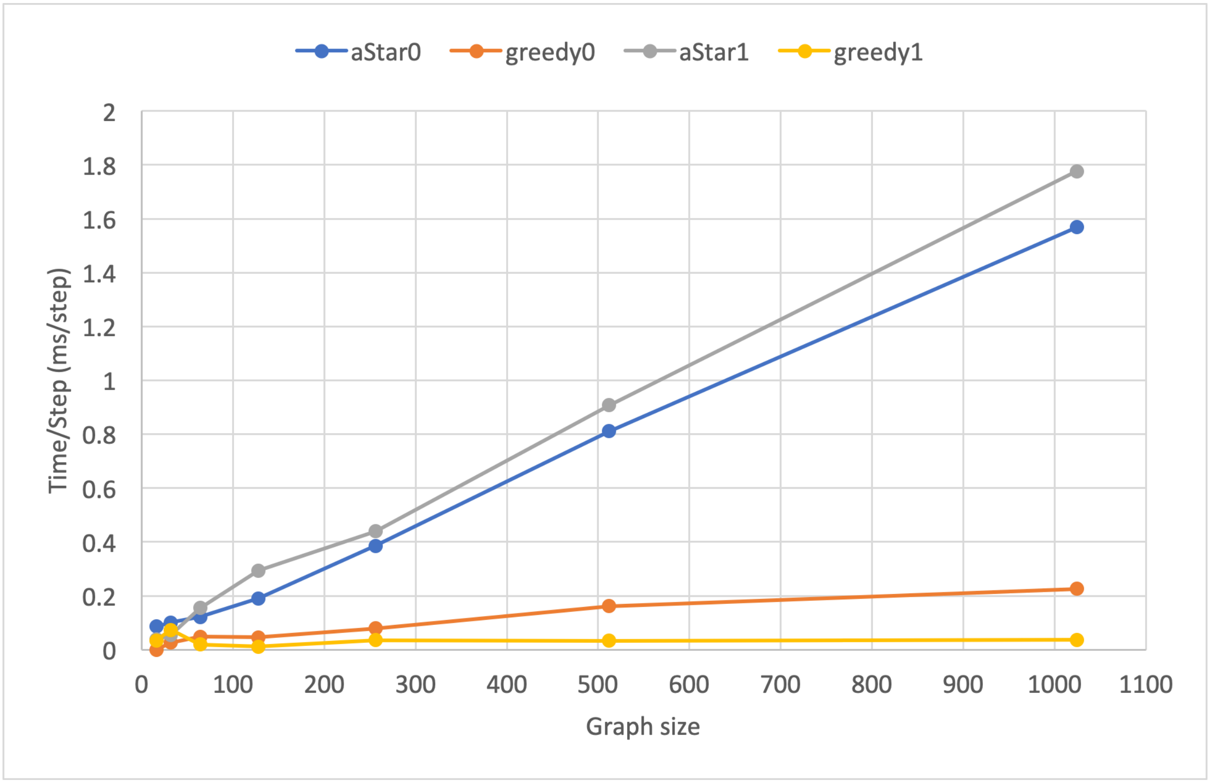

Figure 2. The time/steps ratio for A* and greedy search algorithm with two different modes of movement.

2. Alpha-Beta Pruning

Purpose

The purpose of Alpha-Beta Pruning is to decrease the number of nodes that will be examined and optimizes the minimax function by cutting off branches that are unnecessary to look at. We are using the Minimax algorithm to pick the best move a computer can achieve by looking at the values in the leaf nodes.

Data Structure

The dictionary data structure is used to store the information of the game tree. In our code, the dictionary is named tree. It is believed that using a dictionary data structure would be most appropriate for this problem. Compared to other data structures, dictionaries are known to be an unordered set of key: value pairs. In the case of this problem, each letter that represents a node would be a key in the dictionary, and it is assumed that each node will be unique. In correlation to the key of the dictionary, there will be three types of information stored:

- *kind* stores a string of either MAX or MIN.
- *children* stores the nodes that come after it, whether it be the next node or the leaf node. There can be more than one child, therefore a list is used to contain all children of the key node.
- *has_child_node* stores a boolean to check if the node(s) after it is considered a leaf node.

Technique[2]

```
class alpha_beta:
```

(constructor) It is used to store and keep track of data in a tree and the number of leaf nodes examined.

```
def __init__(self):
      self.tree = {}
      self.LeafNodesExamined = 0
```

The purpose of the *prepareTree* function is to organize input values from alphabeta.txt into a dictionary so the code can use the data to get the *Score* and the *LeafNodesExamined*.
@param *line*: a line from the input file that represents a tree.

```
def prepareTree(self, line):
      Get nodes and edges from a line in alphabeta.txt

      get rid of brackets and commas in nodes
      and for each node
            Store the kind of the node
            Store the children of the node
            Store whether or not it has leaf nodes as children

      get rid of brackets and commas in edges
      and for each edge
            Check to see if the second parameter of the pair is a
            digit. If it is a digit then it should be set to True,
            otherwise False.
            Add next node or value to children.
```

The purpose of the *minmax* function determine the *Score* of the tree, the *LeafNodesExamined*, and what sections to prune.
@param *node*: the current node.
@param *alpha*: stores alpha value. It is initially set to -infinity.
@param *beta*: stores beta value. It is initially set to infinity.
@returns value (or score) of a leaf node.

```
def minmax(self, node, alpha=-float('inf'), beta=float('inf')):
      Save node information into variable, nodeInfo

      If the current node does not have child nodes (meaning that
      they are leaf nodes) then
            If the current node is MAX, then
                  Set the value to -infinity
            Otherwise
                  Set the value to infinity

            Create a temporary variable to store the alpha and beta
            values, alphaTemp, betaTamp

            For the child_node in the children in the node,
                  If the node type is MAX, then
                        If alphaTemp is greater than or equal to
                        betaTemp
                              Return alphaTemp
                        If child_node is greater than alphaTemp
                              Set alphaTemp to equal child_node
                        Increase the number of LeafNodesExamined by 1
                        Set value to alphaTemp
                  Otherwise
                        If betaTemp is smaller than or equal to
                        alphaTemp
                              Return betaTemp
                        If child_node is greater than alphaTemp
                              Set betaTemp to equal child_node
                        Increase the number of LeafNodesExamined by 1
                        Set value to betaTemp
                  Set alpha to alphaTemp
                  Set beta to betaTemp
            Return value

      If the node is MAX, then
            Set the max evaluation to be -infinity, maxEval
            For each child of the node, nodeInfo['children']
                  If beta is smaller than or equal to alpha
                        Exit out of for loop
                  Set Eval = to store the recursive process to get
                  the value of the leaf node
                  Set maxEval to be the bigger value between maxEval
                  and Eval
                  Set alpha to be the bigger value between alpha and
                  maxEval
            Return maxEval
      Otherwise
            Set the min evaluation to be infinity, minEval
            For each child of the node, nodeInfo['children']
                  If beta is smaller than or equal to alpha
                        Exit out of for loop
                  Set Eval = to store the recursive process to get
                  the value of the leaf node
                  Set minEval to be the bigger value between minEval
                  and Eval
                  Set beta to be the bigger value between beta and
                  minEval
            Return minEval
```

The purpose of the *getLeafNodeExamined* function is to get the value stored in *LeafNodesExamined* that is initialized in the constructor.
@returns the number of leaf nodes examined.

```
def getLeafNodeExamined(self):
      Return number of leaf nodes examined
```

The purpose of the *getRoot* function is to get the first node in the tree.
@returns root of the tree.

```
def getRoot(self):
      Return root of the tree
```

The purpose of the *writeFile* function to create a text file to append the results calculated from the alpha-beta pruning.
@param *result*: a concatenated string of results calculated from the *minmax* function.

```
def writeFile(result):
      Create and append the result to a text file named alphabeta_out.txt
```

The purpose of the *main* function is to call functions to perform the task of alpha-beta pruning.

```
def main():
      Open the text file with and for each line
            Prepare tree and save data into a dictionary

            Get the score value from the minmax(getRoot()) function (make
            sure to send the root of the tree into minmax

            Get the leaf nodes examined
            Create result line and send to the writeFile()function to be
            appended to the text file
```

Results

As mentioned in the purpose of this section, Alpha-Beta Pruning optimizes the minimax function by cutting off branches that are unnecessary to look at. This will decrease the computation time. While testing the algorithm, it is evident that depending on how the tree is configured there will be branches and leaf nodes that will not be evaluated. We tested the algorithm with different trees modifying the branching factor and the depth of the tree added to our version of alphabeta.txt file.

Table 3. Summary of number of leaf nodes examined with depth 3 and varying branching factor.

Depth	Branching factor	Total number of leaf nodes	Leaf nodes examined (average of 10 runs)
3	8	512	241.3
3	10	1000	530.9
3	12	1728	874.9

Table 4. Summary of the number of leaf nodes examined with branching factor 10 and varying depth.

Depth	Branching factor	Total number of leaf nodes	Leaf nodes examined (average of 10 runs)
3	10	1000	530.9
4	10	10000	3525.4
5	10	100000	17570.9

Through this data, we can see the difference between the total number of leaf nodes and the leaf nodes that are actually examined (on average). The number of leaf nodes examined is always around half of the total number of leaf nodes or lower. The larger the tree, the more significant the decrease is.

Conclusion

In conclusion, both problems specified for this assignment has been fulfilled. In pathfinding, A* algorithm yielded a better result than the greedy search algorithm in terms of step count, while the greedy algorithm did better in terms of time efficiency. The time/step ratio increased linearly as graph size increased - applying A* algorithm while diagonal moves are allowed yielded the largest growth rate and utilizing greedy search while diagonal moves are allowed produced the smallest. For the Alpha-Beta algorithm, we were able to reduce the number of nodes examined which decreases the process time almost by half while performing a deeper search at the same time. This demonstrates the advantage of being able to eliminate branches in the Alpha-Beta Pruning algorithm compared to a simple minimax algorithm.

References

1. Givigi, S. (2019). Assignment 2. [PDF file]. Retrieved from Queen's University.
2. Lague, Sebastian. YouTube. April 20, 2018. Accessed March 7, 2019. https://www.youtube.com/watch?v=l-hh51ncgDI.