Project 1

CS520 - Intro to Artificial Intelligence

Muzzammil Shahe Ansar (fm501), Harshankar Reddy Gudur (hsg64)

1

Representation and Algorithms

1.1 Grid

The grid is the backbone of the entire simulation, so much time was devoted to making it easy to use. The grid is a simple 2D array of attributes. The entire grid is wrapped in a class with multiple helper functions to make it as easy to use as possible. Each cell in the grid has the following attributes-

- Open
- Alien ID
- Bot Occupied
- Captain Slot (Mainly used for visualization purposes)
- Traversed (Used for path planning and visualization)

This representation was used instead of maintaining separate lists and dictionaries because it was easier to reason about when writing code. Certain optimizations also exist to prevent the recreation of the entire grid when gathering data (Further information can be found in the optimization section). Managing the grid in a class makes the entire experience significantly easier. Attached below is a part of the code that manages the grid-

```
    class GridAttrib:
    __slots__ = ('open', 'bot_occupied', 'traversed', 'alien_id', '
   captain_slot')
    def __init__(self):
        self.open = False
        self.bot_occupied = False
        self.traversed = False
        self.alien_id = -1
        self.captain_slot = False


class Grid:
    def __init__(self, D=30, debug=True):
        self.D = D
        self.grid = []
        self.debug = debug
        self.gen_grid()
```

Since we will be accessing the cells very often during path planning and other operations, we use slots for class attributes in GridAttrib. The Grid class itself stores the dimensions of the grid, a debug variable used to print out details, and the grid itself. The gen_grid function generates the grid according to the given instructions. The function splits the work between itself and another function, gen_grid_iterate. The first function populates the grid and randomly opens a cell. The second function is then called in a loop to iterate on the grid till no more changes can be made. Following this, all the dead ends in the grid are found, and half of them are opened randomly. Our grid is finally finished and ready for use in simulation. We also have a visualization function to help in debugging. It uses a colored text grid to represent the grid and the different agents occupying it. The library 'termcolor' was used for printing with color. The Grid class also has a variety of helper functions-

- `valid_index(self, ind)`
  Returns True if given index exists, False if not
- `get_neighbors(self, ind)`
  Returns all neighboring indices
- `get_open_neighbors(self, ind)`
  Similar to the above but only open neighbors are returned
- `get_untraversed_open_neighbors(self,ind)`
  Similar to the above but only open and untraversed neighbors are returned
- `place_alien(self, ind, alien_id)`
  Sets the alien_id of the cell at ind. Used to place aliens on the grid.
- `remove_alien(self, ind)`
  Removes the alien at ind by setting the alien_id to -1
- `has_alien(self, ind, k=1)`
  Returns True if there are any aliens at ind till depth K (which defaults to 1, i.e. only the cell at ind)
- `place_bot(self, ind)`
  Sets bot_occupied of the cell at ind to True. Used to place a bot in the grid.
- `remove_bot(self, ind)`
  Removes bot from the cell at ind. Traversal of a bot is done by removal followed by placement at one of the neighbors.
- `set_traversed(self, ind)`
  Marks the cell at ind as visited.
- `remove_all_traversal(self)`
  Resets all the traversed attributes to False.
- `get_open_indices(self)`
  Returns a list of all the open indices.
- `get_unoccupied_open_indices(self)`
  Same as above but the indices returned are also unoccupied by aliens and the bot.
- `reset_grid(self)`
  Used to reset the grid so another simulation can be run on it without regeneration.
- `__str__(self)`
  Used to visualize the grid using an ASCII 2D grid.

1.2 Path Finding (BFS)

Multiple algorithms exist for finding paths in a 2D grid. The most popular of these is A^*. We use BFS (Breadth First Search). The workings of the algorithm are described by the following pseudocode snippet-

```
Struct PathTree:
    Data = None
    Parent = None
    Children = []
TreeRoot = PathTree{Bot Location, None, []}
Fringe = Queue([TreeRoot])
DestinationNode = None
While True:
    CurrentNode = Queue.get()
    CurrentLocation = CurrentNode.Data
    if Grid.Traversed(CurrentLocation):
        continue
    if CurrentLocation == CaptainLocation:
        DestinationNode = CurrentNode
        break
    Neighbors = Grid.GetOpenUnvisitedNeighbors(CurrentLocation)
    for n in Neighbors:
        CurrentNode.Children.Add(PathTree{n, CurrentNode, []}
    Queue.Add(CurrentNode.Children)
Return TracePath(DestinationNode)
```

The corresponding Python implementation is part of the bot classes and can be found below-

```python
def plan_path(self):
if self.debug:
    print("Planning Path...")  # If path is empty we plan one
self.path = deque([])
self.grid.remove_all_traversal()
captain_found = False
path_tree = PathTreeNode()
path_tree.data = self.ind
path_deque = deque([path_tree])
destination = None
visited = set()
while not captain_found:
    if len(path_deque) == 0:
        self.grid.remove_all_traversal()
        return
    node = path_deque.popleft()
    ind = node.data
    if ind in visited:
        continue
    visited.add(ind)
    self.grid.set_traversed(ind)
    if ind == self.captain_ind:
        destination = node
        break
    neighbors_ind = self.grid.get_untraversed_open_neighbors(ind)
    for neighbor_ind in neighbors_ind:
        # Add all possible paths that do not hit an alien
        if not self.grid.has_alien(neighbor_ind):
            new_node = PathTreeNode()
            new_node.data = neighbor_ind
            new_node.parent = node
            node.children.append(new_node)
    path_deque.extend(node.children)
self.grid.remove_all_traversal()
if self.debug:
    print("Planning Done!")
reverse_path = []
node = destination
while node.parent is not None:
    reverse_path.append(node.data)
    node = node.parent
self.path.extend(reversed(reverse_path))
for ind in self.path:
    self.grid.set_traversed(ind)
if self.debug:
    print("Planned Path")
    print(self.grid)
```

Originally, we were supposed to work with BFS temporarily and then move onto A^*, but with time, we found that BFS worked competently. We had no reason to switch to A^* since it is a more performant algorithm, rather than a more correct one. We found BFS performant enough for our purposes (Although there is more to this decision).

1.3 The Bot

The bots are represented by classes. At any time, it is assumed that only one bot occupies the grid. Each type of bot is a separate class. Bot 1 through 3 are very similar to each other except for a few differences in the plan_path and move methods. We will start with an in-depth explanation of bot 1. We will then build upon it to understand and analyze the remaining 3 bots.

1.3.1 Bot 1

Bot 1 is the simplest of all bots but the algorithms used and the structure of the class itself are very similar to all other bots. For this reason, we will explore this bot in detail. Attached below is part of the Bot1 class definition.

```
class Bot1:
    def __init__(self, grid, captain_ind, debug=True):
        self.grid = grid
        self.captain_ind = captain_ind
        self.ind = random.choice(self.grid.get_open_indices())
        self.grid.place_bot(self.ind)
        self.path = None
        self.debug = debug
```

Bot 1 stores a reference to the grid to keep track of all the aliens. It also keeps track of the location of the captain separately. It's own location is then randomly chosen from all open indices and the grid is then modified to reflect that. The most important function in this class is the move function-

```
def move(self):
    if self.path is None:
        self.plan_path()
    if len(self.path) == 0:
        if self.debug:
            print("No path found!")
        return

    next_dest = self.path.popleft()
    self.grid.remove_bot(self.ind)
    self.ind = next_dest
    self.grid.place_bot(self.ind)
```

If the path planning has never been done before, the path is planned using BFS. If the length of the path is 0 (This condition implies that no path was found to the captain), the bot stays in its original position for the rest of the simulation. If a path is found, it follows that same path every time the function is invoked. This bot does not try to take into account the dynamic nature of its two-dimensional world and chooses to follow the one path it found initially. This works surprisingly well when the number of aliens with respect to the total number of cells is low but it starts failing very rapidly as the number of aliens increases.

1.3.2 Bot 2

Bot 2 is almost a carbon copy of Bot 1, except for one change in the move function-

```
def move(self):
    self.plan_path()
    if len(self.path) == 0:
        if self.debug:
            print("No path found!")
        return
    next_dest = self.path.popleft()
    self.grid.remove_bot(self.ind)
    self.ind = next_dest
    self.grid.place_bot(self.ind)
```

This move function always plans a new path every time it is invoked. This means that the bot is now able to react to new developments in the grid. If an alien crosses its previous path, the bot can now find a new one that avoids the alien. If no path was found, like before, we choose to stay still and not move anywhere. This massively boosts the success rate in all cases.

1.3.3 Bot 3

Bot 3, again, is very similar to Bot 2 except for a few changes in the move and the plan_path function.

```python
def plan_path(self, k=2):
    if self.debug:
        print("Planning Path...")  # If path is empty we plan one
    ...
```

```python
def move(self):
    self.plan_path(2)
    if len(self.path) == 0:
        if self.debug:
            print("Reverting...")
        self.plan_path(1)
        if len(self.path) == 0:
            if self.debug:
                print("No path found")
            return
    next_dest = self.path.popleft()
    self.grid.remove_bot(self.ind)
    self.ind = next_dest
    self.grid.place_bot(self.ind)
```

Bot 3 has a modified version of BFS used by the other bots. The above to bots use has_alien with 'k=1' during path planning. This bot uses has_alien with 'k=2'. This means that when new cells are added to the path, not only is the cell itself checked for aliens, but its neighbors are checked for it too. This effectively always keeps a distance of 2 between our path and the aliens. Practically speaking, no matter what happens, an alien will not be able to capture our bot next turn if we can find this path. Since this move function plans the path every turn too, if we can find a path like this every turn then that implies confirmed success if there are no limits on the number of turns. Of course, in many cases, we are unable to find such paths. In such cases, we revert to Bot 2's behavior of dynamically finding a path with k=1 and choosing to stay still if one cannot be found.

Bot 4 and Bot 5 will have their own dedicated sections

1.4 Aliens

The class Aliens is used to represent each alien. It is a very simple class so the code is shown here in its entirety-

```python
class Alien:
    # This alien_id is used to keep track of every alien
    alien_id = 0
    def __init__(self, grid):
        self.grid = grid
        indices = self.grid.get_unoccupied_open_indices()
        ind = random.choice(indices)
        self.ind = ind
        self.alien_id = Alien.alien_id
        self.grid.place_alien(ind, Alien.alien_id)
        Alien.alien_id += 1

    def move(self):
        # Get all possible locations for the alien
        neighbors = self.grid.get_open_neighbors(self.ind)
        # Filter out the ones that are occupied by other aliens
        neighbors_without_aliens = [neighbor for neighbor in neighbors if self.grid.grid[neighbor[1]][neighbor[0]].alien_id == -1]
        # Randomly choose any of the locations
        if len(neighbors_without_aliens) > 0:
            rand_ind = np.random.randint(0, len(neighbors_without_aliens ))
            self.grid.remove_alien(self.ind)
            self.ind = neighbors_without_aliens[rand_ind]
            self.grid.place_alien(self.ind, self.alien_id)
```

When a new alien is created, it randomly chooses one of the unoccupied open cells in the grid. The cell at that position in the grid then has its alien_id changed to the alien_id of the new alien.
The movement is almost as simple. All it does is to get the unoccupied neighbors again, but this time we ignore the bot and only consider the aliens. We want our alien to capture the bot so avoiding the cells at which our bot is present is not ideal. We then choose one of these neighbors randomly and move to it.

1.5 Optimizations

To begin talking about optimizations, we need to talk about performance issues first. The version of BFS used initially had a bug in it because of which it would revisit certain nodes. This causes major memory and performance issues. We optimized much more than necessary because of these circumstances. Some of the techniques discussed here are overkill. Regardless, it makes for an interesting discussion.

1.5.1 Deque

Most implementations naively use Python's Queue. While it does the job, it is not the most efficient for this particular application. It is intended for interprocess communication. Quite a bit of overhead can be avoided by simply using Python's Deque instead.

1.5.2 Avoiding Needless Grid Generation

The grid is surprisingly demanding to generate. The iterative steps can go on for quite some time. If we are collecting 10 data points for a plot, and each data point involves about 100 iterations, across 4 bots, then we are generating the grid 4000 times. This adds a fair bit of time which can be easily shaved off. Instead of generating a new grid for all 4 bots, we keep the same grid for the 4 bots and just randomize the placement of the different agents in the grid across the 4 bots. This simple technique reduces the impact of grid generation by 1/4. It can be seen in code here -

```
...
for i in range(iters):
    for K in range(K_start, K_end, K_skip):
        self.gen_grid()
        for b in range(NUM_BOTS):
            self.grid.reset_grid()
            self.gen_world(K)
            ...
```

As shown above, the grid remains the same across the 4 bots, but the grid itself is reset without changing the structure. The world is then generated again (Placement of aliens and so on). This results in a massive improvement in performance while preserving some amount of randomness across the bots.

1.5.3 Collect Data for a Sparser Range

Our initial implementation incremented the value of K across a range. This was fine at the start, but of course, collecting 100 data points with 100 iterations each across 4 bots is a huge amount of computation for my modest computational capabilities. Instead of using a single end value for K, we chose to use a range of the form - (Start, End, Skip) to reduce the amount of computations that were taking place.

1.5.4 Using Slot Classes

Python's normal method of accessing attributes in a class has a decent amount of performance and memory overhead. Using slots for attributes (Can be seen in the definition of the GridAttrib class above) cuts down on this overhead.

1.5.5 Using Multiprocessing

This is not necessary. It is incredibly overkill for the problem at hand but because we wanted to speed up our flawed implementation of BFS (of which we were of course unaware), we chose to parallelize the generation of data. The class WorldState is used for this purpose. This allows each process to have a separate copy of a Grid, Aliens, and other necessary information to run the simulations independently. Again, it is not a difficult optimization to do so we will not discuss it further. If interested, refer to the function gather_data in the class World, and the class WorldState.

2

Bot 4: Design, Implementation and Analysis

The design, implementation and its various iterations and observations are stated here.

2.1 Initial Design and Observations

To understand where we are, we must look at where we started. It is as true for bot design as it is for life. The initial idea of Bot 4 begins with the deficiencies of Bot 3 and trying to make up for it. We have already discussed Bot 3, but let's delve deeper into its failure modes.

2.1.1 Bot 3 and its deficiencies

Bot 3 decides between 3 actions-

1. Follow a guaranteed safe path.
2. Follow a risky path.
3. Stay still.

Action 1 is a strong choice since it does not fail by definition. Put differently, whatever we change must be in the other two actions. Let us look at Action 3 first, since it is the simplest. Staying still can be a good idea if we are in a safe position and an awful decision if we are in a risky position surrounded by aliens on multiple sides. This leads us to the first improvement - evasion. We can choose to evade instead of staying still. We will come back to this later. Action 2 is just as important. The question is, do we take this risky path or stay still? Again, it's better to take the risk in certain situations (very close to the captain, for example) and better to fall back during others. The initial idea was to use a risk_limit. Put succinctly, these are the areas where we can improve Bot 3 -

- Evasion instead of staying still
- Deciding which action to take

2.1.2 The first blueprint to Bot 4

2.1.2.1 Tackling evasion

There are innumerable ways of implementing evasion. The method we settled on was the simplest one we could think of - find the closest alien within a predetermined distance and move in the direction that takes the furthest away from it.

```python
def evade(self, offset = 2):
    # From the current position
    # Search +/- offset for aliens
    # Evade in a way that escapes the closest alien
    x_offsets = [i for i in range(-offset, offset + 1)]
    y_offsets = [i for i in range(-offset, offset + 1)]
    aliens = []
    for j in y_offsets:
        for i in x_offsets:
            if i == 0 and j == 0:
                continue
            ind = (self.ind[0] + i, self.ind[1] + j)
            if self.grid.valid_index(ind) and self.grid.has_alien(ind):
                aliens.append((self.ind[0] + i, self.ind[1] + j))
    if not aliens:
        # If there are no aliens nearby just wait
        return
    # Sort by distance
    aliens.sort(key=self.distance) # Euclidean distance
    closest_alien = aliens[0]
    # Find the direction to move in
    ideal_move_dir = (self.ind[0] - closest_alien[0], self.ind[1] - closest_alien[1])
    # Returns a clamped direction
    def condition_dir(d):
        if d == 0:
            return 0
        elif d > 0:
            return 1
        elif d < 0:
            return -1
    # Since diagonal movements are not allowed we use this bit of code to determine which
    # direction to move in. If the direction we want to move in is inaccessible, we return
    if ideal_move_dir[0] == ideal_move_dir[1] or abs(ideal_move_dir[0]) > abs(ideal_move_dir[1]):
        new_ind = (self.ind[0] + condition_dir(ideal_move_dir[0]), self.ind[1])
        if self.grid.valid_index(new_ind) and self.grid.grid[new_ind[1]][new_ind[0]].open:
            self.grid.remove_bot(self.ind)
            self.ind = new_ind
            self.grid.place_bot(self.ind)
        else:
            return -1
    else:
        new_ind = (self.ind[0], self.ind[1] + condition_dir(ideal_move_dir[1]))
        if self.grid.valid_index(new_ind) and self.grid.grid[new_ind[1]][new_ind[0]].open:
            self.grid.remove_bot(self.ind)
            self.ind = new_ind

            self.grid.place_bot(self.ind)
        else:
            return -1
    return 0
```

We first search in the vicinity of the bot to find out all close-by aliens. There is no point in trying to run away from an alien on the other side of the grid after all. What is 'close by' is decided by the parameter offset. Once we have these aliens, we sort them by distance to the bot. We use Euclidean distance here. We then choose the closest bot and calculate a vector from it to the bot. This is the direction we want to move in, ideally. Of course, since we are restricted to only 4 directions, we have to convert it into a direction we can use. We compare both components of our vector. If they are equal, or the x-coordinate is higher, we move left or right. Otherwise, we go up or down. If the place we want to move to is closed or has another alien, or if we are at the edges of the grid, then we just stay still.
This evade function is functional but we can do much better. From the below figure, we can see that around K=35, Bot 4 falls below Bot 3 efficiency. Not ideal. We will get to that when we discuss improvements.

2.1.2.2 Making good(?) decisions

The next problem to tackle is to effectively use all these actions when required. A simple linear model is used (which carries over to the final model). The code for the move function is listed here-

```python
def move(self):
    self.plan_path(2)
    if len(self.path) == 0:
        # The closer the captain is the higher the risk
        # I will scale it such that if the captain is close,
        # the bot will take high risk
        # If far away, it should focus more on evading
        # We use a linear model
        d = self.distance(self.captain_ind)
        # 0.01 because if risk_factor is 0 I want to only evade
        alpha = random.uniform(0.01, 1.0)
        m = -0.086
        c = 1.06
        self.risk_factor = m * d + c
        if(alpha > self.risk_factor):
            self.evade()
        else:
            print("REVERT")
            if self.debug:
                print("Reverting...")
            self.plan_path(1)
            if len(self.path) == 0:
                if self.debug:
                    print("No path found")
                return
        return
```

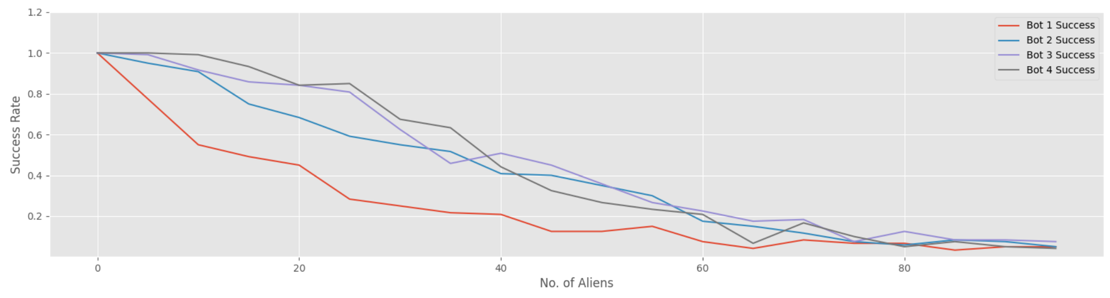


Figure 2.1: Effectiveness of the old Bot 4

If we find a path with k=2, of course, we will choose it every single time. It is safe and it takes us directly to the captain. If it doesn't, we need to decide between taking a risky path and evasion. We decide using a random number. This number(alpha) falls between 0.01 and 1.0. We then use a linear model to compute a risk_factor. If alpha is greater than our risk_factor then we evade, otherwise, we plan the risky path. If we do not find one we stay still. This works, but it has some bugs that need to be fixed and some further modifications that can be made to make it better.

2.1.3 The final Bot 4

Although I call it final, it can still be improved and tuned much further. Working with time constraints, we find this to be a reasonable final Bot 4. We will discuss the improvements to both parts of our previous Bot 4.

2.1.3.1 Improved Evasion

The new version of Bot 4 has significantly better evasion. It does not take into account just one alien. It looks for all the aliens in a sub-grid and chooses the neighbor with the lowest chance of encountering an alien. The code for evasion is listed below. The function is creatively called evade2.

```python
# Calculates a danger level so that we can
# evade in the direction that lets us be in
# the safest possible position
def calculate_danger(self, ind, offset):
    danger = 0
    for j in range(-offset, offset):
        for i in range(-offset, offset):
            x = ind[0] + i
            y = ind[1] + j
            if self.grid.valid_index((x, y)) and self.grid.has_alien((x, y)):
                # The closer the more dangerous. The 0.001 at the end is to avoid div by 0
                danger += 1/(abs(i) + abs(j) + 0.001)
    return danger

def evade2(self, offset=2):
    possible_positions = self.grid.get_open_neighbors(self.ind)
    # The current position might be the safest.
    # We may not want to move in that case so we
    # add it to the possible positions
    possible_positions.append(self.ind)
    dangers = [self.calculate_danger(p, offset) for p in
   possible_positions]
    min_position_i = min(enumerate(dangers), key=lambda x: x[1])[0]
    next_position = possible_positions[min_position_i]
    self.grid.remove_bot(self.ind)
    self.ind = next_position
    self.grid.place_bot(self.ind)
```

This method of evasion is, conveniently, much simpler to understand. We simply go through a small grid centered around our possible locations. We use this sub-grid to calculate a danger value for each of these positions. We then choose the position with the lowest danger. It performs much better than our previous method of evasion. It also automatically chooses to stay still if that is the position with the lowest danger. This means that there is no situation where we would manually choose to stay still. We can always call evade2, and the function will make sure that the bot stays still if that is the safer option. The calculate_danger function is equally simple. We take the inverted Manhattan distance and add it to the variable danger. We want this value to scale up with the number of aliens around and the closeness of these aliens to our bot.

2.1.3.2 Making better decisions

Our new evade method allows us to make better decisions at every point. We have also fixed a few issues from the previous iteration. The new code is listed below.

```python
def move(self):
    self.plan_path(2)
    if len(self.path) == 0:
        # The closer the captain is the higher the risk
        # I will scale it such that if the captain is close,
        # the bot will take high risk
        # If far away, it should focus more on evading
        # We use a linear model
        d = self.distance(self.captain_ind)
        # 0.01 because if risk_limit is 0 I want to only evade
        alpha = random.uniform(0.01, 1.0)
        m = -0.086
        c = 1.06
        self.risk_limit = m * d + c
        # Clamp the value to [0, 1]
        if self.risk_limit > 1:
            self.risk_limit = 1
        elif self.risk_limit < 0:
            self.risk_limit = 0
        if(alpha > self.risk_limit):
            self.evade2()
            return
        else:
            print("REVERT")
            if self.debug:
```

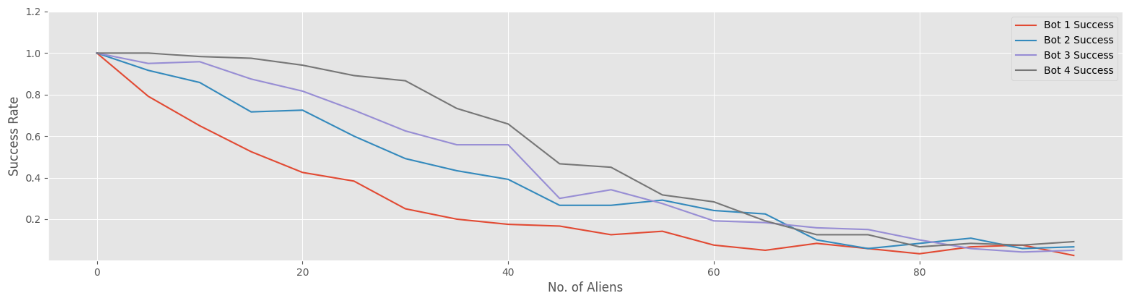


Figure 2.2: Effectiveness of the new Bot 4

```
                print("Reverting...")
            self.plan_path(1)
            if len(self.path) == 0:
                self.evade2()
                if self.debug:
                    print("No path found")
                return

    next_dest = self.path.popleft()
    self.grid.remove_bot(self.ind)
    self.ind = next_dest
    self.grid.place_bot(self.ind)
```

Much of it is the same as before. We fixed a bug with the previous version with the clamping. The previous value of risk_factor, now risk_limit, was unbounded on both ends. We want it to be between 0 and 1 since we are comparing alpha to it. More importantly, with this code we never stay still. If we choose not to take a risk, we are evading. If we choose to take a risk but the path does not materialize, we evade. In both cases, if staying still is the safer option, our function will automatically stay still. The model here can be improved much more though. We are, as of now, not taking the total number of aliens into account. Due to time constraints, we could not explore that venue, but we strongly feel that that would result in a better risk_limit calculation and make this model more effective across a wider range of values of K.

3 Bonus Questions

3.1 Bonus Question 1

Grid generation can be approached in many ways. We take a simple approach. There are two ways in which we can improve the performance of the bots in a grid by changing the grid itself-

- Have more alternative paths
- Restrict the movements of aliens

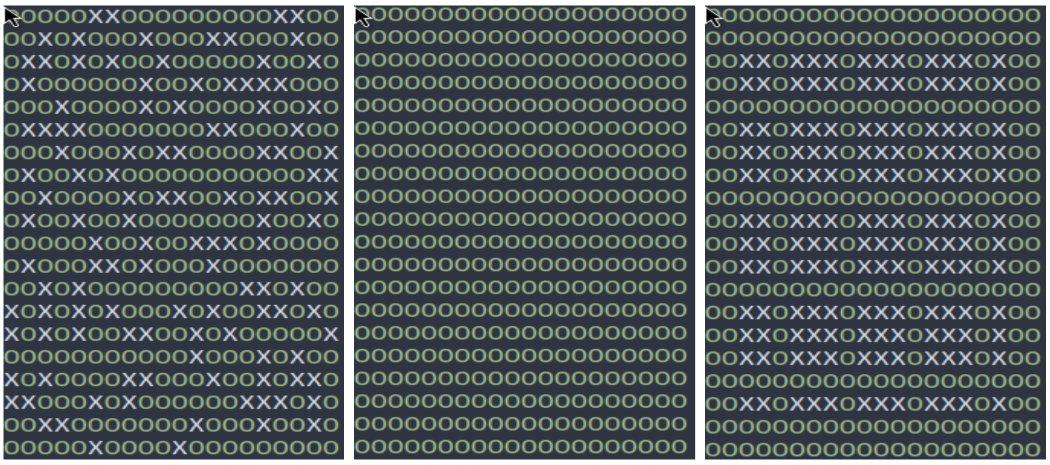
We can apply these ideas and choose to create a completely open grid and a grid that is full of corridors crisscrossing each other. The grids are the exact same as the original grid described before, but the gen_grid function is modified in each.
The gen_grid function for an open grid is-

```
    # Grid generation happens here
    def gen_grid(self):
        for j in range(self.D):
            row = []
            for i in range(self.D):
                g = GridAttrib()
                g.open = True
                row.append(g)
            self.grid.append(row)
```

The open grid is trivial to generate - set all cells to open.
The gen_grid function for a corridor grid is-

```
def gen_grid(self):
    block_dim = (self.D - 8)//3
    for j in range(self.D):
        row = []
        for i in range(self.D):
            g = GridAttrib()
            if i <= 1 or i >= self.D - 2 or j <= 1 or j >= self.D -
    2:
                g.open = True
            else:
                g.open = False if (i % block_dim != 0) and (j %
  block_dim != 0) else True
            row.append(g)
        self.grid.append(row)
```

We make sure that the edges of the grid are always surrounded by open cells of width 2. The corridors are then generated by simply using a modulo function on both axes. We then run simulations on these grids with all 4 bots to understand how they perform. The relevant plots can be seen below. As can be seen from



```
OOOOOXXOOOOOOOOOXXOO
OOXOXOOOXOOOXXOOOXOO
OXXOXOXOOXOOOOOXOOXO
OXOOOOOOXOOXOXXXXOOO
OOOXOOOOXOXOOOOXOOXO
OXXXXOOOOOOOXXOOOXOO
OOOXOOOXOXXOOOOXXOOX
OXOOXOXOOOOOOOOOOOXX
OOXOOOOXOXXOOXOXXOOX
OXOOXOOXOOOOOOOXOOXO
OOOOOXOOXOOXXXOXOOOO
OXOOOXXOXOOOXOOOOOOO
OOXOXOOOOOOOOOXXOXOO
XOXOXOXOOOXOXOOXXOXO
XOXOXOOXXOOXOXOOOOOX
OOOOOOOOOOOXOOOXOXOO
XOXOOOOXXOOOXOOXOXXO
XXOOOXOXOOOOOOXXXOXO
OOXXOOOOOOOXOOOXOOXO
OOOOOXOOOOXOOOOOOOOO
```

(a) Normal Grid, D=20

```
OOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOO
```

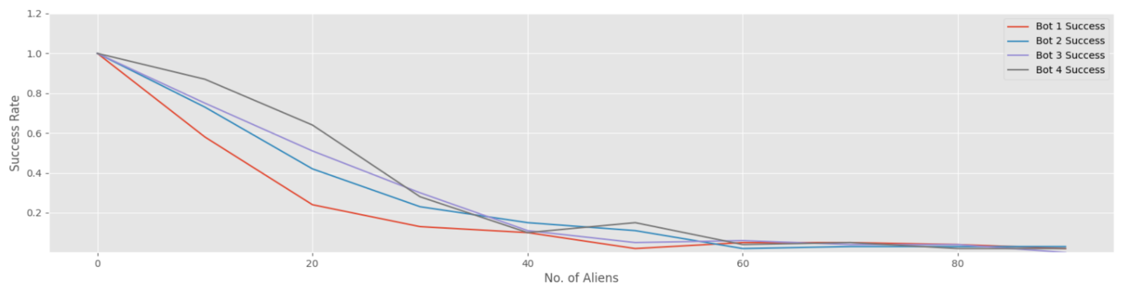
(b) Open Grid, D=20

```
OOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOO
OOXXOXXXOXXXOXXXOXOO
OOXXOXXXOXXXOXXXOXOO
OOOOOOOOOOOOOOOOOOOO
OOXXOXXXOXXXOXXXOXOO
OOXXOXXXOXXXOXXXOXOO
OOXXOXXXOXXXOXXXOXOO
OOOOOOOOOOOOOOOOOOOO
OOXXOXXXOXXXOXXXOXOO
OOXXOXXXOXXXOXXXOXOO
OOXXOXXXOXXXOXXXOXOO
OOOOOOOOOOOOOOOOOOOO
OOXXOXXXOXXXOXXXOXOO
OOXXOXXXOXXXOXXXOXOO
OOXXOXXXOXXXOXXXOXOO
OOOOOOOOOOOOOOOOOOOO
OOXXOXXXOXXXOXXXOXOO
OOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOO
```

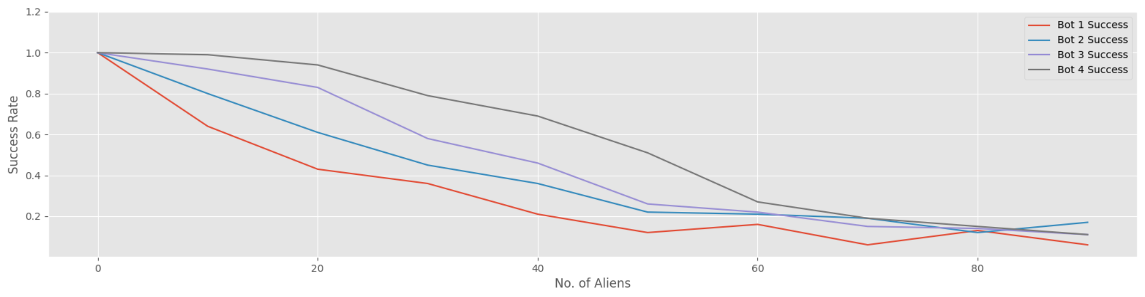
(c) Corridor Grid, D=20

Figure 3.1: Grids Visualized

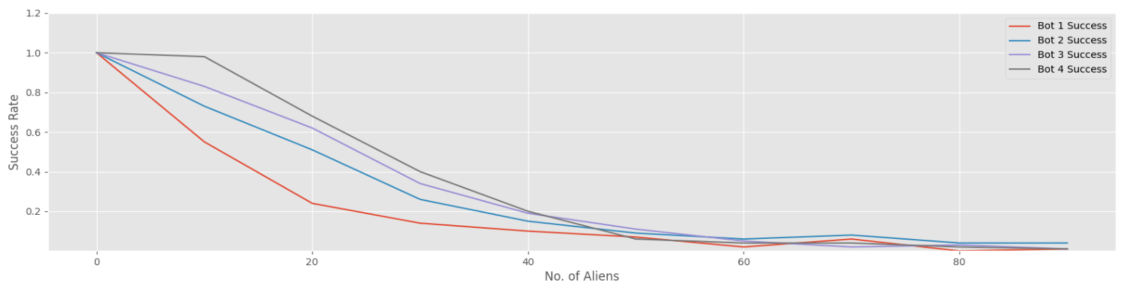
the relevant plots, the success rates between a normal grid and the corridor grid are very similar. It is not significant enough to consider it to be a better grid. The open grid however works well. There is a significant increase in the success rate and it stays that way. Therefore, building a grid that provides for as many alternative paths as possible is better.



(a) Simulation Plot for a normal grid, D=20, 100 runs



(b) Simulation Plot for an open grid, D=20, 100 runs



(c) Simulation Plot for a corridor grid, D=20, 100 runs

Figure 3.2: Comparison of different grids

3.2 Bonus Question 2

We implement a compute limit by counting the number of loop iterations in the BFS method-

```
def plan_path(self):
    if self.debug:
        print("Planning Path...")  # If path is empty we plan one
    self.path = deque([])
    self.grid.remove_all_traversal()
    captain_found = False
    path_tree = PathTreeNode()
    path_tree.data = self.ind
    path_deque = deque([path_tree])
    destination = None
    visited = set()
    compute_counter = 0
    while not captain_found:
        if len(path_deque) == 0 or compute_counter >= COMPUTE_LIMIT
    :
            self.grid.remove_all_traversal()
            return
        compute_counter += 1
        node = path_deque.popleft()
        ind = node.data
        ...
```

Bot 5 is simply Bot 4 with this compute limit attached to it. We could further improve it with heuristics in case no path was found but performance is satisfactory for a reasonable compute limit, as we will see. The remaining bots also have a similar restraint. Bot 3 and Bot 5 both are only allowed half of COMPUTE_LIMIT because they tend to invoke plan_path twice per turn.

```
def plan_path(self, k=2):
    if self.debug:
        print("Planning Path...")  # If path is empty we plan one
    self.path = deque([])
    self.grid.remove_all_traversal()
    captain_found = False
    path_tree = PathTreeNode()
    path_tree.data = self.ind
    path_deque = deque([path_tree])
    destination = None
    compute_counter = 0
    visited = set()
    while not captain_found:
        if len(path_deque) == 0 or compute_counter > COMPUTE_LIMIT
    //2:
            self.grid.remove_all_traversal()
            return
        compute_counter += 1
        node = path_deque.popleft()
        ...
```

4

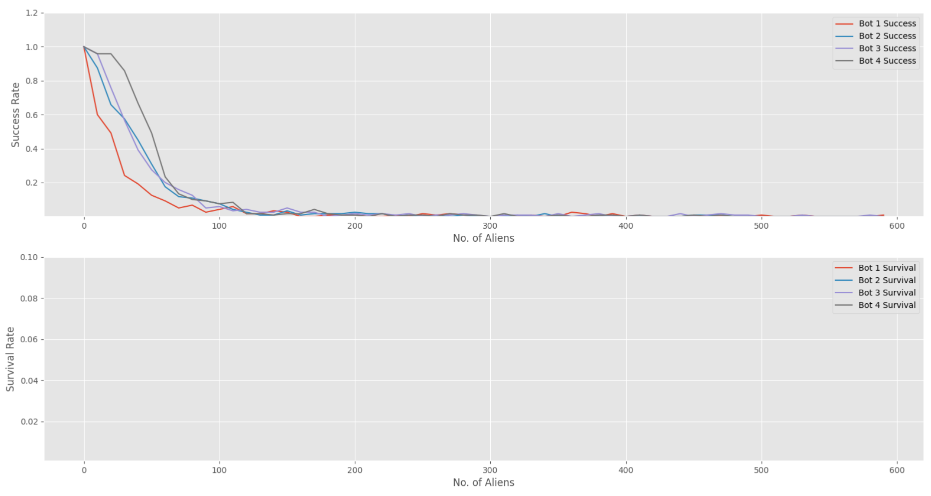
Simulation Results

4.1 Simulation Rules

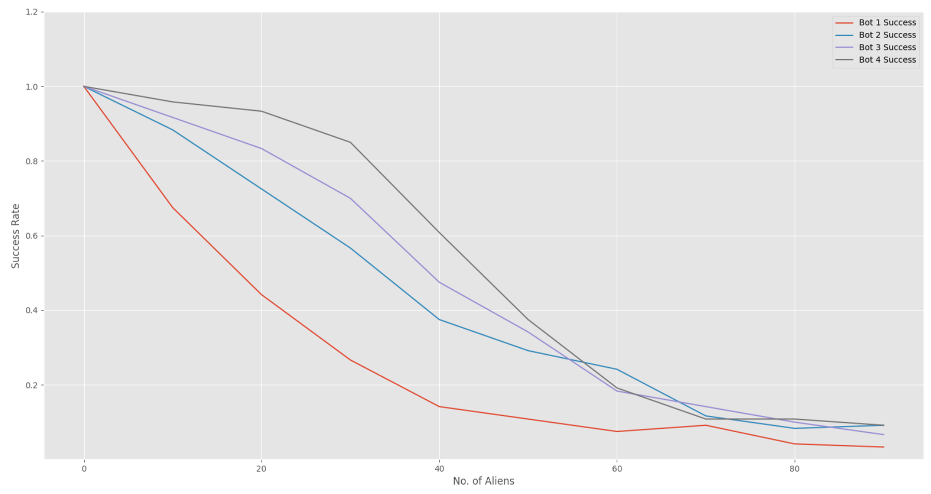
Each bot is simulated in a randomly generated grid of side D=30. These bots are simulated for 1000 steps. If the bot does not die within those 1000 steps then it is considered a survival. If it reaches the captain, it is a success, and if it gets caught by an alien, it is a capture. Each data point is simulated 100 times.

4.2 Plots and Analysis

As can be seen from figure 4.1a, no matter how many aliens are on the grid, a few runs always succeed. This may be due to a miraculous placement of no aliens or the bot spawning close to the captain. So it is better to think of a cutoff at which we can stop generating data. In our case, 0.1 will be the cutoff. For Bot 1, K_{cutoff} is around 60. For Bot 2, Bot 3, and Bot 4, K_{cutoff} is around 80. We shall therefore generate data till K=100 for our plots. Another thing to note is that the survival rate is almost always 0, outside of small blips here and there. We shall therefore not plot the survival rates for now.
Looking at figure 4.1b, we can see that Bot 1 performs the worst. It has a steep descent as K increases and is only viable when K is a small value. It drops below 0.5 before $K \approx 18$. Bot 2 performs significantly better. The success rate decreases relatively slower with K and drops below 0.5 at $K \approx 32$. Bot 3 is a further improvement in most cases and drops below 0.5 at $K \approx 38$. Bot 4 is the best-performing bot under $K = 60$ and drops below 0.5 at $K \approx 45$. Bot 3 drops below Bot 2 in certain scenarios. This can be attributed to variations per run. Throughout the entire graph, the success rates of both Bot 3 and Bot 2 are very close after $K = 60$. This makes sense, Bot 3 will start behaving like Bot 2 as K increases because it will not be able to find a path that will make sure that every alien is 2 cells away. Bot 3 will start behaving more like Bot 2 after this point. We can observe this trend long term in figure 4.1a We can also see that Bot 4 is less effective than Bot 2 and Bot 3 after $K = 60$. We can trace this back to the linear model we use for the risk_limit. Since we do not take into account K when computing the risk_limit we can only scale so much. As the number of aliens in the grid increases, we want our bot to behave more evasively and only take risks when the captain is very close. This would not be

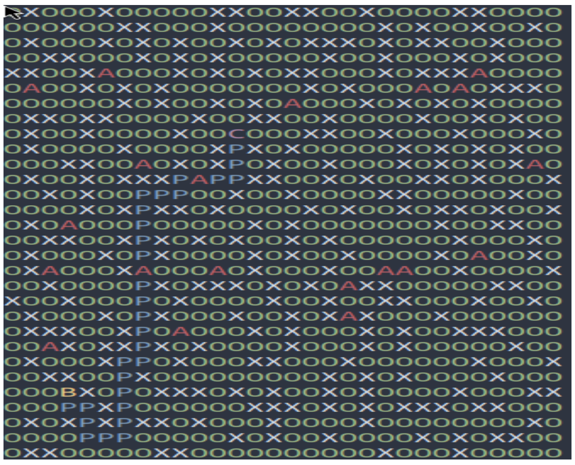


(a) Plot of data, K=(0, 600)

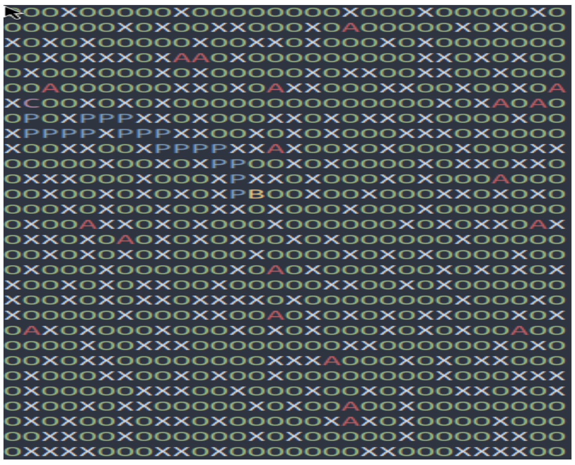


(b) Plot of data, K=(0, 100)

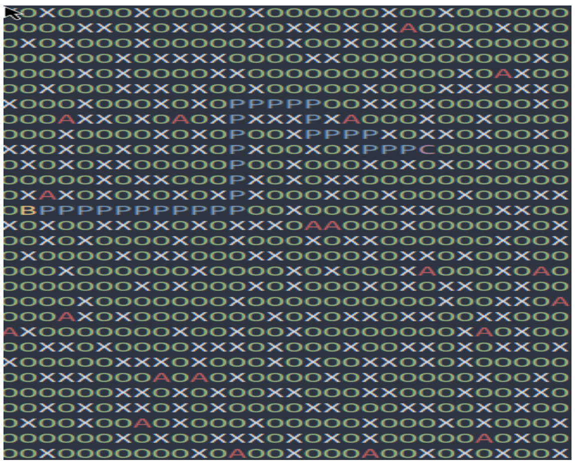
Figure 4.1: Plots of all bots



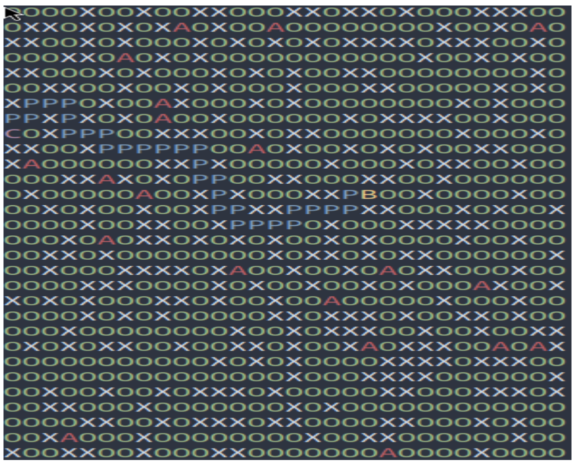
(a) Bot 1 Grid



(b) Bot 2 Grid



(c) Bot 3 Grid

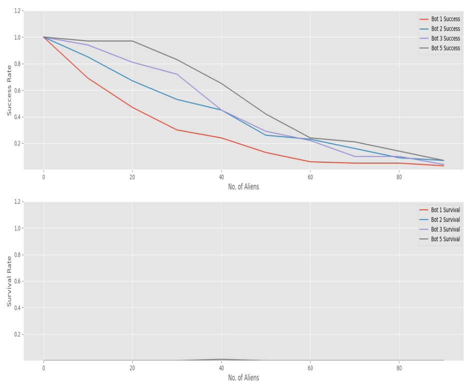


(d) Bot 4 Grid

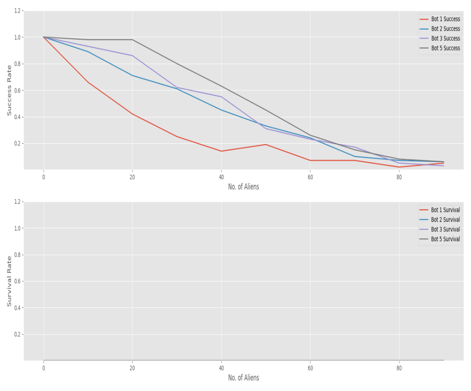
Figure 4.2: Grid View of all bots

4.2.1 Bonus Question 2

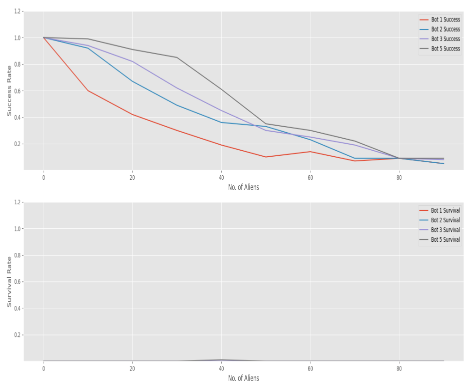
We can see from these plots a simple trend - the more restrained the compute limit is, the more we tend to survive instead of succeed. This is true for all bots, but especially so for Bot 5 which takes evasive actions when a path is not found. This trend is not properly visible till we hit a compute limit of 1000 (Figure **??**). We can see the performance beginning to deteriorate compared to our normal plots above, and the higher compute limit plots. The lowered success rate is much more visible when we examine figure **??**. Bot 5 can be improved even further by making use of heuristics instead of evasion alone.



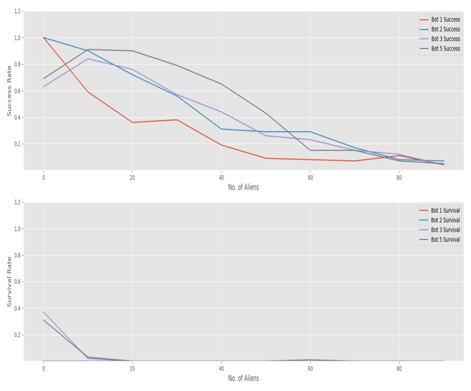
(a) Compute Limit = 10K



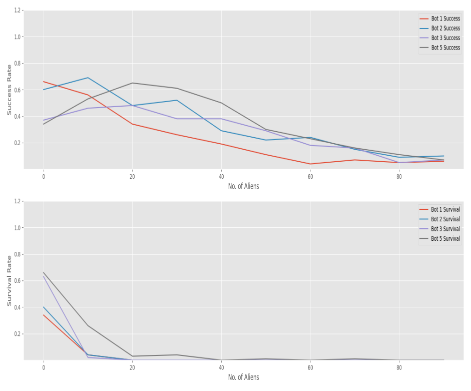
(b) Compute Limit = 5K



(c) Compute Limit = 2.5K



(d) Compute Limit = 1K



(e) Compute Limit = 500

Figure 4.3: Comparison of different compute limits

5
Contributions

The project was equally contributed to by both members. The two members of the team are roommates. Every step from brainstorming ideas to implementing code was done together.