Option 2:

**Introduction:**
I chose to focus my assignment on 2-dimensional Turing machines. Looking through the report specifications, this option struck a chord with me and sounded like it could be some fun. While the example given in the specifications seemed interesting enough, I thought it could be taken a lot further with some though and creativity.

After trying to figure out where I could take this, I thought to try my hands on a rudimentary Turing battle simulator. Given that Turing machines can have multiple states which can affect the output of a specified input, a basic artificial intelligence system that makes use of a 2d Turing system seemed to be a challenging but good direction to handle the problem specified by the assignment, leading to the creation of Turing Battle.

**Implementation Details:**
I chose a screen dimension of 10 rows and 10 columns and an image size of 64x64 pixels. This leads to an overall program resolution of 640 by 640 pixels.

Implementing the system I had in mind proved to be more challenging than I expected, namely because, for an interesting outcome, there had to be many states and instructions given to the machine. This required a lot of thinking and debugging to get the system performing properly. Given the complexity of the system, it's quite difficult to explain it without graphical examples. As such, I will endeavour to give a clearer explanation of the purpose of my implementation when going over the first of four examples.

In the implementation, which uses Java, the data is stored on a 1-dimensional array. While a 2-dimensional array may have been a cuter choice in simulating a 2-dimensional Turing machine, a 1-dimensional array is the same with a defined row and column size in that, map[(row * 2) + 5] where row is the the number of rows, is equivalent to map[2][5].

The heart of the system is a class (named TuringExecutor) which is responsible for the functioning of the Turing machine. This class, which contains the head, retrieves the instructions to perform (what, if anything, to change the value contained at the head to, what state to change to, and where to move the head) based on the current value at the head and the current state the system is in. The internal system uses tape alphabet and not images; the GUI, built on top of the system using the Model-View-Controller paradigm is responsible for drawing a visual representation of the internal 'map'.

Note: I feel that explaining how the application handles the head moving off the screen can be better explained once a better understanding of the system is achieved. For this reason, I will further discuss this issue later on in the discussion.
Examples:

Example 1:

Illustration 1 below shows the first map in its initial state. Before discussing this example, I will take some time to explain the workings of the application. As can be seen clearly, the map is being represented by a battlefield with two opposing armies; a blue army and a red army. In the top right corner of the map, there is a treasure hoard being guarded in an area blocked off by a man-made fortification. The objective of the blue army is to steal the treasure. The objective of the red army is to defend the treasure (achieved by defeating the entire blue army).

The blue army, being the besieging army is prepared and able to build bridges to cross the water and to break down the fortress walls.

The top right tile is outlined with a white square. This indicates the head position. The colour indicates the current state.

Let us now look at the program in action. As can be seen in the Illustration 2, the Turing machine has begun execution. We can see the red combatant, previously in the second tile from the top now waiting at the edge of the water. One of the blue combatants is building a bridge to cross the water.

This requires some explanation which will further shed light on how the Turing Battle system is implemented. When the program starts, it is in a state called scout, which scours each tile in from left to right looking for a combatant. In this case, a red combatant is found. The way the program is set up, a red combatant will always move to the left and a blue combatant will always move right.

As such, the red combatant, finding nothing but grass in the way began to move left. At the first movement, it is in the top left corner. We can take some time here to explain how moving past the edges is handled. When the head moves past the edge, it will switch to the
opposite side of the screen (in this example, the red combatant goes from the left side of the screen to the right side of the screen). The head will also go up to the next row. In this case, because the next row is outside the window, the next row is treated as the last row. It is for this reason the red combatant loops down to the bottom right corner of the screen and continues marching from there. By the same token, if a blue combatant, who moves in the opposite direction, goes past the bottom right corner of the screen, he will 'auto-magically' go back to the top-left corner of the window. While this isn't the most realistic approach to the issue, it simulates, to a certain extent, *old-school* games, sticking with the theme of my implementation.

This example (Illustration 3), a little later on in the program, opens up a few more interesting tidbits and limitations regarding my implementation. We can see that a member of each army has died.

When a blue combatant builds a bridge or breaks down a fortress, which can be seen in this example, he needs some time to recover and so the program will start scouting in the opposite direction. Therefore, once the blue combatant in the last illustration 2 had finished building the bridge, focus went to the red combatant who then had the opportunity to kill the blue combatant. However, once a combatant is victorious, the program again continued to scout (naturally a combatant will need some time to recover from a battle), finding a blue combatant who managed to kill the red combatant.

In Illustration 3, we see something curious. In the top right corner, where a combatant previously stood, is a block of grass. Similarly, where the bridge was built is now a block of grass. This uncanny occurrence stems because of a limitation in the program. Because a Turing machine cannot store a past value in a position, once a combatant walks over an area, be it a bridge, rubble from a destroyed fortress or concrete fall, it will revert back to grass. This doesn't really have an impact (other than visual) on the running of the application so I decided it wasn't worth fixing. To have fixed this issue while remaining true to the workings of a Turing machine, I would have had to store and refer an original clone of the map (so that I can tell whether the block the combatant has walked on is indeed grass or some other passable obstacle) (there may also be other methods), however, I concluded that is likely far beyond the scope of this assignment.

Illustration 3: Woah! The bridge turned to grass!
In Illustration 4 we can see that there is only blue combatant remaining. However, to his credit, he has managed to capture the credit. Upon capturing the treasure, the state is set to halt causing the execution to finish.

Here we can see that despite a red combatant being directly underneath the blue combatant, it made no attempt to fight back, given the single sightedness of the combatants.

*Illustration 4: Lone blue combatant captures the treasure*
Example 2:

Let us as look at the second example to gain a better understanding of the improvements made. This example is strange in design however I feel it will serve well in explaining the changes implemented.

The first example was relatively basic. While shown as a 2D battlefield, the combatants were moving either left or right (depending on the colour of the combatant) which somewhat defeated the purpose. For the second example, I spent some time improving the actions that each combatant can take, vastly increasing the number of states and adding much more depth into the movement of the combatants.

Rather than move in one direction, I expanded the system to allow combatants to move in all four directions. As before, a blue combatant will still try to move right and a red combatant will still try to move left however if this isn't possible, the combatants will look in all directions for another path to take.

Also, to even out the playing field and make the game more balanced, each combatant can only move one square at a time, after which, the application will scour the playing field for a combatant of the other colour. In addition, each combatant will now look around in all directions to see an enemy before moving. If an enemy is found, the combatant will kill him. In this case, play will swap to the other team without that combatant moving (as killing an opponent is counted as a move). Additionally, if a combatant moves and finds an enemy combatant besides him in his new position, he will kill him.

Taking a glance to Illustration 6, we can see that a checkers-board pattern is emerging. This is an excellent example of a combatant striking if possible before making a move. In this example, as each red combatant is surrounded by two blue combatants (and vice versa), initially each combatant will strike out against a surrounding enemy (thus losing their opportunity to
Illustration 7 shows the example a little later on in the program. It's quite apparent that the majority of the combatants have been wiped out. Now the blue combatants can be seen to be work on the walls.

What's interesting to note here, is that if a blue combatant can move or demolish a wall (or build a bridge) it will always choose to demolish the wall. While not the best solution, I felt this was necessary and relatively simple fix granted the scope of the assignment. If this was not the case in this example, the blue combatants will just keep walking up and down their aisle in this example, and no wall breaking will take place.

Note also that the blue combatants are one man up in this example.

Illustration 7: Deconstruction work has begun

Illustration 8 is now a ways into the program and we can see that the tables have turning heavily. The reds have gotten the upper-hand and have whittled down the blue army to only three combatants.

In this whole time, the blues have only managed to wipe out two combatants from the red army. The reason being is quite simple; since the blue army is the besieging army, they are going to have to break down the fortresses of the red army (as well as build bridges over water to get to the red army).

The red army is only concerned with defeating the blue army. As the blue army is preoccupied in destruction, the red army has the upperhand, allowing it to deal severe damage to the blue army.

This is a good example of the variety added to the system by making the blue army a besieging army, responsible for building and destroying objects.

Illustration 8: Red team beginning to dominate
Finally, in this image, we can see that the red army has achieved victory. What is interesting to note is that victory was obtained simply by destroying the entire blue army.

This wasn't easily possible in the first example, however as I changed the system to look for a blue combatant after a red combatant's move, once all the blue combatants were wiped out, the system wasn't able to find any blue combatants, thus being stuck in the searching state without changing any data on the field.

To handle this, I implemented a feature in the system to count the number of head movements before a change in state occurs. If the system remains in the same state for a long period of time (I felt that the number of cells multiplied by 2 should work nicely for this implementation), then it is safe to assume that the program has reached it's completed, and thus the state is changed to 'halt', telling the program to complete.

Before getting on to the third example, I feel that I've adequately explained the features of this system with the first two examples. Therefore, the level of detail found here won't be needed in the next two examples.
Example 3:

In this example, three unfortunate red combatants must protect the treasure stored in the fortress by a surprise attack of blue combatants.

The back wall was added to the fortress to ensure that the red soldiers don't somehow find their way out the back entrance. Since the combatants can now (with constraints) move in all four directions, this was necessary and forces the reds to stay within the fortress as they have no power to break down walls.

In Illustration 11 we can see the blues have successfully crossed the moat (and somehow turned it into grass because of a certain limitation present in the system). Given the reds tendency to walk up in a move to the left isn't possible, we can see that two of them have taken position in the north of the castle.

The third is standing in front of the treasure guarding it valiantly. There is little the reds can do but await the invasion.

Illustration 10: What?! We've been besieged. We're undermanned! Stand tall, men and duke it out to the finish.

Illustration 11: They're approaching men! What are you cowering up there for!
A little later on and the blues have managed to breach the fortress (Illustration 12), taking out two of the red combatants in the process. The valiant combatant standing in the way of the treasure managed to take out one of the blues, but in the end, was overpowered.

An interesting thing to note is that although an opening in the fortress was made, the blues will continue breaking openings in the wall, rather than making use of the other exit, as the system doesn't allow the blues to scout the battlefield to find openings in the fortresses.

Alas, the inevitable has happened, the blue has successfully managed to take hold of the treasure.

One last thing to note here, we can see that the red combatant is still in the top of the fortress in this screen. Because the order of movement is left, up, right and then down for red combatants, this combatant will continue moving from left to right, unable to return back into the fray.

Illustration 12: We've breached the fortress!

Illustration 13: "The treasure is mine, precious!" "Come human friend, we must cast it into the fire to destroy it's evil." ".... The fire? Yes... the fire, precious...."
Example 4:

In this rather cool looking example (Illustration 14), we somehow have four blue and four red combatants trapped in a fortress surrounded by water. This setup is interesting and the outcome is quite unpredictable.

Notice that two of the red combatants are stuck in together in small room. They have no way of escaping the fortress without the help of one of the blue combatants.

Illustration 15 shows an example a little later on. One of the blue combatants broke the wall, entering the reds cell (the red managed to kill him). Unfortunately, while widening the cell, the red combatants are still stuck within their room.

We can also observe one of the blue combatants on a rampage, destroying everything in sight. Unfortunately, the destroyed the barrier surrounding the treasure, but given the workings of the system, just kept walking straight ahead.

Illustration 15: "Victory is mine!" "Why'd you kill him for! We could've used him to break open the wall for us to escape!" "At least now we can spend some time to get to know each other. I mean, we travel together for so long in wars, but we hardly get to know the person behind the stick armour..."
Much later on in the execution of this map, we can see two combatants remaining, slowly converging. Again, given the way the system works, the two are walking towards each other on the horizontal axis rather than taking a shortcut and walking on the vertical axis.

Finally, in the end the blue combatant wins. One final thing to note about the system. We've established previously that the objective of the blue army is to claim the treasure and the the objective of the red army is to destroy the blue army.

However in this case, the system halts after the blue combatant destroys the last red combatant. While the treasure still remains on the playing field, logically there really isn't any point to continue given that the blue combatant can cross any obstacle on the playing field. The fact that he will achieve the treasure is only a matter of time, thus, halting the system at this point makes sense.
Conclusion:
After completing the first example, I felt that the system was somewhat lacking and could benefit with some improvement. Therefore I went back and changed a few things in the code (both versions are distributed with the assignments) including a huge increase in the complexity of the states and transitions. In fact, the system used in example 1 only made use of 7 possible states (including 'halt'). The system used for the remainder of the examples makes use of 51 possible states (including 'halt').

This is a very large number of states, which was rather difficult and painstaking to implement. There needs to be very careful attention to detail to ensure that the system is working fine and bug free with such a large number of levels. Granted, many of the states are very similar (most of the states have four versions, one for the left, right, up and down directions), however I felt this was the best way to implement this.

At this point I'm not sure how completely I've completed the objectives of the assignment. As mentioned previously, this option stood out to me because of the creativity it entailed which appealed to me. I may be wrong on this point, but I feel that I've taken a rather unconventional approach to this option. Rather than expanding into the other areas mentioned in the bonus credit section, I feel that I've done enough for the assignment by taking the idea of 2d Turing machines further than that presented in the assignment specifications and seeing how it applies to a 2d turn-based battle simulator based on rudimentary AI simulating the working of a Turing machine.

In conclusion, I'm rather surprised by what I've been able to complete. AI was always a field of programming that intrigued me and seemed difficult to achieve. I wasn't sure how easy simulating AI via the constraints of a Turing machine-based system will be. While, in the end, the results where rather basic and the AI could be said to be dumb, I feel that there is potential in using this system to create something much more interesting and compelling given enough time and effort.