Automated Snake Game Solvers via AI Search Algorithms

Shu Kong, *80888472, skong2@uci.edu*
Joan Aguilar Mayans, *87286425, joana1@uci.edu*,

Abstract—In this report, we present five different algorithms or methods for a computer to play Snake [1] automatically, including three searching algorithms related to artificial intelligence, Best First Search, A^* Search and improved A^* Search with forward checking, and two baseline methods, Random Move and Almighty Move. These methods can be the core technique in an automated Snake Game Solver, but their performances are quite different. We conducted experiments to compare their performance and explain their differences. Furthermore, we demonstrate how the different methods can get stuck in a dead end and anticipate possible improvements as future work.

Index Terms—Best First Search, Breadth First Search, Greedy Search, A^* Search, Forward Checking, Snake Game, Artificial Intelligence.

1 Introduction

This report is presented as a requirement for the CS 271 Introduction to Artificial Intelligence course. We decided to assess the performance of different AI algorithms by using them to play Snake. This involved creating an implementation of the game together with three different AI algorithms and two baseline methods. We will show the difference between the different approaches and quantitatively demonstrate their performance by extensive experiments.

Snake can be dated back to arcade game Blockade [3], [2], developed and published by Gremlin in 1976. The first known personal computer version of Snake was programmed in 1978, and various versions of snake-similar games were developed afterwards. Traditionally, Snake is played by a human player. The human player plays the game using only the left and right arrow keys which will make the snake turn relative to the direction it is heading. The snake automatically moves forward and it will increase its length and speed every time it reaches a goal, which is referred as *apples* in this report. This makes it difficult to achieve a high score. The final goal of the game is to make the snake eat as many apples as possible without running out of the board or making the snake bite itself. When this happens, the game ends and a final score is returned.

Even if traditional Snake is played by humans, it can also be a good venue to test AI algorithms on a computer, especially searching algorithms. However, because of this test bed purpose, our Snake implementation is much simpler than commercial Snake games. We explain the rules of our Snake implementation in Section 2, followed by elaboration of the algorithms in Section 3. We experimentally show the performance of each algorithm in Section 4, and demonstrate when and how the algorithm reaches a dead end and how we can improve it in Section 5. Finally, we conclude our report in Section 6.

• *The authors contributed equally to this project.*

2 Snake

In this section, we clarify some basic characteristics about our Snake implementation.

In the most general way, our implementation consists of the snake moving on a square board, trying to eat as many apples as possible without biting itself. Once the snake eats an apple, a new apple is placed in a free position on the board and the snake length grows by one unit. When the snake has no choice other than biting itself, the game is over and a final score is returned. In our implementation, we simply calculate the score as the number of apples the snake has eaten or equivalently, the length increased by the snake.

The following are some basic rules followed by our implementation:

1) Goal - the snake tries to eat as many apples as possible, within finite steps[1]. The first priority for the snake is to not bite itself while the second is to increase the score.
2) There are four possible directions the snake can move: *north*, *south*, *east*, and *west*. However, because of the placement of its tail some directions may not be available. The most clear example is that the snake can never swap to an opposite direction *i.e.* north to south, east to west, etc.

1. Although it is possible to keep the snake "alive" with infinite steps, this kind of solutions are not considered in this report.

3) The snake grows by one unit when eating an apple. The growth is immediately reflected by the gained length of the tail, *i.e.* the tip of the tail occupies the square on which the apple was.
4) The board size is fixed to square.
5) After an apple is eaten by the snake, another apple is placed randomly with uniform probability on one available squares of the board. Here the availability of a square is denoted by the fact that it is not occupied by the snake.

3 Algorithms

In this section we present three AI algorithms and two baseline methods to play Snake. The three AI algorithms belong to informed heuristic search while the two baseline methods are developed intuitively based on the character of the game.

3.1 Best First Search

This Greedy Best-First Search algorithm has a one-move horizon and only considers moving the snake to the position on the board that appears to be closest to the goal, *i.e.* apple. We use Manhattan distance to define how close the snake head is to the apple.

This method has almost guaranteed that the snake will be able to eat in an optimal (shortest) way at least the first four apples. The previous statement comes from the fact that a snake with length five cannot construct a circle on the board [2].

However, the one-step horizon also makes it easy to get stuck on local minima and plateaus. The intuitive explanation is that, the snake only looks for the next step that is assumed best or closest to the apple without considering its tail. It is easy for the snake to bite itself once it gets longer. Eventually, this method will stop being optimal after the snake has eaten more than four apples.

In Section 5, we will illustrate how this method can get stuck in a dead end.

3.2 A^* Search

A^* incorporates a heuristic in a multiple move horizon. Before taking action, it considers not only where the goal is and how far it is, but also the current state it has searched so far.

This A^* algorithm uses the Manhattan distance from the head to the apple as a heuristic and the number of steps as the "cost so far". Each iteration of the algorithm lasts until a path is found that leads the snake to eat an apple. It improves the Best First Search algorithm by finding a full path to the apple and not stopping at the first move, this has the advantage of not getting stuck at a dead end on the way to the apple. Without memory or time restrictions, the algorithm is guaranteed to find an optimal path to the apple if one exists.

There are two minor modifications compared to a standard A^* algorithm. First, ties between nodes with same estimated total cost (that is, the sum of heuristic and cost to reach the node) are not broken randomly. Instead, one of the nodes with lowest heuristic is chosen. This helps the algorithm in finding a path with minimum cost to the goal faster. Note that there can be many optimal paths to the goal. Second, the maximum number of nodes expanded is limited. This makes the algorithm stop if a path to goal cannot be found (for any reason). In case the maximum nodes bound is reached, the algorithm will switch back to Best First Search for that iteration.

3.3 A^* Search with Forward Checking

The A^* algorithm introduced in the previous section still has some shortcomings. One of the shortcomings includes the fact that, once the apple is eaten, the snake can reach a dead end which can be avoided with other paths. In other words, the algorithm does not take into account the effects of the selected path once the apple is eaten. To avoid these dead ends, the A^* algorithm is also equipped with a Breadth First Search algorithm that is used to compute if a path to a goal also leads to a dead end. Once an iteration of the A^* algorithm ends, it will then call the Breadth First Search algorithm starting at the goal state found by the A^* algorithm. From here, it will explore the full tree up to a certain number of nodes. If the tree is contained inside this node bound, *i.e.* it is a dead end, the path to the apple is rendered as not good, and the goal node from the A^* algorithm will be discarded (the A^* iteration will continue though).

This dead end check is also used when the A^* algorithm cannot find a path to the goal. It will then select the Best First direction that does not lead to a dead end.

3.4 Random Move

Besides the three Artificial Intelligence methods described previously in this section, we also introduce two baseline methods for comparison. The first one of the two is Random Move.

Just like the name suggests, Random Move selects the next step to move randomly. We impose only one condition: if possible, the move chosen must not end the game. As we can imagine, this method makes the snake spend a long time searching for apples as it considers nothing about the position of those. Moreover, this method can easily lead to a dead end, since it does not consider the full position of the snake on the board.

Even though this method does not seem to work well, it can be a good baseline for experimental comparison with other AI methods, just as shown in Section 4.

3.5 Almighty Move

We finally introduce the second of the two baseline algorithms: Almighty Move.

2. There may exist some cases in which the algorithm does not use an optimal path or it even gets into a dead end. This cases however, are considered as exceptional.

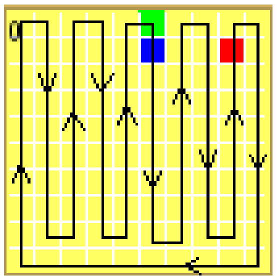

Fig. 1. Almighty Move runs optimally on a board with even-numbered width **or** height. As long as the length of width or height is an even number, Almighty Move can guarantee to make the snake eat the most apples.

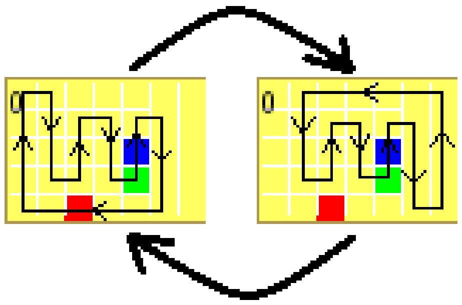

Fig. 2. Almighty Move can also be run on a board with odd-numbered width **and** height. To keep the score increasing, Almighty Move iterates over the two circuiting patterns.

Essentially, Almighty Move makes the snake circuit within the board, just as demonstrated in Fig. 1 and Fig. 2. Note that when the width or height of board are even numbers, it is easy to understand why a maximum score is guaranteed with this method, see Fig. 1. However, in boards with odd-numbered width and height the circuiting pattern needs to be modified a bit, see Fig. 2.

With further analysis, it can be shown that if we start the snake with length 1, *i.e.* with only one unit of tail, then the snake will eat $(hw-2)$ apples[3] if h and w are the even-numbered height and width of the board. However, in a board with odd-numbered height and width, Almighty Move can only guarantee that the final length of the snake is greater than $(h-2)(w-2)$, but less than $(hw-2)$.

In conclusion, Almighty Move is an extreme case that has guaranteed a maximum score by not caring about how much time is needed. Note that it does achieve the goal within finite time though.

4 Experiment and Results

To fairly compare different algorithms and methods, we run each method 100 times and get the averaged performance with standard deviation. We ran the algorithms on different boards with different sizes but in this report, we only consider the board of size 10×10, as it is the largest square board on which all the algorithms can be run with a reasonable amount of time. Each run is stored as a vector with each element being the score at that step. The different outcomes of the experiment are discussed in the following paragraphs.

Fig. 3 shows all the runs for all the algorithms. From this figure we can get information not only about the final scores achieved or how many steps did it take but also about how successful each algorithm is, how consistent, or how fast does it eat the apples. Clearly, only two of the algorithms achieve the maximum score: A^* with forward checking and Almighty Move. However, the way they achieve this maximum score is clearly different. While A^* with dead end checking is much faster all the way to half the maximum score it loses efficiency from there onwards. It is the opposite case for Almighty Move. While at the beginning eats apples way slower than any informed search algorithm, the way the snake is organized at the end makes it very efficient, making it extremely fast at the end games. A^* and Best First Search, match the efficiency of A^* with forward checking at the initial stages of the game but they are unable to reach scores as high as A^* with forward checking or Almighty Move. Random Move is unreliable and inefficient but it serves as a baseline.

Fig. 4 shows the mean score that each algorithm has reached at a certain step. Note that if a run has finished in an earlier step number, it is discarded, that is the reason of the ripple in the figure. The thinner lines in the graph show the values a standard deviation above and below the mean. From this figure we confirm how Best First Search, A^*, and A^* with forward checking share a similar efficiency at the beginning of the game while Almighty Move's is way lower. More importantly, we see that Random Move struggles to increase the score and how chances are that whatever the step number is, Random Move score will be very low.

Fig. 5 shows the ratio of how many runs of each algorithm will reach a certain score *i.e.* how reliable each algorithm is. Unsurprisingly, Almighty Move keeps a success ratio of 1 no matter what the score is, in other words, each run of the Almighty Move will reach the maximum score. However, Almighty Move is an exception and the other algorithms will show a lower success rate. Looking at the figure, it is easy to sort the algorithms in ascending reliability order, from Random Move to Almighty Move.

Finally, Fig. 6 shows the mean final score vs. the mean number of steps needed to achieve it for each algorithm. The ellipses axes are two standard deviations long. The figure shows which region of the score-steps plane the algorithm is most likely to end up. Disregarding the Random Move algorithm, we see that the higher the expected score is the more steps are also expected. Once more, the figure shows how Almighty Move is able to

3. This is because after eating the last apple, the snake occupies the whole board, with its head on one square and total body length of $(hw-1)$. As the snake starts with a 1-length body, it finally eats $(hw-2)$ apples.

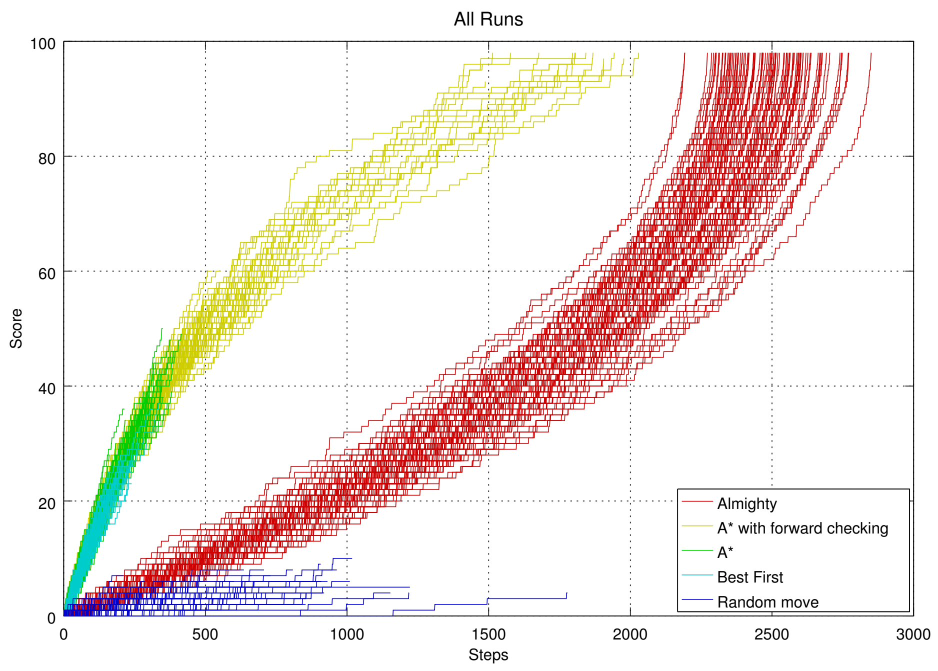

Fig. 3. All the runs for the different algorithms in a 10x10 board. This figure is best seen in color and zoomed in.

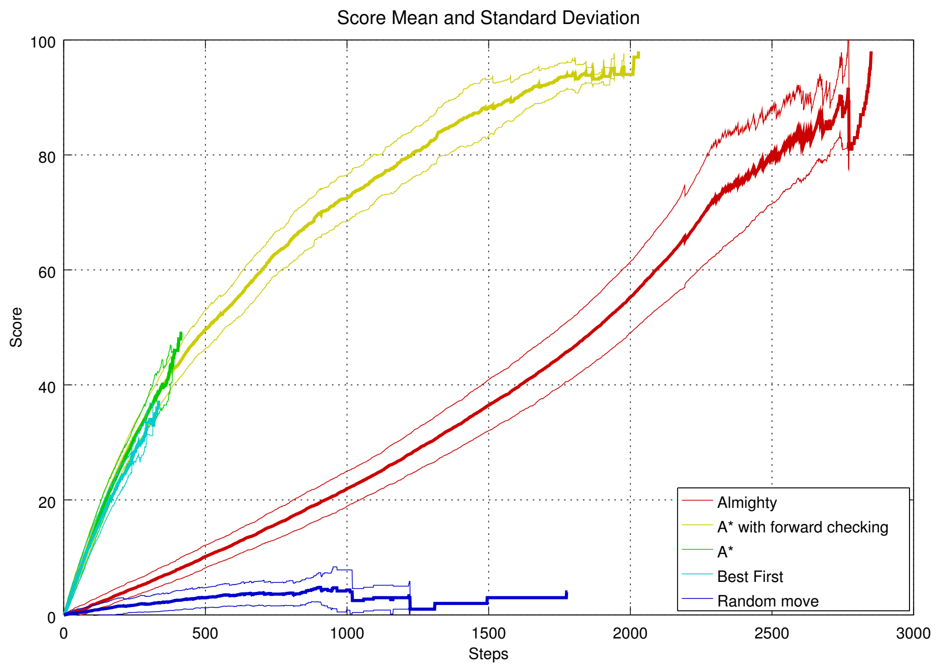

Fig. 4. The thicker lines correspond to the score average for all the runs that made it to that step. The thinner lines show the values one standard deviation above and below the average. This figure is best seen in color and zoomed in.

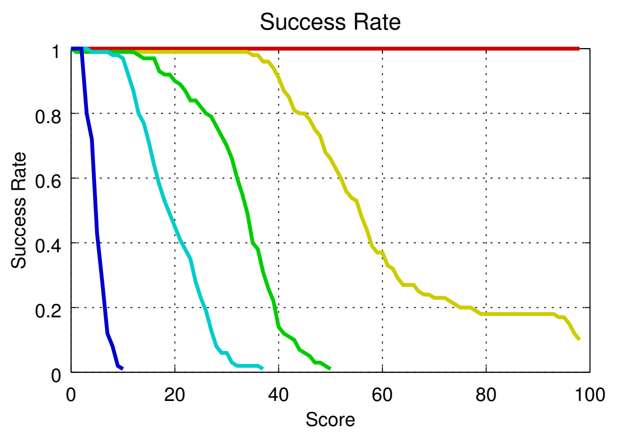

Fig. 5. Success rate expressed as the number of runs that made it to that score divided by the total number of runs. From lowest success rate to highest, the algorithms are: Random Move, Best First Search, A*, A* with forward checking, and Almighty Move.

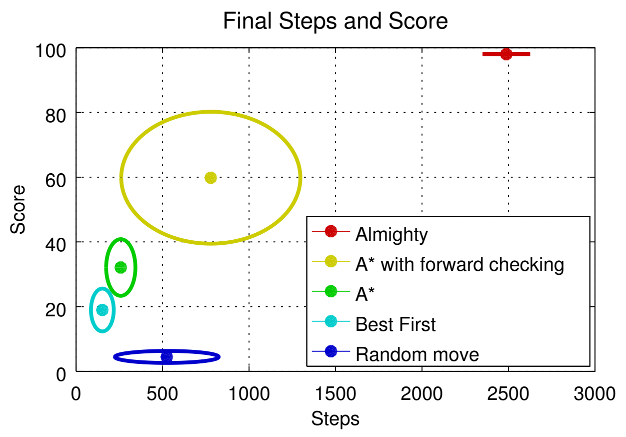

Fig. 6. The central point of the ellipses correspond to the average number of steps and final score for each run. The axes length correspond to two standard deviations of the mentioned metrics.

achieve the highest scores at expenses of using the most steps.

5 Discussion

In this section, we will analyze each method in depth and show how and why each method can get stuck in a dead end and finish the run prematurely. Based on these observations, we will introduce a good way to improve the overall performance by combining the different methods carefully.

5.1 Dead End

5.1.1 Random Future Search

Random Move can easily reach a dead end since it blindly moves forward. For instance, an example of a current state is shown in Fig. 7 (a), if Random Move

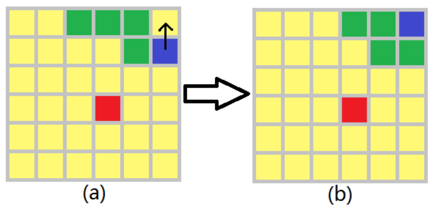

Fig. 7. Random Move can easily reach a dead end since it blindly moves forward. In this example the current state is (a). If Random Move chooses to move up, then a dead end will be reached.

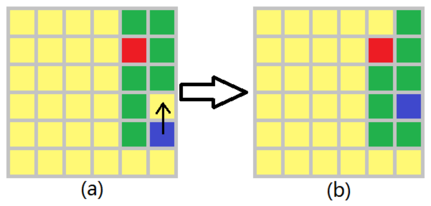

Fig. 8. Best First Search will choose to move up based on the current state (a). Once it moves up, a dead end is reached, as shown in (b). Note that A^* will avoid this dead end.

chooses to move up, as shown in Fig. 7 (b), then a dead end will be reached.

5.1.2 Best First Search

Best First Search has a one-move horizon and selects the next move based on the Manhattan distance to the goal. It is expected to run quite well and fast for the first few apples as discussed in Section 3.1. However, when the snake gets long it can easily lead to a dead end.

For example, in Fig. 8, Best First Search will choose to move up, because the square on the top of the head of the snake is closer to the apple than the one below. However, once the move is done, the snake has reached a dead end and the run will terminate.

5.1.3 A^* Search

A^* is guaranteed to find an optimal path if it exists. In Snake, A^* Search uses the Manhattan distance as a heuristic. Up to some extend, A^* is comparable to Breadth First Search as both can find the optimal path. However, A^* Search considers heuristic information and only expands nodes that can potentially lead to an optimal path, therefore expanding less nodes than Breadth First Search.

Even though it can outperform Breadth First Search or Depth First Search, A^* Search can still lead a dead end if the snake is long enough. Fig. 9 shows such a situation. In this case, A^* Search makes the snake succeed in eating the apple but after that the snake can not do anything but move forward until there are no more moves available. This is incurred by the property of A^* Search that it only considers current situation and how to reach the goal more efficiently without considering possible effects after

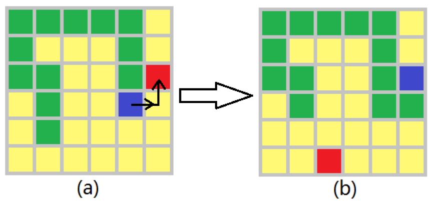

Fig. 9. A^* Search can always find the shortest path as long as a valid path to the goal exists. However, it does not consider the effects when the snake succeeds in eating the apple. In this figure, A^* Search chooses to move right and up. However, after eating the apple, the snake does not have any other choice but to continue going up until no more moves are available. In contrast to Fig. 8 in this case both Best First Search and A^* will get stuck in the dead end.

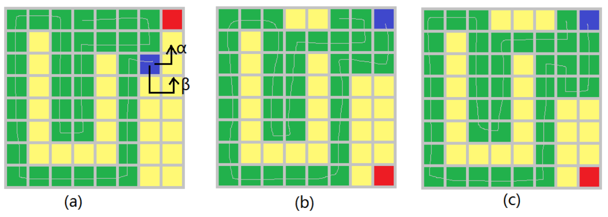

Fig. 10. A^* Search with forward checking can still lead to a dead end. To clearly see how the snake moves, we added a gray line along the snake body to clarify the shape of the snake. With the current state in (a), A^* Search will choose to move right and up. This would lead to a dead end, as shown in (b). However, A^* Search with forward checking will consider possible effects by looking ahead a specified number of steps. For instance, a forward checking depth of three moves would make the snake move down, right, and up. This results in (c). Even though the snake has avoided a first dead end, it will eventually still run into a dead end.

achieving the goal. In the next subsection we discuss how the improvements in A^* help avoiding this type of situations.

5.1.4 A^* *Search with Forward Checking*

As we see how and why A^* Search can still lead to a dead end, we modify it by adding some Forward Checking capabilities as discussed in Section 3.3. This forward checking will check several steps ahead to see if this move incurs in a dead end. This upgrade is expected to improve performance when comparing it to the plain A^* Search.

However, A^* Search with forward checking also can lead to an avoidable dead end. Fig. 10 demonstrates such an example. With the current state in Fig. 10 (a), A^* Search will choose to move right and up. This easily leads to a dead end, as shown in (b). However, A^* Search with a 3-step forward checking will make the snake move down by one unit, before moving right and up. This will result in (c), which is **NOT** a dead end at present, because the snake can move left and the tip of the tail will move downward to give way to the head. This is determined by the snake's growth mechanism. But as the snake keeps moving left for some more steps, the dead end will be eventually be reached. Still, this dead end can be avoided if the snake moves down by 5 steps at (a), which will make room around the tip of the tail. In other words, checking for more steps will eliminate more dead ends that will otherwise occur with fewer forward checking steps.

There is still one more possibility in which A^* with forward checking can lead to a dead end. When using Breadth First Search to look for possible dead ends, apple placement is not considered. The placement of the apple can turn states that were initially thought not to be part of a dead end into one. Imagine a case were the head and the first parts of the snake are moved in a way that fit in a certain region of the board, the placement of the apple in that region will make the snake grow by one unit potentially making the computations made by the Breadth First Search algorithm useless.

Therefore, A^* Search with forward checking can still reach a dead end if the forward check depth is not high enough. Even though checking for more steps will eliminate more dead ends, the time spent on the checking, can slow down the process significantly, making the automated snake game solver unpractical.

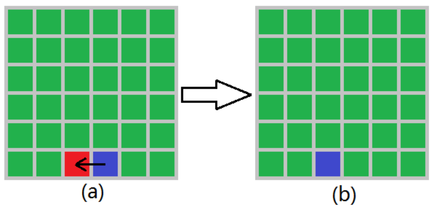

Fig. 11. Almighty Move makes the snake circuit within the board as in Fig. 1. (a) and (b) are the last two states of a run of the Almighty Move in a 6×6 board. Note that the snake reaches the maximum possible length.

5.1.5 *Almighty Move*

As analyzed previously, A^* Search with forward checking eliminates some dead ends by checking a specified number of steps ahead, which means it makes the snake circuit a bit to avoid dead ends by making more room for the head to move. Therefore, it is intuitive to design a sophisticated method for Snake which makes the snake circuit within the board as introduced in Section 3.5.

Fig. 11 shows the last two steps of the snake using Almigthy Move on a 6×6 board. We see that the length of the snake will finally equal the number of the units in the board. This dead end is unavoidable and means the game will end with the maximum score.

5.2 Improvement

Based on experiments and the analysis performed in the previous subsection, we see that we can carefully combine some of the algorithms to get a better method that works more efficiently and effectively.

First, we can run Best First Search algorithm for the snake to eat the first apples. This algorithm has almost guaranteed optimality before the snake eats the first four apples. After eating these apples, we can switch to A^* Search with forward checking as a more reliable alternative. Once the snake reaches a certain score we can use Almighty Move as it keeps maximum reliability while being the most efficient of the methods in the end-game. This final switch will lead to perfect score game.

However, determining the score threshold at which to switch to another algorithm is not easy. For instance, we do not have theoretical analysis on when to switch from A^* Search with forward checking to Almighty Move. We anticipate the threshold depends on the size of the board and the snake's length. This, however, is considered to be outside of the scope of this course project.

6 Conclusion

In this report, we presented five algorithms and methods to build an automated Snake Game solver. We analyzed the different algorithms and conducted experiments to study their performance.

Except Random Move, which is only used as a base-line, the rest seem to offer advantages and disadvantages depending on if the main priority is speed or reliability. While the informed search algorithms can show a reasonable reliability and the highest efficiency at the beginning of the run this properties disappear at the end game. In contrast, Almighty Move is a slow algorithm at the beginning of the run but has guaranteed a maximum score and its efficiency is the highest at the end game. Intuition makes the authors think that a combination of the different algorithms can achieve perfect reliability while keeping the beginning of the game efficiency high.

However, determining at which score threshold the switch between algorithms has to be done is a nontrivial problem which is left as future work. We anticipate the score threshold will depend on the size of the board and the different parameters of each algorithm. Performance can still be further improved by implementing some other AI algorithms as the ones introduced in [4].

References

[1] http://en.wikipedia.org/wiki/Snake_(video_game).
[2] R. DeMaria and J. L. Wilson. *High score!: the illustrated history of electronic games*. McGraw-Hill Osborne Media, 2003.
[3] G. Goggin. *Global mobile media*. Routledge, 2010.
[4] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2009.