TwixT Alive

Jonathon Gibson (jtg98) and Patrick Walsh (prw55)

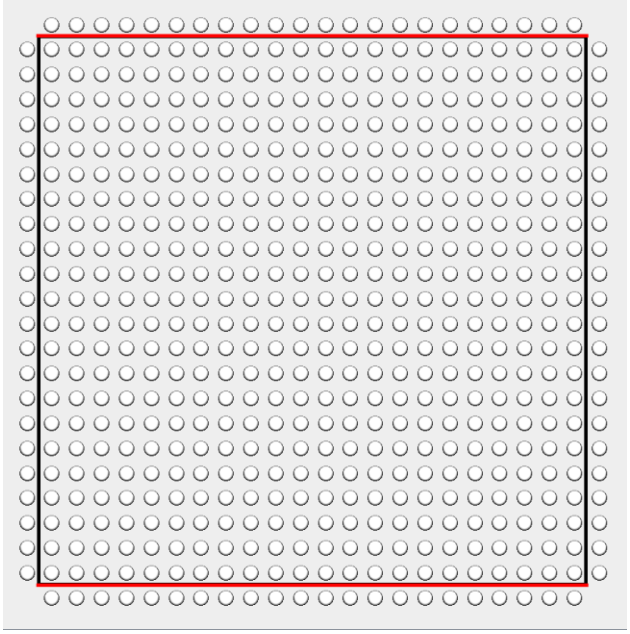
Link to code: https://github.coecis.cornell.edu/jtg98/TwixT

Introduction:

Game playing is an important and commonly studied subcategory within Artificial Intelligence, and there are a variety of techniques and variations on these techniques that can be used for 2-player games of perfect information. At the heart of several of these is Minimax, an adversarial search strategy learned in class which effectively determines the best move one can make given that the opponent will play optimally. To be truly optimal, minimax would need to search the entire game tree, traversing down all the way to every terminal state (win, loss, tie). In reality, most 2-player games that are interesting problems for AI have search space which is exponential in the number of turns taken, and so this is not viable (i.e. if there are 10 choices per turn and on average 20 turns per player, there is somewhere on the order of 10^40 nodes in the search tree!). Instead, an evaluation function can be used which approximates the value of a board state, which allows Minimax to search only a relatively small subset of the game tree, going only a few levels of depth below the current board state, in order to play well, if not completely optimally depending on the heuristic. While there is significant expansion upon it, this technique of using a heuristic evaluation function along with Minimax is the starting point for our project.

Before we can begin to detail the fundamentals that went into our implementation of our game playing AI, it is first useful to introduce the game we designed for: TwixT, a strategy game for two players. Red and Black take turns placing individual pegs on a NxN board (24x24 is common (Figure 1), but often variants of the game are played with smaller boards. So that our AI had a chance to go to a non-trivial depth when using minimax, we generally used a smaller N value in the 6-12 range).

Figure 1 empty twixt board



Whenever two pegs of the same color are placed exactly a knight's move apart, that is, 2 units in one axis and 1 unit in the other, a connection of that player's color is placed between the two pegs, provided that doing so does not intersect an already placed connection of either color

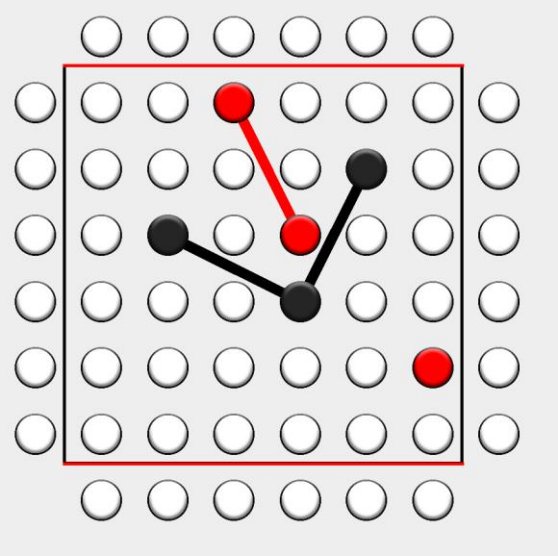


Figure 2 Mid-game

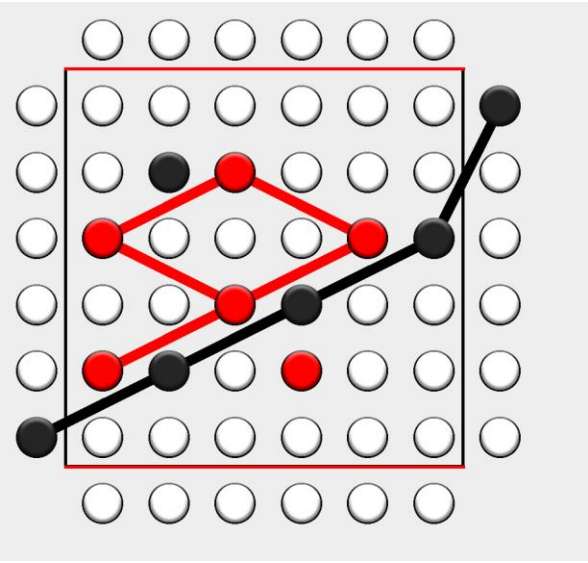


Figure 3 Black Wins

(Figure 2). Only red may place pegs in the first and last row of the board, and only Black may play in the first and last columns of the board (corners can be used by either color). The winner (Black in Figure 3) is the first player to create a continuous chain of pegs connecting the two sides of their color.

TwixT is a game of perfect and complete information, that is, nothing is hidden from either player, and both know the objective of the other player. There is no chance involved at all, and the game is very symmetric except in that one player goes first. Therefore, strategy is incredibly important, and by playing a few games it becomes clear that there is no simple greedy strategy that works. There are a few general rules that help make a player better, but they cannot be used in a vacuum; responding to an opponent's move is just as important to win as working to build a connected path of your own.

For example, suppose an opponent has a connected path that you are trying to block. If you play too close to one end of their path, they can just build connecting paths around your piece, but if you play too far away, they can divert their path with additional moves to avoid your attempt at a block altogether. From an offensive standpoint, on the other hand, it is often beneficial to place pegs with exactly three units between them in a single axis, because an opponent cannot

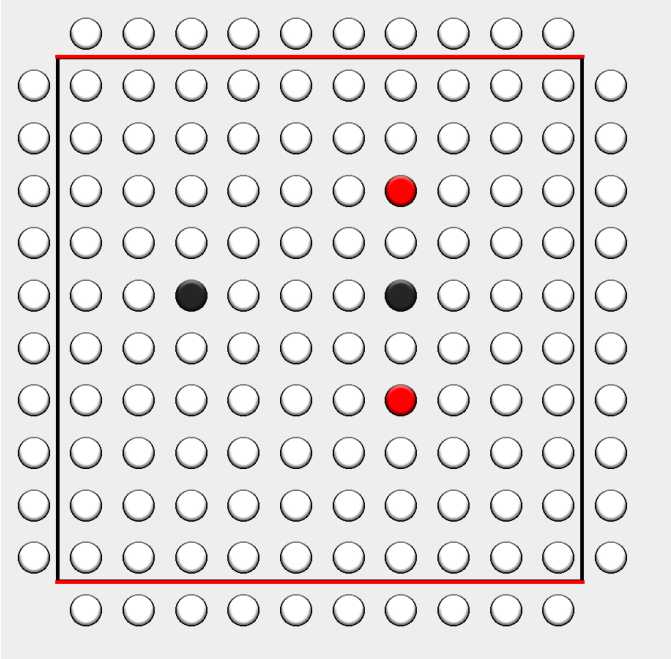


Figure 4 Strategy set-up

block you from connecting them with one additional move unless they already had a peg in the vicinity (Figure 4). However, if this is all you try to do, it is easy for an opponent to plan a blocking path ahead of the "line" you create with your pegs. The point is, while there are several intuitive and tactical strategies which can help, it is unclear if there is a best way to play. There have been TwixT world championships, no winning strategy has been found, the game is unsolved, and we do not know what the outcome of the game is with perfect play (source: https://en.wikipedia.org/wiki/TwixT)

The game of TwixT is PSPACE-complete, meaning that it can be solved using a polynomial amount of memory relative to the input size. In this type of game, a brute force search to the solution is impossible due to the game tree's extreme branching factor caused by the amount of possible next moves. For TwixT, each player has nearly N^2 (where N is the width/height of the board in peg slots) possible moves at the beginning of the game, with only a linear decrease as the game progresses. This means that from the perspective of size of the game tree only, TwixT is far more difficult to solve using a brute force approach than Chess or even Go when using the standard board size of 24x24. Because of this, creating an AI for TwixT is ambitious, and all previous attempts by others that we could find have been very weak. For example, the AI at http://twixtlive.com/Play.aspx is almost trivially beatable, and the developers seem to know this by labelling their AI difficulty settings as "Really Easy", "Easy", and "Medium Easy". Two slightly better AIs can be found here (http://www.johannes-schwagereit.de/twixt/programs.html), but even these are beatable by an amateur. Most approaches use some form of hardcoded tactics/rules/heuristics. From the discussion/analysis on the state of TwixT programming on pages such as this (https://senseis.xmp.net/?Twixt), it seems that it is not clear what the best heuristic functions are.

Strategy and Implementation:

Our AI was programmed in Java. To begin work on it, we first needed to program the logic of the game itself, and we made a GUI so that the game could be played and watched visually. While the design of the game logic itself is largely independent from the design of the AI, we had to be very careful in the representation of the board state so that a board evaluation function could compute the value of any state efficiently and accurately. The most natural representation from a human perspective is probably to represent the grid as an array of peg slots, but this is much less useful for actually trying to determine good moves, because the coordinates of peg slots do not matter. Rather, it is the ability of pegs to connect to other pegs that matters. Therefore, we decided to create Dot objects (Peg slots) as well as Edge objects, and effectively link all peg slots to all potential neighbors, exactly a knight's move away. Further, this allowed us to easily update the graph to indicate which edges can be turned into colored connections and which have already been activated, so that the game did not have to keep track of as much regarding the order of moves (since a board is not completely determined by which pegs are on it, as connections cannot go through previously placed connections).

With this graph-like game representation in place, we first created an AI which plays randomly, just to establish a baseline by which to measure our later AIs by. Next, we worked on a greedy algorithm, which just picks the move which will give it the best board value. To do this, we created a board evaluation function as follows, where P_{player} is the number of pegs required to be placed to connect a path between player's sides and $P_{opponent}$ is the number of pegs required to be placed to connect a path between opponent's sides (if it is impossible to create such a path, P_x is ∞).

$$Board\ Value = P_{player} - P_{opponent}$$

To actually program this function, we implemented Dijkstra's algorithm, but we added some checks to differentiate valid edges/nodes in the graph based on which nodes are owned by the player/opponent, as well as adding some dummy edges between nodes on the borders with weight 0 since there is not a single start node and simple end node, but rather a row/column of start nodes and end nodes. If we have an NxN game board, the runtime of the heuristic function is therefore

$$O(E \log(V))$$

$$= O((8 * N^2 \log(N^2)))$$

$$= O(N^2 \log(N))$$

since there are $O(N^2)$ nodes in the graph, $O(N^2)$ edges (roughly 8 per node), and it must run Dijkstra's algorithm only a constant number of times (twice, once for the player and once for the opponent).

After writing this board evaluation heuristic function, it was easy to expand it to be used by the Minimax algorithm. After all, the greedy algorithm was just a special case of Minimax where the search depth was 1. This followed the Minimax pseudocode given in the textbook, but there were a few interesting details for our specific game and board representation. We had to be very careful about how we modified the game state when simulating moves because given just a peg slot (and color/whose turn it is), it is not trivial to remove that peg from the board so that it as if the peg was never changed. This is because when one places a peg, not only does the peg slot change, but also adjacent edges may become activated, and further, edges intersecting the activated edges must be modified to indicate that they cannot become activated at a later point. To avoid this problem of inadvertently permanently modifying the game state when simulating moves in Minimax, a record of everything that gets modified is saved so that the state can be

completely restored after simulating a given move. Second, we added some randomness so that the AI will not play the same way every time (this was set using a flag that could be easily changed for testing/experimentation purposes).

In terms of the runtime of our Minimax algorithm on an NxN board, we had as follows, where b is the branching factor, d is the depth of the search, and h is the runtime of the heuristic function for evaluating a single game state.

$$O(b^d h)$$

$$= O((N^2)^d h)$$

$$= O((N^2)^d * (N^2 \log(N))$$

$$= O((N^2)^{d+1} \log(N))$$

Our Minimax algorithm worked well enough on small boards, but as can be seen above the runtime grows exponentially at an incredible rate with respect to the board dimensions. To try to cut this down, we added alpha beta pruning to optimize the performance of Minimax, thereby allowing it to work better on larger boards and reach higher depths in a reasonable amount of time. We followed the pseudocode provided in the textbook, and were able to achieve a substantial speedup, though of course alpha beta pruning can only achieve a theoretical speedup from $O(b^d)$ to $O(b^{d/2})$. Using just random ordering of which nodes to expand first, the speedup is only $O(b^d)$ to $O(b^{3d/4})$, so at first this is all we did (performance in the experimentation section). To get close to the $O(b^{d/2})$ theoretical limit, we added a "best" first approximation move for the search to start at in the beginning of each increased depth. Our heursitic for determining the next best move relied upon taking the best move based on a search going only to the previous depth, so this was implemented after we implemented iterative deepening.

Iterative deepening was implemented by just doing a search to depth 1 using Minimax, then 2, then 3, then 4, etc. until either a time has been reached or a specified depth has been reached. The timing feature is useful in that it allows us to specify a maximum amount of time the AI is allowed to think for, and it will then have a move ready to go at that time, unlike a search to an arbitrary depth. Rather than just keep searching until the time limit, the AI is slightly more intelligent so as not to waste time by looking at how much time it took to compute the previous depth, and not attempting 1 depth further if we have already used up 1/(N^2) times a factor so as to be conservative (a.k.a. sometimes attempting to go to the next depth even if it is unclear if there is enough time to finish the computation, so that we always make the best move given the time allotted). Second, we added the option to limit the iterative deepening to a set depth. This may seem like it is exactly the same as just using Minimax to a certain depth without iterative deepening, but it is valuable for two reasons. First, it made testing our AIs against each other easier, since we were able to compare times to compute specified depths even when using iterative deepening. Second, it was helpful for the "best first" heuristic to be used by alpha-beta Minimax.

This heuristic function is quite simple- it just takes the best move from the previous depth and tries this move first when expanding the next level of the tree. This doesn't at all change the result returned, but it allows alpha-beta to cut off useless branches more often. Even if that node is not the best move when considering the next depth level, it is often at least a decent move, and so it gives the ability to cut off some branches more quickly while searching for the optimal branch. Specific results are shown in the experimentation section to show how much the actual speedup gained was, but theoretically, if our heuristic was good enough we could reach a runtime of

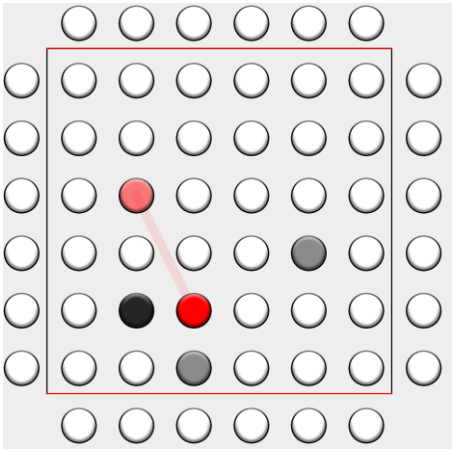
$$O\left(b^{\frac{d}{2}}h\right)$$

$$= O((N^2)^{\frac{d}{2}}N^2\log(N)$$

$$= O((N^2)^{\frac{d}{2}+1}\log(N))$$

We didn't quite reach this in practice, but there was still definite improvement.

Figure 5. Visual Thinking



Finally, we implemented a visual component of the game which allows us to visualize the Minimax search as it happens using transparent pieces which move around the board to show the AI's thinking process (Figure 5). While for the original Minimax AI this was not that informative, for the AI using alpha-beta search it was possible to see that it spent more time "thinking" about moves that were most promising, since it would cut off paths that it knew did not hold the optimal moves and therefore would eliminate those moves quickly.

Experimentation and Results:

To evaluate our AIs, we first played against them to see how well they perform against us. Generally, a decent human strategy could beat any of our AIs on large boards (N>12), but on small boards they began to give us a challenge. On the smallest board that isn't trivial, a 6x6, the player who goes first should always win if they play optimally. Minimax with a search depth of 4 is able to achieve this optimal play since the board is so small. For slightly larger boards (i.e. 7x7 and 8x8), the AIs play near perfectly at a search depth of 4, and we managed to have our fastest AI (alpha-beta with a best-first heuristic an iterative deepening) play optimally on a 7x7 with depth 6 (this game took over 15 minutes). We found that if the search depth is not set to at least

3 the AI is very easy to beat, and if it is set to exactly 3 it may win if the human player is not being careful, but really depth of 4 is needed to play well. Unfortunately, as the boards get larger, not only does searching to a fixed depth get more computation-intensive, but the higher the depth needs to be to beat a human player, since humans can think less locally and with a bigger picture in mind. Therefore, when getting to boards of size 12 and beyond, our AIs were not as effective against human players.

In addition to testing our AIs against ourselves, we matched them up against each other to see how they performed. We found, not surprisingly, that higher search depth AIs beat lower search depth AIs, on average, but it was possible for a depth 3 AI to occasionally beat a depth 4 AI, for example (because of the randomness factor and advantage of going first/second for some board sizes). Initially, we were experimenting with a few different heuristic functions for the board evaluation (for example weighting connections in the middle of the board more highly), and so we tried each against each other using the Greedy AI. Generally, we were satisfied with the results of our AIs, and the best were able to beat us fairly consistently on 8x8 boards given around 20 seconds to compute each move on our computer.

The time to complete a game against two AIs of the same type and depth are shown for each of the AI types in Table 1. As can be seen, the runtime goes up as depth increases incredibly quickly (clearly exponential). There is of course some overhead, and this explains why the time increase does not quite match the expected runtime increases for small depths. Table 2 shows the effect of fixing the AI to be the fastest one and fixing the depth, but instead varying the board size. While there is still a huge increase as N goes from 6 to 14, the growth is slower than for changing depth. Unfortunately, it’s a little tricky to see that, since when varying board size with depth fixed we have a polynomial of a very high order, but as N goes to infinity

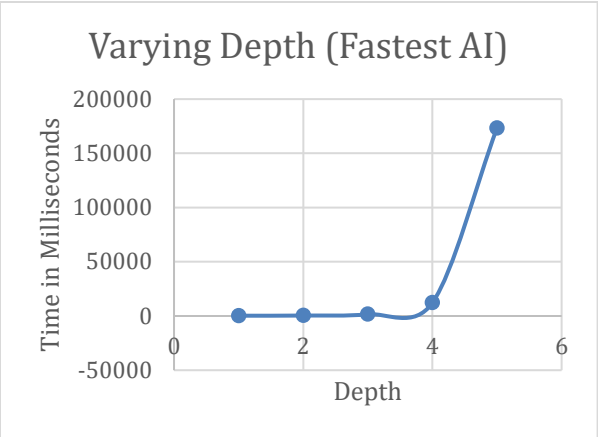
(and this can be seen to some extent by looking at the ratio of the times going from 8 to 10 and then from 10 to 12) it can be seen that the runtime is not exponential in the board size.

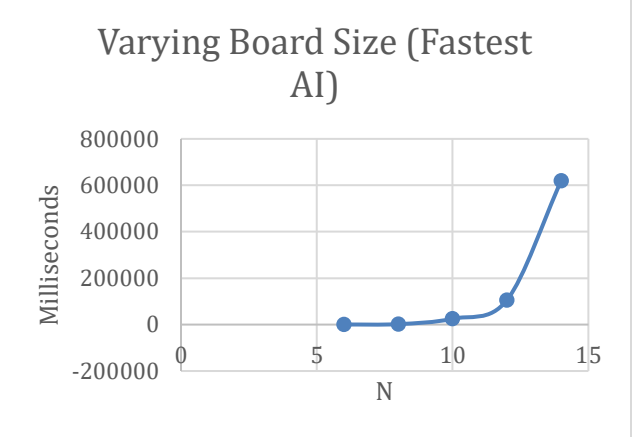
	Depth 1	Depth 2	Depth 3	Depth 4	Depth 5
Minimax	263	765	14478	>200000	>200000
Iterative Deepening Minimax	275	700	11440	>200000	>200000
Alpha-Beta	279	457	4334	30433	>200000
Iterative Deepening Alpha-Beta	271	434	1618	21488	>200000
Alpha-Beta Best First Approximation (w Iterative Deepening)	268	410	1604	12250	173398

Table 1. Time comparison of each AI playing itself on an 8x8 Board. Times are given in milliseconds, and represent the time taken for an entire game to be played until completion (on a specific computer). The average of 3 trials were used for each, and a hard cutoff of 200 seconds was used.

	N=6	N=8	N=10	N=12	N=14
Alpha-Beta Best First Approximation (w Iterative Deepening)	313	1952	25309	105627	691717

Table 2. Time in milliseconds to complete a game of two AIs facing each other on boards of size NxN. The average of 3 trials were used for each. Depth is fixed at 3.





Somewhat unexpectedly, iterative deepening performed considerably better in terms of runtime than its corresponding depth Minimax or alpha-beta search AI, even though it is doing slightly more search since it goes over some nodes in the game tree more than once. This was because we implemented our iterative deepening AI so that it will take a move at a level as soon as it knows it has won or lost the game, whereas our Minimax to a set depth will keep going to that set depth to see if it can at least prolong the inevitable if it is about to lose. On a small board (8x8) with considerable depth (>=3), this effect becomes fairly noticeable, since it is not long before one player knows they will win or lose.

Our best-first heuristic for alpha-beta search was used with iterative deepening, effectively expanding the node that seemed most promising at the previous depth level of the search first in the next depth level. In theory, this could have reduced the runtime considerably, but since the heuristic was greedy and not always helpful, the speedup was much more modest. Nevertheless, at depth 4 and an 8x8 board, it had about a 2x speedup. Looking at the trend, however, the speedup factor is higher for higher depths, which makes since given that a heuristic for picking the best nodes to expand first could theoretically improve the runtime of an alpha beta search from $O(b^{3d/4})$ to $O(b^{d/2})$.

Conclusion:

All in all, the Twixt AI was successful at tackling a hard search problem and seeing how far alpha-beta pruning could go for a game with such a high branching factor. Likely, to create an AI with significantly increased performance and the ability to play on boards of sizes around 24x24, no modification of Minimax could be used. Perhaps a future area of study with the game could involve techniques similar to that used by Google's AlphaGo, since the games are similar in that they have perfect information, a grid of pegs/stones, and a large branching factor. Deep learning could provide a system with the ability to develop more human, large scale strategies, instead of thinking only a few moves ahead. In Twixt, there is a large set of possible moves, but humans can easily eliminate most of them on any turn without performing any sort of exhaustive search. Rather, we can use higher level reasoning about general goals and the bigger picture while also thinking locally when we need to for specific tactics. The way our AI played Twixt was very different, only looking a few moves ahead at a time with a simple heuristic function but being very exhaustive in searching all possible moves within that small future window. Still, it managed to perform quite well with its limited understanding of the game on small boards, and on these boards it was able to beat reasonable human level of play.