Planning Movement on Player-Modifiable Maps

Jurie Horneman

1 Introduction

In 2016 I developed an AI opponent for a 2D turn-based indie game for PC (as yet unannounced and unreleased). In this game, both human and AI players create the level on the fly by rotating and placing up to three arbitrarily shaped board pieces per turn. Making an efficient movement planner for this game was an interesting challenge. I found no information on existing solutions for this particular problem, so I had to develop my own approach.

In this chapter I will describe the problem I had to solve, as well as the data structures and algorithms I implemented to determine where pieces can fit, and to build a plan that involves placing pieces as well as moving along the board. Finally I will describe some possible optimizations that I considered but did not implement.

2 The problem

Within the context of a turn-based strategy game, the movement planner has to find a path on a 15 by 15 cell grid towards a given goal position. At the start of the game, some of the grid cells are passable (Floor), but most of the grid cells are empty (Void) and cannot be moved across. At the start of their turn, each player receives three Tetris-like pieces which they can rotate and place anywhere on the board, as long as it's adjacent to an existing Floor cell. If the movement planner can't find a path leading to the goal position, it should build a plan leading as close as possible to the goal position.

Placing pieces leads to much bushier search trees than simple movement along the four cardinal directions: pieces can be placed in many ways. The fact that the AI can use up to three pieces at a time compounds this problem. Each time the movement planner runs it can generate bushy search trees with relatively complex search states – more than just a 2D position as used in typical 2D movement planning. An early Python prototype which only covered a part of the problem already showed performance problems. So the solution has to make it possible to control the number of search states that are explored and the number of expensive data structures that are built.

In the full AI I developed, the movement planner was called multiple times per turn with different goal positions, each corresponding to different tactical goals. Then one plan was selected using a utility-based approach, which considers the plan's length, as well as other factors. This part of the full AI is not further described in this chapter.

I will now describe the data structures and algorithms I implemented, starting from the core problem of finding whether a piece fits, up to building a movement plan.

3 Finding fits

The movement planner needs to build a plan that includes rotating and placing pieces from a small set of available pieces, in order to construct a valid path. Finding which pieces fit where is the first problem that has to be solved.

A naive approach would be to use A* for movement until reaching a Void (empty) cell, and then to try to fit every single piece at every cardinal direction at every position relative to the cell that could make the Void cell accessible. But since a single piece can be adjacent to many cells, this would mean recalculating the same fit many times over. Instead, I decided to try and precalculate every possible fit at once.

I first pre-rotate all of the pieces. This removes the complexity of having to rotate them later on: I can just process a list of up to twelve pieces.

The planning algorithm starts with a data structure called a **pathability map**. This stores the cell type, for pathfinding purposes, for each map cell – essentially whether the AI can pass through or not. Types are defined as enums so that only one bit is set for each value: this allows me to use bit masks for efficient combined queries. (Only Floor and Void are discussed in this chapter but the game had many more types.) The first pathability map is created from the game state when the AI starts, and then modified during the planning process.

From the pathability map I generate an **adjacency map**, which stores whether a given cell is Void and adjacent to a Floor cell. Pieces can only be placed in empty cells next to a non-empty cell.

From the pathability and adjacency maps I calculate a **fit map**. This stores for each cell which piece rows could fit there, and whether it'd be adjacent to an existing, passable cell. Pieces have a maximum size of 5 by 5 cells, so each cell in the fit map contains the data for the next 5 cells in the map. Effectively, this is a cache so I can check 5 consecutive cells at once. This data is stored as bitfields.

From the fit map and the list of available pieces, it is possible to make a list of all **potential placements** – every piece that could be placed. To do this I iterate over the map, then over each row of each piece. Given the current row of the current piece, also represented as a bitfield, I can determine whether the row would fit using simple bit operations. A row's non-Void cells must not overlap existing non-Void cells, and at least one of the row's non-Void cells must overlap with a cell that is set to true in the adjacency map.

Finally, from the list of potential placements, I build a **potential map**, which stores for each cell a set of the indices into the list of potential piece placements that would make this cell accessible.

The potential map and the list of potential placements are combined into a class called `PieceFitData`, which takes a pathability map and a list of rotated pieces, calculates the data described above, and provides a function which gets all potential piece placements for a given position, which is the question the movement planner needs an answer to.

4 Searching for a plan

In pseudo-code, classical A* works as follows:

```
while the frontier is not empty:
    get the best search state from the frontier
    if that is the goal state:
        exit
    for each state adjacent to the current one:
        calculate the cost of the new state
        if it is lower than our best attempt at getting
            there so far:
            remember this new, lower cost
            put this new state on the frontier
```

To find a path over a 2D grid, the "search state" is simply the position on that grid. In my case I need to track more data, namely:

- The pathability map, since the whole point is to be able to find plans that involve changing that.
- Which pieces are available for use, since each can only be used once.
- The number of moves that are left, since each player only has a limited number of moves per turn.

All of this is encapsulated in a class called `SearchState`.

Switching frontiers

As well as storing more data in the search state, I also added some complexity to the implementation of A* I used, in order to maintain acceptable performance.

`PieceFitData`, the data structure described above, allows the movement planner to easily determine which piece placement creates a path to a given position. However, it's not cheap to build, and it doesn't solve the bushiness problem described at the start. Each potential piece placement is a branch that the planner might explore, and that might lead to another instance of `PieceFitData` being built.

To keep this under control, whenever the planner determines it could move to a position by placing a piece (meaning it finds a valid adjacent search state), it adds this possible next step not to the main A* frontier, but to a second frontier. After normal movement has been exhausted, this frontier is pruned to the top 16 entries, and then planning resumes. (I arrived at the number 16 empirically, by looking at the number of search states that were generated versus the planner's ability to find plans in various test situations.) This approach is effectively a variant of Beam Search (Bisiani 87).

For movement, finding the adjacent search states is straightforward: I simply look at the pathability map for each neighboring position to see if it is a Floor cell.

For laying pieces the logic is similar. I call a function which, given a search state, returns the appropriate piece fit data. This function will be described in more detail further down. I then create a new search state for each potential piece placement on each Void cell neighboring the current position. These new search states are added to the second priority queue mentioned previously, which is then pruned once the current planning pass is over.

Listing 1 C# code for the multiple planning pass logic.

```
public void Plan()
{
    if (nrPlaceablePieces == 0)
    {
        nextFrontier = new PriorityQueue<SearchState>();
    }

    for (int pass = 1; pass <= nrPlaceablePieces; pass++)
    {
        nextFrontier = new PriorityQueue<SearchState>();

        ExecutePlanningPass();
        if (FoundAPlan) return;

        if (nextFrontier.Count == 0) return;

        frontier = nextFrontier;
        frontier.Prune(16);
    }

    ExecutePlanningPass();
}
```

In the code above, `ExecutePlanningPass()` performs the core A* logic, starting with the first frontier and returning either a goal state, or a second frontier. The `Plan()` function has to do as many planning passes as the AI player has pieces to place, and then one more, because after having potentially placed a third piece, we can maybe still move *over* that piece.

After each planning pass, the planner either found a plan that leads to the goal position, or nothing, or a list of promising states that involve piece placement. In the latter case, the list is pruned down to a small number, moved to the main priority queue, and the next pass begins.

Finding the plan that gets as close as possible to the goal position is important but simply involves keeping track of the search state with the best score and A* heuristic value.

5 Hashing of search states and pathability maps

To find adjacent search states that require the placement of a piece, I call a function that returns the piece fit data for a given search state. Since piece fit data is expensive to calculate, I want to build it only when necessary, and reuse it if possible. In order to achieve this I use a dictionary that maps a hash value to a fit data instance.

The `SearchState` class has a method called `HashForFitData()` which XORs the hash values for the available pieces (a bitfield) and the pathability map. These are the only factors that determine whether the piece fit data would be correct for a given search state.

The pathability map has a hash function that uses Zobrist hashing (Zobrist 69) to efficiently calculate a hash value. This hash function was very carefully written and tuned for this caching operation.

6 Possible optimizations

My optimization strategy was as follows:

- Ignore low level optimizations: these can be taken care of by profiling during a later optimization pass.
- Focus on algorithmic optimizations that **reduce and control** the number of expensive operations.
- Identify and make possible **localized optimization** possibilities, where I initially use a simple correct implementation, but know that I can later replace it with a faster, but more complex algorithm.

There are a number of optimizations I didn't implement, for the reasons described above, but which are worth mentioning.

Faster fit maps

The fit map caches 5 cells of fit data for each position. Given that this data takes 2 bits per cell, and the maximum size of pieces is 5 by 5, it would be possible to cache an entire 5 by 5 area's worth of fit data in just 64 bits of data (32 for the pathability data, 32 for the adjacency data). This means determining whether a piece fits could be done with 2 binary operations instead of 10.

Quantum search

I tried to ensure that creating new search states for piece placement was as cheap as possible. Quantum search is my name for an optimization which avoids creating most of them altogether.

After taking the second A* frontier and pruning it down to the 32 most interesting potential piece placements, I generated another map that stored for each cell which of these

32 piece placements would make this cell accessible. This **quantum map** effectively superimposes all of the different piece placements that take us to a given location. I then wrote a second A* implementation which starts at a Void cell adjacent to a Floor cell, takes the bit mask for that position in the quantum map, and for each neighbor in turns ANDs it with the bit mask there. If the result is zero, there is no way to build a path between the two cells using piece placement. (This is a fast way of storing sets of possibilities per cell, and calculating the intersection of these sets during traversal.) This way it is possible to search through all of the piece placement search states simultaneously, without having to create pathability maps until more of the search space had been explored.

I found this an exciting technique. I wrote a prototype that worked for a single pass. However, extending it to multiple passes would have introduced significant complexity, especially given some of the special logic in the game, so I chose a less clever approach.

7 Conclusion

Pathfinding in games where players are allowed to modify the game board by placing complex board pieces presents some special challenges compared to simple pathfinding. Efficiently determining which pieces fit where can be solved by precalculating all of the fits at once. Judicious pruning of the search tree, like in beam search, helps keep the number of search states that are explored under control. Finally, caching ensures that expensive data structures are not built more often than necessary.

8 References

[Bisiani 87] Bisiani, R. 1987. Beam search. In *Encyclopedia of Artificial Intelligence*, ed. S. Shapiro, 56–58. New York: Wiley & Sons.

[Zobrist 69] Zobrist, A. L. 1969. A New Hashing Method with Application for Game Playing, Technical Report 88. Madison: University of Wisconsin.