Trix - A tetris playing intelligent agent
CITS4211 Project, 2013

Darcy Laycock (20369588)

May 28, 2013

Abstract

This report investigates the implementation and design of "Trix", an intelligent agent written in Python designed to play a modified version of the popular video game, Tetris.

Contents

1 Introduction

2 Problem Analysis / Breakdown

When designing trix, there were several prime concerns initially in understanding the game of tetrix. One thing worth nothing is that the design of the game of tetris itself is complex - each state is dependendant heavily on the actions building up to it, thus the game has an incredibly large combination of possible world states and hence is infeasible to precalculate much of the world.

Likewise, at any stage of play, the branching factor is very high (the version given in the project documentation is approximately 26 - This increases with the number of items in the buffer), leading to a world that is hard to brute force search.

The goal of the game is to find an end state that minimizes the end height of the board. One of the games condition is that when all of the cells of a given row are filled, that row is cleared - thus, lowering the height by 1 and increasing the number of cleared cells by 1 simultaneously. Likewise, by minimizing the height we also aim to maximize the number of rows cleared (a similar goal, but not always map 1-to-1 with height due to the placement of pieces).

Because of these, with Trix we aim to find an algorithm for the agent that is capable of:

- Performing in a minimal amount of time.
- Considering the effects of placing items in a buffer.
- Considering the effect of placing the item directly.
- Considering the effect of playing a piece from the buffer.
- Aims to end with the mimimal total height.

Given the broad number of options, our hope is thus that we can attempt to simplify the problem by breaking it down to a simpler, easier to optimise problem. The problem at hand, by default, one of search - how do we find the state with the minimal final height?

The problem with this question lies in the fact that minimal is a qualitative measure instead of a quantitative measure. In order to say a final height is minimal, we need something to compare to. To make it something which an actual goal (something we can say is a goal node based only on the information present in the node), we must thus rephrase it into a different question. Instead, We wish to final the path that ends with a board of height X. Thus, by searching iteratively, increasing X (from a default value of zero - meaning the game ends with nothing on the board) until we find a value of X that terminates the search with a result.

Also worth noting at this stage is there is a second way of expressing this goal - maximizing the number of rows cleared at a given point. Trix chose against this since there is a lot more variation in the optimal goal (e.g. what is the maximum cleared of the board) than the minimum height -

we can assume zero for height and work upwards, hitting the target a lot easier than going in the inverse direction.

If we were to use an uninformed search algorithm for it, we may still have an absurd number of combinations due to the relatively high branching factor (one that varies even further with changes to width of the board and the games buffer size), Hence we need to simplify even further in order to find an approach that will run in reasonable time.

For this, we can take advantage of several factors. First, we witness that there are portions of the game that zero out - that is, we get chains of action / piece combinations that can zero out the rows. In any given game, it's possibly for the height to vary from zero to a maximum height and back to zero again, even on portions of input many pieces long. Hence, we don't need to necessarily search for a zero over the whole game, we only need to search for a series of actions that effectively cancel themselves out.

Next, we can perform some optimizations based on our choice of the search algorithm and implementation to further improve the performance by tweaking how we search and what exactly we're searching for.

With all of this in mind, we thus need some test cases which exhibit the behaviour of this - along with some traditional input. Hence, we test situations (and board sizes) that are known to have simple solutions (e.g. even width and square blocks) and cases such as alternating s / z which are hard to place and fill and sequences of the I pieces which require good look ahead to choose an optimal placement (e.g. ideally we just place them vertically, since they take up width 1).

3 Agent Design

Using the understanding of tetris and the issues associated with it as found in section 2, an agent program called "Trix" for playing tetris.

3.1 General Structure

The agent program for trix is implemented in Python - a language chosen primarily for it's quick feedback / testing flow. The structure itself follows the general agent approach specificed in AIMA with an Agent class responsible for making decisions and classes for the Environment and Percept.

When exploring the code, the following bits are likely to be of prime interest:

trix.environment.Piece - A piece in tetris, capable of performing rotations and various debugging-related operations on the piece.

trix.environment.Board - The board is the actual playing field of tetris - pieces are placed onto the board.

trix.environment.Environment - A combination of buffer, history and environment that combines the game.

trix.agents.Action - the base action of the object, subclassed for the varying types of the actions.

trix.agents.Agent - An abstract agent interface tying it all together. Subclassed for our actual agent.

3.2 Algorithm Design

For testing purposes, the agent algorithm in Trix started out in two simple forms. The first, used exclusively for testing the framework in which Trix runs, was a random agent that simply chooses a random placement and rotation for each piece encountered.

The second and more useful approach, was a very basic breadth-first-search based algorithm that expands all nodes up to a given depth. For early technical reasons, this approach doesn't use the buffer at all and the default maximum depth (a term explained shortly) is one. Due to the high branching factor (which means the options to explore are exponentially related to the depth and easily run out of time and space). Unfortunately, this isn't a suitable agent for real world game play (with the example input file, run time went from under one second for a max depth of 1, a run time of 3-4 minutes for a max depth of 2 and 20-30 minutes for a max depth of 3. For obvious reasons, tests were not run with a depth higher than that.

The final, and more suitable agent used with in Trix is based on a variant of A-star search with several different optimisations in place to make it suitable for use in the game. This is the default agent and what you use when running trix in the current form.

First, before exploring the implementation, it's worth expanding a little on the structure of the world as used by Trix. In terms of graph search (or, more accurately in this form, tree search), a node is simply a combination of four things: A tetris board state, a buffer and a history of actions applied to the board leading to that state. Finally, it's composed of a list of upcoming pieces for the game board.

Actions are defined in three forms: PlaceNextPiece, which will takes the next incoming piece, consumes it from the list of upcoming pieces and then places it at a given left offset on the board. The second form, AddToBuffer, simply consumes the next piece from the pending list and places it in the specified environments buffer. Finally, PlaceFromBuffer takes a piece from the buffer and places it at a given position.

Hence, the initial search state begins with an empty board, a full list of items, empty history and an empty buffer. In search, we thus define the children of that node as the results of possible actions - and then we update that to construct the root node as time goes on).

Using the search algorithm as such, our goal is to find a series of actions (where depth is the number of successive actions applied) which leads to the target end state. Since the smallest height

isn't a quantitative value (or, more aptly, can't be found without first finding all possible heights), we instead start by searching for a node with the height of 0 and then terminating the search either when we've found the goal node or we've exhausted the search.

Since we're not guaranteed there is going to be an end state with a given height, we thus intend to iterate over the heights, starting with zero and increasing to a maximum (defined as the height of the starting board for a search), as was described in the analysis section.

Since we're using A-star, the key part of the search algorithm is hence a distance metric (from the goal node) and a suitable heuristic to use in the calculation of this. Since choosing an admissible heuristic isn't incredibly important (we don't wish to have a complete search, and we don't want an optimal result - just one pretty good for speed reasons), Trix currently uses a fairly trivial approach with the intention (unfortunately, not yet done) to improve upon it in the future.

In this case, the current version of Trix uses a distance measure based on the height of the board (where we optimise for the most rows cleared - a height at the end means we've cleared more rows) and the metric being the maximum height for performing a given action.

Adding to a buffer is the current height + 0, any piece placement finds the maximum depth at the current position / span for the piece, subtracts that from the piece height and then adds the resultant number to the current height - for instance, if we're placing a piece 4 cells tall in a valley that is 2 under the maximum height at it's deepest and the current board height is 3, our estimated best-case height after (noting we don't consider clearing rows) is 5. If, on the other hand, the current board is 6 high but the valley is 5, the estimated distance is 2 (since we can clear up to 4 rows at once). This heuristic is far from optimal, but in test cases, appears to give reasonable results (more time is expected to lead to a better heuristic and distance measure).

Finally, as two levels of performance optimisation, several tweaks were added to the algorithm to make it run in reasonable time:

First, the A-star search cuts off after exploring 200 nodes. This, unfortunately, means a much smaller space is explored (without an optimal distance / heuristic metric) but means that the algorithm generally, in testing, runs at approximately 5 pieces per second tested. Likewise, we cutoff of expansion of a node if it's further ahead in the future than the current cutoff depth (defaulting to the buffer size times 2 - to allow adding in, replacing, and taking out - a full buffer). We default to a minimum cutoff depth of three, to ensure it performs acceptably with empty buffers.

Lastly, since the initial goal node has a target height zero, and we increase that for each subsequent search - we can take advantage of the fact that the search set for 1 is likely to be reached before the search set for 0. Since changing the target height doesn't change the node ordering, we thus node that searches for height X+1 will be included in every failing search for X. Because of this, we will have already performed a search for all heights greater than zero when we perform the search for zero.

Hence, we thus keep track of the best path to a given height. In this case, we define the best path

as the longest path and, when the given path has the same length of others, the one ending in the best end state. We define the best state using utility metric also used in BFS-based agent, defined based upon a weight sum of features such as height, cleared rows and unfilled cells.

Thus, once the search for the zero finishes, if we haven't found a path of actions that resulted in the goal node, we look at what the minimum height we did encounter was and use that instead. This approach greatly cuts down the number of duplicate searches (especially with height heights) to a single search and from analysis and testing gives the same results.

3.3 Running the Agent

Running Trix requires only access to Python 3 - all other code is implemented in Trix itself. From a command line, if you have a unix system with python in the path, your command to see usage is roughly:

```
./bin/trix --help
```

Or, if you're using windows, simply invoke trix using python like so:

```
python3.exe ./bin/trix --help
```

Normal usage, ignoring invocation, is roughly:

```
./bin/trix input-file output-file
```

For example,

```
./bin/trix data/in.a data/out
```

Which will run it using the pieces in data/in.a and output to data/out.

Also note that you can use the -w option to specify the board width (defaulting to 11) and -b to specify the size of the buffer (defaulting to 1).

4 Agent Analysis

This will be broken down into two sections, the first being observation analysis (based on observing sequences of games, using the modified version of DrawPlacement.py in the tools/ directory of the code which produces a gif of the game) and analysis based on the search size used in the game.

4.1 Observational Analysis

Running the program, and comparing the choices it makes at each stage and the results it gives, the following was observed about the performance of trix in it's current incarnation:

- With basis, traditional looking games (e.g. with a fairly even distribution of pieces), trix does a reasonable job of placing pieces. It allows the height of the board to grow at times, making odd looking decisions at times but placing pieces that successfuly clears out rows when available.
- With the simple square cases and even width, the bot performs as expected - clearing out all rows (the easy one).
- When playing any board size that isn't four with the I pieces, the algorithm unfortunately doesn't look far ahead enough to notice that it can just place them vertically - instead opting to use the flat bar-style placement to minimize height locally.
- In alternating sequences of s and z, it does a reasonably job in initial conditions of clearing the board (doing so at a reasonable pace), but still allows it to out pace itself rather quickly, building up a large height.
- Without the hard limit of the search, the implementation is still far to slow (suggesting a better heuristic and possibly distance method needs to be devised), but still gives a reasonable result when terminating the search early.

4.2 Experimental Analysis

Removing the hard coded limit on searches (allowing the algorithm to run a full search), we unfortunately see that the time it take to operates ballons.

For a series of measured tests (capturing the number of nodes expanded on each iteration and the max depth encountered), we defined the following test cases:

data/in.a A repeating series of I pieces (1 repeated)

data/in.b A repeating series of the square pieces (2 repeated)

data/in.c A chain of I, Square and the two hook pieces repeating. (1245 repeated)

data/in.d Cycling s and z pieces (67 repeated)

data/in.e Each of the pieces, cycled in order repeatedly.

data/in.f An arbitrary set that is heavy on 1-4.

data/in.g The example input

In each of these cases, the depth the search hits tended to be the maximum cutoff limit (as defined by 3 times the buffer size - in this case 1) - with very few (either, only trivial choices - such as those that clear rows and the estimated distance was accurate).

For each example above, with a buffer size of one, the branching factor was, respectively:

data/in.a Branching factor = 7.525

data/in.b Branching factor = 7.003

data/in.c Branching factor = 8.080

data/in.d Branching factor = 7.847

data/in.e Branching factor = 7.989

data/in.f Branching factor = 8.403

data/in.g Branching factor = 7.936

This is an good improvement on the base branching factor of approximately 26, but still far from optimal.

Unfortunately, increasing the buffer size led to an expontential increase in the search time, showing that it didn't unfortunately improve upon the results - That is, the run time of the algorithm from experimental analysis was exponential with regards to the buffer size (in it's naive form).

5 Conclusions

All in all, the end results show that whilst Trix can play the modified game of tetris as given, it doesn't necessarily do very well. It's ability to look into the future being severly limited meant that its results only worked well for cases where it tends to have an fairly even distribution of pieces leading to easy clearing patterns.

In the two given easy cases (which require decent look ahead), it still managed to choose suboptimal placements, especially in data/in.a which would end with either zero or four rows, managed to end up with a larger height due to the limited look ahead.

This shows that whilst Trix does improve on the raw, brute force approach, it still needs much more improvement to make it a viable competitor - but it was a good learning excercise none the less.

5.1 Future Directions

The design of trix leaves much room for future improvement. The following are future options that could be investigated to provide further improvement in performance and how they would actually help:

Better search heuristics and distance: The current method of calculating distance from the goal node (and an associated heuristic for it) are far from ideal. A better representative (e.g. space filled) would lead to better performance by prioritizing better boards first. Another would be to test and see if cleared rows (versus height) is a better option.

Improved Utility measurements: The current utility function (used primarily in choosing the best option when comparing choices for a given height) is pretty naive, using fixed weights and measuring only height, cleared rows and the number of cleared items. Future improvement

could use reinforcement learning to improve this value as we learn more about different pieces and their placements on the board.

References

- Russell, S & Norvig, P 2010, *Artificial Intelligence - A Modern Approach,* 3rd edn. Pearson Education, New Jersey.
- *aima-python*, 2011. Available from: <https://code.google.com/p/aima-python/>. [18th of May, 2013].
- *Playing forever - tetrisconcept*, 2009. Available from: <http://tetrisconcept.net/wiki/Playing_forever>. [2nd of May, 2013].