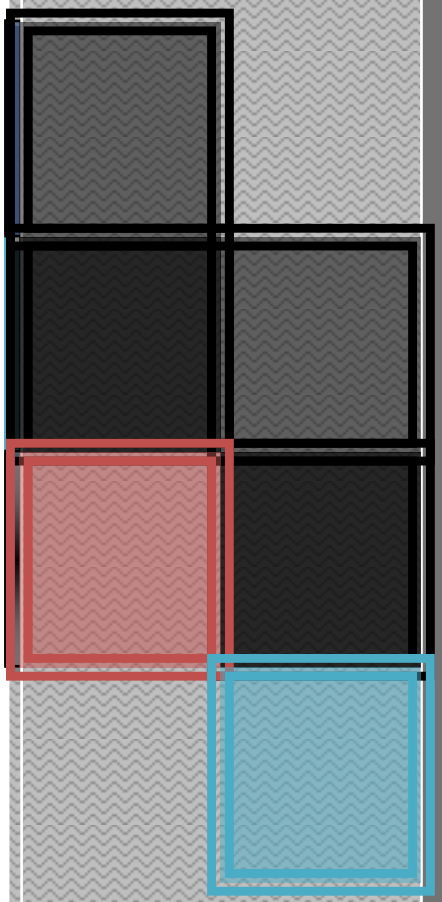
2011

TETRIS

Local Searching an Intelligent Tetris playing Agent

In this project we tried to create the ultimate Tetris game player.
The approaches we used are modifications of Local Search – Stochastic Hill climbing and Beam Search based on weighing game playing heuristics.



Gil , Alex and Michael
Hebrew University of Jerusalem
3/2011

TABLE OF CONTENTS

INTRODUCTION TO TETRIS

Tetris is a popular one player game in which a random tetromino (from a possible seven **tetrominoes**) falls at a set speed down the field while the player moves it about to make it land in such a way that full horizontal lines of tetrominoes are created.

Figure 1 shows the **possible tetrominoes in Tetris**, where each is a different combination of four joined squares. The player can control the horizontal movement of the falling **tetromino** as well as being able to rotate the tetromino 90 degrees in either direction, provided the action can be done without intersecting another block or the boundaries of the field.

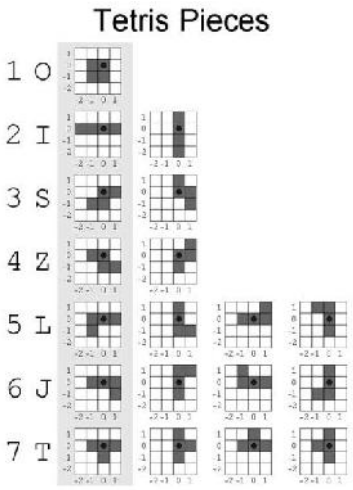


Figure 1: Possible tetrominoes

In the standard definition of **Tetris**, the field size is set as 10 blocks wide by 20 blocks high and the probability of each piece is equal. In most versions of **Tetris** where a human is the player, the next tetromino to come is displayed as additional information to help players set up strategies. This is known as two-piece **Tetris**. In one-piece **Tetris**, only the current falling tetromino is known.[1] (Sarjant, 2008)

[1] Introduction section is heavily based on the introduction section in the cited paper

PROJECT OVERVIEW

Our first choice to tackle the **Tetris challenge** was to try and implement a reinforcement learning algorithm, and design our agent to learn **Tetris** through rewards. After searching the internet we reached the understanding that solving the **standard Tetris** (10 blocks wide on 20 blocks high with the standard pieces) is virtually impossible since the state space is 2^{200}. Of course relaxation techniques in board representation are possible and will limit the state space by a large factor, but relaxation holds many easily seen disadvantages. For example, a popular relaxation is treating a board as 8 different boards of width 4 (the widest piece in Tetris), does limit the state space, but the board is not treated as an atomic unit which a change in one part can effect the other.

Another relaxation option we saw in previous research in this field was using smaller tetrominoes, but our initial aim was to create **a real Tetris player agent**, so we decided on a different approach.

We experimented with various Tetris **playing heuristics**. Each heuristic is a board evaluation function, which assigns a score to any given board. For each board and each falling tetromino, our agent will try all options of placing the piece on top of the pile. Since there are 10 columns and a maximum of 4 ways to rotate each piece, for each piece our agent will evaluate 9 (on the 'O' piece) to 32 (on the 'L', 'J' and 'T' pieces) different boards and decide on the preferred location for this current piece.

Since we are playing the one-piece **Tetris** game, there is no more information which the agent can use.

We implemented **5** heuristics and decided on a sum of 1,000 points to distribute between the heuristics. Each game receives a different tuple of five integers and throughout the whole game will use the same weighting tuple on each piece decision. The sum of all the tuple values is a constant in order to prevent different tuples from generating the same decision function (essentially [0,1,2,3,4] is equal to [0,2,4,6,8]). The larger the sum of the weights is, the larger the search space for an optimal weighting is. A more detailed explanation will be given in the "**Local Search and Implementation**" chapter.

After deciding on the method of play for our agent and our chosen heuristics, the only question left unanswered was how to weight each board evaluation function in order to create the best **Tetris playing agent**.

SOLUTION SPACE:

The number of possible solutions in whole numbers to the equation $x_1+x_2+x_3+x_4+x_5=1,000$ is $\binom{1,000+5-1}{1,000}$ = **42,084,793,751**

Since we cannot cover the entire search space we decided to use **local search algorithms**. Since we want to compare the different tuples, we can't run each game (with that tuple) on a random game. There are easier games (e.g. a game consisting solely of the 'O' piece) and harder games (e.g. a game consisting of the sequence 'Z', 'S') and we want to compare and evaluate each node in our search space in order to find the global maximum.

In order to compare the nodes, we generated **10 random games of 7,000 pieces each**. The average piece count on running the agent on this training set with the current weight tuple is the score of the search node. After a game reached **20,000** pieces we stopped the game with that score (to prevent infinite loops).

After running the local search algorithm on **3,663** search nodes (**36,630 Tetris games**), we chose the 7 best agents and ran them on a test set - 15 games of **12,000** random pieces (again stopping a game that reached 20,000 pieces). The winner of the championship tournament was declared as the **best Tetris playing agent**.

VOTING

Deciding between the best players after the championship was a problem of its own. We did not want to choose solely on the highest average, since there is no guarantee that the agent scoring **highest on average** in the second training set, will score highest on any random game. Each of our contestants was better in different of the random games, and we want to ensure an all around best player. An agent that has three very high scores, but plays poorly on other games, may have a high average, but is not the balanced reliable player we are looking for.

Following that, we used different **voting techniques** in order to decide who our champion agent is. We ranked the agents on how they scored at each game, and treated each game ranking as a single voting ballot. Of course different voting techniques produced different champions, but the overall sum of voting methods, produced the agent we believe has won.

BOARD EVALUATION HEURISTICS

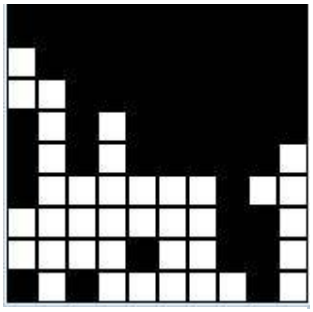


Figure 2: Max Height = 8, Average Height = 4.7, Holes = 9, Bumpiness = 15, Valleys = 2

All the heuristic functions are in fact board evaluation function. They are all minimum functions, meaning they all evaluate the empty board with value 0 and the worst board (in their opinion) with a higher value.

The values of the functions do not necessarily return values of the same magnitude, since they are weighted later. A function with a relatively high value could possibly receive a lower weight in order to even the effect.

MAX HEIGHT HEURISTIC

This heuristic measures the maximum height of the pile. An empty column has a height value of zero. The value of the function ranges between **0** and **20**. In **Figure 2**, the max height value is **8**.

AVERAGE HEIGHT HEURISTIC

This heuristic measures the average height of the pile. That is, the sum of all column heights divided by the board width. Since we originally thought that this heuristic is very important, we doubled the value returned. The value of the function ranges between 0 and 39.

In **Figure 2**, the average height value is **9.4 (4.7*2).**

HOLES HEURISTIC

This heuristic counts the total number of holes in the board. A hole is defined as an unoccupied unit which is underneath an occupied unit. In another sense, this is a part of the board that cannot be filled directly down from the top of the board. The value of the function theoretically ranges between **0** and **160**, but it rarely passes the value of **40**.

In **Figure 2**, the holes value is **9** even though some of the holes are not surrounded by occupied units from all directions.

BUMPINESS HEURISTIC

This heuristic is a measure of the bumpiness of the board, more accurately, the sum of the absolute height differences between neighbouring columns. In **Figure 2**, the contour vector of the board is **{8,7,4,6,4,4,4,1,4,5},** thus the height difference vector is **{-1,-3,2,-2,0,0,-3,3,1}**. The bumpiness value of **Figure 2** is **15**.

The value of the function can theoretically range between **0** and **171**, but it rarely passes the value of **60**.

VALLEYS HEURISTIC

This heuristic is the only one that is piece oriented. It counts the number of units which fit solely the **'I' piece**. More formally it sums the unoccupied units (x,y) that have at least two unoccupied units directly above them (**y+1**, **y+2**), and that the column height of the two neighbouring columns are at least the height of **y+2**. These units are in a **valley**, and only the 'I' piece can reach them.

In **Figure 2** the value of this function is **2**. They are at locations **(1,4)** and **(8,2)**. The unit (9,1) is not a valley since the height of column **8** is only one. Had location **(1,3)** in **Figure 2** been unoccupied, the valleys function value would have been **3**.

The value of the function can theoretically range between **0** and **80**, but it rarely passes the value of **10**.

LOCAL SEARCH OVERVIEW AND IMPLEMENTATIONS

In order to find a "best agent", we use a modified version of **stochastic random-restart hill-climbing search** with some additional elements of **beam search**. This will be explained in this chapter. See references for further information regarding the searches.

HEURISTICS:

We use 5 heuristics to evaluate the agent's moves (all described above). The influence of each of the heuristics are different and are set by a "**weighting vector**" which consists of 5 numbers that represent the weight of each heuristic accordingly (the numbers are normalized and sum up to a 1000).

Every move, the agent evaluates each possibility with each one of the heuristics, these values are multiplied by the heuristic's weight and give the overall score – the action with the highest score is chosen to be played.

We started with a random "weighting vector". The basic idea we implemented is to "improve" the weights in a way that will make the agent play better.

Each "**weighting vector**" is evaluated ("played") on a constant set of "tetrominoes" up to **20,000** pieces. The achieved game score is stored as an evaluation for the run.

All the runs are stored in a **database** file, each tuple is a combination of a weighting vector and score evaluation.

IMPROVING THE WEIGHTING VECTOR:

In order to improve a weighting vector, we start with one of the vectors we had already evaluated and have its score, and try to change the weight of one of the heuristics in a way that will generate a higher score.

First a weight vector is chosen (how to choose the vector is chosen is explained later), then we iterate over each one of 5 heuristics, change their weights (see weight iteration) and evaluate a new vector on a constant set of shapes.

WEIGHT ITERATION:

We go over the vector and change 1 heuristic's weight at a time. We increase the weight by a **100**, normalize the vector and evaluate the result. The result is stored in the **database**. Then we do the same for the other four heuristics' weight in the vector (changing only one heuristic from the original vector at a time).

After this iteration is over we chose the highest scoring vector (from the 5 variations and the original). This will now be our new original vector and we repeat the process but increasing the weights by only **90**. We run the evaluation and add the results to the database. This process is repeated but we increase by values of **80, 70, ..., 20, 10** and **5** as the iteration continues.

This part of the **Local Search** is the **hill climbing** part. We start from an initial vector, and climb to its best neighbour. All together we perform 10 continuous climbs from our starting vector. Each climb has a smaller step.

Different ranges (**5-100**) of "step sizes" were chosen to be checked for three reasons. The **first** is that if we are currently in a local one step maximum, taking the same sized step will only lead us to check the same neighbours again. We will remain in place.
The **second** reason was that we want to advance towards the maximum relatively fast on the one hand, but reach the maximum in a close approximation on the other hand. The large steps will bring us up the hill faster and the small steps will help us find the actual peak with high proximity.
Third is that taking a good playing vector and modifying one of the heuristics even a little bit - can lead to drastic changes in the score.

CHOOSING AN INITIAL VECTOR:

The last aspect we wanted to take care of is how to "**intelligently**" choose a "**good**" vector to start with each time. It obviously has to be done so that we can use the previously found **good solutions**, and progress faster.

The basic idea was to choose the best existing vector (highest scoring vector) from the **database** and work on it. **BUT!** In this way we can be threatened by finding a

"**local maximum**" and not finding the **BEST** solution. So, we use stochastic elements to avoid this.

Stochastic and Beam Search elements:

For choosing a vector, we find the 10 best playing vectors we have already checked in the database, one of which is then randomly chosen to be the starting one. This is the general idea of **a beam search**.

This however, can also lead us to a local maximum, since we are always focusing on a very specific group of vectors, while the solution may lie in a totally different region of the search space. Therefore, to avoid this, we used **stochastic random-restart hill-climbing**. We allow the algorithm to switch the entire search location with some predefined probability.

In our case we do this with probability of **50%**. Half the runs, the initial vector to climb from will be generated randomly, which allows further exploration and also moving to different parts of the solution space. The other half of the time will be distributed stochastically between the currently best search vectors (**5%** chance for each).
50% seems like a high probability, when in generally commonly known techniques, much smaller numbers are used. Our motive for using such a large random probability was that we have no idea of where the best solution may lie. That is since we have 5 different heuristics each one of which is independent and can improve algorithm all by itself. We did not want to get stuck in local maxima even though it could look good at that moment.

After drawing one of the currently best search vectors, a slight randomization is performed on it in order to increase state-space flexibility and to decrease the chance of running the same vectors many times through the game. The action performed is to go over each of the weights and add to the weight a number in the range (-20,20), each addition is in equal probability ($^1/_{39}$). Of course after the slight randomization to each weight, the whole vector is normalized to sum to 1,000.

DATA SHARING:

As mentioned above, **each run** is stored in the **database**. That is done in order to save time on future runs. Before evaluating a vector, we check if this vector was

already run and skip the game (which can take 2-7 minutes) if it was. This saves a lot of time, and allows for checking a larger number of "**weighting vectors**".

MULTI THREADING:

Since each run of the algorithm is independent, we can run many instances of it at once. All of them share the same **database** and add the results to it. The fact that they are running together, sharing the database and choosing from the current best vectors gives us a huge advantage in speed of exploring the solution space.

GENERAL SEARCH TOPICS SUMMARY:

LOCAL SEARCH:

Local search is a **meta heuristic** for solving computationally hard optimization problems. **Local search** can be used on problems that can be formulated as finding a solution maximizing a criterion among a number of candidate solutions. **Local search algorithms** move from solution to solution in the space of candidate solutions (the search space) until a solution deemed optimal is found or a time bound is elapsed. (Wikipedia, Local search)

HILL CLIMBING:

Hill Climbing is an iterative algorithm that starts with an arbitrary solution to a problem, then attempts to find a better solution by incrementally changing a single element of the solution. If the change produces a better solution, an incremental change is made to the new solution, repeating until no further improvements can be found. (Wikipedia, Hill climbing)

STOCHASTIC HILL CLIMBING

This is a variant of the basic **Hill Climbing** method. While basic hill climbing always chooses the steepest uphill move, **stochastic** hill climbing chooses at **random** from among the uphill moves. The **probability** of selection may vary with the steepness of the uphill move. (Wikipedia, Stochastic hill climbing)

BEAM SEARCH:

Beam search is a **heuristic search algorithm** that explores a graph by expanding the **most promising node in a limited set**. Beam search is an optimization of **best-first search** that reduces its memory requirements. **Best-first search** is a graph search which orders all partial solutions (states) according to some heuristic which attempts to predict how close a partial solution is to a complete solution (goal state). In beam search, only a predetermined number of best partial solutions are kept as candidates. (Wikipedia, Beam Search)

GENERAL VOTING TOPICS SUMMARY:

CONDORCET WINNER

The Condorcet candidate or **Condorcet winner** of an election is the candidate who, when compared with every other candidate, is preferred by more voters. Informally, the Condorcet winner is the person who would **win a two-candidate** election against each of the other candidates. A Condorcet winner will not always exist in a given set of votes. (Wikipedia, Condorcet Winner)

COPELAND'S VOTING METHOD

Copeland's method is a Condorcet method in which candidates are ordered by the number of **pairwise victories**, minus the number of **pairwise defeats**.

Proponents argue that this method is easily understood by the general populace, which is generally familiar with the sporting equivalent. In many round-robin tournaments, the winner is the competitor with the most victories.

When there is no Condorcet winner this method often leads to ties. (Wikipedia, Copeland's method)

INSTANT RUNOFF VOTING

Instant runoff voting is a voting system used typically to elect **one winner by ranked choice voting.**

The winner is decided as follows:

1. In the **first round**, votes are counted by tallying first preferences (in the same way as plurality voting, or First-past-the-post).

2. If **no candidate has a majority of the votes**, the candidate with the fewest number of votes is eliminated and that candidate's votes are counted at full value for the remaining candidates according to the next preference on each ballot.
3. This **process repeats** until one candidate obtains a majority of votes among the remaining candidates. (Wikipedia, Instant-Runoff Voting)

BORDA COUNT

The **Borda count** is a single-winner election method in which voters rank candidates in order of preference.

The method determines the winner of an election by giving each candidate a certain number of points **corresponding to the position** in which he is ranked by each voter. Once all votes have been counted the candidate with the most points is the winner. Because it sometimes elects broadly acceptable candidates, rather than those preferred by the majority, the Borda count is often described as a **consensus-based** electoral system, rather than a majoritarian one. (Wikipedia, Borda Count)

MINIMAX REGRET

Regret (often also called **opportunity loss**) is defined as the difference between the actual payoff and the payoff that would have been obtained if a different course of action had been chosen.

The **minimax regret** approach is to minimize the worst-case regret. The aim of this is to perform as closely as possible to the optimal course. Since the minimax criterion applied here is to the regret (difference or ratio of the payoffs) rather that to the payoff itself, it is not as pessimistic as the ordinary minimax approach. (Wikipedia, Minimax Regret)

IMPLEMENTATION ISSUES

The project implementation is spread on two main levels -

- The JTetris module (written in Java), which contains the Tetris game itself, and the 5 heuristics mentioned above.
- The TetrisAnalyzer module (written in Python), which is used to do the local beam search, manage the result database, run the "championship" games, and other utilities.

JTETRIS

Our JTetris module is an improved, modified version of an OOP exercise given to Stanford students in 2001[2].
We made some major changes in order to make JTetris compatible with our needs.
All of the java source files were included in the original Stanford exercise except for GMABrain.java, which contains our implementation of the heuristics.

Some of these changes/additions are listed below:

- The whole weight system and concept is new.
- We added "simulation mode", in which a driver can automatically run the JTetris on given predefined games and weights.
- The full results can be written to the screen for a human user to read.
- Alternatively, a summarized result can be written to a file for convenient automatic use.
- GUI: We changed the whole color scheme, buttons, labels, scoring, etc.

The file of interest is GMABrain.java which we mentioned above. The rest of the files contain the infrastructure and driver of the game.

JTETRIS USAGE

- java JTetris Regular Mode.

- java JTetris w1 w2 w3 w4 w5
  Regular mode with the given weights for the AI player.
  Prints all game information to the standard output.

- java JTetris <games_file>
  Simulation mode with predefined heuristic weights.

[2] See reference regarding Stanford Tetris Project

- java JTetris <games_file> w1 w2 w3 w4 w5
  Simulation mode with w_i as heuristic weights.
  Prints all game information to the standard output.

- java JTetris <games_file> w1 w2 w3 w4 w5 <file_out>
  Simulation mode with w_i as heuristic weights.
  Prints to file_out only the average piece number of the games.

- java JTetris <games_file> w1 w2 w3 w4 w5 <file_out> noanim
  Same as the above, but when the 'noanim' string is added, no animation will be displayed. The game runs at the same speed.

The java source files reside in the "JTetris Source" folder.

TETRIS ANALYZER

This is a collection of tools written in the Python language.

The main components of the module are:

- TetrisAnalyzer.py — Runs a single thread of the local search agent.
- BestPlayersChampionship.py — Runs the 7 best agents in 15 predefined 12000 piece games.
- DBUtil.py — DB utility file.
- runBestDBPlayer.py — Runs the best agent extracted from the DB file.
- RandGen.py — Generates random games (i.e. sequences of number in range [0,6]) and prints them to the screen.

In this module, the file of interest is TetrisAnalyzer.py.
It selects a start vector (see CHOOSING AN INITIAL VECTOR), and goes through the "steps" described in WEIGHT ITERATION.

The collected data (weight list, game score) is pickled (Python's way of saving objects in files) into a DB file which is used to share information and avoid duplicate runs of the game.
Several TetrisAnalyzer.py processes can be executed concurrently - they all use the same DB file, thus implementing the data sharing of beam search. (See Figure 3)

We used RandGen.py to generate the predefined games files mentioned earlier.

The module runBestDBPlayer.py can be used to view the best player in the DB playing a new randomly generated game.

BestPlayersChampionship.py runs the CHAMPIONSHIP TOURNAMENT that will be elaborated later in this document.

The DB file (tetrisDB.dat) contains a hash table in which weight lists are keys, and game scores are values[3].
If no DB file is present, running TetrisAnalyzer.py will create a one.

The Python source files reside in the "TetrisAnalyzer" folder.

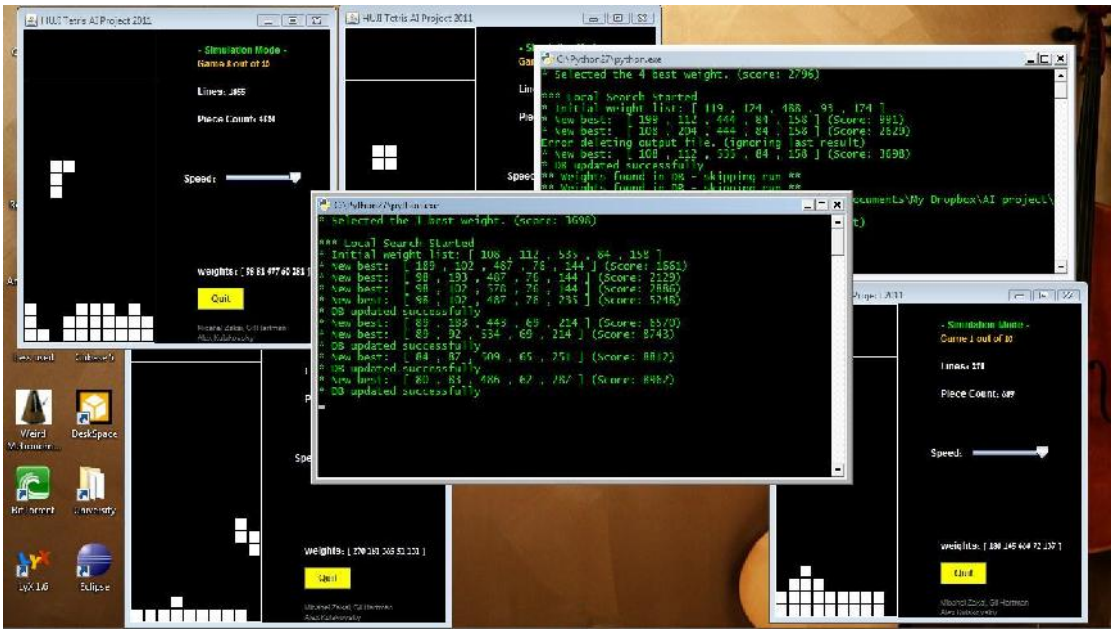


Figure 3: Screen shot of a home computer runnig 4 TetrisAnalyzer agents.
In the center agent, it is possible to observe that although the weights can only increase, because the vector is normalized after each change, some wieghts infact decrease while the others stay in place.

[3] In fact, the key is a string representation of the weight list. This was done because lists are not "hashable types" in Python.

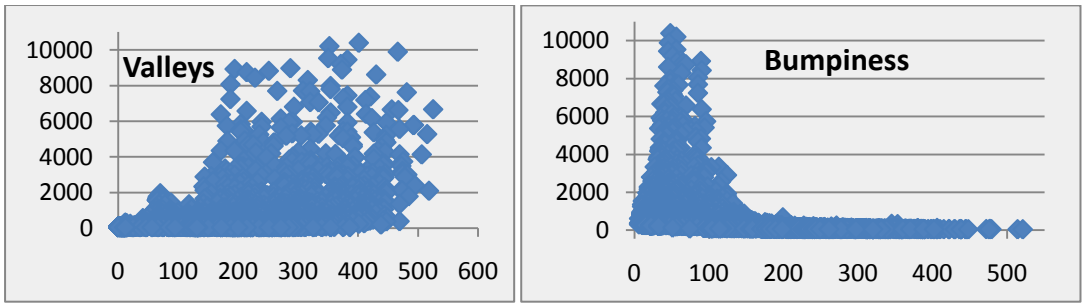
RESULTS

HILL CLIMBING ON TRAINING SET

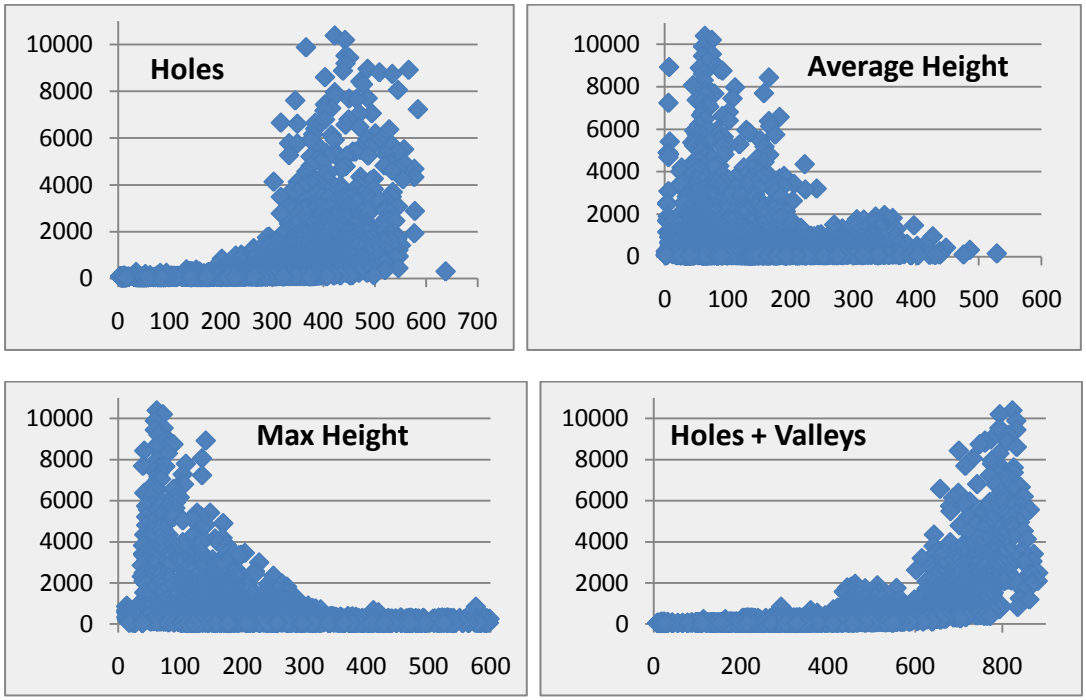
We ran a **series of parallel** local search agents in order to generate the database and perform the hill climbing. We ran **4-8 agents in parallel** at a time on a basic home computer. Each agent updated **50 values** on the database (**500 Tetris games**) and ran for 1-7 hours (depending on how well the agents performed). All together running time of the local searches was approximately **80 hours** (3 whole days).

All agents updated the same **database** and were run on the same 10 games, each containing 7,000 randomly generated pieces[4].

After running the agents, we now essentially have the values of part of a function with 5 variables (more precisely 4, since the sum of all 5 is always 1,000).
Trying to plot the graph of the function or part of it is a hard task. Nevertheless we attach **graphs of the contour** at each variable against the value of the function. A sixths graph of the sum of **holes and valleys** against the value of the function is attached.



[4] The games are attached in the file 10 games of 7000.txt with the following number-piece dictionary:
0 – I, 1 – L, 2 – J, 3 – S, 4 – Z, 5 – O, 6 - T



From the graphs it is possible to see that we did not explore during the local search all the parts and corners of the function in hand. At each weighted value of any variable where there are high values; there are also many low results. Although it is easy to see that some heuristics require larger weights in order to receive a peak in the explored graph, they have an optimal weight which should not be enlarged.

The **Holes + Valleys** graph is interesting since, when the sum of the two heuristics is between 882 and 782, the **average value** score in the part explored is **4,589**, and the minimum value is 720. That cannot be said about any range of values of a single variable.

The entire database collected and an enlarged version of the graphs can be viewed in the attach **sortedDatabase.xlsx** file.

CHAMPIONSHIP TOURNAMENT

After running the local search algorithm, we extracted the 7 highest scoring tuples. These tuples underwent a second round on a test set in order to determine which weight is best scoring on an average random Tetris game.

In this tournament, each agent played the same **15** randomly generated games of **12,000** each, while the maximum score per games was set to **20,000** pieces.

Out of the 7 agents, two agents (ranked 2nd and 4th in the original training set) varied from each other by only 2 points of weight. Agent 2's weights were **[72, 75, 442, 56, 352]**, while agent 4's weights were: **[73, 75, 443, 56, 350].** The weight order is: **Max Height, Average Height, Holes, Bumpiness, Valleys**. Not surprisingly they received close scores on some of the games, but varied on up to 12,000 pieces on other games. This supports the hypothesis that there is no magical maximum to our search space, the heuristic weight component heavily relies on the given game.

Among the 15 games played, at least one of the seven agents scored at least **5,000** pieces, an excellent score for any human player. In **7** out of the **15** games, there was an agent that scored over **18,000** pieces.

Checking the best average score in the championship games, revealed that the best playing agent was **Agent 2** scoring an average of **10,148** pieces per game (the agent's average in the training set was very close - **10,197**). However that same agent scored the minimal score amongst the participants on **3** of the games. That fact led us to explore other methods of ranking the best agent.

We ranked the agents in each game according to the number of pieces they survived, and treated each ranking as a voting ballot. We then used different voting methods in order to try and decide which of the agents won the championship. **Agent 2** was almost the **Condorcet winner** (lost only to agent 1) of the competition, but won **Dodgson's** methods and came in **tied** with agent 4 on the **plurality and Copeland** methods.

Agent 4 (very similar to agent 2 in weights) won in **instant runoff, Borda count**, and came in **tied in plurality and Copeland**.

The only reason not to choose agent 2 or 4, came from the **minimax regret** approach of decision theory. The approach advises you to minimize the worst case regret, or in other words chose the agent whose worst game is the best. Following this approach we should choose **agent 7 or 3** who **did not score under 1000 pieces** in any of the 15 games and were ranked 3rd and 4th in average pieces and in Borda count.

In conclusion, we chose Agent 2 as the winner of the championship over the test set. Agent 4 (a close relative) came close by a short margin. Agent 2's weights are: **[72, 75, 442, 56, 352]**.

All detailed scores, averages, rankings and pairwise matrix can be viewed in the attached **championshipScores.xls** file.

SUMMARY AND CONCLUSIONS

The project which started in reading dozens of learning articles and ended with local stochastic beam search and analyzing a 6 dimensional function (the five weights as variables and the piece average as function value) has been very enlightening and rewarding. Starting this project, **our optimistic guess** was to reach an agent which can survive **400 pieces** on average. After trying to explore the search space by ourselves, we hoped to reach 2,500 pieces a game. But the results of the project, an **agent that survives 10,000 pieces** on an average game, surprised us. The project showed us the power of AI heuristics and searching algorithms that we learned about in class the past semester.

The project can extended and probably improved by **adding more features** to the project: adding more heuristics to the search space, searching more time on the training set, enlarging the training and test set to more games, reducing the chance of random restart vectors over time, changing heuristic values throughout different situations of the Tetris game, looking a few step forwards while playing the Tetris game, and many more.

All of the heuristics we suggested were used in the best agents. Runnig a variation of the championship winner by removing some heuristic and maintaining the ratio between the other weights resulted surprisingly (or not) in a fairly poor playing agent[5].

Our best playing agent is the best playing agent **we have found on the training and test set**. That agent is not guaranteed to be the best playing agent on those sets. Furthermore, had we tested on a different training set, we would have probably reached a slightly different agent.

[5] The championship winner values are [72,75,442,56,352] and running the agent with the values [72,75,442,0,352] averaged around 400 pieces per game

BIBLIOGRAPHY

Carr, D. (2005). Adapting Reinforcement Learning to Tetris. Retrieved from http://research.ict.ru.ac.za/g02c0108/HnsThesis.pdf.

E. Demaine, S. H.-N. (2005). Tetris is Hard, Even to Approximate. MIT. Retrieved from MIT: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.85.3171&rep=rep1&type=pdf

Livnat, Y. B. (2000). Reinforcement Learning Playing Tetris. Retrieved from http://www.math.tau.ac.il/~mansour/rl-course/student_proj/livnat/tetris.html

Lorincz, I. S. (2006). Learning Tetris using the noisy cross-entropy method.

Motenko, Y. M. (2010). Effectively applying Reinforcement Learning to multi-state problems via the game of Tetris. HUJI .

Parlante, N. (2001). Stanford Tetris Project. Retrieved 3 2011, from Stanford CS Ed Library: http://cslibrary.stanford.edu/112/

Sarjant, S. J. (2008). SmartAgent | Creating Reinforcement Learning.

Wikipedia. (n.d.). Beam Search. Retrieved 3 2011, from Wkipedia: http://en.wikipedia.org/wiki/Beam_search

Wikipedia. (n.d.). Borda Count. Retrieved from Wikipedia: http://en.wikipedia.org/wiki/Borda_count

Wikipedia. (n.d.). Condorcet Winner. Retrieved 3 2011, from Wikipedia: http://en.wikipedia.org/wiki/Condorcet_winner

Wikipedia. (n.d.). Copeland's method. Retrieved 3 2011, from Wikipedia: http://en.wikipedia.org/wiki/Copeland%27s_method

Wikipedia. (n.d.). Hill climbing. Retrieved 3 2011, from Wikipedia: http://en.wikipedia.org/wiki/Hill_climbing

Wikipedia. (n.d.). Instant-Runoff Voting. Retrieved 3 2011, from Wikipedia: http://en.wikipedia.org/wiki/Instant-runoff_voting

Wikipedia. (n.d.). Local search. Retrieved 3 2011, from Wikipedia: http://en.wikipedia.org/wiki/Local_search_(optimization)

Wikipedia. (n.d.). Minimax Regret. Retrieved 3 2011, from Wikipedia: http://en.wikipedia.org/wiki/Regret_%28decision_theory%29

Wikipedia. (n.d.). Stochastic hill climbing. Retrieved 2011 3, from Wikipedia: http://en.wikipedia.org/wiki/Stochastic_hill_climbing