CS221 Final Project: Bananas for Bananagrams

Saahil Agrawal
Stanford University
saahil@stanford.edu

David Kwok
Stanford University
kwokd@stanford.edu

Abstract

Bananagrams is a game in which players compete to build a crossword grid using a random set of letters in the fastest time. In this project, we build and optimize an efficient search algorithm to build a complete grid of valid words from 21 random letter tiles in under 3 seconds, by leveraging a novel representation of the lexicon. We also model a Markov decision process and use reinforcement learning to teach the agent the optimal strategy for incorporating a new random letter into an already completed crossword grid, using a feature representation of the state space. When evaluated on test cases, the learned policy has a 11% higher completion rate than a baseline; we perform error analysis on these results and explore directions for further improvement.

1 Introduction

Bananagrams is a game introduced in 2006 in which players compete to build words on a grid using letter tiles. It is a variant of the game Scrabble, which has been widely used in studying Artificial Intelligence techniques, as it contains many properties of interest that align with the challenges that AI-algorithms are adept at tackling. Bananagrams maintains many of these properties of interest, but the variation on the rules introduces new challenges that have not yet been addressed in the past by Scrabble-playing algorithms.

The rules of Bananagrams can be summarized in just a few steps. First, each player receives 21 random letter tiles (from a bag of 144 tiles). When the game begins, the players attempt to form a crossword grid composed of intersecting words. Whenever any one player has no tiles left in his hand, he yells "Peel," and each player draws one new letter from the bag. Gameplay continues until there are fewer tiles left in the bag than the number of opponents. A player is free to rearrange his tiles freely throughout the game, and the game ends when any one player has used all the tiles in his hand and there are none left to be drawn[1]. For our project, we propose to solve two separate problems that are taken from the gameplay of Bananagrams:

1. Given a set of random tiles, find an efficient search algorithm to construct a crossword grid.
2. Given a completed crossword grid and a new random letter, find the best strategy to incorporate the new tile.

2 Literature Review

Bananagrams is a variant on the popular, well-studied game of Scrabble. Over the course of AI research, there have been many algorithms developed for playing this multiplayer, turn-based game. For

[1] In the official rules, players may also swap out a letter tile for three new ones from the bag, but we ignore this for the purposes of our implementation.

example, one early implementation of a Scrabble agent from Appel & Jacobson (1988) uses an efficient backtracking algorithm and a representation of the lexicon (dictionary of valid words) called a DAWG (Directed Acyclic Word Graph), a condensed representation of a trie whose edges are labelled with letters, and where each word corresponds to a path from the root. The backtracking algorithm takes an input letter from the grid and checks for all valid left parts of words and corresponding right parts of words, ultimately selecting the option with the highest score to play. This algorithm is able to generate a move (playing one word) in one or two seconds.

A more modern and the most well-known Scrabble AI is called MAVEN (Sheppard, 2002), which uses the same DAWG representation of the lexicon. MAVEN uses a series of heuristics to tradeoff the agent's current score against future score (by trying to prevent the opponent from being able to play high-scoring words on the board). In other words, MAVEN has been optimized for the specific rules and point assignments unique to Scrabble and is less generalizable to other AI tasks; for our project, we choose to limit the scope to focus only on the crossword generation task of Scrabble, so we found Bananagrams to be a useful variant to work with.

3 Approach

Unlike Scrabble, Bananagrams has no point values assigned to different letters or grid positions, and the goal of the game is simply to complete the crossword in the fastest time possible. Hence, the metrics for evaluating our agent will be 1) whether it creates a complete crossword without any unused tiles (completion rate), and 2) the speed of completion (completion time). A baseline we will use for comparison is human-level performance; in our tests and observations, a human player is able to construct a complete crossword in an average of 3 to 4 minutes; a human player can also append a letter to an existing crossword in about 5 to 10 seconds (both of these estimates assume that the player is actually able to find a complete solution; in many cases, the player is left with several tiles he is unable to play on the board). We expect our algorithms to relatively easily beat both of these baselines, as the agent has the advantage of having access to the entire dictionary of words.

As an oracle for both of our search tasks, we design a simple algorithm that pulls a random word from the dictionary and plays it on the board horizontally. Next, the algorithm pulls another random word from the dictionary with the same first letter as the word on the board and plays this word vertically. Then, the algorithm continues to use the other letters already played on the board as the first letter of the next word to be played, until the total number of letters on the board exceeds 21 (the size of the initial hand in Bananagrams). This algorithm is able to form a complete crossword in under one second and always finds a complete solution. Though this algorithm is obviously unrealistic for gameplay, it sets up the necessary infrastructure to determine how words can be played on the board.

The main inputs to the algorithms we design are the complete set of 144 letter tiles standard to Bananagrams (from which letters will randomly be drawn), as well as a complete dictionary of valid words[2]. In order to optimize our search algorithms, we introduce an anagram map that increases the speed with which we are able to identify whether a string of letters is a valid word in the dictionary. In particular, for each word in our official dictionary, we sort the letters of the word alphabetically and add them as elements to a set. While our official dictionary has 276,643 words, the set of anagrams has only 244,991 elements. While this may not seem like a huge difference, because each element of the anagram map is sorted alphabetically, determining whether a set of letters can form a word in the dictionary is much simpler, as we only need to check for the presence of letters, rather than for each possible permutation of letters. This is a novel representation of a lexicon that makes the ordering of letters irrelevant to checking for a valid word, and we compare the performance of this representation against the more well-studied DAWG representation (which does require maintenance of letter order) used in prior Scrabble algorithms.

[2]We use the official Scrabble dictionary in our implementation.

4 Task I: Construct Crossword Grid

Both of the search algorithms specified below proceed thusly: first, determine the first word to place on the grid using some heuristic (which varies by algorithm), in combination with the anagram map to determine valid words. Remove each of the letters used from the hand. Second, given the word on the board, use each of the letters of the inscribed word as a seed and find the second word to place on the board using some heuristic. The purpose of inputting a seed (from the second step onwards) is that each of the candidate words considered by the algorithm must contain this seed in order to be a valid candidate, so that in the end, a crossword-like grid of words can be formed. In addition, we enforce the constraint that new letters cannot be placed adjacent to any existing letters (except, of course, for the seed), so that no overlapping, invalid words will be present. After the second word has been placed on the board, repeat the same procedure (as for the second word) until there are no more tiles present in the hand, or until no further valid words can be formed.

4.1 Algorithm I: Breadth-First Search on Letters-in-Hand

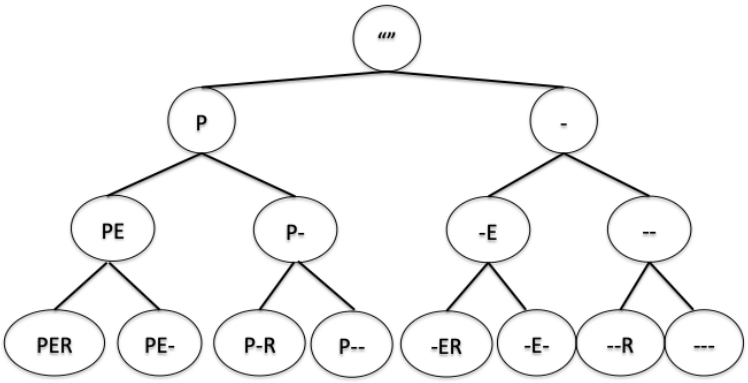

Figure 1: Algorithm I Search Tree

Our initial approach for creating a crossword grid given a random set of tiles is to search across the possible letter combinations in the hand that can be made into words (adapted from Piech, 2016). For this approach, we use a breadth-first search algorithm by first fixing the order of letters in the hand and then adding (or excluding) letters in sequence until valid words are found. One key feature of this algorithm is that we use a heuristic to score each of the words based on its component letters, where letters that are less frequent in the complete pool (and are therefore more difficult to play) are assigned a higher number of points. This heuristic is reasonable because starting the grid with letters that are more difficult to play allows for the more flexible use of letters that are easier to play later on. The algorithm is implemented as follows:

1. Select a seed. For the first word, the seed is an empty string, and for subsequent words, the seed is a letter tile that has already been played on the grid.
2. Find the score of the current letter combination, using a heuristic that assigns each letter a point value inversely correlated with its frequency in the pool of possible letters. Each word is scored as the sum of the points of its component letters.
3. If the candidate letter combination is in the set of anagrams, save this word if it has the highest score of any words seen thus far. Otherwise, continue to the next step.
4. After every node, explore two children states: a) adding the next letter in the hand to the existing letter, and b) omitting the next letter (represented by a dash -). Thus after every iteration, two child nodes are added to the queue (see figure 1).
5. Repeat steps 2 to 4 for 50,000 iterations, and then return the best word found. Play this word on the grid.

6. From the existing words that have been played, find the letters that can be used as seeds for the next word, based on the spacing around them, and find the next-best word using steps 1 to 5 for all the spots, and select the spot with the best score.
7. Repeat until either the agent is left with no tiles, or it is not possible to use any of the remaining tiles to create a word.

4.1.1 Results / Error Analysis

Empirically, while this algorithm almost always finds a complete solution, its performance is suboptimal in completion time. In a sample of 20 runs, the algorithm found a complete solution 80% of the time in an average of 400 seconds (6.7 minutes). As shown in figure 1, the algorithm is inefficient as it explores equivalent states multiple times (e.g., the state -E is equivalent to state -E-). See Appendix A for examples of input-output pairs generated by this algorithm; as expected, this algorithm tends to first generate words that use less-common letters first and then appends easier-to-generate words subsequently.

4.2 Algorithm II: Breadth-First Search on Dictionary

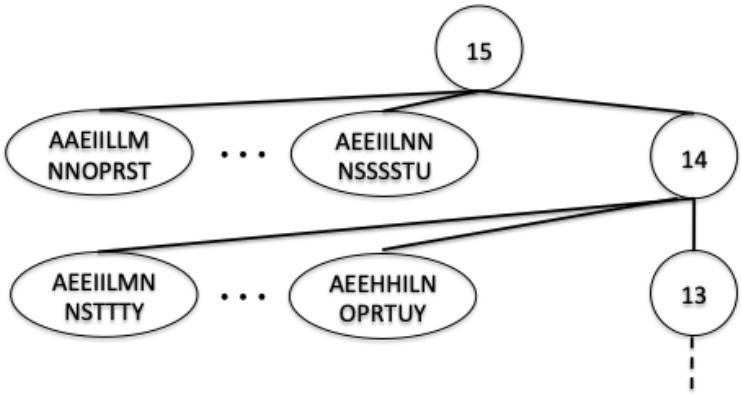

Figure 2: Algorithm II Search Tree

A second algorithm we introduce to solve the Bananagrams search problem takes greater advantage of the fact that the agent has access to the full dictionary of words. Hence, instead of creating a search tree based on the tiles in hand (as we did in Algorithm I), we create a search tree based on the elements of the anagram map created from the official dictionary. In particular, we define the root node of the search tree as the number of letters n that are currently in the agent's hand. Each of the children of the root node corresponds to an anagram combination from the dictionary with n number of letters. The last child of the root node is the quantity *(n-1)*, and its children are defined the same way recursively. The breadth-first search algorithm proceeds thusly:

1. Begin at the root node. For each child node, check to see if the combination of letters is either a complete match or a subset of the letters currently in hand. If so, return arbitrarily one of the words associated with the anagram, place this word on the grid, and continue to step 3. If not, continue to step 2.
2. If all the children of the root node have been searched without a valid word being found, continue to the node *(n-1)* and continue searching down the tree recursively until the first subset or complete match has been found.
3. When a match is returned, remove the letters of the match from the hand and place the word corresponding to the anagram on the board. Then, using one of the letters on the board as the seed, rerun the algorithm starting at node *(n'+1)*, where n' is the remaining number of tiles in the hand, introducing the additional constraint that a valid match must contain the letter specified

as the seed. If no matches are found with the current seed, rerun this step with the next possible seed.

This algorithm implicitly uses the heuristic of repeatedly finding the longest word that can be made from the tiles in the hand, anchoring on the seed letters that have already been placed on the board. This is a reasonable heuristic to use, as placing longer words earlier in the game (regardless of letter composition) subsequently allows for more combinations of words that can be added to the grid, as there are more letters available to be used as seeds.

4.2.1 Results / Error Analysis

Empirically, the performance of this algorithm far exceeds the performance of Algorithm I. Simulating 100 times shows that the average run time for the algorithm is 2.7 seconds, and it finds a complete solution in 79% of cases. This runtime obviously beats any known human performance, taking advantage of the fact that the agent has access to the entire dictionary of valid words. See Appendix A for example input-output pairs from this algorithm. As expected, this algorithm generates very long words first and then appends a series of short words in subsequent runs. For the last several words generated, the algorithm very often selects obscure two- or three- letter words from the dictionary.

Though Algorithm II runs extremely quickly in comparison to Algorithm I for a typical 21-tile hand, we also explored how the performance of each algorithm varied with the number of random tiles initially drawn. Below in Table 1, we tabulate the relationship between completion time/rate across different numbers of starting letters (10, 15, and 21). Interestingly, Algorithm I is able to find a solution even more quickly than Algorithm II when there are only 10 starting tiles (average of 0.2 seconds). However, with more tiles, Algorithm I's running time explodes to an average of 4.4 seconds with 15 tiles, and a whopping 400 seconds (6.7 minutes) with 21 tiles, since the number of states explored by Algorithm I scales exponentially with the number of tiles. Algorithm II, however, has a much more consistent completion time at around 2.6 seconds regardless of the number of tiles, as it simply repeats the same O(1) algorithm (search on the static dictionary of words) multiple times until the tiles are depleted. On average, Algorithm II's completion rate (about 80%) is only slightly below the completion rate of Algorithm I (about 85%).

Table 1: Results Across Different Numbers of Starting Tiles

	Algorithm I		Algorithm II	
Number of Letters	Time (sec)	Completion Rate	Time (sec)	Completion Rate
10	0.2	88%	2.6	80%
15	4.4	85%	2.6	85%
21	399.5	80%	2.7	79%

5 Task II: Adding New Letter to Grid

Having developed an efficient algorithm to solve the search problem of constructing a crossword grid from a set of random tiles, we next address the second task of Bananagrams. Namely, given a completed crossword grid and a random new letter ("Peel!"), we want to find the best strategy that incorporates the new letter into the crossword grid. In particular, we model this problem as a Markov decision process (MDP), in which the agent can take two possible actions: 1) search the existing crossword for a space to play the new letter, or 2) break down the crossword into its component letters and reconstruct the crossword from scratch with the addition of the new letter, using Algorithm II from Task I.

5.1 MDP Definition

We define the state of the MDP as the current crossword grid and the new letter tile in the agent's hand. The end state is always reached after one step. Taking either action as defined above deterministically

yields some probability of successfully completing the grid (which occurs if there are no tiles in the agent's hand in the end state), and some reward associated with each action and end state. Intuitively, the action of appending the letter to the existing crossword will take less computational time but is also more likely to fail, whereas the action of reconstructing the crossword from scratch will take more computational time but is more likely to complete the grid successfully. To capture both effects, we define the reward of completing the crossword as (100 minus the number of seconds elapsed in computation time) and the reward of not completing the crossword as 0.

$$s_{start} = (< \text{Current crossword} >, < \text{Tiles in hand} >, 0) \tag{1}$$

$$Actions(s) = \begin{cases} Append\ tiles\ to\ existing\ crossword \\ Reconstruct\ crossword\ with\ new\ tiles \end{cases} \tag{2}$$

$$Successor(s, a) = (< \text{Updated crossword} >, < \text{Updated tiles in hand } >, 1) \tag{3}$$

$$Transitions(s, a, s') = \begin{cases} p, & \text{if s'[1]} = \emptyset \\ 1 - p, & \text{otherwise} \end{cases} \tag{4}$$

$$Reward(s, a, s') = \begin{cases} 100 - (\text{time taken in seconds}), & \text{if s'[1]} = \emptyset \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

$$IsEnd(s) = s[2] = 1 \tag{6}$$

5.1.1 Reinforcement Learning

The Markov decision process defined above has unknown transitions and rewards for state-action pairs. Therefore, we use reinforcement learning in order to determine the optimal policy for a given state. One additional challenge is that the state space in the MDP is enormous and would be impossible to explore comprehensively; hence, we test several feature representations of the state space ϕ, and define weights w associated with each feature.

The optimum reward is therefore defined as:

$$\hat{Q}_{opt} = w * \phi(s, a) \tag{7}$$

The update of the weight vector at each step is given by:

$$w \leftarrow w - \eta * [w * \phi(s, a) - r]\phi(s, a) \tag{8}$$

5.1.2 Results / Error Analysis

We hand-crafted and tested a number of different feature representations of the state by applying the training algorithm described above on 1,000 iterations of crossword and new tile inputs. See Appendix B for a tabulation of resulting weights for select key features. One observation from the weights, for example, is that it is more optimal for the agent to append the new letter when there are a larger number of letters in the longest word on the board. This finding makes intuitive sense, since the presence of longer words on the input crossword allows for more possible spaces where the new letter can be played to form a valid word. From the weights, we also observed that both actions struggled to find a solution when there were letters like **Q** in the set of new tiles. Both reconstructing and appending often failed to

incorporate these new letters, which is not surprising, as certain letters can only form valid words when used in combination with other letters (e.g., **Q** must be paired with an existing **U**).

In order to test the actual performance of the learned weights, we compared the realized rewards on 100 new test cases not used during training under two cases: 1) a baseline of selecting either the "append" or "reconstruct" action with a 50% chance, and 2) selecting the action according to the optimal policy given the state (the weights that yield the higher Q-value). In the baseline, the agent completed the crossword **73%** of the time, compared to **84%** when using the learned weights. These results indicate that our reinforcement learning setup makes significant progress in understanding the dynamics of the problem; in our final section, we discuss the limitations of our approach and the next steps we plan to pursue.

6 Discussion / Future Work

Our goal for this project was to explore novel algorithms and strategies a computer agent could use for playing Bananagrams. We accomplished this by splitting the game up into two single-agent tasks that we optimized independently.

First, we developed an efficient search algorithm that could reliably construct a crossword grid given a random set of 21 letter tiles in an average of less than 3 seconds, with a completion rate of over 75%. This novel algorithm was a particular success of our project and can easily defeat any human player. However, this algorithm has only a moderately high completion rate that could potentially be improved; in particular, the algorithm currently takes a greedy approach in forming subsequent words (i.e., words are played on the board in succession without directly accounting for what future words can be formed by adding to the grid). One possibility for future exploration would be to model this problem also as a Markov decision process, where we train weights to select a more optimal word to play, with the objective of depleting all letter tiles in hand. Though this modified approach would undoubtedly raise the computational time, we could potentially improve the completion rate significantly.

In our second task, we defined an MDP to find an optimal policy for incorporating a new letter into an existing crossword grid given two possible actions: appending the new tile directly to an existing letter, or reconstructing the grid from scratch with the new letter in hand. We tested and trained weights using a number of hand-generated features to represent the states, and we confirmed that the resulting weights made intuitive sense. Finally, we tested the learned weights against a baseline (random selection of actions) and found that they yielded significantly better performance. For future work, we identified two ways in which we could potentially improve the agent's performance even more. First, given the relatively large state space in our MDP, we may be able to train the weights for more iterations, in order to make sure that they see sufficiently many different states in order to differentiate between the rewards realized by the two actions. Second, as a starting point, we hand-crafted a number of feature representations of the state space, but did not have the time to more exhaustively test all our hypotheses or non-linear combinations of the features. Given the complexity of this problem, there are many more hypotheses that would be worthwhile to test. As next steps, we could try testing more of our hypotheses on relevant features, or use deep learning to generate more complex feature representations of the state space. One specific idea for developing a more comprehensive representation of the state space is to represent the input crossword as a 2D image and develop feature vector representations of the board, similar to Glove embedding in NLP. This richer feature representation can then be used for Q-learning in the MDP defined.

Appendix A

Example of Algorithm Performance in Task I
The randomly drawn tiles are:
'N', 'E', 'D', 'L', 'E', 'V', 'I', 'W', 'T', 'O', 'T', 'A', 'D', 'T', 'I', 'E', 'K', 'J', 'T', 'Y', 'G'

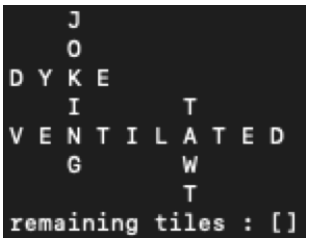

Figure 3: Algorithm I Output

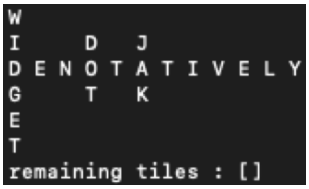

Figure 4: Algorithm II Output

In the above example, Algorithm I is able to finish the crossword grid in about 68 seconds. Algorithm II is able to finish the task in about 2.1 seconds.

Example of Algorithm Performance in Task I
The randomly drawn tiles are:
'T', 'R', 'S', 'K', 'I', 'T', 'R', 'I', 'G', 'N', 'G', 'O', 'U', 'L', 'A', 'I', 'O', 'M', 'R', 'E', 'A'

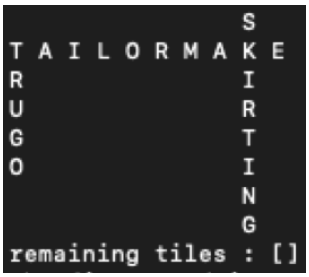

Figure 5: Algorithm I Output

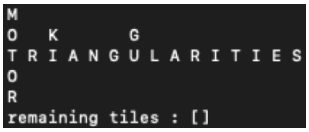

Figure 6: Algorithm II Output

In the above example, Algorithm I completes the grid in 64 seconds, while Algorithm II completes the task in 0.5 seconds.

Appendix B

Below is a table of the learned weights of some key features after training the Markov decision process on 1,000 iterations.

Feature	Append	Reconstruct
Length of longest word on board	69.5	57.3
No. of vowels on board	52.4	72.5
Tile V in hand	-52.0	0.2
Tile Q in hand	-4.7	-2.0

Table 2: Weights of select key features

The absolute weights have little meaning, but have to be analyzed in comparison with weights corresponding either to the other action or to other features.

Bibliography

1. Abraham, P.J. (2017). "A Scrabble Artificial Intelligence Game." Master's Projects. 576. https://scholarworks.sjsu.edu/cgi/viewcontent.cgi?article=1574&context=etd_projects
2. Appel, A.W. and Jacobson, G.J. (1988). "The Worlds Fastest Scrabble Program." Programming Techniques and Data Structures, 31, 572-585. Retrieved from https://www.cs.cmu.edu/afs/cs/academic/class/15451-s06/www/lectures/scrabble.pdf
3. Piech, C. (2016). Anagrams and Bananagrams [Web log post]. Retrieved Oct 22, 2018, from https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1172/handouts/bananagrams.html
4. Sheppard, B. (2002). Towards Perfect Play of Scrabble. Retrieved from https://cris.maastrichtuniversity.nl/portal/files/1504714/guid-b2c8ed9d-4f1f-4eb2-bf0f-7ad95196f373-ASSET1.0
5. Young, S.H. (2013). WordSmith - Building a Program that Plays Scrabble. Retrieved from https://www.scotthyoung.com/blog/2013/02/21/wordsmith/