Ashley Cooray (anc73), Pooja Kanumalla (pk457), Sohni Uthra (su58)

Final Report

Introduction

Creating crossword puzzles requires selecting words and ordering them given constraints. These constraints include a requirement that letters overlap in various words, that no words are generated through overlapping words incorrectly, and many more. In artificial intelligence, the Constraint Satisfaction Problem (CSP) is defined as finding and assigning values or states to variables (in this case, letters and words) given some constraints or limitations over those variables (Mittal, Falkenhainer). Because of the several constraints involved in creating a crossword, we can define this problem as a non-binary (constraints handle more than 2 variables) Constraint Satisfaction Problem (Bacchus). Through this project, we sought out various constraint optimization algorithms for generating crosswords, scoring the resultant crossword puzzles, and selecting the puzzle with the highest score to display to the user. We primarily focused our attention on creating different methods of generating crossword puzzles such that we can maximize the 'score' of the crossword.

We planned to create a program that generates automatic crossword puzzles from an inputted size. Given multiple online dictionaries, we found definitions of a plethora of words. We took one such selection of words and their associated definitions from the dictionary to form words and clues in the crossword puzzle. Once we had a list of words and associated clues, arranged in a JSON file, we were able to find words that have letters in common to use for our crossword, and narrow down our word selection to these words.

A naive approach to crossword generation as a CSP problem would be to check each possible word for each possible position and check if that word satisfies all constraints; however this is severely inefficient and will result in poor performance (Pearson, Jeavons). As a result, we instead developed five different algorithms to generate a crossword puzzle, and instead of checking every word, we ranked words so we only had to check the top ranked words (which results in less iterations). Each of our five algorithms proceeds fairly similarly, however each selects the starting word (and the rest of the words in some of the algorithms) differently. We run all five algorithms once a user inputs a desired size, and score each of the generated crossswords to determine the "best" crossword. This chosen crossword is then what is then displayed to the user to fill in.

Lastly, we created a GUI for the user to input the size of the crossword they want and then input answers for the clues presented for the displayed crossword. The user can ask to check the crossword where the words inputted fill the grid of the crossword and incorrect tiles have a red border. Once the user

successfully completes the crossword, an end screen is displayed with the amount of time it took for the user to complete the puzzle.

Prior Work

As we worked to build our solution, it was interesting to learn about the different approaches taken by others. There have been multiple constraint satisfaction approaches to creating an AI crossword generator. One example is a research project conducted via a partnership in 2020 with Swarthmore College, Otis Peterson, and Michael Wehar. This team used different heuristic algorithms and data structures to achieve the same goal as ours: choosing an optimal and maximal set of words to construct a solvable crossword puzzle. Specifically, they approach the problem by storing a list of possible words for each space, inserting a word into the space with the smallest list, and updating all lists. If any space has an empty list, their algorithm removes the most recently added word from the grid and list, and reverts all lists that were changed because of that most recent placement. This process is repeated until the crossword is filled with valid words. The team found that this approach was able to search through potential words quickly, which improved overall performance (Wehar).

Another approach to the problem was developed by a team from the University of Siena (Italy) in 2011. Their algorithm leverages website crawling to input a list of web pages. Then, NLP techniques are used to extract the words and definitions from the HTML pages. Their constraint satisfaction approach to actually constructing the puzzle involves defining the skeleton of the crossword and a schema, a skeleton partially filled with words. Words can be added to the schema if it starts and ends at a black cell and does not alter a previously inserted word. The team also employs a priority queue to store a set of possible schemas (Rigutini et al).

Finally, we investigated research from the University of Guelph in 1976. There was a very interesting difference between this approach and ours and others that we researched; this team decided to use a letter-by-letter method to form words. More specifically, letters are first placed on the board, and the words formed are cross-checked with a dictionary to ensure validity of the local words. The goal behind this strategy is that if the puzzle is valid locally, the entire puzzle will eventually be valid as well. To determine which letters to place in which cell, the researchers created a tasking algorithm, which assigns a “weight” to each cell and maintains a precedence stack that pops spaces with the highest weight first. The weight of a cell is equal to its degree of freedom plus the number of directly connected cells. To select a letter for the cell, the algorithm maintains a vector for each cell. The vector contains occurrence ratios for each of the 26 letters (each occurrence ratio is determined by taking the minimum occurrence ratio from a group of different occurrence ratios about each letter), and the letter with the highest ratio is chosen because it will advance the puzzle to a valid solution faster (J.Mazlack).

Methods

Data Importing, Cleaning & Initial Setup

1. We found a CSV online (Clues) containing words and clues from various esteemed crossword puzzle websites, such as *The Daily Telegraph, The New York Times, The Hindu, The Guardian,* and more. However, any dataset with 'clue' and 'answer' columns can be inputted.
2. We import the data via Pandas and clean it by dropping all null values using Numpy.dropna().
3. We create a dictionary mapping the resulting cleaned words to their respective clues.
4. We drop any words (and their respective clues) that are longer than the user-inputted size of the crossword itself, as these words would not fit in our puzzle. This is done in the function *clean_words*. We also removed any words containing less than 3 letters, as we did not want to clutter our puzzle with smaller words, especially since we noticed our dataset contained a lot of single letters as "words." The clues as given by the dataset (found online) contain numbers in parentheses at the end, which we remove. We remove any spaces in "words," as some words in the puzzle were actually multiple words but we do not want spaces in our puzzle; also, if a user types a space in their answer, that space is removed so it matches the new word without spaces in the dictionary. We had also seen several other crossword puzzles include two word "words" in the crossword puzzle without any separation. This was done in puzzles such as the *New York Times* crossword puzzle. Lastly, we create a dictionary to keep track of the counts of letters among all words corresponding to each letter (this is used later in one of our word ranking functions detailed below).
5. We create a 2D array for our grid and initialize it with a blank space at every position. We make our grid a square with edges of length matching the user-inputted size.

Ranking Words

All five algorithms we came up with for generating our crossword puzzle involved a ranking function to determine how to place words on the board. We created five ranking functions (detailed below) to return a ranked list of words that was used in our crossword generation algorithms.

Ranking Function 1: Rank by Number of Intersections

First, we made *ranked_by_num_intersections* that takes in a list of words in the puzzle to rank against as well as the entire dictionary of words and clues to rank. It ranks words in the dictionary based on how many intersections they currently have with all of the words that are already in the puzzle. This, however,

means that the letters at the intersection in the grid are being highly valued as the letter would be in both the words which intersect.

Ranking Function 2: Rank by Common Letters

In order to prevent the overvaluation of letters, we created *ranked_by_common_letters* which takes in the grid at the time and the entire dictionary of words and clues to rank. Instead of ranking the words by having letters in common with the words on the board it ranks the words by having letters in common with the letters on the board. This now means that letters at an intersection were being valued equally to all other letters on the board.

Ranking Function 3: Rank Without Intersections

Next, we created another ranking function that devalues letters at intersections. When a letter is being used by two words, i.e. it is an intersection point, no other words can use that letter (limitation of two dimensions). This means that ranking a word by the number of intersections with *any* letter on the board once again overvalues the letters at the intersection point. Since we sought to rank the words based on how they could fit on the board, using letters which would inhibit rather than aid placement does not help. For these reasons, we created *ranked_without_intersections,* which only ranks words in the dictionary which have letters in common with the letters on the board that are not *yet* an intersection point. This was done through use of the *is_intersection* function, which leverages boolean logic to determine if a letter is an intersection point.

Ranking Function 4: Rank Without Intersections and Unique Letters

Building upon ranking function 3, we noticed that a greater subset of words would be able to be placed on the board if we introduce more letters (that have not already been used) onto the board with each additional iteration. So, we created *ranked_without_intersections_and_unique_letters*, which ranks words in the dictionary by the number of common letters they have with the letters on the board that are not yet an intersection point first, and by the number of unique letters they introduce to the puzzle. Since this allows additional variation in the letters placed on the board, it would allow additional words to be placed.

Ranking Function 5: Rank by Letter Score

Finally, we considered a letter scoring metric which is independent of the words on the board. This function takes in the clues dictionary as well as the ranked letters based off of their frequency among all word clues. We chose to assign each letter a score based on their frequency in the dataset. We decided each letter would receive a score of 6, 5, 4, 3, 2 or 1. Letters that receive 6 points are the most frequently

found in the dataset, letters that receive 5 points are the next most frequently found, etc. Once the letter scores are determined, we rank all words in the dictionary using their letter scores. For example, say we have the word “AND.” Let us say “A” has a score of 6, “N” has a score of 6, and “D” has a score of 4. The total score for “AND” is therefore: 6 + 6 + 4 = 16. We input “AND” into our ranked words dictionary with its calculated score of 16. We then return the sorted list of words based on their respective calculated scores.

Whitespace

We define whitespace as spaces that are blank (do not contain a letter) on the grid. As we add words to the grid, we decrease whitespace by the number of new letters added to the board. Additionally, we made the distinction in our puzzle that words cannot be directly next to each other. This was done to satisfy the constraint that we cannot generate words on the crossword as a consequence of placing a word on the board. For example, if two words are placed vertically next to each other on the board then for the length of the shorter word, we would have generated two letter words which may or may not be real words. Therefore, when we add a word to the board, we mark spaces directly left/right of vertical words and above/below horizontal words with a ‘-’ character to prevent other words from being placed in those spaces. We make a distinction that words can be placed in these positions if they are in the other direction (ie. a vertical word can have a letter placed above a horizontal word) in order to allow intersections. We count these marked spaces as used whitespace and remove this from our whitespace count. This whitespace count is independent of the whitespace scoring algorithm which will be discussed later in the paper. Once every space on the grid either has a letter or a mark that a letter cannot be placed there, we can no longer place words on the grid. This whitespace counter is used as a loop condition to indicate to the algorithm that no additional words can be placed.

Placement Algorithms:

Algorithm 1: Highest Ranked Word First

With this algorithm, we place words on the grid simply based on rank. We begin with the highest ranked word, and continue on to the next highest ranked word and so on until there is no more room, or whitespace, as defined above, on our grid. We do this with *generate_puzzle_highest_ranked_first*, which takes in the user-inputted size, the grid, our words to clues dictionary, the ranking function to use, and our ranked letters dictionary (this will be used if the ranking function is *Rank by Letter Score*).

1. We first rank all of the words in the dictionary using the inputting ranking function. If the ranking function is *Rank by Letter Score*, we pass in the dictionary mapping words to clues as well as the

ranked letters dictionary into *rank_by_letter_score*. Else, we pass the current grid and the dictionary mapping words to clues into the chosen ranking function. The first word is ranked by determining which word in the dictionary has the most intersections with all words in the dictionary.

2. We sort our ranked words in descending order, based on the ranking function given.
3. We then place our first word by calling *place_first_word*. It places our first word horizontally in the middle of the grid. After placing the word, we initialize whitespace to account for this newly placed word as well as spaces above and below it.
4. We create an array to hold the letters positions that have been placed on the board and a dictionary to map these words to their starting x and y positions as well as a boolean indicating whether the word is horizontal (True) or vertical (False).
5. While whitespace (positions that have not been filled by a letter or a marker preventing a word from being placed there) exists on the grid:
    a. We re-rank (via the inputted ranking function) and sort the words by the letters in words currently on the grid after updating the current words in the puzzle.
    b. If re-ranking the words returns no words (no more words intersect with the current words in the puzzle), we break out of the loop and our algorithm is done.
    c. Next, we loop through the ranked words to determine which words a given word intersects with on the board by calling our helper function *contains_intersection*. We loop until we reach the end of ranked words or no word is found.
    d. We then find the ideal placement of the word to be placed on the grid by passing the grid, the word to be placed, the words on the board that intersect with the word to be placed (found from c), and the dictionary of words and their positions on the board into another helper function *find_placement*. In this function, we loop through the words on the board that the word to be placed contains intersections with, find the letters at which they intersect, determine if that letter is a valid intersection, and find the placement vector once a valid interaction is found.
    e. We loop through all of the possible intersections between the word to be placed on the board and the words on the grid. This is found by identifying all common letters between the words on the board and the word to be placed. Next, we check to make sure a word could be placed at one of the intersections on the board: we call *is_valid_intersection* with the direction (horizontal/vertical) of the word on the board we want to intersect with, its x and y positions, the index of the intersection of the word on the board, the index of the intersection of the word we are attempting to place, the grid, and the word to be

placed. Our function *is_valid_intersection* uses another helper function *check_spacing* to check spacing around where the word is to be placed on the grid to make sure it is either empty or is the current word on the board we are trying to intersect with. Once we determine with *check_spacing* that spacing above/below for horizontal words and spacing left/right for vertical words exists and is free, we return True from *is_valid_intersection.*

f. Back in *find_placement*, now that we've determined the intersection is valid and the word can be placed there, we call *determine_position* to calculate the x and y positions of where to place the word. This function returns the x position, y position, and horizontal/vertical alignment via a boolean (True for horizontal and False for vertical) indicating where and how the word to be placed should be placed on the board. This result is then returned by *find_placement.*

g. Once a placement that we found to be valid by checking spacing and area is found, we call *check_layering* which prevents words from being layered on top of each other. One such example is the case of AND and BANDAID. If the word AND exists in the crossword and we seek to place the word BANDAID on the grid directly overlapping with the word AND, we technically would not violate any spacing problems; however, we want to prevent encapsulation or layering of words inside each other. If the word to be placed on the board is determined to be horizontal, *check_layering* will loop through all words currently on the board; if any of these words are also horizontal, it will start by checking if it has the same starting x position. From this point, there can be two possible cases. The first case is if the horizontal word already on the board has a smaller x starting position than the horizontal word to be placed (the word already on the board is above the word to be placed). We set a variable called start to be the ending x position of the word on the board and a variable called an end to be the starting position of the word to be placed. This gives us the space between the two words. In the other case, the horizontal word already on the board has a larger starting x position than the horizontal word to be placed (the word already on the board is below the word to be placed). The start variable will then be set to the ending x position of the word to be placed, while the end variable will be set to the starting position of the word already on the board. In either case, if the start variable is greater than or equal to the end variable plus 1, we know that there is overlap and we cannot place that word on the board and we return False. Similar logic was then implemented for words to be placed that are vertical. If True is returned from this function, we can then return a final valid placement from *find_placement*. If False is

returned, our algorithm will iterate at most 5000 times to find another word to place in the puzzle. If no word is found, we will break and our crossword is complete.

h. If we find a valid placement, we decrease whitespace by calling *determine_whitespace_to_remove* which decreases whitespace by the length of the word (minus used intersections found from *determine_num_used_intersections* in order to avoid double counting intersections as used spaces) as well as decreasing whitespace by spaces around the word (as detailed above) based on the horizontal/vertical alignment of the word. Markers are used to indicate whitespace next to words in order to prevent double counting of this space as well. Since we want to prevent words from being placed next to each other, the space around the word is also removed from whitespace as explained unless it had already been decremented (as indicated by the marker).

i. We are now ready to place our word on the board. We do this by calling *place_on_board* on the grid, the word to be placed, the placement we found, and the dictionary of words on the board with their positions. This function adds the word to be placed and its placement to the dictionary of words on the board with their positions. It then adds the word in its placement (x position, y position, and alignment) to the grid.

6. Once we have filled all of the whitespace or hit a maximum number of iterations, then we return the resulting grid and positioned words. We have successfully created a crossword of the given size.

Algorithm 2: Highest Ranked Longest Word First

With this algorithm, we make a slight distinction in which words are placed on the board: instead of choosing the word with the most number of intersections, we choose the highest ranked word (the one with the most number of intersections with other words) of maximum length. The maximum length is the longest word in the list of words. Essentially, the way we choose words to place on the board slightly differs from *Algorithm 1: Highest Ranked Word* to *Algorithm 2: Highest Ranked Longest Word First* by first requiring that the word to be placed is the longest word in the dictionary. We do this with *generate_puzzle_highest_ranked_longest_first,* which takes in the user-inputted size, the grid, our words to clues dictionary, the ranking function to use, and our ranked letters dictionary (this will be used if the ranking function is *Rank by Letter Score*). The steps are as follows:

1. We first rank all of the words in the dictionary using the inputting ranking function. If the ranking function is *Rank by Letter Score*, we pass in the dictionary mapping words to clues as well as the ranked letters dictionary into *rank_by_letter_score*. Else, we pass the current grid and the dictionary mapping words to clues into the chosen ranking function.

2. We sort our ranked words in order of descending length and descending number of intersections. The first word, therefore, is the longest word that has the highest rank.
3. We then place our first word by calling *place_first_word*. It places our first word horizontally in the middle of the grid. After placing the word, we initialize whitespace to account for this newly placed word as well as spaces above and below it.
4. We create an array to hold words placed on the board, and a dictionary to map these words to their x and y positions as well as a boolean indicating whether the word is horizontal (True) or vertical (False).
5. While whitespace exists on the grid:
    a. We re-rank (via the inputted ranking function) and sort the words first by length *and* second by rank after updating the current words in the puzzle.
    b. The rest of the steps are the same as parts b through i in step 5 detailed above in *Algorithm 1: Highest Ranked Word*.

Algorithm 3: Random First Word

With this algorithm, *random_first_word*, we make another slight distinction from *Algorithm 1: Highest Ranked Word*. This time, we change only the first word placed on the board, by randomly choosing it instead of choosing it based on a rank. We do this with *enerate_puzzle_random_first_word,* which takes in the user-inputted size, the grid, our words to clues dictionary, the ranking function to use, and our ranked letters dictionary (this will be used if the ranking function is *Rank by Letter Score*). The steps are as follows:

1. We choose a random word in our dictionary of all words to be placed first on the board.
2. We then place our first word by calling *place_first_word*. It places our first word horizontally in the middle of the grid. After placing the word, we initialize whitespace to account for this newly placed word as well as spaces above and below it.
3. We create an array to hold words placed on the board, and a dictionary to map these words to their x and y positions as well as a boolean indicating whether the word is horizontal (True) or vertical (False).
4. While whitespace exists on the grid:
    a. We rank (via our inputted ranking function) and sort the words after updating the current words in the puzzle.
    b. If ranking the words returns no words, we break out of the loop and our algorithm is done.

c. The rest of the steps follow parts b through i in step 5 of *Algorithm 1: Highest Ranked Word* exactly.

Algorithm 4: Require Alternation

With this algorithm, *require_alternation*, we once again only make a slight distinction from *Algorithm 1: Highest Ranked Word* where we constrain how words can be placed on the grid. Through testing, we concluded that requiring the direction of which words were placed to alternate not only creates a more complex crossword, but also introduces a more interesting layout. To accomplish this, a new find placement method was created, *find_placement_direction_constrained*, such that the placement was only valid if the word was being placed in the opposite direction as placement. We do this with *generate_puzzle_require_alternation,* which takes in the user-inputted size, the grid, our words to clues dictionary, the ranking function to use, and our ranked letters dictionary (this will be used if the ranking function is *Rank by Letter Score*).

Algorithm 5: Require Alternation Random First Word

This algorithm, *require_alternation_random_first_word*, combines *Algorithm 3: Random First Word* with *Algorithm 4: Require Alternation* by requiring that the words alternate in alignment (horizontal/vertical) but select the first word randomly rather than by highest rank as was done in *Algorithm 4*. We did this as a result of scoring metrics produced from Algorithm 3; after seeing the success of random word selection, we hypothesized that requiring alternation in conjunction with random word selection may provide the most compact crossword with greatest intersections. We achieved this with *generate_puzzle_require_alternation_random_first_word,* which takes in the user-inputted size, the grid, our words to clues dictionary, the ranking function to use, and our ranked letters dictionary (this will be used if the ranking function is *Rank by Letter Score*).

Scoring

We score each algorithm in order to return the “best” crossword as determined by its score. In determining the score of a crossword, we created 3 different algorithms which each prioritized various metrics.

1. Minimize Whitespace: This algorithm scores crosswords based on their ability to minimize the whitespace, or the blank characters on the board. This means that we are prioritizing the crosswords which pack in the most letters as possible.
2. Maximize Intersections: This algorithm scores crosswords based on how many intersections the crossword contains. Since a crossword with more intersections means that the words in the

crossword have greater overlap and a greater degree of similarity, maximizing the number of intersections means we are maximizing the denseness of a crossword.

3. Unique Letters: Crossword puzzles have several features which increase the difficulty and build upon the complexity. One such feature is the ability to contain as many unique letters as possible. With more unique letters, the crossword generated is more unique and grabs more interest.

The score returned by each of these algorithms is the metric (ex: number of intersections for the second algorithm) divided by the size of the grid in order to make sure the result is a proportion. These three algorithms were developed to determine the best word placement algorithm. The results of these scoring criteria are discussed below.

GUI

We leveraged Python's PySimpleGUI library to develop the GUI for our crossword puzzle. We chose this library because of its ease of use, myriad of functions and classes, and abundance of thorough documentation. Our GUI contains three possible screens: home, play, and end.

1. The home screen gives the user an introduction to the crossword. At a high level, the user can input the desired size for the puzzle and can then either press "Make Crossword" to enter the play screen, or "Close" to exit the game. We developed *create_home_layout()* to create the layout for the home screen; the layout contains an sg.Text component to welcome the user, an sg.Text component to prompt the user to input a size, an sg.InputText component to allow the user to type in the size, an sg.Button for the close button, and an sg.Text component for the error message.
2. The play screen is where the user will play the crossword by entering answers to the clues. We developed *create_play_layout()* to create the layout for the play screen; this layout contains an sg.Text component to welcome the user to this screen, an sg.Button component for the close button, an sg.Button component to go back to home, and an sg.Graph component where the actual crossword will be.
3. The end screen is where the user will see the completed and correct crossword in addition to the time taken to complete the puzzle. We developed *make_end_window* in order to not only create the end layout, but to also display the correct grid. The end layout contains an sg.Text component to congratulate the user, an sg.Text component to display the time, an sg.Graph component to display the grid, an sg.Button component for the close button, and an sg.Buton component to go back to home.

Each button in each layout is given a key so that we can easily track what has been clicked. Now that we have introduced the screens, it is time to dive deeper into when and how these screens are displayed, and

the logic used to achieve the desired goals. We have 4 boolean variables (home_screen, end_screen, play_screen, and exit_game) to keep track of which screen is currently displayed.

The puzzle will naturally start with only home_screen as True. An sg.Window component is created with the home layout returned from *make_home_layout.* As previously mentioned, we will check if the size inputted by the user is valid. If so, the play layout will be created and *generate_crossword* will be called to start the game. We will now walk through the steps *generate_crossword* will execute to successfully generate the puzzle.

1. The dictionary containing all words and clues for the given size is obtained using the *clean_words* function described earlier in this report.
2. Then, grid arrays and positioned words dictionaries with keys as the words on the grid and values as the word's x and y starting position, and a boolean indicating whether the word is horizontal (True) or vertical (False) are generated from each for the 1st, 2nd, and 4th algorithms.
3. Five puzzles each are generated for *Algorithm 3: Random First Word* and *Algorithm 5: Require Alternation Random First Word.* For both algorithms, the highest scoring grid (determined based on the scoring logic discussed above) is assigned to a variable in addition to its corresponding positioned word dictionary. We now have five grid options.
4. To determine which grid and dictionary of positioned words to use for the puzzle, each of the five calculated grids is scored and the highest-scoring grid is finally selected for the puzzle.
5. Next, we assign numbers to the words in the selected dictionary using *create_word_numbers.* This function loops through the entire grid. If a spot is taken by a letter, we will loop through all the positioned words and assign a number to that word if the positioned word's x and y position matches the current position on the grid. We store word numbers for horizontal and vertical words in separate dictionaries mapping the word to a number. This is to avoid overlapping numbers on the grid.
6. We then call *generate_clues* to create a dictionary mapping 'Horizontal' to a list of (word number, clue) tuples, and 'Vertical' to a list of (word number, clue) tuples.
7. We then place the clues on the sg.Graph component using *format_clues*. We create horizontal and vertical sg.Column elements to append to the play layout. We iterate through the 'Horizontal' and 'Vertical' tuple lists mentioned in step 6 separately and create sg.Text elements for the clues, so that we can add these texts to their appropriate columns. Another facet of this function is creating a mapping of clue numbers to word numbers and their corresponding word. Horizontal clue numbers start from 0 to the length of the 'Horizontal' list minus one, and vertical clue numbers start from the length of the 'Horizontal' list to the length of the 'Vertical' list minus one. Because

these numbers differ from the word numbers written on the crossword boxes, it is important that we create a mapping so we can easily check the correctness of the crossword.

8. We are finally ready to draw the crossword. A play window is created using the play layout passed into the function and the empty crossword is drawn using *generate_grid*, which utilizes PySimpleGUI's draw_rectangle and draw_text functions. White blocks are drawn for boxes requiring a letter and black blocks are drawn for spaces that contain an empty space or a '-' in our algorithms.
9. To fully transfer to the play screen, we will close the home window, set home_screen to False, and set play_screen to True.

On the play screen, the user can enter their answers to the crossword clues and then click the "Check" button to check the answers. This is done in the *check_puzzle* function. This function returns a boolean called correct so we can check if the entire puzzle is right.

1. In this function, the correct grid is generated but not displayed so we can use it to check the user's answers against.
2. Then, we will iterate over each user input and determine if the inputted word matches the answer word stores in the mapping of clue numbers to word numbers. If so, an empty block is drawn for each correct letter. If no answer or a wrong answer is submitted, each wrong or empty block is drawn with a red border.
3. We also add the word numbers to each block again in addition to each letter that the user input.

If the puzzle is correct, we end the timer and make an end window using the logic outlined previously. We then close the play window, set end_screen to True, and set play_screen to False. The game has completed and the end screen is displayed with the features described above.

Results

Here are some visuals from our GUI using a custom dataset as input. Note that our GUI implementation includes minimal styling as we wanted to focus on the implementation of our algorithms and AI components:

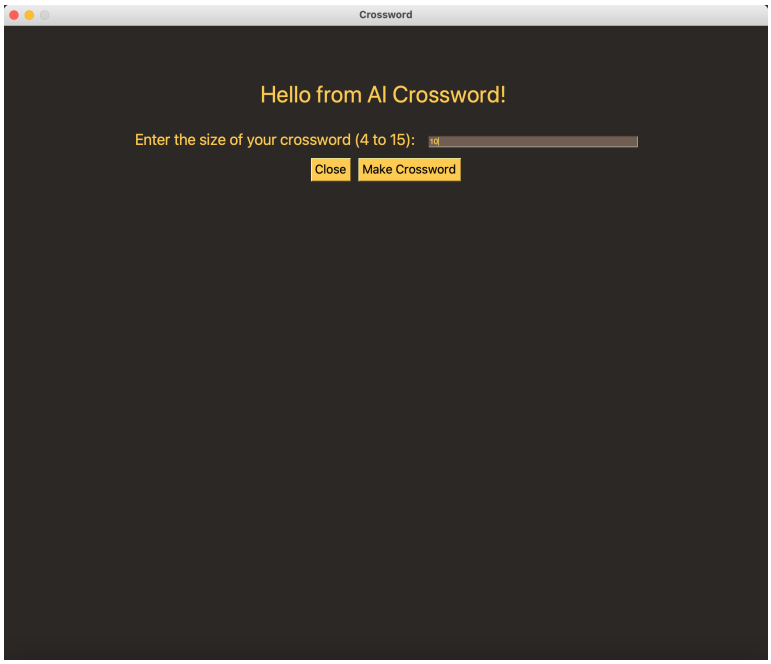

Figure 1: Home Screen

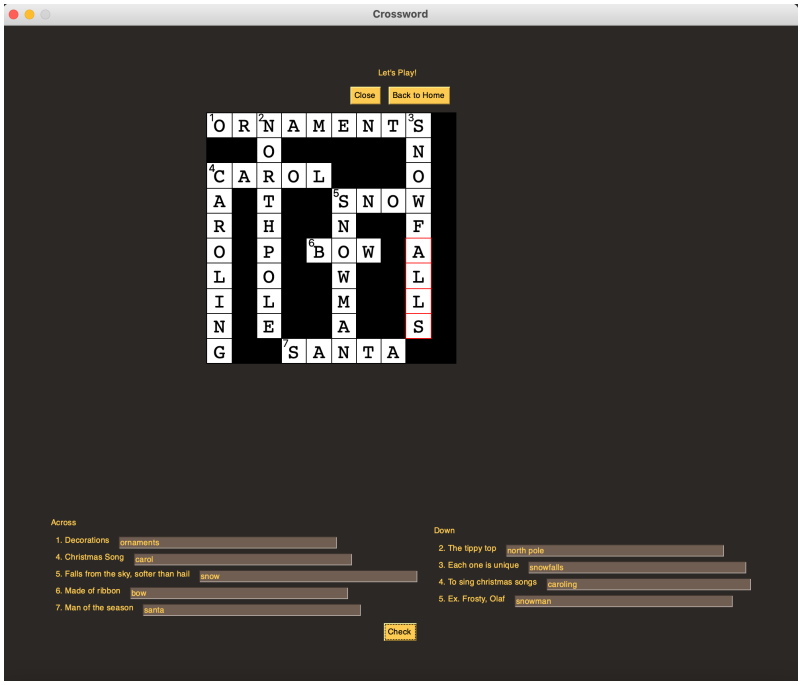

Figure 2: Play Screen (10x10 crossword with an incorrect answer)

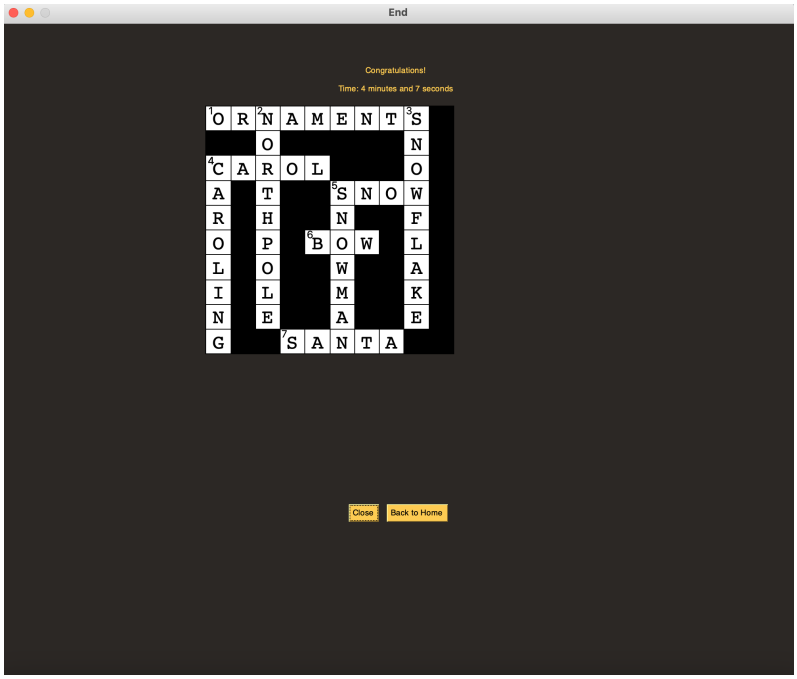

Figure 3: End Screen (after all answers in the 10x10 crossword are correct)

Additionally, through rigorous testing, we were able to not only determine which algorithm (on average) performed the best for each scoring metric, but we were also able to determine which scoring algorithm prioritized what we valued most in a crossword. After running rigorous testing models which evaluated the various ranking algorithms, placement algorithms, and scoring algorithms, we accumulated the following results from a run of the test file:

	Minimize Whitespace	Maximize Intersections	Unique Letters
Highest Ranked	0.45453	0.20731	0.18891
Highest Ranked Longest Word	0.45453	0.20731	0.18891
Random First Word	**0.46170**	**0.23801**	**0.20069**
Require Alt.	0.45232	0.22758	0.18139
Require Alt. Random First Word	0.43246	0.22262	0.19082

Figure 4: Results of each scoring algorithm combined with each placement algorithm.

	Common Letters	Letter Score	Number of Intersections	Without Intersections	Without Intersections and Unique Letters
Highest Ranked	0.46358	0.48102	0.46020	0.45453	**0.48513**
Highest Ranked Longest Word	0.46358	0.48102	0.46020	0.45453	**0.48513**
Random First Word	**0.46424**	**0.48364**	**0.46313**	**0.46320**	0.45997
Require Alt.	0.45951	0.48120	0.45049	0.45232	0.44960
Require Alt. Random First Word	0.43443	0.46355	0.43873	0.43446	0.43825

Figure 5: Results of each ranking function combined with each placement algorithm (using the minimize whitespace scoring algorithm).

Discussion

The testing results above were produced by running our code on a custom dataset containing winter-themed words and clues. This highlights the capability for the user to input a desired dataset and generate crossword puzzles for those words. However, throughout the project, we also worked with the large CSV containing words and clues from various esteemed crossword puzzle websites (mentioned in the Methods section).

Figure 4 highlights the scores from using each scoring algorithm (top panel) with each placement algorithm (left panel). The *Minimize Whitespace* scoring algorithm produced the highest scores overall. This tells us that the minimizing whitespace scoring algorithm was the best for crossword generation if we are prioritizing a high score. Further, as one can see, the *Random First Word* placement algorithm tended to produce the highest score with each of the scoring algorithms. *Figure 5* also shows that the *Random First Word* placement algorithm produces the highest score when combined with 4 out of the 5 ranking functions. This is an indicator that perhaps the *Random First Word* placement algorithm, specifically in conjunction with the *Minimize Whitespace* scoring algorithm generates the best crossword puzzles. This might be because generating a random word can increase the chance of finding intersecting words and thus doing well with other algorithms due to increased variation. However, for puzzles that used either the *Without Intersections* or *Without Intersections and Unique Letters*, we might want to consider using the *Highest Ranked* or *Highest Ranked Longest Word* placement algorithms. On the other hand, using the *Unique Letters* ranking function tended to produce low scores, relative to other ranking algorithms, when run with each of the placement algorithms. If high scores are the ultimate goal, we probably should not use the *Unique Letters* algorithm.

Our project centers around using constraint satisfaction as our main AI component. The algorithms and functions that we developed were designed and implemented to satisfy constraints. Constraint Satisfaction was an ideal AI component which we were able to leverage due to the complexities of the solution set. As several valid crosswords can be generated as long as they fit certain constraints, a constraint satisfaction approach allows the largest solution set to be generated. We considered several constraints when creating the algorithms which placed words on the board, some of which are as follows:

1. No two letters from different words unless the words intersect may be directionally one square away. This was done to ensure that we were not generating words as a result of word placement. When words are placed side by side, additional words due to the intersection of grid squares are generated. To avoid this, the constraint of no adjacent letters to any letter in the word tile (excluding the intersection point) was included.
2. No words can be placed if the entire word does not fit on the grid. Words must be able to be fully shown by the grid size allocated by the user.
3. Only one letter can fit in a grid square. In order to present a valid crossword, words can only intersect if they share a letter at the shared letter.

Further, we investigated the performance of our system. Because most crossword datasets, including the large CSV containing words and clues from various esteemed crossword puzzle websites, are very large, it is natural that generating a crossword will take several minutes, especially for large sizes. However, even for large sizes, the code runs to completion and takes polynomial time to execute. Our solution is not an NP-hard problem.

Conclusion

Through this project we determined that there are various algorithms for crossword generation. Surprisingly, as shown by the results, the algorithm which chose the random starting word performed the best for nearly all algorithms of placing additional words. Additionally, we were able to build upon this framework by creating our own dictionary of words and their clues (themed specifically to generate a ‘winter’ themed crossword). With this extension, individuals would be able to input a word list with clues or they would be able to scrape for the clues themselves in addition to the size of the crossword they are seeking to generate their ideal crossword.

Works Cited

Bacchus, Fahiem, et al. *Binary vs. Non-Binary Constraints - Unsw Sites*. https://www.cse.unsw.edu.au/~tw/bcvwaij02.pdf.

"Clues." *Data: Clues: 648,046 Rows*, https://cryptics.georgeho.org/data/clues. (Dataset mentioned in the Methods section)

J.Mazlack, Lawrence. "Computer Construction of Crossword Puzzles Using Precedence Relationships." *Artificial Intelligence*, Elsevier, 21 Feb. 2003, https://www.sciencedirect.com/science/article/pii/0004370276900199#section-cited-by.

Mittal, Sanjay, and Brian Falkenhainer. *Dynamic Constraint Satisfaction Problems*. 1990, https://www.aaai.org/Papers/AAAI/1990/AAAI90-004.pdf.

Pearson, Justin, and Peter Jeavons. *A Survey of Tractable Constraint Satisfaction Problems*. 14 July 1997, http://www.cis.uoguelph.ca/~yxiang/6890/csp-survey.pdf.

Rigutini, Leonardo, et al. "(PDF) Automatic Generation of Crossword Puzzles - Researchgate." *ResearchGate*, June 2012, https://www.researchgate.net/publication/229589133_Automatic_Generation_of_Crossword_Puzzles.

Wehar, Michael. "Ai Generated Crossword Puzzles." *Artificial Intelligence* +, 24 Oct. 2022, https://www.aiplusinfo.com/blog/ai-generated-crossword-puzzles/.