MP2 Report

Dennis J. McWherter, Jr. (dmcwhe2@illinois.edu) 4 credit
Joe Fetsch (jfetsch2@illinois.edu) 4 credit
Revanth Reddy (rreddy4@illinois.edu) 4 credit

October 24, 2016

1 Parts 1.1 and 1.2

This section describes our implementation details for part 1.1. Solutions can be found in the Solutions section (section 5)

1.1 Variables, Values, and Constraints

Our implementation for the Word Sudoku solver used the **words** from the word bank as *variables* and **linear horizontal or vertical groupings of cells** as *values*. Values are represented by a 3-tuple specified as $(orientation, position, length)$. $orientation$ is specified as either 'H' or 'V' for horizontal or vertical, respectively. The starting position refers to the (row, col) grid location where the beginning of the word is placed. Finally, $length$ is the value that specifies the total number of cells from this position that defined the total set of cells described by this grouping.

Having modeled the problem this way, there are several constraints. The first is that for any given value, the $length$ must be equal to the word length of our selected variable (i.e. only exact fits into cell-groupings are significant). Similarly, for any assignment, the current state of the board must follow the Sudoku constraints (i.e. unique letter per row, column, and 3x3 cell). Likewise, any assignment must also consider each assigned square on the board. If an assignment is overlapping a previous assignment, it cannot change any existing character on the board.

More formally, our constraints are as follows:

$$\text{sudoku}(x) = \begin{cases} 1 & \text{diff}(x, y), \forall y \in \text{adjacent}(x) \\ 0 & \text{otherwise} \end{cases}$$

$$\text{overlap}(pos, word) = \begin{cases} 1 & \text{assignmentUnchanged}(pos, word) \\ 0 & \text{otherwise} \end{cases}$$

$$\text{consistent}(board, word) = \min_{\forall pos \in board} (\text{sudoku}(pos), \text{overlap}(pos, word), \text{inBounds}(pos, word))$$

Where `diff` returns true iff x and y are assigned (i.e. not the '_' character) and distinct, `adjacent` returns all values for a position's row, col, and 3x3 cell, `inBounds` returns whether or not a word can fit at that starting position based on length, and `assignmentUnchanged` ensures that for any character in a *word* start at *pos*, placing that word does not change any existing assignments (i.e. overlaps are compatible). It should be noted that the `sudoku` is checked after placement to ensure the incoming word does not violate this constraint.

1.2 Implementation

Our implementation followed the standard form for backtracking search (using the pseudocode provided in lecture). The more interesting bit of our implementation follows our selection and ordering heuristics. For our *most constrained variable*, we selected the word with fewest possible positions on the Sudoku board. For our tie-breaking strategy, we used the *most constraining variable* heuristic in which we selected the longest word from the list. For value ordering, we filtered remaining values to those with valid length. From there, we used the *least constraining value* heuristic and selected the value with the fewest number of unassigned grid squares within its grouping.

Due to how the problem is modeled, it's important to note that our implementation need not explicitly assign *each* word to a position. Where words overlap, it's possible that a valid assignment is already made implicitly. Specifically, consider the case in for part 1.1's result along the right-most column. The words **PYTHON** and **NECK** were explicitly assigned. However, when combined next to each other, **ONE** is already implicitly assigned.

Moreover, hints were not directly handled as a special case. Where hints existed on the board our algorithm considered like all other constraining squares.

2 Part 1.3

Our implementation in part 1.3 varied since we had to check all possible variables for a given step. Specifically, since we were aware of the existence of unplaceable words, if a specific word could not be placed our algorithm had to try another in its place (out of a list of valid words for that step). That being said, the fact that a word could not be placed in a single instance also did not specifically make it an impostor. Instead, the remaining configuration may have just been improperly setup. Consequently, this resulted in a required search of the entire space. When a solution was found, it was added to a list of solutions and flushed out to disk instead of returning upon the first result.

3 Part 2.1

This section describes our implementation and evaluation functions for part 2.1. Similarly, we discuss general trends and conclusions observed in this scenario. For our solutions, please see section 5.

3.1 Implementation

Our implementation follows the standard form for minimax search and alpha-beta pruning. To fully formulate our solution for alpha-beta search, we supplemented the lecture materials with a Cornell lecture [1]. We use a depth-limited search with a maximum depth of 3 to explore nodes and choose the appropriate strategy at each step.

Our implementation was pluggable to allow for a greater variety of adjustment with maximum code reuse. That is, features such as strategy, search depth, etc. were configurable on a per-agent basis. Each agent used the same basic code to execute their strategies, but it ran with their specified strategy. Moreover, we would run two **distinct** agents against each other. That is, each agent would run a separate copy of their solver without any knowledge of the opponent's strategy or solving mechanism.

3.2 Evaluation Functions

We implemented two different heuristic functions for our Breakthrough game solver. The **offensive** heuristic function prioritizes capturing pieces and making progress toward the opponent's goal, while penalizing having pieces threatened by the opponent. More formally:

$$\text{offense}(P, O) = 50 * \text{stolenPieces} - 10 * \text{numThreatened} + \max_{x \in P} (\text{goalRow}(x) - \text{distanceToGoal}(x))^2$$

where P and O are the current player's and opponent's pieces, respectively. The `distanceToGoal` function takes a piece and finds its closest linear distance to the opponents goal. The functions `numThreatened` and `stolenPieces` return the number of threatened pieces for the current player and the number of pieces lost by the current player, respectively.

The **defensive** heuristic, on the other hand, prioritizes keeping pieces alive, maximizing the opponent's distance to the goal, and, finally, penalizes for the number of pieces threatened by the opponent. Formally,

$$\text{defense}(P, O) = 10 * \text{stolenPieces} + 50 * \text{remainingPieces} - \text{closestOpponentToGoal}^2 - 5 * \text{numThreatened}$$

where P, O, `distanceToGoal`, `numThreatened`, and `stolenPieces` retain their definitions from earlier. The `closestOpponentToGoal` provides the score of how close an opponent is to the current player's home row. This score increases as the player moves closer to winning the game.

Regardless of the heuristic used, the evaluation function always returns ∞ or $-\infty$ for match wins or losses, respectively. Therefore, the heuristics are only used to estimate game outcomes and are not used when the outcome is known.

3.3 Conclusions

Overall, our observations about alpha-beta and minimax search were largely in-line with expectations. That is, in our trials we noticed that alpha-beta always performed fewer node expansions. Consequently, it also had a fast average decision time in making moves to play the game. While the searching mechanism does not alter the outcome of the game, it does alter the overall runtime.

At a depth of 3 for both players, we observed that whether or not a player wins depends largely on the opponent's strategy and in which order a player starts. For instance, assuming player 1 always goes first, player 1 will win if both players pick an offensive strategy. However, when both players pick a defensive strategy, the second player wins. If different strategies are chosen, the defensive player wins every time.

As expected, things do change when you alter the depth. For instance, if the opponent is playing a depth of 1, the player playing a depth of 3 will always win regardless of strategy or when the player starts. However, it seems that in these cases, the player at depth 3 is forced to expand a greater number of nodes in alphabeta search than when playing an optimal opponent. This ultimately slows down the decision time for making the next move and increases the overall number of moves taken during the gameplay.

4 Part 2.2

This portion of the project extends the functionality of the board and gameplay in two way. Thanks to the extensibility of the code for part 2.1, we only had to alter relatively few lines of code to achieve the desired changes while the overall simulation offensive and defensive evaluation functions remain the same.

We added gives an option for the user to choose a different end goal rule, where the winner is the first player to get three workers to the opponent's base, instead of the just one. This involved parsing the input argument integer and then adjusting the end goal checking function.

We gave the user to an option simulate an alternate rectangular board with the dimension of 5x10 with each player having 20 pieces (two rows of 10) instead of just 16 pieces for the standard square 8x8 board. The alteration of the code basically consists of replacing the existing hardcoded values for the square board to more generalized variables for number of board rows and columns that can be set based on the board chosen.

An interesting observation from this setting is that player 2 performs better for all scenarios for the oblong-shaped board when both agents run their search algorithms at depth 3. However, when reducing the depth of search for player 2 to 1, player 1 can win. On the other hand, player 1 performs better when both players choose the same strategy if playing on the "3 workers to base" ruleset.

5 Solutions

Below are various solutions to each problem.

5.1 1.1 Solution

Part 1.1 solved the board by expanding **28 nodes** in **0.16 seconds**.

```
L I G H T E N M P
C O N F U S E A Y
S U P W I N D R T
E T U N D R A V H
M R F I C K Y E O
I A O M S H P L N
N G L B A U O I E
A E K L V M U N C
R D S Y E P T G K
```

(a) Result board

```
V, 0, 7: MARVELING
V, 1, 1: OUTRAGED
H, 0, 0: LIGHTEN
H, 1, 0: CONFUSE
V, 2, 0: SEMINAR
V, 0, 8: PYTHON
V, 3, 3: NIMBLY
H, 3, 1: TUNDRA
H, 2, 1: UPWIND
V, 4, 2: FOLKS
V, 5, 8: NECK
V, 5, 6: POUT
V, 5, 5: HUMP
H, 4, 3: ICKY
V, 5, 4: SAVE
```

(b) Assignment ordering

Figure 1: Part 1.1 solution

5.2 1.2 Solution

Part 1.2 solved the board by expanding **187 nodes** in **0.38 seconds**.

```
D C O Q U E T R Y
R L B O A T I N G
I A S V G L O B E
V M T E S O A L B
E P I N Y C R U X
L D N B M K P I S
S O A I B J U D P
U W C R O A N E I
B N Y D L W K A T
```

(a) Result board

```
V, 0, 2: OBSTINACY
V, 0, 1: CLAMPDOWN
H, 0, 1: COQUETRY
V, 1, 3: OVENBIRD
V, 2, 5: LOCKJAW
V, 0, 0: DRIVELS
H, 1, 2: BOATING
V, 3, 4: SYMBOL
H, 2, 4: GLOBE
V, 5, 8: SPIT
V, 5, 6: PUNK
V, 5, 7: IDEA
H, 4, 5: CRUX
H, 7, 3: ROAN
H, 5, 6: PIS
H, 3, 6: ALB
V, 6, 0: SUB
```

(b) Assignment ordering

Figure 2: Part 1.2 solution

5.3 1.3 Solution

Part 1.3 solved the board by expanding **3429518 nodes** in **6590.58 seconds** for the very first solution. While we couldn't complete searching the entire subspace, we did dump intermediate state of our algorithm as solutions were found (located in the state.pickle file). The best solution we found contained only **17 word assignments**. As before, all words that exist in the final puzzle assignment (i.e. *JAGS*) do not necessarily have to be explicitly assigned.

```
S O R I E N T A L
N P A T C H I E R
I E G R Y P H O N
C R O S D W E L T
K A N U B G R I M
E T I L R J A G S
R I Z K A E B U C
A N E W I T L S O
M G D O N S Y T W
```

(a) Result board

```
V, 0, 1: OPERATING
V, 1, 2: AGONIZED
H, 0, 1: ORIENTAL
H, 1, 1: PATCHIER
V, 0, 0: SNICKER
H, 2, 2: GRYPHON
H, 3, 4: DWELT
V, 4, 4: BRAIN
V, 5, 8: SCOW
V, 5, 7: GUST
V, 5, 6: ABLY
V, 3, 3: SULK
V, 5, 5: JETS
H, 7, 3: WIT
H, 4, 5: GRIM
V, 6, 0: RAM
H, 8, 2: DON
NA: AMBUSH
NA: ENLARGE
NA: JAGS
NA: ANEW
NA: STRIDE
```

(b) Assignment ordering

Figure 3: Part 1.3 solution

5.4 2.1 Solution Boards

The boards below are the final board states represented by player. All games were played with both players playing at a **depth of 3**. Any location containing a 1 or 2 corresponds to players 1 or 2, respectively. Any location with a 0 means that it is a free, unclaimed location. Similarly, the matchup is described in the caption in the form of **Player1 Strategy v. Player2 Strategy**. It should be noted that we have reduced the number of boards (from 16 total games to 4 outcomes) since the outcome is strictly dependent on strategy. As denoted in the tables below, the search algorithm only affects the number of nodes expanded and running times of the game. The tables in the following section, however, enumerate all of the values for each simulation.

```
1 0 0 0 2 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 2 0 0 0 2 2
0 0 2 2 2 0 2 2
0 0 0 0 0 0 2 2
1 0 0 0 0 0 0 2
0 0 0 0 0 0 2 2
```

(a) Offense v. Defense

```
1 1 0 0 0 0 0 0
1 1 1 1 1 0 0 0
1 1 1 1 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 2
0 0 2 0 2 2 0 0
0 0 0 0 0 0 0 2
0 1 0 0 0 0 0 0
```

(b) Defense v. Offense

```
1 1 1 0 1 1 1 0
1 1 0 0 0 0 0 0
0 0 0 0 0 0 0 2
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 2
0 0 0 0 0 0 0 2
1 0 0 0 0 0 2 0
```

(a) Offense v. Offense

```
0 0 0 0 0 0 2 0
1 1 1 0 0 0 1 0
1 1 1 1 0 0 0 0
1 1 1 1 1 0 2 2
0 0 0 0 0 0 2 2
2 2 2 0 2 2 2 2
0 0 1 2 0 0 0 2
0 0 0 0 0 0 0 0
```

(b) Defense v. Defense

5.5 2.1 Solution Statistics

Below are the tables enumerating various output statistics of each simulated matchup.

	Total Exp. Nodes	Avg. Nodes per Move	Avg. Time per Move (ms)	Pieces Captured
Player 1	17457	416	701	2
Player 2	17932	427	840	14

Total Turns: 84
Game Winner: Player 2

Figure 6: Minimax Offense vs. Minimax Defense

	Total Exp. Nodes	Avg. Nodes per Move	Avg. Time per Move (ms)	Pieces Captured
Player 1	16433	513	1305	11
Player 2	15811	510	1134	4

Total Turns: 63
Game Winner: Player 1

Figure 7: Minimax Defense vs. Minimax Offense

	Total Exp. Nodes	Avg. Nodes per Move	Avg. Time per Move (ms)	Pieces Captured
Player 1	12614	420	817	12
Player 2	11968	413	708	7

Total Turns: 59
Game Winner: Player 1

Figure 8: Minimax Offense vs. Minimax Offense

	Total Exp. Nodes	Avg. Nodes per Move	Avg. Time per Move (ms)	Pieces Captured
Player 1	19817	601	1641	2
Player 2	19565	593	1430	2

Total Turns: 66
Game Winner: Player 2

Figure 9: Minimax Defense vs. Minimax Defense

	Total Exp. Nodes	Avg. Nodes per Move	Avg. Time per Move (ms)	Pieces Captured
Player 1	14708	350	469	2
Player 2	17932	427	844	14

Total Turns: 84
Game Winner: Player 2

Figure 10: Alphabeta Offense vs. Minimax Defense

	Total Exp. Nodes	Avg. Nodes per Move	Avg. Time per Move (ms)	Pieces Captured
Player 1	11105	347	727	11
Player 2	15811	510	1128	4

Total Turns: 63
Game Winner: Player 1

Figure 11: Alphabeta Defense vs. Minimax Offense

	Total Exp. Nodes	Avg. Nodes per Move	Avg. Time per Move (ms)	Pieces Captured
Player 1	10615	353	537	12
Player 2	11968	412	710	7

Total Turns: 59
Game Winner: Player 1

Figure 12: Alphabeta Offense vs. Minimax Offense

	Total Exp. Nodes	Avg. Nodes per Move	Avg. Time per Move (ms)	Pieces Captured
Player 1	12671	384	802	2
Player 2	19565	593	1436	2

Total Turns: 66
Game Winner: Player 2

Figure 13: Alphabeta Defense vs. Minimax Defense

	Total Exp. Nodes	Avg. Nodes per Move	Avg. Time per Move (ms)	Pieces Captured
Player 1	17457	416	703	2
Player 2	12686	302	537	14

Total Turns: 84
Game Winner: Player 2

Figure 14: Minimax Offense vs. Alphabeta Defense

	Total Exp. Nodes	Avg. Nodes per Move	Avg. Time per Move (ms)	Pieces Captured
Player 1	16433	514	1300	11
Player 2	13226	427	592	4

Total Turns: 63
Game Winner: Player 1

Figure 15: Minimax Defense vs. Alphabeta Offense

	Total Exp. Nodes	Avg. Nodes per Move	Avg. Time per Move (ms)	Pieces Captured
Player 1	12614	420	814	12
Player 2	7948	274	358	7

Total Turns: 59
Game Winner: Player 1

Figure 16: Minimax Offense vs. Alphabeta Offense

	Total Exp. Nodes	Avg. Nodes per Move	Avg. Time per Move (ms)	Pieces Captured
Player 1	19817	600	1647	2
Player 2	12522	379	777	2

Total Turns: 66
Game Winner: Player 2

Figure 17: Minimax Defense vs. Alphabeta Defense

	Total Exp. Nodes	Avg. Nodes per Move	Avg. Time per Move (ms)	Pieces Captured
Player 1	14882	354	462	2
Player 2	12881	306	550	14

Total Turns: 84
Game Winner: Player 2

Figure 18: Alphabeta Offense vs. Alphabeta Defense

	Total Exp. Nodes	Avg. Nodes per Move	Avg. Time per Move (ms)	Pieces Captured
Player 1	11150	348	750	11
Player 2	13579	438	627	4

Total Turns: 63
Game Winner: Player 1

Figure 19: Alphabeta Defense vs. Alphabeta Offense

	Total Exp. Nodes	Avg. Nodes per Move	Avg. Time per Move (ms)	Pieces Captured
Player 1	12521	342	528	12
Player 2	8876	306	418	7

Total Turns: 59
Game Winner: Player 1

Figure 20: Alphabeta Offense vs. Alphabeta Offense

	Total Exp. Nodes	Avg. Nodes per Move	Avg. Time per Move (ms)	Pieces Captured
Player 1	12390	375	812	2
Player 2	12521	379	788	2

Total Turns: 66
Game Winner: Player 2

Figure 21: Alphabeta Defense vs. Alphabeta Defense

5.6 Part 2.2 Solutions

This section describes the final state of representative games using our heuristics to solve part 2.2. Included are the statistics requested from part 2.1 as well as the final board state. The description of each scenario can be found in the caption and follows a similar form as before.

	Total Exp. Nodes	Avg. Nodes per Move	Avg. Time per Move (ms)	Pieces Captured
Player 1	12690	604	1068	7
Player 2	9071	432	537	12

```
1 1 0 1 1 0 0 1 2 0
1 1 0 0 0 0 0 0 0 0
0 0 2 2 2 0 0 2 0 2
2 0 0 2 0 1 0 2 2 2
2 0 0 0 0 0 0 0 0 2
```

Total Turns: 42
Game Winner: Player 2

Figure 22: Rectangular board: Offense Alphabeta vs. Defense Alphabeta

	Total Exp. Nodes	Avg. Nodes per Move	Avg. Time per Move (ms)	Pieces Captured
Player 1	11123	556	753	7
Player 2	8932	447	805	10

```
1 1 1 1 1 1 1 0 2 0
1 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 2 2 0 2
2 2 2 2 1 0 0 0 2 2
2 0 0 0 2 0 0 0 0 2
```

Total Turns: 40
Game Winner: Player 2

Figure 23: Rectangular board: Offense Alphabeta vs. Offense Alphabeta

	Total Exp. Nodes	Avg. Nodes per Move	Avg. Time per Move (ms)	Pieces Captured
Player 1	13993	341	461	2
Player 2	13129	320	461	16

```
0 0 0 0 0 0 0 0
0 0 2 0 0 0 0 0
2 0 0 0 0 0 0 0
0 0 2 0 0 0 0 2
0 0 0 2 2 2 0 2
0 0 0 0 0 2 2 2
0 0 0 0 0 0 0 2
0 0 0 0 0 0 2 2
```

Total Turns: 82
Game Winner: Player 2

Figure 24: 3 workers to base: Offense Alphabeta vs. Defense Alphabeta

	Total Exp. Nodes	Avg. Nodes per Move	Avg. Time per Move (ms)	Pieces Captured
Player 1	11468	244	347	12
Player 2	9580	208	246	8

```
1 1 0 0 0 0 2 2
1 1 0 0 0 0 2 0
1 0 0 0 0 0 0 2
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
1 1 0 1 0 0 0 0
```

Total Turns: 93
Game Winner: Player 1

Figure 25: 3 workers to base: Offense Alphabeta vs. Offense Alphabeta

6 Statement of Individual Contribution

This MP was completed as a joint effort between Dennis, Joe, and Revanth. The solution to part 1 was a full group collaboration; the discussion of design and pair programming involved all three parties. Dennis started the initial draft of the report while Revanth and Joe revised and edited. Part 2 was initially sketched out by Dennis but later finalized and completed by all 3 parties.

References

[1] Cornell.edu, 'Minimax search and Alpha-Beta Pruning', n.d. [Online]. Available: `https://www.cs.cornell.edu/courses/cs312/2002sp/lectures/rec21.htm`. [Accessed: 16- Oct- 2016].