CS 440 MP1
Section Q4
4 Credits

Christian Howard
howard28@illinois.edu

Luke Pitstick
pitstck2@illinois.edu

Abstract

Within this report, an analysis was performed on solutions to a set of games, Flow Free and Breakthrough. To handle Flow Free, it was modeled as a Constraint Satisfaction Problem (CSP) and a solver was built to tackle it in this form. It was found that using more sophisticated for constraint propagation leads to significant improvements in the solver efficiency with various test problems to back it up against different level of sophisticated CSP solvers. For Breakthrough, a framework was developed to build and play different agents against each other with different heuristics. First, it was found that using Alpha-Beta pruning and a Utility-estimate based mechanism to select branches of the search tree to investigate slimmed down the number of nodes searched relative to Minimax search. It was also found that coming up with more dominant Defensive heuristics was easier across the three Breakthrough variations required of 4 credit students, while the Offensive heuristic was challenging. It was found that Defensive heuristics generally perform better than Offensive heuristics, reasons which could be chance or something more fundamental to the game.

Contents

1 Part 1 — Constraint Satisfaction Problem - Flow Free **3**
- 1.1 Problem Formulation . . . 3
- 1.2 Smart Implementation . . . 3
- 1.3 Smarter Implementation . . . 3
- 1.4 Results . . . 3

2 Part 2 — Game of Breakthrough **5**
- 2.1 Action Ordering for Alpha-Beta . . . 5
- 2.2 Heuristic Definitions . . . 5
  - 2.2.1 Provided Heuristics . . . 5
  - 2.2.2 Custom Defensive Heuristics . . . 6
  - 2.2.3 Custom Offensive Heuristics . . . 6
  - 2.2.4 Particle Swarm Optimization based Heuristics . . . 6
- 2.3 Matchup Results . . . 7
  - 2.3.1 8×8 Board with Nominal Rules . . . 7
  - 2.3.2 5×10 Board with Nominal Rules . . . 8
  - 2.3.3 8×8 Board with Modified Rules . . . 9

3 Contributions **12**

4 List of Potential Extra Credit **13**

Appendices **14**

A Puzzle Solutions for Part 1 **14**

B Breakthrough Final Board Configuration **18**

C Open Source Software Used **21**

1 Part 1 — Constraint Satisfaction Problem - Flow Free

1.1 Problem Formulation

The variables in this CSP were the coordinates of each individual cell. These could be represented as a set of tuple variables like $V = \{(x_i, y_i)|i = 1, \cdots, n\}$. Some of these variable are already assigned as source nodes and cannot be changed. The other variables can take on any value from within the domain or no value before being assigned. The domain is a set of colors represented by a capital letter $D = \{A, B, \cdots, Z\}$. There are only two constraints. First, all cells must be assigned a color. Second, source nodes must have 1 adjacent cell of the same color, and the others must have 2 adjacent of the same color. The result of the second constraint covers the requirement that both source nodes are connected to each other. It also takes care of not allowing zig-zags.

1.2 Smart Implementation

The Smart implementation adds features to reduce the number of attempted assignments. On a first pass, it constrains any locations where a source node only has one empty neighbor, and therefore must be a certain color. Every time an assignment is added, the maze is re-checked to make sure it did not violate any constraints itself, or for other nodes. This is in addition to the local checks made before allowing the assignment to take place.

1.3 Smarter Implementation

The Smarter approach takes advantage of constraint propagation. When a node is added and it has two neighbors of the same type, then we know that it's other neighbors cannot be the same value. This arc consistency is checked when a node is added. This reduces the valid domain of unassigned nodes, which means when they are selected sooner to be assigned by the Minimum Remaining Values heuristic. This causes faster failures on bad branches, thus exploring less assignments.

1.4 Results

The results show the magnitude of improvements of each of the features mentioned above. Although the simpler 5×5 case shows only small improvement, the Dumb approach could not solve anything beyond it in reasonable time. The Smart approach had no problem with the 7 to 9 dimension problems. The runtime and attempted assignments increase similarly with an increase in problem size. Each increase in problem size sees a 100x performance hit in both areas.

We see a similar improvement with the Smarter approach. Where the Smart approach could not solve the 10×10 problem in a reasonable time, the Smarter

approach could. In the other cases, it was able to reduce the attempted assignments and run-time by a factor of about 2. Other inferences or deeper constraint propagation could still be added to further improve this. One attempted inference was to prioritize the variable with the most neighbors. This is a constraint that is not always covered by the reduced domain case, but it proved to not be useful.

Note that visuals for the inputs and solutions for each puzzle can be found in the Appendix A.

	5×5	7×7	8×8	9×9	10×10 v1	10×10 v2
Dumb	67	N/A	N/A	N/A	N/A	N/A
Smart	63	1,131	15,345	1,487,384	N/A	N/A
Smarter	63	1,131	13,125	634,987	3,965,214	3,124,689

Table 1: Number of Attempted Assignments

	5×5	7×7	8×8	9×9	10×10 v1	10×10 v2
Dumb	0.016	N/A	N/A	N/A	N/A	N/A
Smart	0.009	0.056	1.17	125.3	N/A	N/A
Smarter	0.01	0.068	1.01	56.4	3,168	3,551

Table 2: Run-Time in Seconds

2 Part 2 — Game of Breakthrough

2.1 Action Ordering for Alpha-Beta

The order of actions to try when using the Alpha-Beta search algorithm was based on using the Heuristic function to estimate the utility of some action and then choose the action that would most likely prune the search the most. Given that the utility function is sufficiently accurate, it should be expected that this pruning will be effective and not result in missing too many search tree branches worth checking.

2.2 Heuristic Definitions

Before going into the specific heuristics, the features constructed based on the game state will be defined. The following features represent those found and used to define heuristics:

f_1 = Number of pieces
f_2 = Mean distance from top of board to pieces
f_3 = Standard deviation of distance from top of board to distance
f_4 = Standard deviation of horizontal locations of pieces
f_5 = Average barrier size, where barrier defined as horizontal chain of pieces
f_6 = Number of pieces in the goal row of the board
f_7 = Number of pieces in the home row of the board
f_8 = Number of pieces 1 row away from goal row of board
f_9 = The maximum distance of some friendly piece from the home row

Note that these features can be viewed as being generated by some feature mapping $M(\cdot)$ such that the following is true for some game state s:

$$f = M(s)$$

where $f = [f_1, \cdots, f_9]^T$. This mapping is used in the software to make it (hopefully) easier to construct the heuristics. Additionally, let us define the operator $F(\cdot)$ as one that returns the friendly value for some input feature component and $E(\cdot)$ as one that returns the opponent's value for some input feature component, e.g. $F(f_1)$ returns the number of friendly pieces and $E(f_5)$ returns the average barrier size for the opponent's team.

2.2.1 Provided Heuristics

For this assignment, two heuristics were provided to use as baseline heuristics when we go to come up with our own. The two heuristics were defined as the following:

$$h_{pd}(f) = 2F(f_1) + \omega$$
$$h_{po}(f) = 2\left(30 - E(f_1)\right) + \omega$$

where $\omega \sim U(0,1)$, $h_{pd}(\cdot)$ is the provided defensive heuristic, and $h_{po}(\cdot)$ is the provided offensive heuristic.

2.2.2 Custom Defensive Heuristics

The custom defensive heuristics are defined as the following:

$$h_d^{(1)}(f) = 2F(f_1) - 50F(f_3) - 50E(f_1) + \omega$$
$$h_d^{(2)}(f) = 2F(f_1) - 20F(f_3) - 50E(f_9) + \omega$$
$$h_d^{(3)}(f) = 2F(f_1) - 20F(f_3) - 10E(f_1) - 100E(f_6) + \omega$$

where $\omega \sim U(0,1)$, $h_d^{(1)}(f)$ is the defensive heuristic for the 8×8 board with nominal rules, $h_d^{(2)}(f)$ is the defensive heuristic for the 5×8 board with nominal rules, and $h_d^{(3)}(f)$ is the defensive heuristic for the 8×8 board using modified rules such that it takes 3 pieces in the goal row to win and an agent will lose if less than 3 pieces are on the board.

2.2.3 Custom Offensive Heuristics

The custom offensive heuristics used in the later analysis are the following:

$$h_o^{(1)}(f) = 100F(f_6) - 20\left(F(f_3) + F(f_4)\right) - 100E(f_1) + \omega$$
$$h_o^{(2)}(f) = 100F(f_6) - 10\left(F(f_3) + F(f_4)\right) - 100E(f_1) + \omega$$
$$h_o^{(3)}(f) = 100F(f_6) + 10F(f_8) - 50F(f_3) - 100E(f_1) + \omega$$

where $\omega \sim U(0,1)$, $h_o^{(1)}(f)$ is the offensive heuristic for the 8×8 board with nominal rules, $h_o^{(2)}(f)$ is the offensive heuristic for the 5×8 board with nominal rules, and $h_o^{(3)}(f)$ is the offensive heuristic for the 8×8 board using modified rules such that it takes 3 pieces in the goal row to win and an agent will lose if less than 3 pieces are on the board.

2.2.4 Particle Swarm Optimization based Heuristics

It should be noted another set of custom heuristics that were created, though were not used in the analysis to follow. These heuristics were formed by using a Particle Swarm Optimization (PSO) algorithm to try and train a given

agent to find an optimal linear combination of the features to win against some designated foe (e.g. Defensive Heuristic 1 or Offensive Heuristic 1 based agents).

The idea of this approach was to model the problem as the following optimization problem:

$$\begin{aligned} \beta^* &= \arg\min_{\beta} -p(\beta) \\ \text{subject to } &-100 \leq \beta_k \leq 100 \ \ \forall k \\ \text{where } h(f) &= \sum_{k=1}^{9} \beta_k f_k \end{aligned}$$

This basically means the cost function was the negative of the probability of the desired agent winning, $-p(\cdot)$, through use of a parametrized heuristic function via the parameter vector β. The probability would then be estimated, given β, by playing a Monte Carlo sample of games against one particular foe and gradually improving the heuristic β weights via the PSO algorithm.

This method proved to find some interesting results, but it was not robust enough due to having to use lower search tree depths to make the run-time manageable. It is believed this could prove very useful if parallelized appropriately such that greater search depths could be used during the optimization procedure.

2.3 Matchup Results

Before going forward, keep in mind that all matchups were done over 20 games, so results in any tables are based on averages across the games. The visual of a final board, however, will just be one sample board at the end of one of those 20 games, each of which can be found in Appendix B. Note as well that the Alpha-Beta agent uses a depth of 4 while the Minimax agent uses a depth of 3.

2.3.1 8×8 Board with Nominal Rules

Tables 3, 4, 5, 6, 7, 8 refer to data for a set of matchups between different agents. Their associated final board configuration can be found in the Appendix. Note that X refers to Player 1 and O refers to Player 2. For a given matchup, you can frame it as Player 1 vs Player 2 to know which agent refers to which symbol.

In terms of things to comment on, it was particularly interesting to see how difficult it was to come up with an offensive heuristic function that could beat the baseline Defensive Heuristic. For whatever reason, it was a real challenge to find a heuristic that could outmatch the baseline Defensive Heuristic. Conversely, finding a strong Defensive Heuristic was simpler. Perhaps it was a lack of experience in Breakthrough by the author, but it almost appears as though coming about a good offensive strategy is harder than a defensive one for the game Breakthrough.

In terms of Heuristic function patterns, it appears that Defensive strategies perform the best in general. Additionally, it was interesting to see how many fewer nodes the custom heuristics looked at relative to the provided ones. It appears Alpha-Beta pruned a lot, but at the same time it seems feasible the pruning cut out potentially good paths to go down. It appears this is feasible due to the way the actions to try were ordered, which was based on the heuristic functions themselves.

As an additional note, it is obvious that noise in the heuristic function does cause a variation in the performance. This is interesting and is something that would be intriguing to explore in a theoretical manner.

Winner	Win Rate	Pieces captured by Minimax h_{po}	Pieces captured by Alpha-Beta h_{po}
Minimax h_{po}	85%	9.85	9.85

Agent	Total Nodes Expanded	Mean Nodes Expanded Per Move	Mean Move Time (ms)
Minimax h_{po}	423,560	12,038	4.69
Alpha-Beta h_{po}	594,402	17,317	24.78

Table 3: 8×8 Nominal — Minimax h_{po} vs. Alpha-Beta h_{po}

Winner	Win Rate	Pieces captured by Alpha-Beta $h_o^{(1)}$	Pieces captured by Alpha-Beta h_{pd}
Alpha-Beta h_{pd}	100%	4.15	6.55

Agent	Total Nodes Expanded	Mean Nodes Expanded Per Move	Mean Move Time (ms)
Alpha-Beta $h_o^{(1)}$	176,274	4,832	18.24
Alpha-Beta h_{pd}	526,551	14,594	23.17

Table 4: 8×8 Nominal — Alpha-Beta $h_o^{(1)}$ vs. Alpha-Beta h_{pd}

Winner	Win Rate	Pieces captured by Alpha-Beta $h_d^{(1)}$	Pieces captured by Alpha-Beta h_{po}
Alpha-Beta $h_d^{(1)}$	100%	12.35	3.6

Agent	Total Nodes Expanded	Mean Nodes Expanded Per Move	Mean Move Time (ms)
Alpha-Beta $h_d^{(1)}$	173,399	5,260	18.27
Alpha-Beta h_{po}	642,267	20,151	26.04

Table 5: 8×8 Nominal — Alpha-Beta $h_d^{(1)}$ vs. Alpha-Beta h_{po}

2.3.2 5×10 Board with Nominal Rules

In this variation of the game, it was interesting to see that the nodes expanded between the custom heuristics and the baseline got much closer, as Tables 10 and

Winner	Win Rate	Pieces captured by Alpha-Beta $h_o^{(1)}$	Pieces captured by Alpha-Beta h_{po}
Alpha-Beta h_{po}	80%	12.75	7.6

Agent	Total Nodes Expanded	Mean Nodes Expanded Per Move	Mean Move Time (ms)
Alpha-Beta $h_o^{(1)}$	124,289	2,666	10.22
Alpha-Beta h_{po}	576,628	12,471	17.29

Table 6: 8×8 Nominal — Alpha-Beta $h_o^{(1)}$ vs. Alpha-Beta h_{po}

Winner	Win Rate	Pieces captured by Alpha-Beta $h_d^{(1)}$	Pieces captured by Alpha-Beta h_{pd}
Alpha-Beta $h_d^{(1)}$	90%	8.65	5.6

Agent	Total Nodes Expanded	Mean Nodes Expanded Per Move	Mean Move Time (ms)
Alpha-Beta $h_d^{(1)}$	234,860	5,901	21.57
Alpha-Beta h_{pd}	578,163	114,889	22.79

Table 7: 8×8 Nominal — Alpha-Beta $h_d^{(1)}$ vs. Alpha-Beta h_{pd}

11 show. This shows that either pruning was reduced for the custom heuristics or increased for the baseline ones as the shape of the board changed. Table 12 also shows that for this different shaped board, the custom Defensive Heuristic did much better at dominating the custom Offensive Heuristic. The baseline Defensive Heuristic also continued to be a challenge to beat by the custom heuristics, as seen in Table 9.

2.3.3 8×8 Board with Modified Rules

Tables 13 and 14 cover some matchups for this Breakthrough variation. Some of the interesting things to make note of was the custom Defensive heuristic ended up not being as strong as in the previous Breakthrough variations. It was interesting to see that the custom Offensive heuristic also closed the gap a little better against the baseline Defensive heuristic in this variation, though still not winning against it overall.

Winner	Win Rate	Pieces captured by Alpha-Beta $h_o^{(1)}$	Pieces captured by Alpha-Beta $h_d^{(1)}$
Alpha-Beta $h_d^{(1)}$	65%	8.05	15.1

Agent	Total Nodes Expanded	Mean Nodes Expanded Per Move	Mean Move Time (ms)
Alpha-Beta $h_o^{(1)}$	146,576	3,187	11.01
Alpha-Beta $h_d^{(1)}$	174,551	3,822	13.53

Table 8: 8×8 Nominal — Alpha-Beta $h_o^{(1)}$ vs. Alpha-Beta $h_d^{(1)}$

Winner	Win Rate	Pieces captured by Alpha-Beta $h_o^{(2)}$	Pieces captured by Alpha-Beta h_{pd}
Alpha-Beta h_{pd}	100%	5.35	8.55

Agent	Total Nodes Expanded	Mean Nodes Expanded Per Move	Mean Move Time (ms)
Alpha-Beta $h_o^{(2)}$	77,380	4,209	19.13
Alpha-Beta h_{pd}	97,463	5,305	10.95

Table 9: 5×10 Nominal — Alpha-Beta $h_o^{(2)}$ vs. Alpha-Beta h_{pd}

Winner	Win Rate	Pieces captured by Alpha-Beta $h_d^{(2)}$	Pieces captured by Alpha-Beta h_{po}
Alpha-Beta $h_d^{(2)}$	95%	11.8	10.45

Agent	Total Nodes Expanded	Mean Nodes Expanded Per Move	Mean Move Time (ms)
Alpha-Beta $h_d^{(2)}$	115,719	5,352	26.30
Alpha-Beta h_{po}	141,405	6,854	15.95

Table 10: 5×10 Nominal — Alpha-Beta $h_d^{(2)}$ vs. Alpha-Beta h_{po}

Winner	Win Rate	Pieces captured by Alpha-Beta $h_d^{(2)}$	Pieces captured by Alpha-Beta h_{pd}
Alpha-Beta $h_d^{(2)}$	85%	7.45	8.65

Agent	Total Nodes Expanded	Mean Nodes Expanded Per Move	Mean Move Time (ms)
Alpha-Beta $h_d^{(2)}$	102,706	5540	35.12
Alpha-Beta h_{pd}	101,179	5850	15.81

Table 11: 5×10 Nominal — Alpha-Beta $h_d^{(2)}$ vs. Alpha-Beta h_{pd}

Winner	Win Rate	Pieces captured by Alpha-Beta $h_o^{(2)}$	Pieces captured by Alpha-Beta $h_d^{(2)}$
Alpha-Beta $h_d^{(2)}$	100%	9	12

Agent	Total Nodes Expanded	Mean Nodes Expanded Per Move	Mean Move Time (ms)
Alpha-Beta $h_o^{(2)}$	90,624	4558	27.27
Alpha-Beta $h_d^{(2)}$	78,623	3874	26.82

Table 12: 5×10 Nominal — Alpha-Beta $h_o^{(2)}$ vs. Alpha-Beta $h_d^{(2)}$

Winner	Win Rate	Pieces captured by Alpha-Beta $h_o^{(3)}$	Pieces captured by Alpha-Beta h_{pd}
Alpha-Beta h_{pd}	60%	6.1	11.3

Agent	Total Nodes Expanded	Mean Nodes Expanded Per Move	Mean Move Time (ms)
Alpha-Beta $h_o^{(3)}$	310,650	5,152	20.82
Alpha-Beta h_{pd}	695,549	11,597	22.26

Table 13: 8×8 Three Workers — Alpha-Beta $h_o^{(3)}$ vs. Alpha-Beta h_{pd}

Winner	Win Rate	Pieces captured by Alpha-Beta $h_d^{(3)}$	Pieces captured by Alpha-Beta h_{po}
Alpha-Beta $h_d^{(3)}$	70%	13.4	4.85

Agent	Total Nodes Expanded	Mean Nodes Expanded Per Move	Mean Move Time (ms)
Alpha-Beta $h_d^{(3)}$	256,314	5,274	16.93
Alpha-Beta h_{po}	731,261	15,264	20.86

Table 14: 8×8 Three Workers — Alpha-Beta $h_d^{(3)}$ vs. Alpha-Beta h_{po}

3 Contributions

- Christian Howard
  - Built a generic CSP Framework Skeleton
  - Built generic framework for agents playing games based on rules, applied to Breakthrough
  - Developed Particle Swarm Optimization code to learn optimal heuristics via Monte Carlo of games
  - Implemented Breakthrough and the variations asked of for students taking course for 4 credits
  - Did analysis for set of Breakthrough test problems
  - Wrote Breakthrough portion of report
- Luke Pitstick
  - Wrote most of the code for Part 1 of MP (CSP Flow Free)
  - Did analysis for set of test problems for CSP Flow Free
  - Wrote CSP Flow Free part of report

4 List of Potential Extra Credit

- Development of optimization algorithms for use in finding optimal heuristic functions

Appendices

A Puzzle Solutions for Part 1

```
B__RO
____Y_
__Y__
_RO_G
_BG__
```

Figure 1: 5×5 Input

```
BRRRO
BRYYO
BRYOO
BROOG
BBGGG
```

Figure 2: 5×5 Solution

```
____O____
_B__GY_
____BR__
____Y____
_________
__R_____
G____O__
```

Figure 3: 7×7 Input

```
GGGOOOO
GBGGGYO
GBBBRYO
GYYYRYO
GYRRRYO
GYRYYYO
GYYYOOO
```

Figure 4: 7×7 Solution

```
___R__G_
_BYP____
___O_GR_
___P____
_______Y_
_____BOQ_
_Q______
_________
```

Figure 5: 8×8 Input

```
YYYRRRGG
YBYPPRRG
YBOOPGRG
YBOPPGGG
YBOOOOYY
YBBBBOQY
YQQQQQQY
YYYYYYYY
```

Figure 6: 8×8 Solution

```
D__BOK___
__O__R___
__RQ__Q__
DB_______
_G_______
___P____G
__Y___Y__
_______KP_
__________
```

Figure 7: 9×9 Input

```
DBBBOKKKK
DBOOORRRK
DBRQQQQRK
DBRRRRRRK
GGKKKKKKK
GKKPPPPPG
GKYYYYYPG
GKKKKKKPG
GGGGGGGGG
```

Figure 8: 9×9 Solution

```
RG_________
_____O___O_
_YP_Q___Q_
__________
__G________
_______R___
_______B___
P__________
_Y_______B_
__________
```

Figure 9: 10×10 v1 Input

```
RGGGGGGGGG
RRRROOOOOG
YYPRQQQQQG
YPPRRRRRRG
YPGGBBBBRG
YPPGBRRBRG
YYPGBRBBRG
PYPGBRRRRG
PYPGBBBBBG
PPPGGGGGGG
```

Figure 10: 10×10 v1 Solution

```
__________
_B_________
__TPFBTV__
__________
__________P
F__________
__________
__SNHSNH__
_________V_
__________
```

Figure 11: 10×10 v2 Input

```
TTTPPPPPPP
TBTPFFFFFP
TBTPFBTVFP
TBBBBBTVFP
TTTTTTTVFP
FNNNNNNVFF
FNSSSSNVVF
FNSNHSNHVF
FNNNHHHHVF
FFFFFFFFFF
```

Figure 12: 10×10 v2 Solution

B Breakthrough Final Board Configuration

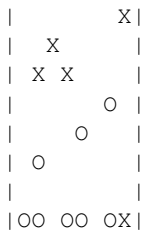

Figure 13: 8×8 Nominal — Minimax h_{po} vs. Alpha-Beta h_{po}

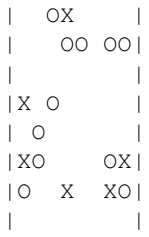

Figure 14: 8×8 Nominal — Alpha-Beta $h_o^{(1)}$ vs. Alpha-Beta h_{pd}

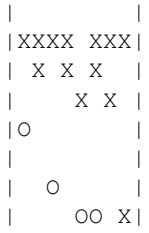

Figure 15: 8×8 Nominal — Alpha-Beta $h_d^{(1)}$ vs. Alpha-Beta h_{po}

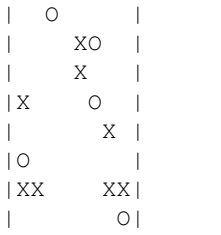

Figure 16: 8×8 Nominal — Alpha-Beta $h_o^{(1)}$ vs. Alpha-Beta h_{po}

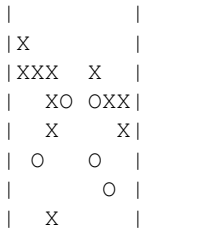

Figure 17: 8×8 Nominal — Alpha-Beta $h_d^{(1)}$ vs. Alpha-Beta h_{pd}

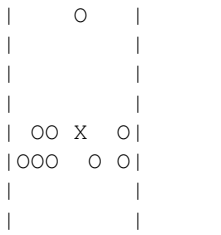

Figure 18: 8×8 Nominal — Alpha-Beta $h_o^{(1)}$ vs. Alpha-Beta $h_d^{(1)}$

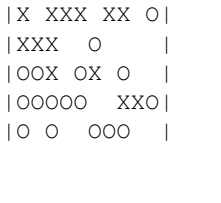

Figure 19: 5×10 Nominal — Alpha-Beta $h_o^{(2)}$ vs. Alpha-Beta h_{pd}

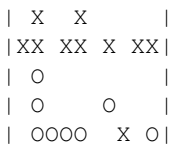

Figure 20: 5×10 Nominal — Alpha-Beta $h_d^{(2)}$ vs. Alpha-Beta h_{po}

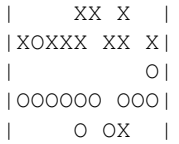

Figure 21: 5×10 Nominal — Alpha-Beta $h_d^{(2)}$ vs. Alpha-Beta h_{pd}

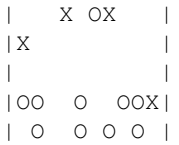

Figure 22: 5×10 Nominal — Alpha-Beta $h_o^{(2)}$ vs. Alpha-Beta $h_d^{(2)}$

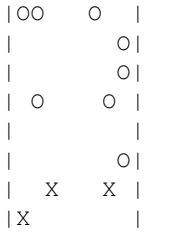

Figure 23: 8×8 Three Workers — Alpha-Beta $h_o^{(3)}$ vs. Alpha-Beta h_{pd}

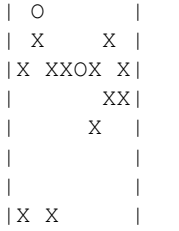

Figure 24: 8×8 Three Workers — Alpha-Beta $h_d^{(3)}$ vs. Alpha-Beta h_{po}

C Open Source Software Used

No open source code or packages were used in the development of the codes for this project outside of standard C++ libraries.