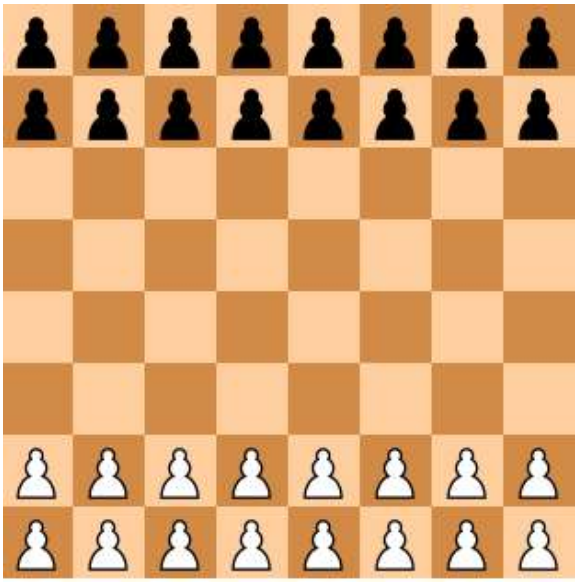


CS440/ECE448 Fall 2017

Assignment 2: Constraint Satisfaction Problems and Games

Deadline: Monday, October 30, 11:59:59PM

As on Assignment 1, you have the option of working in groups of up to three people. You are free to either stay with the same team or pick a new one, but as before, all the group members must be enrolled in the same section for the same number of credits (except for online students, who can work with Section Q students enrolled for the same numer of credits).

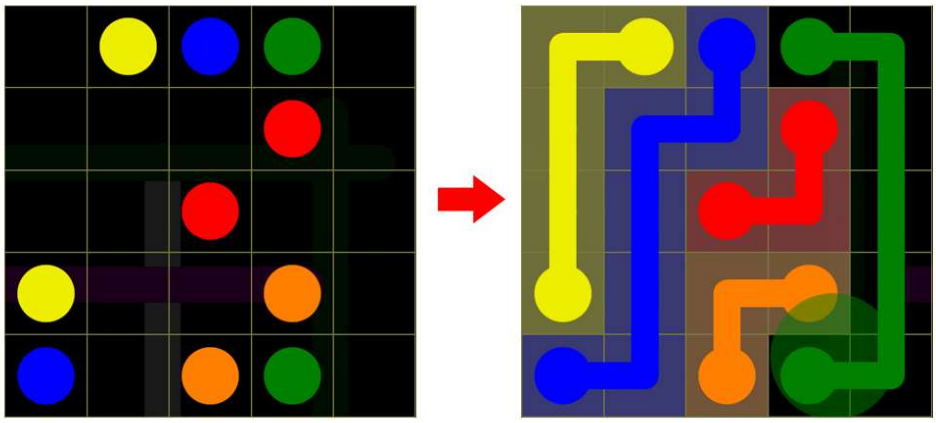
Contents

Part 1: CSP - Flow Free

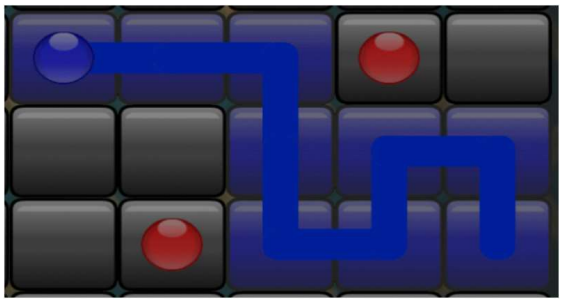
Created by Litian Ma and Jialu Li based on Flow_Free

Introduction and rules

d Flow Free is a puzzle game for iOS and Android released by Big Duck Games on June 2012. The goal for this game is to fill in colors on all empty grid cells to form "pipes" such that the colors are able to flow between two given sources. You will be given a puzzle with multiple color inputs. Each color has two sources on the grid, and you will need to draw a pipe of the same color to connect them. The pipes cannot intersect with each other, and a complete solution cannot contain any empty grid cell. For this assignment, each puzzle only has one unique solution. The following picture shows an example of the start state and goal state for one Flow Free instance.



The other constraint for this assignment is that no zigzag pattern is allowed. This means that for each non-source cell, its four-connected neighborhood (consisting of cells above, below, to the left, and to the right, if they exist) should have exactly two cells filled with the same color. For each source cell, its neighborhood should have exactly one cell filled with the same color. The following picture shows an example of the disallowed "zigzag" pattern.



First, you need to formulate Flow Free as a CSP. In your report, give your definitions of variables, domains, and constraints. Next, with these in mind, implement backtracking search. You should create two versions of your implementation: the "dumb" one with random variable and value ordering (and no forward checking), and the "smart" one that includes any variable and value ordering heuristics that make sense for your formulation (which would probably include forward checking). In the report, describe your "smart" implementation and give a brief explanation of why you chose to use your particular combination of heuristics and inference techniques.

You need to run both your solvers on several input puzzles. You will use the input and output formats illustrated by this sample 5*5 puzzle and the corresponding solution.

For each of the inputs below, please include in your report:

A. Your solution, i.e., the filled game board that satisfies all the constraints like the example above.
B. For both the "dumb" and the "smart" implementation, the number of attempted assignments and the execution time of the algorithms (in seconds or milliseconds). If your "dumb" implementation takes too long on a certain input, feel free to cut it off and say so in your report (you will not lose points).

1.1 Smaller inputs (for everybody)

In this part, you are required to solve the following three puzzles:

1. 7*7 puzzle
2. 8*8 puzzle
3. 9*9 puzzle

1.2 Bigger inputs (for four credits only)

In this part, you should improve your "smart" implementation with a better inference technique or constraint propagation technique (arc consistency). Then use both the "smart" and the new "smarter" implementation to solve the following bigger

puzzles. As in Part 1.1, report your solution, attempted assignments, and running time for both methods.

1. 10*10 puzzle
2. 10*10 puzzle

For bonus points

- Try to solve the following larger inputs: 12*12 puzzle, 12*14 puzzle, 14*14 puzzle.
- Create nice visualizations of your output (similar to the above graphics).

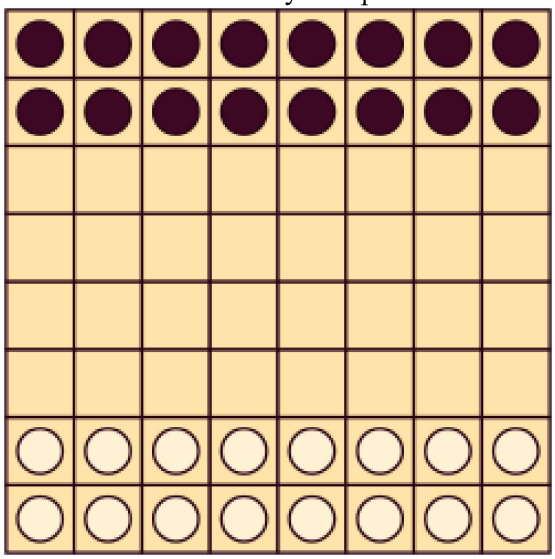
Part 2: Game of Breakthrough

Created by Shreya Rajpal, revised by Akshat Gupta

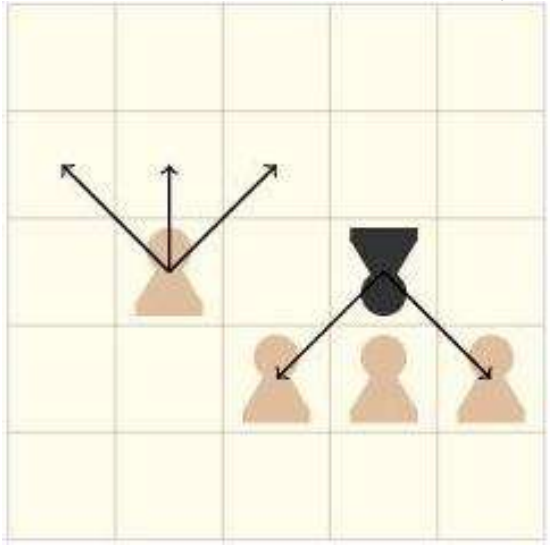
The goal of this part of the assignment is to implement an agent to play a simple 2-player zero-sum game called Breakthrough.

Rules of the game

- The 8x8 board is initially set up as shown below. Each player has 16 workers in their team.



- Players play alternating turns, and can only move one piece (of their own workers) at a time.
- In each turn, a worker can only move *one square* in the forward or diagonally-forward directions, as shown below. Moreover, a worker can 'capture' workers of the enemy team if they are placed diagonally forward from it, as shown in the illustration below. Note that if enemy workers are directly in front of the player, then a capture isn't possible.



- The game finishes when (a) a worker reaches the enemy team's home base (the last row); or (b) when all workers of the enemy team are captured.

For more information, check out Breakthrough's Wikipedia page.

2.1 Minimax and alpha-beta agents (for everybody)

Your task is to implement agents to play the above game, one using **minimax search** and one using **alpha-beta search** as well as two evaluation functions - one which is more **offensive** (i.e., more focused on moving forward and capturing enemy pieces), while the other which is more **defensive** (i.e., more focused on preventing the enemy from moving into your territory or capturing your pieces). The evaluation functions are used to return a value for a position when the depth limit of the search is reached. Try to determine the maximum depth to which it is feasible for you to do the search (for alpha-beta pruning, this depth should be larger than for minimax). The worst-case number of leaf nodes for a tree with a depth of three in this game is roughly 110,592, but in practice is usually between 25,000 - 35,000. Thus, you should at least be able to do minimax search to a depth of three.

We provide the following two dummy heuristics:

- **Defensive Heuristic 1**: The more pieces you have remaining, the higher your value is. The value will be computed according to the formula `2*(number_of_own_pieces_remaining) + random()`.

- **Offensive Heuristic 1**: The more pieces your opponent has remaining, the lower your value is. The value will be computed according to the formula `2*(30 - number_of_opponent_pieces_remaining) + random()`.

  NOTE: `random()` generates a random float uniformly in the semi-open range [0.0, 1.0) as per Python's random(). This noise is added to provide an easy way to break ties.

Your task for this part is:

- Implement minimax search for a search tree depth of 3.
- Implement alpha-beta search for a search tree of depth more than that of minimax.
- Implement **Defensive Heuristic 1** and **Offensive Heuristic 1**.
- Design and implement an **Offensive Heuristic 2** with the idea of beating **Defensive Heuristic 1**.
- Design and implement an **Defensive Heuristic 2** with the idea of beating **Offensive Heuristic 1**.

Then play the following matchups:

1. Minimax (Offensive Heuristic 1) vs Alpha-beta (Offensive Heuristic 1)
2. Alpha-beta (Offensive Heuristic 2) vs Alpha-beta (Defensive Heuristic 1)
3. Alpha-beta (Defensive Heuristic 2) vs Alpha-beta (Offensive Heuristic 1)
4. Alpha-beta (Offensive Heuristic 2) vs Alpha-beta (Offensive Heuristic 1)
5. Alpha-beta (Defensive Heuristic 2) vs Alpha-beta (Defensive Heuristic 1)
6. Alpha-beta (Offensive Heuristic 2) vs Alpha-beta (Defensive Heuristic 2)

For each of the above matchups, report the following:

A. The final state of the board (who owns each square) and the winning player.
B. The total number of game tree nodes expanded by each player in the course of the game.
C. The average number of nodes expanded per move and the average amount of time to make a move.
D. The number of opponent workers captured by each player, as well as the total number of moves required till the win.

If you have time, try to run each matchup multiple times to determine whether the noise in the evaluation function has any effect on the final outcome.

Finally, you should summarize any general trends or conclusions that you have observed. How does the type of evaluation function (offensive vs. defensive) affect the outcome? How do different combinations of evaluation functions do against each other?

Tips

- Pseudocode for alpha-beta pruning is given in Figure 5.7, p. 170, in the 3rd edition.
- For alpha-beta pruning, try to come up with a move ordering to increase the amount of pruning. Discuss any interesting choices in your report.
- For offensive vs. defensive evaluation functions, you may want to define the evaluation function as having two components: your score and your opponent's score. You would then weight these two components differently or combine them in different ways to get a more offensive or more defensive strategy. You can also read the Breakthrough Wikipedia page for possible features of a position to look for and prioritize for offensive vs. defensive play. Better yet, you should try playing the game yourself to get a better sense of what features or formations can occur.

2.2 Extended rules (for four-credit students)

Modify your implementation and evaluation functions to support the rule changes below and run several matchups for alpha-beta agents only with the maximum depth you can manage. You can choose any combination of offensive or defensive agents. Report outcomes of 2-4 representative matchups (or averages over several matchups of the same type), and summarize any interesting trends and differences from part 2.1.

1. **3 Workers to Base**: To win, 3 workers of a player's team must reach the opponent's base (as opposed to 1 in the original game). A player automatically loses when less than three of their pieces are left on the board. Therefore, the two ways to win the game would be to (a) move 3 pieces to the enemy's home base; (b) capture n-2 of the enemy's pieces, where n is the total number of players the enemy has at the beginning of the game.
2. **Long Rectangular Board**: For this variation, stick to the original, 1-worker-to-enemy-base win criterion. Instead, change the board shape to an oblong rectangle of dimensions 5x10.

For bonus points

- Design an interface for the game that would allow you to play against the computer. How well do you do compared to the AI? Does it depend on the depth of search, evaluation function, etc.?
- Implement any advanced techniques from class lectures or your own reading to try to improve efficiency or quality of gameplay.
- Implement a 1-depth greedy heuristic bot, and describe the differences between the gameplays of the greedy bot and the minimax bot.

Report Checklist

Your report should briefly describe your implemented solution and fully answer the questions for every part of the assignment. Your description should focus on the most "interesting" aspects of your solution, i.e., any non-obvious implementation choices and parameter settings, and what you have found to be especially important for getting good performance. Feel free to include pseudocode or figures if they are needed to clarify your approach. Your report should be self-contained and it should (ideally) make it possible for us to understand your solution without having to run your source code. For full credit, your report should include the following.

Part 1:

1. For everybody: Give your CSP formulation (variables, domains, and constraints) and briefly describe your "smart" backtracking search implementation. For each of the puzzles of size of 7*7, 8*8, and 9*9, report the solution and compare the number of attempted variable assignments and execution time for your "dumb" and "smart" implementations.
2. For four-credit students: Describe your "smarter" implementation. For each of the two 10*10 puzzles, report the solution and compare the number of attempted variable assignments and execution time for your "smart" and "smarter" implementations.

Part 2:

1. For everybody: Describe your implementation, especially your choice of Offensive and Defensive Heuristic 2. For matchups 1-6, report items A-D. Identify and discuss any general trends.

2. For four-credit students: For two types of extended rules in 2.2, describe your modified implementation (and especially evaluation functions) and report items A-D for matchups 2 and 3. Discuss any interesting trends that you observe.

Extra credit:

- We reserve the right to give **bonus points** for any advanced exploration or especially challenging or creative solutions that you implement. Three-unit students always get extra credit for submitting solutions to four-unit problems. **If you submit any work for bonus points, be sure it is clearly indicated in your report.**

Statement of individual contribution:

- All group reports need to include a brief summary of which group member was responsible for which parts of the solution and submitted material. We reserve the right to contact group members individually to verify this information.

WARNING: You will not get credit for any solutions that you have obtained, but not included in your report! For example, if your code prints out path cost and number of nodes expanded on each input, but you do not put down the actual numbers in your report, or if you include pictures/files of your output solutions in the zip file but not in your PDF. The only exception is animated paths (videos or animated gifs).

Submission Instructions

As for Assignment 1, **one designated person from the group** will need to submit on **Compass 2g** by the deadline. Three-unit students must upload under **Assignment 2 (three credits)** and four-unit students must upload under **Assignment 2 (four credits)**. Each submission must consist of the following two attachments:

1. A **report** in **PDF format**. As for Assignment 1, the report should briefly describe your implemented solution and fully answer all the questions posed above. **Remember: you will not get credit for any solutions you have obtained, but not included in the report.**

   As before, all group reports need to include a brief **statement of individual contribution**, i.e., which group member was responsible for which parts of the solution and submitted material.

   The name of the report file should be **lastname_firstname_assignment2.pdf**. Don't forget to include the names of all group members and the number of credit units at the top of the report.

2. Your **source code** compressed to a **single ZIP file**. The code should be well commented, and it should be easy to see the correspondence between what's in the code and what's in the report. You don't need to include executables or various supporting files (e.g., utility libraries) whose content is irrelevant to the assignment. If we find it necessary to run your code in order to evaluate your solution, we will get in touch with you.

   The name of the code archive should be **lastname_firstname_assignment2.zip**.

Multiple attempts will be allowed but only your last submission will be graded. **We reserve the right to take off points for not following directions.**

Late policy: For every day that your assignment is late, your score gets multiplied by 0.75. The penalty gets saturated after four days, that is, you can still get up to about 32% of the original points by turning in the assignment at all. If you have a compelling reason for not being able to submit the assignment on time and would like to make a special arrangement, you must send me email **at least four days before the due date** (any genuine emergency situations will be handled on an individual basis).

Be sure to also refer to course policies on academic integrity, etc.