Suarez, Trotter, Gunter
CSCI 523 Artificial Intelligence
Dr. Wilkins
Fall 2015

Conniption - System Description

Evaluation Functions

1) Sols: There is a specific set of winning positions across the board. Sols iterates over that set, identifying those solutions that have exactly one player's tile(s) inside of it. For each of these solutions, it count the number of tiles inside and assigns a weight to that solution accordingly. The weights are defined as 1 for one tile, 4 for two tiles, 16 for three tiles, and 1024 for four tiles. Winning states are assigned the highest priority. Each player's score is the sum of its weights. Player 2's score is always subtracted from Player 1's score.

2) Cells: Returns a raw summation of weights placed on each tile on the board. Each weight is calculated by the number of solutions possible from a single position on the board. For example the corner tiles each award a value of 3 since victory and only be achieved in three separate cases (once horizontally, once vertically, and once diagonally). The center positions ([4,3] and [4,4]) on the other hand are rated much more highly with a total of 13 possible solutions, which would return a value of 13.

3) Hybrid: Returns the sum of both the Sols and Cells evaluation functions. During the start of the game, Cells accounts for most of the total evaluation of the two functions. This is because of two reasons: first at the start of the game, the board is generally empty thus there are many chains of tiles which results in a lower return from Sols. Secondly, the value returned by Sols happens to be exponential which means that single tiles, or chains of two matter much less than chains of three or four, which allows for the value Cells to be more influential in the beginning of the games. One last observation is that during the middle to late game, Cells also serves as a tiebreaker for Sols, generally choosing the center most tiles when all else is equal in Sols.

4) Flip: When testing each of our evaluation functions, over time we noticed that flipping the board proved to be one of the most powerful mechanics in the game. Often times a player could flip aggressively whenever an advantage was present to quickly snowball to victory. However, we observed that our evaluation functions would greedily flip at any point it would improve the board state. Since flips are a finite yet powerful resource, we considered adding a cost to flips in order to encourage conservative play. The system state records every flip that is made and subtracts the overall value returned from the evaluation function by 10 for every flip made. This resulted in the program deciding to only flip either when it greatly increased the board state (to the point where victory would be assumed) or defensively to prevent the opponent from winning. As a result, we saw incredible improvements to the win percentages when Flip was applied to Player 2 against the previous evaluation functions. When applied to Player 1 Flip as a perfect win record with zero losses and zero ties. More detail of each matchup will be discussed in the results.

5) Random: Returns a random number from 0 to 10 as the evaluation function's value.

Architecture

The game program is separated into four primary modules: *resource*, *game*, *evaluation*, and *main*. The *resource* module is the foundation of the program, providing supporting data types for use in the game's main loop and in the AI's searching algorithms. The *game* module contains higher level classes implementing the algorithms and methods to correctly interact with game states. The *evaluation* module simply contains the various functions used by configurable AI, and the *main* module is the entry point of the application where the primary game loop is located. There are other modules alongside these, but they either serve as helper libraries or are for configuration.

The *resource* module contains the classes *Node*, *Move*, and *SystemState*; and these classes serve as the basis of the entire program. The *Node* class is an elementary data type used elsewhere for easier sorting of various objects. It maps a value to an item and comparisons are performed with respect to that value rather than the item. This is particularly useful for concisely and intuitively sorting *Move* objects according to an evaluation function. The *Move* class represents the three kinds of actions a player can take during a turn. These actions are represented by one of three keys 'none', 'flip', or 'place'. A *Move* object also stores an integer of 0 or 1 representing a *Move* taken by either Player 1 or Player 2 respectively, and it supports recording of a chosen column. If the action is 'place', this column variable is the index of a column in the board. Otherwise it is -1. Because a *Move* object only represents one portion of a ply, a single ply cannot be represented by a single *Move* object. All possible turns can be defined in terms of up to three appropriate *Move* objects, and our model always uses exactly three objects for this. Thus the minimax trees produced by this model have the same number of leaf nodes as a single-move model, but we can more easily verify legality and more easily accept user input.

The *SystemState* is a large class which encodes the relative position of tiles in the board, whether the board is flipped or not, the current player, the number of flips made by each player, and so on. The primary means of interacting with a *SystemState* object are through the methods *update()*, *validMove()*, *genMoves()*, and *isGoal()*. The *update()* method accepts a *Move* object and outputs a new *SystemState* object by applying that move. The new object is completely separated from its parent, so use of the *update()* method is immutable. This sacrifices some efficiency in exchange for stability when generating large minimax trees and renders inverse *Move* objects unnecessary. This method also updates the various attributes serving as counters and flags for the *SystemState*. This method also assumes that the provided move is valid. This validation is performed separately by the *isValid()* method. It too accepts a *Move* object and it uses the *SystemState* attributes to verify the Conniption rules. In a similar manner, *genMoves()* creates the complete list of currently valid *Move* objects.

The *isGoal()* method is used for identifying the end of the game and delegates this function to one of three helper functions depending on whether the previous *Move* object had an action of ‘place’, ‘flip’, or ‘none’. It returns a 2-tuple where the first element is a Boolean for whether the game is over or not, and the second element is the integer representing the winning player. If there is no winning player, this integer is 2. In the case of a ‘none’ *Move*, the method defaults to returning *(false, 2)* but can be told to verify it as a ‘flip’. In the case of a ‘flip’, it searches all possible solution locations in the board for a winning chain of tiles with memoization and a statically generate graph to prune overlapping non-winning locations. In the ‘place’ case, it only searches the neighborhood of the previously placed tile; and all cases check the number of placed tiles in case of a filled board with no winner.

When the *resource* module is loaded, it also runs a function that generates two globally accessible data structures, *SOLS_GRAPH* and *CELL_MAP*. These two structures store adjacencies between solution locations and cells. Specifically, *SOLS_GRAPH* is an undirected graph mapping solutions to sets of solutions using ordered tuples of cell locations. Thus, solutions which share a cell or a pair of cells are adjacent. This allows identification of a solution’s neighborhood in constant time. *CELL_MAP* is a dictionary mapping cells to sets of solution locations, allowing lookup of a cell’s neighborhood in constant time.

The *game* module builds on this with the *Game* and *Player* classes. The *Game* class is a mutable wrapper for the *SystemState*, maintains the mapping between a *Player* object and its index in a *SystemState*, and keeps the history of applied *Move* objects. It also supports functions for checking whether the game is over, getting the current *Player* object, and logging to either a pickle or text file.

The *Player* class is an abstract class that stores a name and features the *choose_move()* method. This method expects a *SystemState* object and outputs a *Move* object. It is implemented by the child classes *Human* and *AI*. The *Human* class uses the *SystemState* object to determine the correct prompt for the user and maps keyboard input to the appropriate *Move* object. If a *Move* object is not valid, it repeats the prompt until a valid one is specified.

The *AI* class is configurable with an evaluation function, a recursion depth in units of two-plies, and a selection function. The *AI* implementation of *choose_move()* delegates the task to a recursive minimax search. This search is parameterized by the root state and maximum. Given a state, it checks whether the depth limit has been reached or whether the state is a goal state. In these two cases, it uses the provided evaluation function to score the state and returns that score. Otherwise, it gets the list of valid *Move* objects, creates a new *SystemState* object with each, and recurses. When the recursion returns, it keeps track of the best choice for that player as well as the alpha and beta values for alpha-beta pruning. It returns the best score when all successors have been tested or when alpha surpasses beta. When the search is complete, it creates a *Node* object for each possible *Move* from the root state with the resulting score as each *Node* object’s value. The list of nodes is passed to the selection function.

All selection functions begin by sorting the *Node* list based upon whether the current player is minimizing or maximizing. It then pops the highest scoring *Move* objects. Selection functions differ in their handling of ties. The default selection function built into the *AI* class chooses a random *Move* from the tie list.

The evaluation and selection functions used to configure an *AI* object are in the *evaluation* module. Here we defined one alternative selection function which prioritizes *Move* objects based on their action. The ‘none’ type receives highest priority, and the ‘flip’ type is lowest. Our other evaluation functions have already been described, but all of them accept a *SystemState* object as their only parameter; and several of them make use of the *SOLS_GRAPH* and *CELL_MAP* structures to mitigate redundant calculations.

The entry point for the application is the *main* module where we can tweak the code manually to either run a debugging test, a more interactive playthrough, or a series of automated games. The latter two both support choosing to allow a user to play or to run an *AI* with a particular evaluation function. In these cases, the program enters a outermost loop which repeatedly plays games. Each game is a loop of prompting the players to choose a move, and it uses the current *Game* object to determine which player to prompt. At the end of a game, it calls the *Game* object’s *save()* method to append its data to a pickle file for use in data collection or recreation for debugging.

Personal Contributions

1) Gunter: I primarily focused on designing the architecture of the program and spent a lot of time helping implement the algorithms that used it. In particular, I made the decision to represent a ply with three moves and constructed the static data structures used in the *isGoal()* method and the evaluation functions. I also spent a lot of time pair programming with Armando to implement the minimax search.

2) Suarez: I primarily worked with Alex in building the architecture of the program. In particular, was in charge of developing the evaluation functions, helped in the implementation of those as well as applying the alpha-beta pruning to the minimax search. I was in charge of conducting the testing, analyzing the results and writing the report.

3) Trotter: I designed, coded, debugged the front end of the Conniption application. This included implementation how information was to be displayed to the user, streamlining user input through capturing raw character inputs from the console and displaying the output in a seamless state by clearing the terminal for each plie. Secondly, I also performed extensive playtesting, making observations on how to improve our evaluation functions. As a result I devised the heuristic for adding a cost for each Flip action in order to facilitate more conservative play which significantly improved its win percentage.