David Jones
16105441
AI

Backgammon Bearing Off Application

Major Classes

searchControl: This object carries out the building of the database. It takes a board and figures out the values recursively while filling the database when any state is resolved.
View: This class handles the user interface. Prompts, user input (boards and dice rolls), and data output (suggesting rolls and showing board stats) is handled through the view.
BoardGenerator: This is a collection of functions that together produce all possible boards that can proceed a given board or given board and dice roll. It uses the backgammon bearing off rules.
Main: This section makes use of BoardGenerator, searchControl, and View to carry out the program. It is the central control of the program.

Generating Boards Based on Current Board and Die Roll:
A set of unique boards is generated from any combination of board and die roll. The number on stones in each position is represented as ints in a vector. The positions are scanned to determine what state it is in:

1. The higher die and lower die can both be used fully.
2. Only the lower die can be used fully.
3. The higher die can be used fully, but not the lower die. (because if only one can be used fully, the rules state it must be the higher one)
4. Neither die can be used fully.

Each state makes use of various functions:

1. popStone: removes a stone from the highest occupied position
2. moveStone: increments and decrements positions to simulate one stone moving
3. genAll: makes a set of all boards that fully use a given die (single die)

genAll handles cases where a die can be fully used and popStone handles the case in which a die removes the highest position stone. popStone and moveStone return the result rather than altering the original, so they can be used iteratively to simulate all possible moves. The method for generating boards is done twice in the case of doubles, thereby simulating four moves. The method getBoards analyzes the board and makes use of the functions accordingly, eventually returning a set of vector<int>s to represent all boards accessible for a given die and board combination.

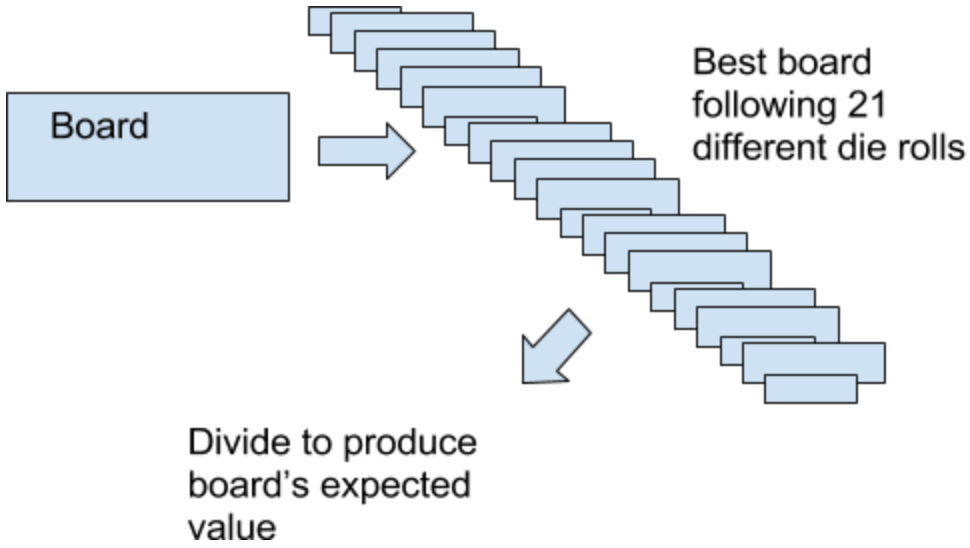
The Database:
The database is built using the unordered_map data structure. It resided in the searchControl class, as it is the searching that populates it. There are two types of entities, both stored as key-value pairs of strings:

1. Board:
    a. Key - a string of six characters representing the stones on each position.
    b. Value - a string in the format of “minimum maximum expected value” (e.g. “1 2 1.25”).

2. Board and Die Roll
    a. Key - a string of eight characters representing the stones on each position and the die roll being accounted for after (e.g. “00020012 is a board of 000200 and dice of 1 and 2).
    b. Value - a string in the format of “minimum maximum expected value suggested move” (e.g. “1 2 1.25 suggested move: 110000”).

Building the database:

The database is built in a depth-first pattern. A board (represented as a vector<int>) is analyzed by analyzing 21 future boards (each being the best move for every die roll). To get those 21 boards, the same method is called recursively. Eventually the search gets to a next board of 000000, which is seeded with values of all zero, and as the call stack rewinds, the boards for higher tiers can be assessed. The rewinding call stack is shown partially in the picture below (omitted are the 21 boards branching off from each of those boards used to gain the expected value of the 21 boards following the far left board):



Notice that some future boards are smaller. That illustrates the reduced probability of the rolls of double. The following are the rules and definitions are used to justify my implementation of expected value processing:

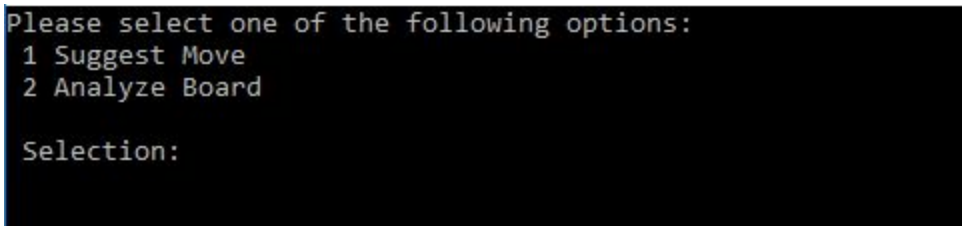
EV = expected value
Pr[V] = probability of V
V = expected value of a board that follows the current roll and board
EV = $Pr[V_1]*V_1 + Pr[V_2]*V_2 \ldots + Pr[V_n]*V_n$ where n is the number of possible values (21 in our case)
In this case, that EV calculated is the expected value following the turn. Therefore, the desired result is EV + 1.

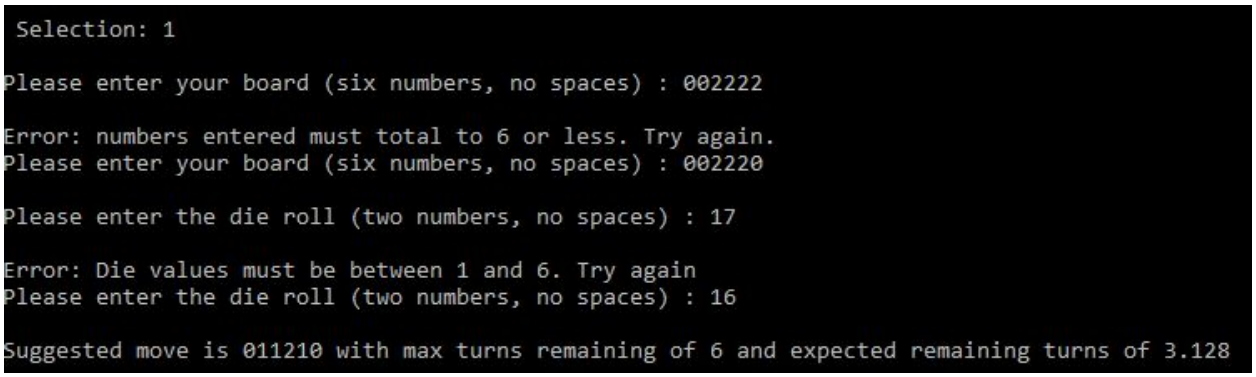
David Jones
16105441
AI

For any board, I am considering 21 possible values. These 21 values are the lowest expected value for boards following each die combination (i.e. the optimal choice). Doubles have a probability of 1/36 and other roles have a probability of 2/36. In my program, I simply doubled any values resulting from non-doubles and divided the sum by 36. Finally, the result is offset by one to reach account for the turn needed to get to any other board. The min and max are simply the smallest min and the largest max of those 21 boards, again increased by one.

Using the expected value process to populate the database:
All combinations of 0, 1, 2, 3, 4, 5, and 6 stones are generated iteratively as vector<int> objects and passed through the process explained above. Many times a board will have already been reached before, so the value is simply returned rather than searched. Despite the great number of boards examined briefly more than once, this process ensures that every single possible board is examined and recorded in the database. The variable MAXSTONES in main() sets the amount of stones to be find all combinations of (in this case, it is six).

User Interface:





David Jones
16105441
AI

```
 Selection: 2

Please enter your board (six numbers, no spaces) : 000222

For the board 000222 the following values are turns remaining not including the current turn :

Dice Values | Minimum Possible Turns | Maximum Possible Turns | Expected Number of Turns
-------------------------------------------------------------------------------------------
    11                  2                   9                  3.936
-------------------------------------------------------------------------------------------
    12                  2                   9                  4.132
-------------------------------------------------------------------------------------------
    13                  2                   9                  3.936
-------------------------------------------------------------------------------------------
    14                  2                   9                  3.832
-------------------------------------------------------------------------------------------
    15                  2                   8                   3.71
-------------------------------------------------------------------------------------------
    16                  2                   8                  3.596
-------------------------------------------------------------------------------------------
    22                  1                   8                  3.413
-------------------------------------------------------------------------------------------
    23                  2                   9                  3.832
-------------------------------------------------------------------------------------------
    24                  2                   8                   3.71
-------------------------------------------------------------------------------------------
    25                  2                   8                  3.619
-------------------------------------------------------------------------------------------
    26                  2                   8                  3.493
-------------------------------------------------------------------------------------------
```

```
-------------------------------------------------------------------------------------------
    26                  2                   8                  3.493
-------------------------------------------------------------------------------------------
    33                  1                   6                  2.986
-------------------------------------------------------------------------------------------
    34                  2                   8                  3.596
-------------------------------------------------------------------------------------------
    35                  2                   8                  3.493
-------------------------------------------------------------------------------------------
    36                  2                   7                  3.404
-------------------------------------------------------------------------------------------
    44                  1                   5                  2.648
-------------------------------------------------------------------------------------------
    45                  1                   7                  3.279
-------------------------------------------------------------------------------------------
    46                  1                   7                  3.188
-------------------------------------------------------------------------------------------
    55                  1                   4                   2.42
-------------------------------------------------------------------------------------------
    56                  1                   7                  3.069
-------------------------------------------------------------------------------------------
    66                  1                   3                  1.716
```

```
Would you like to continue using the program?
 1 for yes
 2 for no
Your choice:
```