Report on Ludo CS

Ranidu Prematilaka

September 1, 2024

1 Data Structures and Efficiency

1.1 Board Representation

The board is represented using an array of structures. Each structure within the array contains:

- A pointer to a piece or a block
- A piece count

This design decision prioritizes performance. While iterating through all pieces to calculate moves was possible, it introduced performance overhead. Thus taking up extra bit of memory for the sake of increasing performance felt understandable especially since memory constraints were not specified.

1.2 Piece Representation

Pieces are represented using a structure that includes the following fields:

- Pointer to the player
- Position
- Captures
- Approach passed count
- Index
- Mystery Effect and duration
- Name
- Location (base, standard board, home, home straight)
- Rotation

This structure facilitates easy access to piece attributes, simplifying computations. The pointer to it's player is a easy way of accessing the global attributes each player has.

1.3 Player Representation

Players are represented using a structure that includes the following fields:

- piece array
- home straight space
- pieces at Play
- pieces at base
- pieces at home
- index
- starting position
- approach position
- name
- which podium position

Each player has an array of four elements storing their piece structures, establishing a clear link between player and their respective pieces. It also includes global attributes applicable to all pieces are stored within the player structure to eliminate redundancy. Home straight is also implemented to each player since it's a unique space for each player and this is done using the same board structure as above to avoid memory wastage.

1.4 Block Representation

Blocks are represented using a structure that includes the following fields:

- pointer array for the pieces
- mystery effect
- name
- rotation

The block structure is used so that it only stores the unique attributes that only a block would hold and nothing more reducing memory wastage.

2 Program Efficiency

2.1 Modularity and Readability

The primary focus during development was to create a modular and redundancy-free program. This approach enhances code readability and simplifies future modifications. The game is structured into three main components which were worked on independently:

Game Logic

↓↑

Interface (mediates communication between game logic and bots/players)

↓↑

Bots

Even though this interface is an overhead for the given requirements it's much easier for development being able to divide and concur the program.

As an added upside to this is it enables to easily implement a way to implement manual play from a human player as well.

2.2 Error Handling

Due to the way of development where the game logic and bots/players are implemented together through the interface error handling is at minimal since the intermediary interface handles on which inputs are possible and bots/players take decision based on those.

2.3 Reduced Computation

The program always try to do the least amount of computation for a given process and once a computation is done its output is used throughout instead of recomputing almost acting like a cache.

This can be a balance between performance or memory usage since they seem to be inversely proportional in most cases. for a few of the function it is possible to introduce a way of caching so that the same computation wouldn't be executed but haven't implemented due to coming to the conclusion of the performance overhead is negligible.

2.4 Block Movement

Block movement all including eliminations are handled by the same functions as for single piece movements thus reducing repeated code and save memory

2.5 Block Implementation

Even though block implementation can be done with static memory this requires to have unused variables if no blocks exists wasting memory. due to this and the fact that creating a block has a low probability on average thus i choose to dynamically allocate memory for only the required block when needed reducing memory usage.

2.6 Mystery cell duration implementation

There was two options in mind of development for this feature,

- decreasing count every round from 4n to 4(n-1)(takes off effect)
- storing the round count when the effect occurred

i came to the conclusion of picking the second option due to the reasons. The first option for each round we have to iterate through every piece and decrement by 1 if an effect exists while the second only you only have to check weather the effect is still valid for the current round reducing the time complexity of the requirement dramatically. Thus even though it needs an extra variable to store the effect occurrence round it outweighs the benefits in comparison.

2.7 Home straight using a board structure

The way that the home straight is implemented is through using the same structure definition as the board. this can seem unnecessary due to the pieces not being able to make partial movements after entering the home straight. But The reason for my decision is that since creating a block on the home straight is deemed possible it's much more performance efficient to only check weather if the home straight cell has pieces and weather a block is needed to be created compared to iterating through the pieces to see weather a piece exits in the "about to land" cell. This also gives an added benefit of being able to use the same functions as used in the board movement for the home straight.

2.8 Time complexity of functions

Most functions are kept at a constant time complexity in the worst case scenario. Even though there are functions that either iterate through the some sort of array such as checking weather a block exists between start and end or if iterating through the players or its pieces, due to the behaviour of the game since the n (in O(n)) has a upper bound the time complexity to run the program is kept to a minimum.

2.9 Optimization and Future Improvements

While there is always room for further optimization, the current code is considered production-ready. Its modular nature provides a solid foundation for future enhancements and optimizations as needed.

3 Conclusion

This Ludo CS implementation prioritizes efficiency and maintainability through modular design, mindful data structures, and minimized computation. Strategic memory allocation and function design contribute to optimized performance and resource usage while being able to ensure that the program is adaptable to future requirements or fixes.