Monsters and Munchkins

Harris Bubalo, Jamie Lin,
Anna Yang, and Michael Huang

Abstract - Logic puzzles have garnered the attention of people of all ages due to their thought-provoking nature. In a typical game, players utilize a limited set of truths to answer a given question. Among them include the classic river crossing logic puzzle, "Missionaries and Cannibals". This article focuses on another variation of the game: "Monsters and Munchkins". We demonstrate the validity of our solution to this problem, implemented and modeled in ACL2s through a generalized algorithm and simulator.

Index terms - termination, validity

I. Introduction

The relationship between logic and games is prevalent in reasoning, argumentation, education, and many other fields of study. Games have been increasingly popular in academic settings. For example, they have been used professionally as semantics to infinitary languages, discovery techniques in logic papers, and as a tool to study the structures of debates [1]. Specifically, games model the relationships and structures that exist in society at large. Players can use their understanding of the game in order to further interpret the world around them This type of participation has great potential to spark ideas about new technologies. Not only do these games have a curricular quality, but the inherent challenge is also fun, which is essential in pushing a player to continuously engage with the game and the concepts that it is offering [2].

Puzzles, a single-player game genre, are popular for testing ingenuity and knowledge. Solvers are expected to follow a logical order to arrive at the correct solution for a puzzle. A player achieves the final goal by following a sequence of moves or steps from the starting point. These problems have only two possible outcomes - winning or losing. Although the problem can be solved in multiple ways, the goal's reachability or termination is the winning condition. In this paper, we will be focusing on a river crossing puzzle. It is a "transport" perfect-information game, where the objective is to carry items from one side of the river to the other. Usually, there are restrictions like the number of items that can be boarded, which items can simultaneously exist on both sides. Specifically, we will be looking at the "Monsters and Munchkins" variation of the river crossing puzzle.

The problem consists of moving all monsters and munchkins from one side of the river to the other with a boat of a set capacity. At least 1 creature has to be in the boat to move the boat to the opposite side of the river. The rule that the number of monsters cannot exceed the number of existing munchkins at any side or in the boat, unless there are no munchkins, is in effect at all stages of the game. The game can be generalized by changing the number of monsters and munchkins in the starting state. This paper explains that the problem is not generally solvable if the boat's capacity remains at 2. Increasing

the capacity to 3 provides solutions for up to 5 pairs. Games with a boat capacity of 4 or more are solvable for any number of pairs. For our purposes then, we would like to show in ACL2s that, for any boat capacity ≥ 4, the game is solvable for any number of pairs. Our tests will allow us to have reason to believe that our algorithm to solve the Monsters and Munchkins problem works. These tests entail that if there is a solution, we can compute it and return steps to reach the goal. If there is no solution, our algorithm will throw an error.

A. Contributions

It is important to note that the "Monsters and Munchkins" problem and its variations are not unsolved problems. Proofs of varying kinds pertaining to the solvability of certain starting conditions already exist. Students from Harvey Mudd and Pomona College wrote a variation of the algorithm for the "Missionaries and Cannibals" problem in their paper, *Graphical Solution of Difficult Crossing Problems*. Their graphical approach utilizes coordinates on a graph to specify different game states, while our algorithm uses ACL2s and a simple pattern to return a working output. They also examine various different solutions by drawing lines between the transition of states on their graph to reach a solution, while ours only considers one possible path that we have calculated. The students concluded that a boat capacity of 4 allowed all numbers of missionary and cannibal pairs to successfully be moved to the other side. Because of these conclusions, our algorithm considers a boat with a capacity of 4 [3].

Keeping these studies in mind, we model an algorithm that solves the "Monsters and Munchkins" variation in ACL2s. Using both trial and error and deductive reasoning, we drew out patterns that helped us develop our algorithm. We described the conditions of the games in input and output contracts, as well as wrote them explicitly within certain functions. Lastly, our main metric for success will be to develop properties for a final game state and utilize ACL2's *test?* to see if our algorithm produces correct solutions that satisfy such properties (these properties are explained in Section IV). Through our work of modeling the puzzle, developing a generalized algorithm, and mechanically demonstrating its correctness in ACL2s, we hope to add to the progress in modeling more complex games and puzzles in similar environments.

B. Road-Map

We organize the paper into the following sections: Section II elaborates on the tools we will use and other background information on our topic. Section III goes more into depth about the Monsters and Munchkin game and other similar problems. Section IV lays out the process in writing out our traditional algorithm. Section V generalizes our traditional algorithm and discusses broadening the scope of our problem. Section VI focuses on our findings and results. Section VII outlines possible additions to our work and progress with other variations of the problem. Section VIII is our conclusion.

II. Background

A. *Model and Notation*

We use ACL2s for the notation of our algorithm. We represent monsters and munchkins as natural numbers, and the *side* that the boat is either the symbol ‘left or ‘right.

The data definition *count* is a list of two elements, where the first element is the number of monsters and the second is the number of munchkins. This data type is used alongside *side* to represent the number of monsters and munchkins currently on a specific side of the river.

We indicate the types of game moves with the data definitions, *move-left* and *move-right*, which are lists of a sequence of symbols and quantitative values to represent the direction of the boat and the number of monsters and munchkins being moved. These moves are generalized in another data definition called *move*. Furthermore, these single moves are organized in a list of moves, with the data definitions *lom-start-left*, *lom-start-right*, and *lom* to accommodate these lists of moves. *Lom-start-left* is a move that starts from the left side of the river and moves monsters, munchkins, and the boat to the right. *Lom-start-right* is a move that starts from the right side of the river and moves monsters, munchkins, and the boat to the left. *Lom* is either a *lom-start-left* or *lom-start-right*.

We created a function called *move* to convey a step to be taken in our algorithm. This function prints out how many monsters and munchkins to move to which side of the river in list form.

Later, we refer to the type *state*, which is a list of two *count* values, a *side* to represent the game state, and a boolean error flag. The quantities of monsters and munchkins are represented on the left in the first *count* value, and the right quantities are represented in the second *count* value. The side in which the boat is on is the *side* value.

Lastly, the boolean indicates whether or not the current *state* has been reached by an invalid move. This field is crucial when checking for correctness. The boolean flag is used to show that not only do we can have a valid starting state and a valid end state, our algorithm ensures that all intermediate steps and game states are also valid. An invalid move refers to when the boat takes a monster/munchkin capacity of over 4 across the river, the boat takes more monsters/munchkins than are available, there are no monsters and munchkins on the boat to move it, and if the boat does not alternate river banks. When invalid moves occur, the boolean flag is changed from its initial false state to true and remains true until the end of the solution. This means that even if there are valid counts of monsters and munchkins and the boat is on the right side in the end state, the solution is not valid if the boolean flag is true. We will be using all these tools as well as additional ACL2s contracts to represent our algorithm.

B. *The Game Structure*

To represent our game, we will have 3 starting variables - the *count* of monsters and munchkins on the left side of the river, the *side* the boat is on, and the *count* of monsters and munchkins on the right side. Each state of our game will keep track of these variables. Based on

these variables, the algorithm then prints a list of instructions of each following game state that leads to the correct solution.

C. *Goals*

A formula is valid if it is true for all values of its terms. In our work, we will be showing through tests that our algorithm is valid for all scenarios. This means that given a valid starting state, a valid end state can be reached through printed instructions where all intermediate states also abide by all game rules. Valid states and properties are explained in Section IV.

To reasonably show validity and deem our algorithm a success, we will write our algorithm with contracts that set the invariants of the game at the beginning, develop properties that describe correct game states and solutions, and use ACL2s' *test?* to show—through thousands of examples—that our algorithm produces solutions that abide by these properties. Because of these invariants, our algorithm automatically rejects cases that are unsatisfiable.

III. The "Monster and Munchkins" Game

The "Monsters and Munchkins" game is another variation of the classic river crossing logic puzzle, "Missionaries and Cannibals", where the goal of the game is to move all of the missionaries and cannibals from one side of the river to the other. The earliest river-crossing puzzle is found in the manuscript "Propositiones ad Acuendos Juvenes", translated in English as "problems to sharpen the young". This manuscript contains 3 river-crossing problems, which include the "Fox, Goose, and Bag of Beans" puzzle, the "Jealous Husbands" problem, and the "Bridge and Torch" problem. The "Jealous Husbands" variation is the problem that the "Missionaries and Cannibals" puzzle" is modeled after.

The starting conditions of the game restrict all of the monsters and munchkins that begin on the left side of the river. The number of monsters and munchkins must be equal, and at no time or location should the number of monsters exceed the number of munchkins. To win, all monsters and munchkins must be on the right side of the river.

This paper demonstrates the process we took to show that our algorithm is valid. Validity is achieved when a working initial state and the output of our algorithm results in an acceptable end state. We created a simulator function that, when run under *test?,* returns the appropriate end state when a viable list of moves and a valid start state is inputted. This enables us to demonstrate that the algorithm terminates and does exactly what we intended it to do.

IV. Methodology of Traditional Algorithm

We began by drafting our possible steps on paper. Eventually, we discovered a repeated pattern in our solutions that could be used to solve the problem for any number of pairs. Moves are represented with the following notation: (move *numMonsters numMunchkins direction*). According to our algorithm, correct

solutions should alternate between (move 2 2 'right) and (move 1 1 'left), until there are 4 munchkins remaining on the left, after which the last moves should be:

```
(list (move 0 4 'right)
      (move 1 0 'left)
      (move 4 0 'right)
      (move 1 0 'left)
      (move 2 0 'right))
```

We proceeded to write out the pseudocode, and draw out a visual representation of what the steps looked like.

For cases with less than or equal to 4 monsters and 4 munchkins in the initial state, we found that the pattern we discovered for quantities greater than 4 was not applicable. We decided to hard code these specific cases: For 2 pairs of monsters and munchkins or less, we could move all beings across in a single move. For initial states with 3 or 4 pairs of monsters and munchkins, we could also achieve termination without the last 5 moves stated above. Cases with more than 4 pairs were delegated to a helper function that applied the pattern recursively to solve the problem.

During our initial coding process, our helper algorithm would not be admitted by ACL2s. Confirming that the code worked in DrRacket, we continued debugging the ACL2s code. In our initial implementation of *alg-help*, we had the function accept game conditions where the boat was on either the left or the right side. We eventually found that doing this made ACL2s spend a lot of time "expanding abbreviations", and because of this, the function failed to admit. To make the function simpler and have ACL2s consider fewer cases, we altered the algorithm to handle only the cases where the boat was on the left. We accomplished this by hard-coding right side moves, thus allowing the function to have a more specific contract and to finally terminate properly.

We also wanted to ensure that the lists of moves produced by our algorithm were consistent, so we defined another property that made certain the number of moves produced followed a $(5+2(m-4))$. The 5 describes the hard-coded last moves when there are 4 munchkins remaining on the left, the 2 is demonstrative of the recursive call between (move 2 2 'right) and (move 1 1 'left), and the minus 4 accounts for the hard-coded ending steps. We expressed this in a function called *is-valid-lom*.

We represented the conditions for the starting and end states in the functions *is-valid-start* and *is-valid-end*. For a starting state to be acceptable, the boat and all of the monsters and munchkins must begin on the left side of the river, and the number of monsters and munchkins must be equal. We also restricted the starting state to have an equal amount of monsters and munchkins on the right side of the river, set the number of monsters and munchkins on the left side to be greater than 4, and ensured that the error flag would not be activated. For the valid end state, the boat must be on the right, there must be zero monsters and munchkins on the left, there must be equal numbers of monsters and munchkins on the right, and the error flag must still be *nil*.

The final step was to implement a simulator, *simulate,* that when given a starting state and a list of steps would

return a valid end state. We created a *simulate-move* function that would only simulate a valid move, and called on it recursively in the main simulator function.

We created two *test?* expressions that test the previously described properties, and thus would show through thousands of examples that our algorithm works properly. The first *test?* checks if, given a valid start state, is the solution produced by running *alg-help* consistent with *is-valid-lom* and the expected game length. The second *test?* checks if, given a valid start state, does simulating the moves produced by *alg-help* on the state bring us to a valid end state. These *test?* expressions could fall under one *test?*, but we chose to distinguish checking solution list properties and checking end state properties.

V. Generalization of Traditional Algorithm

Our traditional algorithm considers states where monsters and munchkins start on the left side of the river and the boat capacity is 4. A boat capacity of 4 is chosen because lesser capacities do not fully solve the problem. A capacity of 1 trivially does not work, as no net change could occur (1 monster or munchkin is required to drive the boat). For a capacity of 2, up to 3 pairs can cross [4]. For a capacity of 3, up to 5 pairs can cross [4]. A boat capacity of 4 is chosen because for all number pairs of monsters and munchkins, a complete crossing is possible [3].

We have also considered cases where there are no restrictions on the number of monsters and munchkins in the starting position (i.e. not only in pairs). A start state with more monsters than munchkins has no successful end state, so we reject it automatically. When there are more munchkins than monsters, the solution is trivial because we can treat this problem as if there are an equal number of monsters and munchkins. After following the existing algorithm, all remaining munchkins on the left riverbank can then be brought to the right riverbank, making the problem solvable.

Additionally, if we increase the boat capacity beyond 4, there is guaranteed to be a possible solution. Even if we increase the boat capacity, we can treat the problem as if the boat capacity is 4 and follow the existing algorithm. This would produce a working solution that satisfies all conditions.

VI. Experimental Results and Metrics

We were able to reasonably show the validity of the traditional problem algorithm. By encoding our algorithm in ACL2s, proving termination of the function, and showing many examples of correctness with *test?* expressions and our *simulate* function, we have successfully completed this part of our project. Through these *test?* expressions, we demonstrated that our generated list of moves reached a proper end state in a predictable number of moves. According to our algorithm's logic, it should output a list of moves equal to $(5+2(m-4))$, where m is the number of pairs. Our first *test?* reasonably shows this property of length and pattern of moves through 2500 correct examples and 0 counterexamples. In our second *test?*, we used our simulator to execute the list of moves generated by

our algorithm and reasonably show—through 6000 correct examples and 0 counterexamples—that the end state is valid.

Due to difficulties with ACL2s contracts, we used defun rather than definec to allow our *simulate* function to work with ACL2s. However, the *simulate* function and both *test?* expressions work as expected. Thus, we are confident that despite the lack of explicit contracts, our algorithm works as intended.

Our results and reasoning show that we have found that it is possible to solve the monsters and munchkins problem for any number of starting monsters and munchkins, given a boat capacity of 4 or more.

VII. Further Progress

Future steps include providing a more expanded and explicit mechanical proof of correctness. Due to difficulties with ACL2s, defining our *simulate* function with contracts was not possible; so, adding such contracts would add more confidence to our algorithm's validity. Another step for further progress would be to consider additional variables in the problem. Our results generalize only within the scope of the above conditions of the monsters and munchkins game. We did not allow for factors such as having 3 variables to move across the river like in the wolf, lamb, and cabbage problem or having islands in the river crossing.

Another consideration is that our algorithm only provides one possible solution. Alternative crossings, or determining the optimal crossing, is not in the scope of our algorithm and project as a whole.

VIII. Conclusion

This paper analyzes a variation of a river-crossing problem: the “Monsters and Munchkins” game. This includes creating an algorithm in ACL2s and a simulator.

Our general algorithm works with the specified invariants and sufficiently determines that for any number of pairs greater than 4, a crossing is possible. However, it does not expand on other arbitrary starting states. It also has hard-coded sections to deal with individual cases, but we acknowledge that there are other possible solutions to this river-crossing logic puzzle. At this time we can make no provable judgements on whether our solution is optimal.

Overall, our algorithm and the surrounding work hopefully do contribute to the progress of understanding games and puzzles to a higher, more formalized degree. We learned that there are many ways to model complex systems and that there are patterns that can be deduced from similar game states. Our implementation of “Monsters and Munchkins” provides a mechanical model in ACL2s that demonstrates that our algorithm does terminate in a representation that others can comprehend. While our project is relatively limited in scope, our work can help to expand the modeling, understanding, and mechanical proving of other problems. Through this game model, we hope to add to the understanding of pattern deduction and reasoning in both games and other fields.

VIII. References

[1] B. B. Marklund, P. Backlund and H. Engstrom, "The Practicalities of Educational Games: Challenges of Taking Games into Formal Educational Settings," 2014 6th International Conference on Games and Virtual Worlds for Serious Applications (VS-GAMES), Valletta, Malta, 2014, pp. 1-8, doi: 10.1109/VS-Games.2014.7012170.

[2] Hodges, Wilfrid and Jouko Väänänen, "Logic and Games", *The Stanford Encyclopedia of Philosophy* (Fall 2019 Edition), Edward N. Zalta (ed.), URL = <https://plato.stanford.edu/archives/fall2019/entries/logic-games/>.

[3] Fraley, Robert, et al. “Graphical Solution of Difficult Crossing Puzzles.” Mathematics Magazine, vol. 39, no. 3, 1966, pp. 151–157. JSTOR, www.jstor.org/stable/2689307. Accessed 18 Apr. 2021.

[4] Franci, Raffaella (2002). "Jealous Husbands Crossing the River: A Problem from Alcuin to Tartaglia". In Dold-Samplonius, Yvonne; Dauben, Joseph W.; Folkerts, Menso; van Dalen, Benno (eds.). From China to Paris: 2000 Years Transmission of Mathematical Ideas. Stuttgart: Franz Steiner Verla. pp. 289–306. ISBN 3-515-08223-9.

Link to Project Repository:
https://github.com/mihalehang/monsters-munchkins