Mini-Project 1: Sheep & Wolves Agent

Burak Aksu

baksu3@gatech.edu

***Abstract*—This report describes an intelligent agent that solves the Sheep and Wolves problem for any number of starting animals. I built the agent using a breadth-first search algorithm that guarantees finding the shortest solution when one exists. The implementation successfully handles all test cases and correctly identifies unsolvable configurations. My approach prioritizes both finding correct solutions and ensuring they use the minimum number of moves.**

1 How The

1.1 Core Algorithm Design

My agent uses breadth-first search because it guarantees finding the optimal solution with the fewest moves. This was crucial since the grading rewards both correctness and optimality. The agent represents each state as a tuple containing the number of sheep and wolves on the left side, plus which side the boat is currently on.

The search process works by maintaining a queue of states to explore and systematically trying all possible moves from each state. I represent states as (sheep_left, wolf_left, boat_side) where boat_side indicates whether the boat is on the left (0) or right (1) side of the river. This gives me complete information about the current situation since I can calculate the right side by subtracting from the totals.

1.2 State Generation Process

From any current state, my agent considers five possible moves: taking one sheep, one wolf, both, two sheep, or two wolves. The boat must carry between 1-2 animals, so empty trips aren't allowed. For each potential move, the agent checks if enough animals are available on the current side and whether the resulting state would be valid.

The validation function implements the core constraint that wolves cannot outnumber sheep on either side unless no sheep are present. I check both sides of the river after each potential move. If sheep exist on a side but are outnumbered by wolves, that state gets rejected immediately.

1.3 Search Strategy

I chose breadth-first search over depth-first search because BFS explores states level by level, guaranteeing that the first solution found uses the minimum number of moves. This level-by-level exploration means shorter paths are always discovered before longer ones.

To prevent infinite loops and improve efficiency, I maintain a set of previously visited states. When the agent encounters a state it has seen before, it skips exploring that path since it already found a shorter route to that configuration. I also keep track of parent relationships so I can reconstruct the solution path by working backwards from the goal state.

2 Performance Analysis

2.1 Correctness and Robustness

My agent performed well across all test scenarios in Gradescope. It successfully solved standard cases like (3,3) and handled more challenging configurations

with unequal numbers of sheep and wolves. When no solution exists, the agent correctly returns an empty list rather than getting stuck in an infinite search.

The breadth-first approach ensures completeness, meaning if any solution exists, my agent will find it. The systematic exploration of all possible moves from each state means no valid paths get missed. This thoroughness was essential for handling edge cases that might trip up more heuristic approaches.

2.2 Efficiency Characteristics

For small to medium problems (up to about 10-15 animals), my agent solves very quickly, usually within milliseconds. The algorithm's time complexity grows exponentially with problem size, but this matches the inherent difficulty of the problem space rather than reflecting inefficient implementation.

The space complexity comes from storing the queue of states to explore and the set of visited states. In practice, this remains manageable for all realistic problem sizes since most solvable configurations have relatively short optimal solutions.

2.3 Challenging Cases

Balanced configurations where sheep and wolves start in equal numbers tend to require more complex solution sequences. These scenarios need careful coordination to maintain valid states throughout the crossing process. My agent handled these well because the systematic search doesn't rely on shortcuts that might miss complex but necessary move sequences.

Problems with significantly more sheep than wolves often have simpler individual moves but require more total moves due to the larger number of animals. The BFS approach naturally finds efficient solutions for these cases since it always explores shorter sequences first.

3 Algorithmic Design Decisions

3.1 State Representation Choices

I chose tuple representation for states because it provides fast hashing for the visited set while remaining easy to work with. The compact `(sheep_left, wolf_left, boat_side)` format contains all necessary information without redundancy. Python's built-in tuple hashing makes lookup operations very efficient.

3.2 Memory Management

Using a deque for the BFS queue provides optimal performance for the frequent enqueue and dequeue operations. Sets for visited state tracking offer constant-time membership testing, which becomes important as the search space grows. These data structure choices keep memory usage reasonable while maintaining fast operations.

3.3 Solution Reconstruction

My parent mapping approach allows efficient solution reconstruction without storing complete paths during search. Each state remembers which state and move led to it, letting me build the solution by following this chain backwards. This saves significant memory compared to maintaining full paths for every explored state.

4 Algorithmic Design Decisions

4.1 Strategic Similarities

Human solvers typically use similar high-level strategies like ensuring no invalid states occur and working systematically toward the goal. Both humans and my

agent tend to think about balancing the animals on each side and planning several moves ahead.

4.2 Computational Advantages

My agent surpasses human capabilities in several key areas. It never forgets previously explored states, while humans often revisit the same configurations without realizing it. The agent also guarantees finding the shortest solution, whereas humans might stop at the first working solution they discover.

The systematic exploration ensures no possible moves get overlooked due to cognitive limitations. Humans sometimes miss non-obvious move sequences that turn out to be part of the optimal solution.

4.3 Human Intuition vs Systematic Search

Humans often rely on pattern recognition and intuitive leaps that can be faster for simple cases but less reliable for complex scenarios. They might quickly identify promising directions but struggle with the detailed verification needed to ensure every intermediate state remains valid.

My algorithmic approach trades human-style intuition for computational thoroughness. While this might seem slower for obvious cases, it provides reliability and optimality guarantees that human solving cannot match consistently.

5 Algorithmic Design Decisions

5.1 Algorithm Selection Rationale

I selected breadth-first search specifically because the scoring system rewards optimal solutions. Other approaches like depth-first search or heuristic methods might find solutions faster but could miss the shortest path. Given that half the

grade depended on optimality, BFS was the natural choice despite its higher memory requirements.

5.2 Implementation Insights

The most challenging aspect was correctly implementing the state validation logic. Initially, I had bugs where the agent would incorrectly reject valid states or accept invalid ones. Careful testing and debugging of the constraint checking proved essential for getting reliable results.

I also learned that the choice of data structures significantly impacts performance. My initial version used lists for some operations that sets handle much more efficiently. Switching to appropriate data structures improved runtime noticeably.

5.3 Design Trade-offs

The main trade-off in my design is space complexity versus optimality guarantees. A depth-first approach would use less memory but might not find the shortest solution. A heuristic search could be faster but might miss optimal paths. I decided that guaranteeing optimality was worth the additional memory usage for this problem.

6 CONCLUSION

My sheep and wolves agent successfully combines theoretical computer science principles with practical problem-solving requirements. The breadth-first search approach proved effective for this domain, providing the optimal balance of correctness, optimality, and efficiency within the expected problem constraints.

The systematic approach demonstrates how algorithmic thinking can solve complex constraint satisfaction problems reliably. While the exponential search space limits scalability to very large instances, the agent performs excellently

within reasonable bounds and provides a solid foundation for understanding state-space search methods.

The project reinforced the importance of careful algorithm selection based on problem requirements. The scoring system's emphasis on optimality made BFS the clear choice, even though other algorithms might have different advantages in different contexts. This experience highlighted how problem constraints should drive algorithmic decisions rather than defaulting to familiar approaches.