Phase 1 Update - **Hardware Optimization of Sokoban Solver**

Benjamin Huang, Yen-shi Wang, and Zachary Armendariz

1. Project Update

1.1. Algorithm Description

The algorithm is a Sokoban Puzzle Solver (SPS). We are using Rolling Stone as a reference (for the algorithm and the baseline code). On a high level, the algorithm searches the state tree of the game, attempting to find a path from the start state to the end state. The overall search uses iterative deepening A* search to traverse the tree. Because the search space of Sokoban is so large, a number of additional techniques have to be applied to the problem as well. The main techniques we will be focusing on in our project are as follows:

- Collapsing player movement: although in the game itself the movement of the player might be considered as transitions into different game states, player movement is reversible. As such, two game states differing in only the player's position would be collapsed into a single game state for the solver if the player is able to move between the two positions without moving any stones.
- Heuristics: The algorithm uses a heuristic to estimate the distance from the goal state. It decides, based on this estimator, the depth to search the current sub-tree before moving on to another state sub-tree.

Most of the features of the game state are constant throughout the level (the bounds, goal squares). The only variables are the positions of the stones and the position of the player (with the optimization described above). As such, the way game states are stored in Rolling Stone is as a bitvector, with each bit representing a tile on the level and the bit value indicating whether a stone is present or not.

1.2. Performance Peak

Profiling was carried out on the baseline algorithm (solved 46 of the 90 problems) to determine which parts of the algorithm were the bottleneck. `gprof` was used to determine the time the algorithm spent in each function. The functions in which most of the computation took place varied from level to level, but some functions appeared to take a lot of time in many of the levels. We took the five functions from each level that took up the most time as the bottlenecks for that level, and found the bottlenecks common to the most levels. We also determined the total time the program spent in those functions over the course of solving all 46 levels.

Name of function	Number of Levels	Total time (sec)
`mark.c::MarkReach`	44	44.04
`moves.c::Moves`	43	52.274
`conflicts.c::GetPenalty`	27	44.23
`mark.c::UnReach`	21	9.07
`macro.c::MarkReachGRoom`	20	5.39
`weights.c::GetOptDist`	17	0.67
`bitstring.c::AllBitsSetBS`	16	10.65
`moves.c::DistToGoal`	10	1.19

Several of the functions, including `Moves`, `MarkReach`, `UnReach`, `MarkReachGRoom` are all variations of a flood-filling algorithm in order to determine possible stone moves and player positions. `Moves` is done using BFS most likely because it also records the shortest distance from the search origin, while the other functions use DFS. We plan to first look into the flood-filling algorithm, specifically `Moves` and `MarkReach`.

1.3. Design of the kernel

A flood-fill will require visiting every filled tile at least once, to check that it is unoccupied and to mark it as visited. Additionally, all tiles surrounding the filled area will need to be checked at least once (to find out that they are occupied). This is the bare minimum computation for a flood-fill, but generally algorithms require a data structure to keep track on the frontier of the search. This data structure is typically a stack (DFS) or a queue (BFS). Often, there are multiple routes to get to a given tile, so a single frontier tile is stored multiple times in the data strucutre. This results in additional checks for each square to see if it has already been visited. The total number of checks is the number of adjacent filled tiles. This could be as much as 4 checks (3 redundant checks).

In terms of instructions, a check for tile occupancy will involve a load, a bitmask, and a compare. Marking a tile visited will involve a load, a bit operation, and a store. Additionally, updating distance will involve an increment, and a store. Popping a frontier tile from the data structure will require a load, while pushing frontiers to the data structure will require an index computation (an add/subtract) followed by a store.

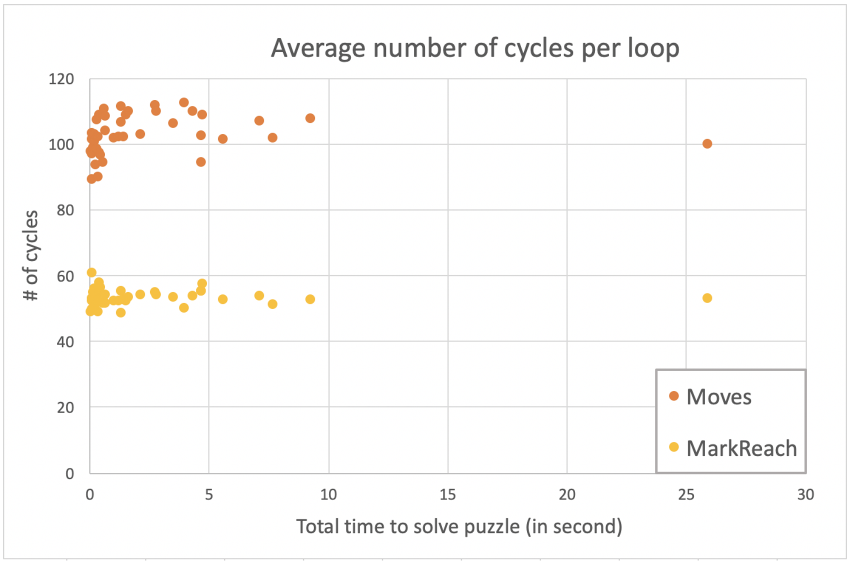
Immediately it is obvious that the flood-fill is very data-bound rather than compute-bound, with almost all processes having one or two instructions and then needing to either load and/or store. Additionally, aside from the initial state where the frontier is very small, the instructions are generally independent since different parts of the frontier do not depend on each other. (In the case of DFS the ordering of the frontier exploration does depends on the computation immediately before to finish, but in the context of this solver algorithm, strict DFS ordering is not necessary for the functions that use it.)

For the kernel, we will design flood-filling to avoid redundant work and avoid unnecessary cache misses. For one, the flood-fill should be aligned with memory; that is, the order of exploration should be along the axis which tiles are adjacent in memory. This will allow bitvectors that have been loaded into registers to be used as much as possible before have to store them back in memory. Fixing a direction also allows less items to be stored on the stack or queue, since we know the tiles in the reverse direction will have already been explored.

This method of flood-filling will be more effective if the area being filled is longer on the axis which tiles are adjacent in memory. In fact, we could transpose the game level so that the game level is longer on the axis which tiles are adjacent in memory.

2. Performance Baseline

2.1. Performance Numbers



We have tried to get the number of cycles for each call to `MarkReach` and `Moves`. Specfically, we measured the average number of cycles for a single iteration of the while loop (this performs the necessary computation for a single tile). We did this for the loops which did not terminate early. Each dot is a puzzle, and we use the amount of total time it takes to solve the puzzle as the measurement of different sizes of input.

2.2. Distance from Theoretical Peak

A single flood-fill loop will involve the following steps:

- Pop the tile index from the data structure. (load: 4 cycles (assume L1 cache), increment: 1 cycle)
- (Load occupancy from the level.) (load with indexing: 5 cycles (assume L1 cache))
- (Load the bitvectors indicating visited tiles.) (load with indexing: pipelined with previous)
- Check if the tile is occupied (`and`: 1 cycle, skipped branch) (here we assume the tile is unoccupied).
- Check if the tile is visited. (`and`: 1 cycle, skipped branch) (here we assume the tile is not yet visited).
- Mark the tile as visited. (`or`: 1 cycle)
- (Store the bitvectors indicating visited tiles.) (store: 1 cycle)
- Calculate adjacent tiles indices (4x add/sub: 4 cycles)
- Push the four adjacent tile indexes to the data structure. (4x store: 4 cycles, 4x increment: 4 cycles)

The steps enclosed by parentheses would not be necessary if the bitvectors and data structures were already in the registers. Excluding these, the theoretical peak is only 20 cycles for a single loop in `MarkReach`. Including these, the theoretical peak is 26 cycles. Our measured cycle count for a single loop in `MarkReach` is between 50-60.

References

[1] Dorit Dor and Uri Zwick. "SOKOBAN and other motion planning problems". In: *Computational Geometry* 13.4 (Oct. 1999), pp. 215–228. ISSN: 09257721. DOI: `10.1016/S0925-7721(99)00017-6`. URL: `https://linkinghub.elsevier.com/retrieve/pii/S0925772199000176`.

[2] Nils Froleyks. "Using an Algorithm Portfolio to Solve Sokoban". PhD thesis. Karlsruhe Institute of Technology, 2016, p. 56. URL: `https://baldur.iti.kit.edu/theses/SokobanPortfolio.pdf`.

[3] Anand Venkatesan, Atishay Jain, and Rakesh Grewal. "AI in Game Playing: Sokoban Solver". In: (2018). arXiv: `1807.00049`. URL: `http://arxiv.org/abs/1807.00049`.

[4] Huaxaun Gao et al. "A Sokoban Solver Using Multiple Search Algorithms and Q-learning". In: ().

[5] James Hyun and Seung Hong. *sokoban-solver*. URL: `https://github.com/jameshong92/sokoban-solver` (visited on 09/16/2019).