Lance Chisholm - 001177098
Joshua Vandenhoek - 001177098

3740 Project report

We attempted to make a prolog program that will use breadth first recursion in order to find the shortest list of correct moves to solve the puzzle, given a starting configuration and a goal configuration.

The first attempt that we made was similar to the solution we used for in our 3620 assignment that solved the 8 puzzle in c++. This way we were calculating the manhattan distance of the whole configuration and attempting to use a version of A* search in order to find the cheapest cost of moves to solve the puzzle. However this approach proved far too difficult and complex for our understanding of prolog and we were in way over our heads.
Our second approach was to simply use the nondeterministic nature of prolog to our advantage, this way we could simply try all valid moves given a configuration and check every time if that new configuration is the same as our goal configuration. If it was, then we would add what move we made into the list of moves at each level of the recursive call going “back up” the recursive tree.
Our second approach ended up working out because we are performing a breadth first search of the tree, which means that we will check each valid move and check the new configuration against our goal, before we move onto trying a new move. By structuring our recursion this way we ensure that we will get the shortest list of moves to solve the puzzle every time. We can get more lists of valid moves if we wished, however it seems redundant to collect information on these longer lists of moves so we added a cut operator after our call to solve.

We used our segment of all valid moves from our first implementation because it was easy to see that we would obviously need to know what moves were valid and where. If a particular move is valid, then it will return a character that represents the valid move that was made, as well as the new configuration of the board. Once this happens we set the head of the list for the list of moves to be the one that was just made. Then we recursively call solve on the tail of the list so that we will try new moves with that move that was just made.

Initially we found a tutorial that was similar to the same solution that we used for our puzzle solver in C++, however we realized that this method was far too complex for our understanding of prolog. What we then did was use the nondeterministic attribute of prolog to not try and make “the best guess” for a move to try, but to instead just try and guess every possible move.

We had issues with the difference between the ‘=’ and the ‘is’ operator. This was giving us problems because we were attempting to assign the head of the path list to a character (which won’t work with the is operator).

Lance Chisholm - 001177098
Joshua Vandenhoek - 001177098

Another issue that we encountered was ensuring that our program was actually doing a breadth first attempt of the problem, instead of a depth first. The issue we had was that we were using our append call in the wrong place (inside the solve function). After consulting our notes we figured that we needed to make another function “solve1” that would use the append call to set up breadth first search and then call solve to obtain the breadth first recursive condition.

We also had an issue with the cut operator because like the append call it was in the wrong place. This was a problem because it would freeze our program in places that it shouldn’t have been frozen.

Our prolog solution to the 8-puzzle problem is much faster than our c++ solution that we completed in another class. However it cannot do very complex problems, when it is presented with a complex problem it would take far too long to solve than actually trying to solve the problem yourself.

Now we will provide an example of a query we can perform using our program.
| ?- solve1(0/1/2/3/4/5/6/7/8, 1/2/3/4/0/5/6/7/8, P).

P = [r,r,d,l,l,u,r,d,r,u,l,l,d,r]

- The first parameter for the solve1 function is the initial configuration of the puzzle. Each value is separated by a ‘/’ character. We visualize this list as:

0	1	2
3	4	5
6	7	8

- The second parameter is the goal configuration:

1	2	3
4	0	5
6	7	8

By performing the moves in the same order that the list P provides, we will solve the puzzle according to the goal configuration.
note : each character represents the direction that 0 has to move; r is right, l is left, u is up, and d is down.

Compared to our c++ solution to the 8-puzzle solver, this prolog version was much easier to implement and conceptualize. It is also easier to understand. However our c++ implementation can handle much more complex problems because in that version we are

using an informed algorithm, whereas in contrast we are not using an informed algorithm in this prolog version.