Assignment 1

CSCI 3136: Principles of Programming Languages

Due Febuary 4, 2019

Assignments are due on the due date before 23:59. All assignments must be submitted electronically via the course SVN server. Plagiarism in assignment answers will not be tolerated. By submitting your answers to this assignment, you declare that your answers are your original work and that you did not use any sources for its preparation other than the class notes, the textbook, and ones explicitly acknowledged in your answers. Any suspected act of plagiarism will be reported to the Faculty's Academic Integrity Officer and possibly to the Senate Discipline Committee. The penalty for academic dishonesty may range from failing the course to expulsion from the university, in accordance with Dalhousie University's regulations regarding academic integrity.

General Instructions: How to Submit Your Work

The following submission instructions apply to **all** assignments, programming assignments and theory assignments. I will assume that you are familiar with using Subversion, as this was covered in CSCI 2132. If your knowledge of Subversion is rusty, consult your CSCI 2132 notes or look for Subversion tutorials online.

Throughout this course, assignments are managed via the FCS Subversion server `svn.cs.dal.ca`. The course repository is at `https://svn.cs.dal.ca/csci3136`. Only I and the TAs have access to the whole repository. You have access to the subrepository `https://svn.cs.dal.ca/csci3136/CSID`, where `CSID` is your CSID (your `bluenose` login).

Create a working copy of your subrepository in your `bluenose` home directory or on your personal laptop or desktop at home using `svn co https://svn.cs.dal.ca/csci3136/CSID`. To work on assignment *n*, make sure you have a directory `CSID/a`*n* in your working copy. For some assignments, `svn co` or (once checked out) `svn update` will create this directory for you and populate it with some files that provide a template for your work. For other assignments, you have to create the directory yourself.

All files you are asked to submit as part of your assignment answer must be placed into this directory `a`*n*, placed under subversion control using `svn add`, and submitted using `svn commit`. You can submit as often as you want before the submission deadline. On the deadline, I will take a snapshot of the contents of your assignment directory for marking.

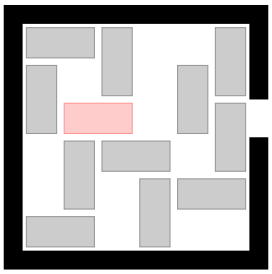
For programming assignments, each assignment states specifically which files you should submit.

For each theory assignment, you must submit a single **PDF file** `a`*n*`/a`*n*`.pdf` containing your assignment answers. Both typeset solutions or handwritten solutions are acceptable. If you submit handwritten solutions, you need to scan them to produce the PDF file to be submitted. If you submit typeset solutions, you can typeset your answers in Word, LaTeX or any other tool you like that is capable of producing PDF files. **Word documents, LaTeX source files or any application-specific documents are not acceptable.** If you use Word, for example, you can export your document in PDF format from Word.

After you have submitted your assignment, the markers will review your submission and provide feedback including marks in a single file `MARKS.a?.txt` or multiple files `MARKS.a?q?.txt` if the assignment has multiple questions. These files will be placed inside the relevant `CSID/a?` directory. Thus, running `svn update` and checking for the existence of these files in your assignment directories, you can check the feedback from TAs and keep a record of your assignment marks so far. If I find the time to do it, I am planning to also maintain a summary file in your `CSID` directory that provides an overview of the marks you obtained in all assignments so far. I can guarantee that the infrastructure to create these files will not be in place for the first two assignments.

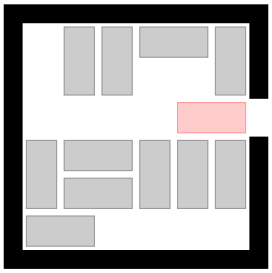
Rush Hour

Rush Hour is an entertaining puzzle game. You are given a 6×6 grid with *cars* (horizontal or vertical 3×1 or 2×1 rectangles) placed on it. From here on, I call this grid the *board* because the original Rush Hour game was a board game. The board is enclosed by walls, but there is an opening on the right side of the third row:



Horizontal cars are allowed to move left and right but only as far as they can go without hitting other cars or the wall. ("Hit" means "occupy the same grid cell as".) Vertical cars are allowed to move up and down subject to the same rules. By moving cars out of the way, you create new possible moves for other cars. The third row has a single horizontal 2×1 car on it (as well as possibly parts of some vertical cars). In the example above, this car is red. Your goal is to find the shortest sequence of moves that allows the red car to move through the gap on the right of the third row. A single move moves a single horizontal car left or right or a single vertical car up or down. Whether you move this car by 1, 2, 3 or 4 positions is unimportant; it still counts as only one move as long as the move is legal (does not hit the wall or other cars). Since the board is a 6×6 grid and every car is at least 2 cells long, no car can legally move by more than 4 positions.

To simplify the computational modelling of the problem, we will consider the red car to be *free* if it is in the 5th and 6th columns, that is, its right end is right next to the opening in the wall. Clearly, if we can move the red car to this position, we can also move it through the opening without any additional moves. The following is a possible solution state reachable from the state above:



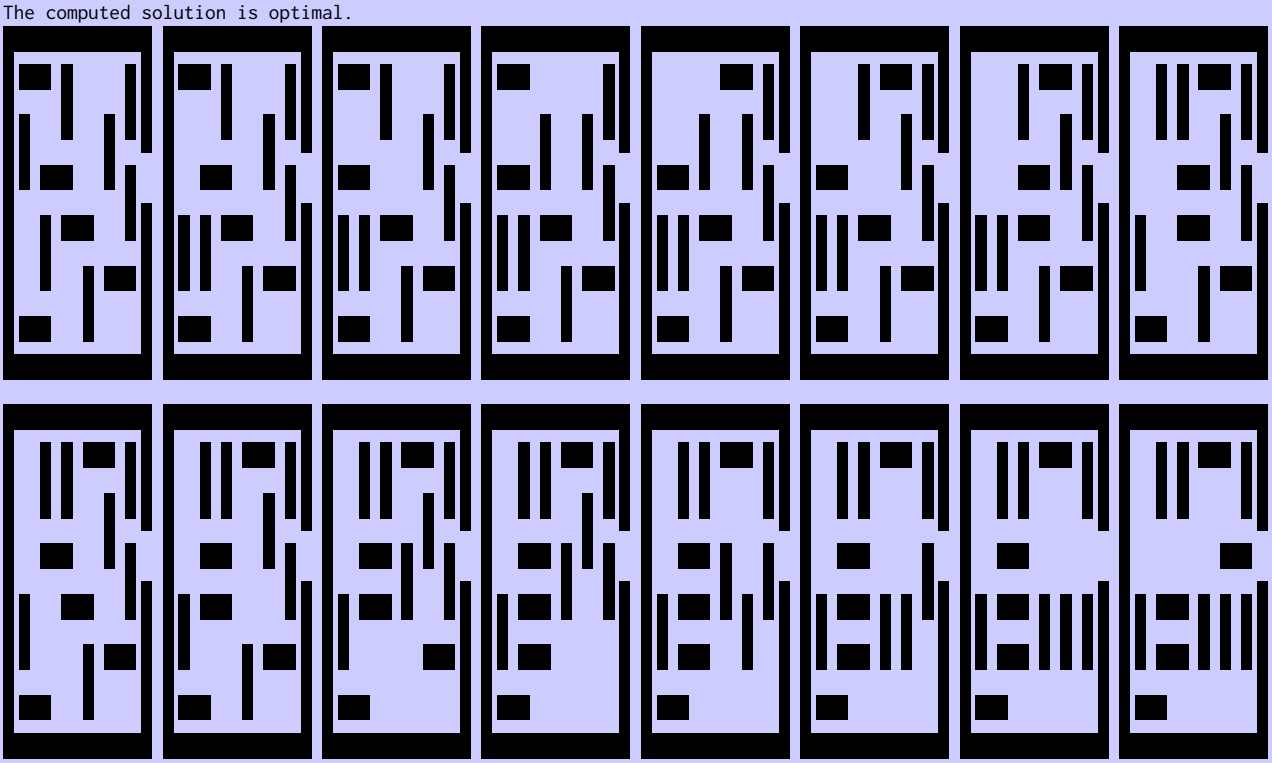
Your task is to write two programs that find an optimal solution (one taking the minimum number of moves) for any Rush Hour puzzle. The first one is to be implemented in Scheme, the second in Prolog. As part of the skeleton files you download using `svn co` or `svn update`, you get a database of almost 500,000 Rush Hour puzzles and a reference implementation of a Rush Hour solver in Python. To run the solver on the 84,386th puzzle (which happens to be the hardest puzzle in the database), you run

```
$ cd Python
$ ./rush_hour_solve.py 84386
BBHooKFoHoJKFAAoJLoGCCoLoGoIDDEEoIoo
(3,1)+2
(3,3)-1
(2,3)+1
(1,2)+3
(3,3)-1
(3,2)+2
(5,2)-3
(3,4)-1
(4,4)-1
(6,4)-2
(5,6)-3
(3,5)+2
(4,4)+1
(4,6)+1
(3,3)+3
```

The first 36-character string is the textual encoding of the puzzle in the database. Since this puzzle takes 15 moves to solve, this line is followed by 15 lines, each storing one move in the optimal move sequence. Each move is represented in the form (*row*,*col*)±*offset*. Both *row* and *col* are integers between 1 and 6 and refer to the position of the rightmost (bottommost) square of a horizontal (vertical) car to be moved. For a horizontal car, a negative offset means the car is moved *offset* positions to the left; a positive offset means the car is moved *offset* positions to the right. For a vertical car, a negative offset means move up; a positive offset means move down.

The Python directory contains a second script `rush_hour_check.py` that you can use to both check that you produced a correct solution and obtain a visual representation of the solution. To this end, it is useful to redirect the output of `rush_hour_solve.py` to a file:

```
$ ./rush_hour_solve.py 84386 > 84386.sol
$ ./rush_hour_check.py 84386.sol
Optimal moves     = 15
Moves in solution = 15
Search space size = 308818
The computed solution is optimal.
```



In this example, `rush_hour_check.py` is happy with its input and reports that the solution is optimal. It also prints the sequence of board states produced from the start state by the sequence of moves in the solution. The first state (top left) is the start state encoded by the string "`BBHooKFoHoJKFAAoJLoGCCoLoGoIDDEEoIoo`". The last state (bottom right) is the solved state reached at the end of the sequence of moves. Each state is obtained from the previous state to its left by moving one car to the left, to the right, up or down as instructed by the output of `rush_hour_solve.py`.

If the first string in the input file of `rush_hour_check.py` is not a puzzle in the database, a move is not a valid move, the final state is not a solution state (the car in the third row is not in the rightmost position) or the number of moves is not optimal, `rush_hour_check.py` will print an error message.

One useful piece of information you see in the output is the search space size. This is the total number of board states that are reachable from the initial state. The largest search space is the one of puzzle 84385, which has over 500,000 states in it. The general rule of thumb is that a larger search space size means the puzzle is harder to solve, at least for a computer. I recommend that you use puzzles with small search space sizes for your initial tests and move to bigger search spaces only once and if you are ready to test the efficiency of your code. The Python code can solve any puzzle in the database in under 4s on `bluenose`. I have a Rust implementation that takes no more than 0.08s to solve any puzzle on my laptop. My Scheme and Prolog implementations, on the other hand, take up to 19s and 25s, respectively, to solve the harder puzzles on `bluenose`. Since I always praise Haskell, I also have a Haskell implementation that takes no more than 8s to solve any puzzle in the database on my laptop. Note, however, that I put significantly more effort into optimizing the Python implementation than into optimizing any of the other solutions. For Scheme and Prolog, I tried to write as idiomatic code as possible, which ends up being slower than the Python code in addition to the inherent speed difference

between the Python interpreter and the Scheme and Prolog interpreters. For Haskell and Rust, I simply did not have the time to optimize them as aggressively, since they are not the focus of this assignment.

To find out which puzzles have small search spaces, look at the database file `rush_no_walls.txt`. The puzzle number refers to the line in this file that the puzzle is stored on, starting from 0 for the first line. Each line has 3 fields: the optimal number of moves to solve it, the 36-character string encoding the puzzle, and the size of the search space.

The General Solution Strategy and the Python Reference Code

I am making the Python reference code available so you have a solution that demonstrates how to approach the problem and replicates the split into multiple modules that your Scheme and Prolog implementations will follow.

Algorithmically, finding a sequence of moves that transforms the start state into a solved state is nothing but a graph traversal problem. The nodes of the graph are all possible states. There is an edge between two states if one of them can be transformed into the other using a single move. Then, what we are looking for is a path from the start state to a solution state in this graph. Since we are looking for a *shortest* move sequence, we are looking for a *shortest* path from the start state to a solution state in this graph. This suggests to use breadth-first search (BFS) to solve the problem.

In general BFS is a poor choice for solving this type of search probem because the solution space to be searched can get very big and BFS may need a lot of space to remember all states it has visited already. If the size of the search space increases exponentially with the search depth, an iteratively deepening depth-first search is more space-efficient and essentially as time-efficient as BFS. For the Rush Hour puzzle, BFS is feasible in terms of its space consumption (and significantly faster than an iteratively deepening depth-first search, as I verified experimentally) because a maximum search space size of around 500,000 states is rather moderate even if we have to store every state.

The Python implementation does not in fact explicitly build the graph to be explored. It simply starts at the start state and marks this state as visited. Then, for every state on the current frontier (the current BFS level), it generates all its neighbours (all states that can be reached using a single move from this state). For each neighbour, it checks whether this state is a solution state. If so, an optimal solution has been found. If not, it checks whether the neighbour has been seen before. If so, BFS does not explore it again. If the state has not been seen before, it is marked as seen and added to the list of states to be explored on the next BFS level.

To maintain the set of seen states, the Python implementation uses a Python set (which is a hash table under the hood.)

In order to report an optimal move sequence when a solved state is found, every state on the current frontier stores, along with it, the sequence of moves taken by the breadth-first search to reach this state from the start state. When a solved state is found, it then suffices to report this sequence of moves.

The implementation is split into three files: the main file `rush_hour_solve.py`, the solver module `rush_hour.solver` stored in the file `rush_hour/solver.py`, and a module `rush_hour.state` stored in the file `rush_hour/state.py` that provides primitives to manipulate board states. The dependency graph is that `rush_hour_solve.py` imports `rush_hour.solver` and `rush_hour.solver` imports the primitives to manipulate the board state from `rush_hour.state`. For your Scheme and Prolog implementations, I provide the equivalent of `rush_hour_solve.py` and `rush_hour.state` to you while you have to implement `rush_hour.solver` using the primitives provided by `rush_hour.state`.

`rush_hour_solve.py` takes care of parsing the command line arguments, reading the right puzzle from the puzzle database, invoking the solver provided by `rush_hour.solver`, and printing the computed solution to `stdout` in the format required by `rush_hour_check.py`.

`rush_hour.solver` provides a single public function

```
rush_hour.solver.run(puzzle)
```

This function takes the 36-character representation of the puzzle loaded from the database as its only argument. It returns the sequence of moves in an optimal solution, each represented as a 16-bit integer (see the `rush_hour.state.make_move` function below).

To find a solution, `rush_hour.solver.run` implements the above BFS strategy and relies on the following functions provided by `rush_hour.state` to do so:

```
rush_hour.state.from_string_rep(string)
```

This function constructs a compact internal representation of the board state from its 36-character string representation. The internal representation consists of four 64-bit integers. If we pad the 6×6 grid with two extra rows and two extra columns to represent the walls surrounding the board, we obtain an 8×8 grid. The 64 grid positions are numbered row by row from the top left to the bottom right. The ith bit in the first integer is 1 if the ith grid cell is occupied by a car. For the second integer, the ith bit is 1 if this car is a horizontal car. For the third integer, the ith bit is 1 if the car is a vertical car. For the fourth integer, the ith bit is 1 if the ith grid cell is the rightmost grid cell of a horizontal car or the bottommost grid cell of a vertical car. From here on, *state objects* refer to this 4-integer representation.

You normally shouldn't have to worry about the details of this internal representation, but for debugging purposes, this information may be useful.

If the provided 36-character string is not a valid representation of a Rush Hour board, the function returns `None`.

```
rush_hour.state.make_move(pos, offset)
```

Given a position `pos` of a piece to be moved and an offset `offset` by which to move it, this function returns a *move object* representing this move, to be used in the output of `rush_hour.solver.run`.

A move object is a 16-bit integer whose highest 8 bits store the position `pos` and whose lowest 8 bits store `offset` + 4. (Valid offsets are between −4 and 4, so this ensures that the offset in a move object is represented by a non-negative integer.)

```
rush_hour.state.horizontal_move(state, pos, offset)
```

This function tries to move the car occupying grid cell `pos` in state `state` by `offset` positions. On success, it returns the new state resulting from the move. On failure, the function returns `None`.

This operation succeeds if `pos` is the rightmost grid cell of a horizontal car and this car can be moved |`offset`| positions to the left (negative offset) or right (positive offset) without hitting the wall or another car. In any other scenario, the operation fails.

```
rush_hour.state.vertical_move(state, pos, offset)
```

This function tries to move the car occupying grid cell `pos` in state `state` by `offset` positions. It produces the same return value as `rush_hour.state.horizontal_move`.

This operation succeeds if `pos` is the bottommost grid cell of a vertical car and this car can be moved |`offset`| positions up (negative offset) or down (positive offset) without hitting the wall or another car. In any other scenario, the operation fails.

```
rush_hour.state.is_solved(state)
```

This function returns `true` if `state` is a solved state, that is, one where the car in the third row has been moved all the way to the right (position 30 is occupied by a horizontal car). Otherwise, it returns `false`.

```
rush_hour.state.is_occupied(state, pos)
```

This function returns `true` if position `pos` in state `state` is occupied by a car. Otherwise, it returns `false`.

```
rush_hour.state.is_horizontal(state, pos)
```

This function returns `true` if position `pos` in state `state` is occupied by a horizontal car. Otherwise, it returns `false`.

```
rush_hour.state.is_vertical(state, pos)
```

This function returns `true` if position `pos` in state `state` is occupied by a vertical car. Otherwise, it returns `false`.

```
rush_hour.state.is_end(state, pos)
```

This function returns `true` if position `pos` in state `state` is the rightmost position of a horizontal car or the bottommost position of a vertical car. Otherwise, it returns `false`.

With this description of the functions provided by `rush_hour.state`, you should now be able to read the code in `rush_hour/solver.py` and understand how it finds an optimal solution to the given puzzle.

`svn co` or `svn update` should have created a directory `a1/Scheme` in your working copy. Inside this directory, you find three files: `rush_hour_solve.ss`, `rush-hour/state.ss`, `rush-hour/solver.ss`, and `rush-hour/utils.ss`.

`rush-hour/utils.ss` implements a few utility functions used in the implementation of the functions in `rush-hour/state.ss` and of the functions in my sample implementation of `rush-hour/solver.ss`. Since it is perfectly possible to implement the solver in `rush-hour/solver.ss` without these utility functions, I do not discuss the contents of this file here. It needs to be present because `rush-hour/state.ss` relies on it. You are free to have a look at the functions this file implements and use them in your implementation, but you are on your own with that.

`rush_hour_solve.ss` is the main file analogous to `rush_hour_solve.py`. It takes care of all the boilerplate code such as reading the puzzle from the database and printing the computed solution to `stdout` in the format expected by `rush_hour_check.py`. It imports the library `(rush_hour solver)`, which the file `rush_hour/solver.ss` provides. If you look into this file, there is little code in there beyond a basic skeleton for a Scheme library file. Your task is to implement a function

```
(solve-puzzle string)
```

in this file. `rush_hour_solve.ss` expects `(rush-hour solver)` to export this function and calls it to initiate the solution process.

At the top of `rush-hour/solver.ss`, you will notice the import statement:

```
(import (rnrs (6))
        (rush-hour state))
```

The `(rush-hour state)` library implemented in `rush_hour/state.ss` provides the same functions as `rush_hour.state` in the Python code. However, their return values differ slightly. The following is the list of functions it puts at your disposal:

```
(state-from-string-rep string)
```

This function constructs a compact internal representation of the board state from its 36-character string representation. The internal representation is the same as in the Python code. If the provided string is not a valid representation of a Rush Hour board, this function returns `#f`.

```
(state-make-move pos offset)
```

Given a position `pos` of a piece to be moved and an offset `offset` by which to move it, this function returns a *move object* representing this move, to be used in the output of `solve-puzzle`. Move objects are represented as in the Python code.

```
(state-horizontal-move state pos offset)
```

This function tries to move the car occupying grid cell `pos` in state `state` by `offset` positions. On success, it returns the new state resulting from the move. On failure, it returns `#f`. The success conditions for this function are the same as for `rush_hour.state.horizontal_move` in the Python code.

```
(state-vertical-move state pos offset)
```

This function tries to move the car occupying grid cell `pos` in state `state` by `offset` positions. On success, it returns the new state resulting from the move. On failure, it returns `#f`. This function succeeds under the same conditions as `rush_hour.state.vertical_move` in the Python code.

```
(state-is-solved? state)
```

This function returns `#t` if `state` is a solved state, that is, one where the car in the third row has been moved all the way to the right (position 30 is occupied by a horizontal car). Otherwise, it returns `#f`.

```
(state-is-occupied? state pos)
```

This function returns `#t` if position `pos` in state `state` is occupied by a car. Otherwise, it returns `#f`.

```
(state-is-horizontal? state pos)
```

This function returns `#t` if position `pos` in state `state` is occupied by a horizontal car. Otherwise, it returns `#f`.

```
(state-is-vertical? state pos)
```

This function returns `#t` if position `pos` in state `state` is occupied by a vertical car. Otherwise, it returns `#f`.

```
(state-is-end? state pos)
```

This function returns `#t` if position `pos` in state `state` is the rightmost position of a horizontal car or the bottommost position of a vertical car. Otherwise, it returns `#f`.

To complete this question, implement the `solve-puzzle` function in `rush_hour/solver.ss`. **Your implementation must be purely functional, that is, it must not use local `(define ...)` forms (nested inside function definitions) and it must not use the `set!` function.** You are allowed to use local variables defined using `let`-, `let*`- or `letrec`-blocks. Named `let`-blocks are also allowed.

Your implementation will be graded primarily for correctness (70%). Secondary evaluation criteria with much lower weight are the use of proper functional idioms such as `map`, `filter`, and tail recursion where applicable (20%). The least important criterion is the running time of your implementation. In particular, you will not be penalized for a linear-time implementation that could shave off constant factors by being more clever in certain low-level details. If you provide an implementation that takes superlinear (e.g., quadratic) time in the size of the explored search space, you will lose 10% of the marks. Given that my implementation takes no more than 19s to solve any puzzle in the database on `bluenose`, your implementation should not take longer than 3 minutes to solve any puzzle in the database on `bluenose`.

(Q2) Solve Rush Hour Using Prolog

`svn co` or `svn update` should have created a directory `a1/Prolog` in your working copy. Inside this directory, you find four files: `rush_hour_solve.pl`, `rush_hour/state.pl`, and `rush_hour/solver.pl`, and `rush_no_walls.pl`.

`rush_hour_solve.pl` is the main file analogous to `rush_hour_solve.py`. It takes care of all the boilerplate code such as reading the puzzle from the database and printing the computed solution to `stdout` in the format expected by `rush_hour_check.py`. SWI Prolog's I/O system is painfully slow, to the point where reading the puzzle database takes forever. To circumvent this, the Prolog code uses its own puzzle database stored as predicates in `rush_no_walls.pl`. When you compile your program using

```
$ swipl -o rush_hour_solve -g main -O -q -c rush_hour_solve.pl
```

this database gets compiled into the executable `rush_hour_solve` this creates along with the actual code. This makes loading the desired puzzle significantly faster than parsing a text file.

`rush_hour_solve.pl` loads the module `rush_hour/solver.pl`. If you look into this file, there is little code in there beyond a basic skeleton for your implementation of a predicate

```
puzzle_solution(Puzzle, Solution).
```

`rush_hour_solve.pl` uses this predicate to search for a solution. Given an instantiation of `Puzzle` with a 36-character puzzle definition,

```
:- puzzle_solution(Puzzle, Solution).
```

should instantiate `Solution` with a list of move objects that represent the sequence of moves to solve this puzzle. As in the Python and Scheme implementations, move objects are 16-bit integers.

At the top of `rush_hour/solver.pl`, you will notice the import statement:

```
:- [rush_hour/state].
```

This loads the module `rush_hour/state.pl`, which provides you with the low-level predicates for manipulating state objects and moves. The following is the list of predicates it puts at your disposal:

```
puzzle_state(Puzzle, State)
```

Given an instantiation of `Puzzle` with a 36-character string representing a Rush Hour puzzle, this predicate instantiates `State` with a state object. Similarly to the Python and Scheme implementations, a state object is a list of four 64-bit integers with the same meaning as for the Python and Scheme implementations.

This predicate is only meant to be used as a function: Given an instantiated first argument and an uninstantiated second argument, it instantiates the second argument with the state object corresponding to its first argument. Any other attempted use of this predicate is likely to fail.

```
pos_offset_move(Pos, Offset, Move)
```

This predicate expects `Pos` and `Offset` to be instantiated with a grid position and an offset. `Move` should be uninstantiated. Under these conditions, the predicate succeeds and instantiates `Move` to be the move object representing the move of the car at position `Pos` by `Offset` positions. Move objects are represented as in the Python code.

```
horizontal_move(State, Pos, Offset, NewState)
```

This predicate expects `State`, `Pos`, and `Offset` to be instantiated; `NewState` should be uninstantiated. Thus, it is once again intended to be used as a function that computes `NewState` from `State`, `Pos`, and `Offset`. if the car occupying the grid cell position `Pos` in state `State` can be moved by `Offset` positions, then the predicate succeeds and instantiates `NewState` to hold the new state. Otherwise, the predicate fails. The success conditions are the same as for the function `rush_hour.state.horizontal_move` in the Python implementation.

```
vertical_move(State, Pos, Offset, NewState)
```

This predicate expects `State`, `Pos`, and `Offset` to be instantiated; `NewState` should be uninstantiated. Thus, it is once again intended to be used as a function that computes `NewState` from `State`, `Pos`, and `Offset`. if the car occupying the grid cell position `Pos` in state `State` can be moved by `Offset` positions, then the predicate succeeds and instantiates `NewState` to hold the new state. Otherwise, the predicate fails. The success conditions are the same as for the function `rush_hour.state.vertical_move` in the Python implementation.

```
state_is_solved(State)
```

This predicate succeeds if `State` is a solved state as defined for `rush_hour.state.is_solved`.

```
state_is_occupied(State, Pos)
```

This predicate succeeds if the grid cell at position `Pos` in state `State` is occupied.

```
state_is_horizontal(State, Pos)
```

This predicate succeeds if the grid cell at position `Pos` in state `State` is occupied by a horizontal car.

```
state_is_vertical(State, Pos)
```

This predicate succeeds if the grid cell at position `Pos` in state `State` is occupied by a vertical car.

```
state_is_end(State, Pos)
```

This predicate succeeds if the grid cell at position `Pos` in state `State` is the rightmost grid cell of a horizontal car or the bottommost grid cell of a vertical car.

To complete this question, implement the `puzzle_solution/2` predicate in `rush_hour/solver.pl`. Your implementation will be graded primarily for correctness (80%). A secondary criterion (20%) is the running time of your implementation. Your code should be able to solve any puzzle in the database in at most 4 minutes on `bluenose`.