Programming Contest: Rush Hour

Generated by Doxygen 1.8.5

Tue Oct 8 2013 19:53:34

Contents

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

Class Documentation

3.1 vehicle Struct Reference

Public Attributes

- int number
- int headX
- int headY
- char orientation
- int length

3.1.1 Member Data Documentation

3.1.1.1 int vehicle::headX

3.1.1.2 int vehicle::headY

3.1.1.3 int vehicle::length

3.1.1.4 int vehicle::number

3.1.1.5 char vehicle::orientation

The documentation for this struct was generated from the following file:

- C:/Users/Terence/Google Drive/CS 302/PA06/Rush_Hour/rush_hour.cpp

Chapter 4

File Documentation

4.1 C:/Users/Terence/Google Drive/CS 302/PA06/Rush_Hour/rush_hour.cpp File Reference

This program will solve a given rush hour puzzle using a bounded "try every possible strategy" method via recursion. The reult that is found is the least amount of necessary moves.

```
#include <cstdlib>
#include <iostream>
#include <cstring>
```

Classes

- struct vehicle

Functions

- int setupBoard (int board[BOARD_SIZE][BOARD_SIZE], vehicle ∗&carList)
- int solveJam (int board[BOARD_SIZE][BOARD_SIZE], vehicle ∗&carList, int numCars, int &currentBound, int movesDone)
- bool checkForEnd (int board[BOARD_SIZE][BOARD_SIZE], int movesDone, int currentBound, vehicle &car)
- bool moveVehicle (int board[BOARD_SIZE][BOARD_SIZE], vehicle &car, bool direction)
- bool placeCar (int board[BOARD_SIZE][BOARD_SIZE], vehicle &car, int newX, int newY)
- int main ()

Variables

- const int BOARD_SIZE = 6
- const int CAR_LEN = 2
- const int TRUCK_LEN = 3
- const char HORIZONTAL = 'H'
- const char VERTICAL = 'V'
- const int MAX_MOVES = 10
- const bool FORWARD = true
- const bool BACKWARD = false

4.1.1 Detailed Description

This program will solve a given rush hour puzzle using a bounded "try every possible strategy" method via recursion. The reult that is found is the least amount of necessary moves.

Author

Terence Henriod

Version

Original Code 1.00 (10/8/2013) - Terence Henriod

4.1.2 Function Documentation

4.1.2.1 bool checkForEnd (int *board[BOARD_SIZE][BOARD_SIZE],* int *movesDone,* int *currentBound,* vehicle & *car*)

This function checks for the end of a strategy path either by comparing the number of moves done in the current strategy, or by checking to see if the 0 car has escaped the jam

Parameters

board[BOARD_-SIZE][BOARD_-SIZE]	a 2-D array that will represent the game states as the puzzle is solved
carList	a pointer used to allocate/reference memory for an array that will hold the vehicle structs that have the vehicle information for the puzzle
currentBound	the highest number of moves any strategy we will test may use. decreases as new, more efficient, but successful strategies are found.
movesDone	the count of moves currently executed for a given strategy path
car	the 0 car is passed so that the last space of the row car 0 occupies contains car 0

Returns

endFound indicates the success of finding the end of the strategy path

Precondition

1. int board[BOARD_SIZE][BOARD_SIZE] is initialized to an appropriate puzzle state
2. int movesDone is $>=$ 0
3. int currentBound is $>=$ movesDone
4. vehicle car is a valid car

Postcondition

1. int currentBound will be $>=$ movesDone. it will not be updated, this occurs outside this function if necessary

1. first compares movesDone to currentBound, if movesDone is $>=$ currentBound the end has been found
2. if car 0 made it to the rightmost column (it has escaped), the end has been found

4.1.2.2 int main ( )

4.1.2.3 bool moveVehicle (int *board[BOARD_SIZE][BOARD_SIZE],* vehicle & *car,* bool *direction*)

This function moves a vehicle in the given direction (if possible) and updates the vehicle's position on the game board.

Parameters

board[BOARD_-SIZE][BOARD_-SIZE]	a 2-D array that will represent the game states as the puzzle is solved
car	the car that is to be moved
direction	a flag to indicate which direction the car will be moved

Returns

moveSuccess indicates whether or not the attempted move was completed

Precondition

1. int board[BOARD_SIZE][BOARD_SIZE] is initialized to an appropriate puzzle state
2. vehicle∗ car is a valid car

Postcondition

1. int board[BOARD_SIZE][BOARD_SIZE] will contain the placement data of all vehicles, with -1 representing unoccupied space (if there is a puzzle to be solved), including the just moved car
2. if the vehicle was not successfully moved, the board and car's states remain unchanged
3. first it is decided which direction the car is being moved
4. then it is determined if the space the vehicle would occupy will be valid
5. next a check to see that the intended position is clear to be moved into
6. if all of the above checks pass, the vehicle is then placed on the board in its new location, and any previous/no longer valid spaces are cleared

4.1.2.4 bool placeCar (int *board[BOARD_SIZE][BOARD_SIZE],* vehicle & *car,* int *newX,* int *newY*)

Places a car on the given board, regardless of what may be occupying the spaces.

Parameters

board[BOARD_-SIZE][BOARD_-SIZE]	a 2-D array that will represent the game states as the puzzle is solved
car	the car that is to be moved
newX	the new x coordinate for the "head" of the car "vector"
newY	the new y coordinate for the "head" of the car "vector"

Returns

moveSuccess indicates whether or not the attempted placement was completed (in this version, placement is always successful)

Precondition

1. int board[BOARD_SIZE][BOARD_SIZE] is initialized to an appropriate puzzle state
2. vehicle∗ car is a valid/initialized car

Postcondition

1. int board[BOARD_SIZE][BOARD_SIZE] will contain the placement data of all vehicles, with -1 representing unoccupied space (if there is a puzzle to be solved), including the recently placed car
2. the vehicle will be placed wherever the function is told to place it (BEWARE seg-faults!)
3. the car's "head" is updated to it's new coordinates
4. placement on the board begins with the head coordinates
5. then the tail is placed based on the length of the car into the following spaces on the board (to the right for horizontal, down for vertical)

4.1.2.5 int setupBoard (int *board[BOARD_SIZE][BOARD_SIZE]*, vehicle ∗& *carList*)

This function will either return the signal to stop processing rush hour puzzles or it wll set up the board in prepaation for the next puzzle and return the number of cars that will be in the puzzle.

Parameters

board[BOARD_-SIZE][BOARD_-SIZE]	a 2-D array that will represent the game states as the puzzle is solved
carList	a pointer used to allocate/reference memory for an array that will hold the vehicle structs that have the vehicle information for the puzzle

Returns

numCars the number of cars the puzzle will have

Precondition

1. vehicle∗ carList is a NULL pointer to an empty list
2. int board[BOARD_SIZE][BOARD_SIZE] may be in any state

Postcondition

1. vehicle∗ carList will be allocated and will contain all the cars necessary for soving the puzzle (if there is a puzzle to be soved)
2. int board[BOARD_SIZE][BOARD_SIZE] will contain the placement data of all vehicles, with -1 representing unoccupied space (if there is a puzzle to be solved)
3. the number of cars for the next puzzle to be solved is read in
4. if there are no cars (no puzzle), this number is returned
5. if there are cars, the board is cleared (all spaces = -1) and car placement occurs
6. each car's data is read in, stored in carList, an it is represented on the board

4.1.2.6 int solveJam (int *board[BOARD_SIZE][BOARD_SIZE]*, vehicle ∗& *carList*, int *numCars*, int & *currentBound*, int *movesDone*)

This function will solve a rush hour puzzle using a semi-exhaustive "try all possibilities" approach that is bounded by the lowest number of moves from a previously successful strategy or predetermined upper limit.

Parameters

board[BOARD_-SIZE][BOARD_-SIZE]	a 2-D array that will represent the game states as the puzzle is solved
carList	a pointer used to allocate/reference memory for an array that will hold the vehicle structs that have the vehicle information for the puzzle
numCars	the number of cars the puzzle will have
currentBound	the highest number of moves any strategy we will test may use. decreases as new, more efficient, but successful strategies are found.
movesDone	the count of moves currently executed for a given strategy path

Returns

currentBound the highest number of moves any strategy we will test may use. decreases as new, more efficient, but successful strategies are found.

Precondition

1. int board[BOARD_SIZE][BOARD_SIZE] is initialized to an appropriate puzzle state
2. vehicle$*$ carList is a NULL pointer to a suitable list to fit the current puzzle
3. int numCars matches to the size of carList
4. int movesDone is $>=$ 0

Postcondition

1. int currentBound will represent at least the best number of moves required to solve the current strategy path and at most the
2. int board[BOARD_SIZE][BOARD_SIZE] will contain the placement data of all vehicles, with -1 representing unoccupied space (if there is a puzzle to be solved)
3. first checks for an end of a strategy path either to see if the bound is reached or if the game has been won. if the game has been won, if the score is a new low score, the bound is updated
4. if the end of the strategy path has not been reached, each car is moved
5. first the car is moved forward and the function is called again to solve this new game state. after the reursive calls have resolved, the move is undone the..
6. next the car is moved backward and the function is called again to solve this new game state. after the reursive calls have resolved, the move is undone.
7. the lowest score is returned

4.1.3 Variable Documentation

4.1.3.1 const bool BACKWARD = false

4.1.3.2 const int BOARD_SIZE = 6

4.1.3.3 const int CAR_LEN = 2

4.1.3.4 const bool FORWARD = true

4.1.3.5 const char HORIZONTAL = 'H'

4.1.3.6 const int MAX_MOVES = 10

4.1.3.7 const int TRUCK_LEN = 3

4.1.3.8 const char VERTICAL = ’V’

Index