Terence Henriod

Rush Hour: Breadth First Search

Generated by Doxygen 1.7.6.1

Thu Dec 5 2013 20:23:10

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 GameState::car Struct Reference

Public Attributes

- int xPos
- int yPos
- int length
- char orientation

3.1.1 Detailed Description

Contains all the pertinent data for representing a car, with a pair of coordinates for the car's head, its length, and its orientation.

3.1.2 Member Data Documentation

3.1.2.1 int GameState::car::length

3.1.2.2 char GameState::car::orientation

3.1.2.3 int GameState::car::xPos

3.1.2.4 int GameState::car::yPos

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

- GameState.h

3.2 GameState Class Reference

`#include <GameState.h>`

Classes

- struct car

Public Member Functions

- GameState ()
- GameState (const GameState &other)
- GameState & operator= (const GameState &other)
- ~GameState ()
- void clear ()
- bool move (const int whichCar, const bool direction)
- bool placeCar (const int whichCar, int newX, int newY)
- void readIn ()
- void setNumCars (const int newNumCars)
- bool isWin () const
- string stringify () const
- void printBoard () const

Private Attributes

- char board [kBoardSize][kBoardSize]
- car carList [kMaxCars]
- int numCars

3.2.1 Detailed Description

Contains the full amount of data required to represent and manipulate a Rush Hour - Game state. Includes a representation of a game board, the ability to move cars, and overloaded copy constructor and assignment operator for use with the STL queue.

3.2.2 Constructor & Destructor Documentation

3.2.2.1 GameState::GameState ( )

GameState

The default constructor for a game state. Constructs and initilizes an empty GameState.

Precondition

1. The GameState object is given an appropriate identifier.

Postcondition

1. A new, empty GameState will be initialized.

3.2.2.2 GameState::GameState (const GameState & *other*)

GameState

The copy constructor for a game state. Constructs and initilizes a GameState equivalent to the given parameter GameState other.

Parameters

other	A previously created GameState. Will be cloned into ∗this.

Precondition

1. GameState other is a valid instantiation of a GameState
2. The GameState object is given an appropriate identifier.

Postcondition

1. ∗this will be an equivalent clone of other.

3.2.2.3 GameState::∼GameState ( )

∼GameState

The destructor for the GameState class. Currently doesn't do anything.

Precondition

1. The GameState object is given an appropriate identifier.

Postcondition

1. ∗this will be completely destructed.

3.2.3 Member Function Documentation

3.2.3.1 void GameState::clear ( )

clear

Clears the GameState so that it is an empty one with no car data.

Precondition

1. ∗this is a valid GameState object.

Postcondition

1. ∗this will be a GameState with no cars and a clean board

3.2.3.2 bool GameState::isWin ( ) const

isWin

Indicates if the GameState pertains to a winning one.

Returns

isWin A boolean value indicating if the game has been won, true if it has, false otherwise.

Precondition

1. ∗this is a valid instantiation of a GameState.
2. ∗this must actually contain a GameState in order for this result to produce a meaningful result.

Postcondition

1. The status of whether or not the game has been won is returned.

3.2.3.3 bool GameState::move (const int *whichCar,* const bool *direction*)

move

Modifies the game state to simulate the movement of a vehicle.

Parameters

whichCar	Which car in the current list is to be moved.
direction	Which direction the car will attempt to move. Function should be called using the constants given in GameState.h.

Returns

movementSuccess Indicates the success of the movement so that decisions can be made. True for a successful movement, false otherwise.

Precondition

1. The GameState is in a valid state with legally placed cars.

Postcondition

1. Cars will be in a valid state on the game board and the success of the movement is returned.

1. Depending on the car and direction, the next array location the car will occupy is computed.
2. This location is checked to see if it is in the bounds of the array.
3. The location is then checked to see if it is occupied already.
4. If the location is occupied, the car is placed on the new location and the space previously occupied is cleaned up. True is returned to indicate a successful movement.
5. If for any reason the move cannot be legally made, the attempt is abandoned and false is returned.

3.2.3.4 GameState & GameState::operator= (const GameState & *other*)

operator=

The assignment operator for the GameState class. Clones the given parameter GameState other into ∗this.

Parameters

other	A valid GameState to be cloned into ∗this.

Returns

∗this

Precondition

1. GameState other is a valid instantiation of a GameState
2. The GameState object is a valid instantiation of a GameState.

Postcondition

1. ∗this will be an equivalent clone of other.

3.2.3.5 bool GameState::placeCar (const int *whichCar,* int *newX,* int *newY*)

placeCar

Updates a car to its new coordinates and marks the car's new location on the board. Currently, the boolean return is meaningless, but may become useful in the future.

Parameters

whichCar	Which car in the list is to be placed on the board.
newX	The new x coordinate of the car.
newY	The new y coordinate of the car.

Returns

successful Currently meaningless, in the future may be used to indicate a successful/legal placement.

Precondition

1. Parameters newX and newY are legal locations for the new car to be placed.

Postcondition

1. The board will be marked with a car corresponding to the given car and the given coordinates.
2. Does not perform any clean-up, only placement.

1. Computes the first location to mark to represent the car. =# Appropriatly computes and marks the remaining locations.

3.2.3.6 void GameState::printBoard ( ) const

printBoard

Displays the board on screen for debugging purposes.

Precondition

1. ∗this is a valid instantiation of a GameState.
2. ∗this must actually contain a GameState in order for this result to produce a meaningful result.

Postcondition

1. The GameState will remain unchanged.
2. A planar representation of the game board is displayed on screen.

3.2.3.7 void GameState::readIn ( )

readIn

Reads in data from the standard in to initialize a game state using the given data.

Precondition

1. ∗this is a valid instantiation of a game state.
2. ∗this is a cleared games state to prevent data corruption, or at least this function will perform the clearing operation.

Postcondition

1. ∗this will contain the data of a valid game state.

1. ∗this is cleared.
2. The car data is read in sequentially.
3. The cars are then placed in the array

3.2.3.8 void GameState::setNumCars (const int *newNumCars*)

setNumCars

Used to notify the GameState how much car data is waiting to be read in.

Precondition

1. ∗this is a valid instantiation of a game state.
2. ∗this is a cleared games state to prevent data corruption, or at least this function will perform the clearing operation.
3. Parameter newNumCars corresponds precisely to a set of data waiting to be read into the GameState via function readIn.

Postcondition

1. ∗this will contain a number pertaining to the number of cars in the next Rush Hour problem to be solved.

1. numCars is assigned the value of the given parameter newNumCars.

3.2.3.9 string GameState::stringify ( ) const

stringify

Converts the current GameState's vehicle list to a string for GameState sharing, searching, and comparison.

Returns

gameString A string class object containing the data of the GameState's vehicle list inside it.

Precondition

1. ∗this is a valid instantiation of a GameState.
2. ∗this must actually contain a GameState in order for this result to produce a meaningful result.

Postcondition

1. The GameState will remain unchanged.
2. A string class object with the data pertaining to the car list is returned.

3.2.4 Member Data Documentation

3.2.4.1 char GameState::board[kBoardSize][kBoardSize] [private]

3.2.4.2 car GameState::carList[kMaxCars] [private]

3.2.4.3 int GameState::numCars [private]

The documentation for this class was generated from the following files:

- GameState.h
- GameState.cpp

Chapter 4

File Documentation

4.1 GameState.cpp File Reference

Class implementations declarations for a Rush Hour GameState.

`#include "GameState.h" #include <cassert> #include <iostream> ×`

4.1.1 Detailed Description

Class implementations declarations for a Rush Hour GameState.

Author

Terence Henriod

Rush Hour: Breadth First Search

Version

Original Code 1.00 (10/29/2013) - T. Henriod

4.2 GameState.h File Reference

Class declarations for a Rush Hour Game State.

`#include <cassert> #include <iostream> #include <string> ×`

Classes

- class GameState
- struct GameState::car

Variables

- const bool FORWARD = true
- const bool BACKWARD = false
- const char kHorizontal = 'H'
- const char kVertical = 'V'
- const char kEmpty = '.'
- const int kBoardSize = 6
- const int kBoardSizeSquared = (kBoardSize * kBoardSize)
- const int kMaxCars = 18
- const int kCarXpos = 0
- const int kCarYpos = 1
- const int kLengthPos = 3
- const int kOrientationPos = 4
- const int kCarDataSize = 4

4.2.1 Detailed Description

Class declarations for a Rush Hour Game State.

Author

Terence Henriod

Rush Hour: Breadth First Search

Version

Original Code 1.00 (10/29/2013) - T. Henriod

4.2.2 Variable Documentation

4.2.2.1 const bool BACKWARD = false

4.2.2.2 const bool FORWARD = true

4.2.2.3 const int kBoardSize = 6

4.2.2.4 const int kBoardSizeSquared = (kBoardSize * kBoardSize)

4.2.2.5 const int kCarDataSize = 4

4.2.2.6 const int kCarXpos = 0

4.2.2.7 const int kCarYpos = 1

4.2.2.8 const char kEmpty = '.'

4.2.2.9 const char kHorizontal = 'H'

4.2.2.10 const int kLengthPos = 3

4.2.2.11 const int kMaxCars = 18

4.2.2.12 const int kOrientationPos = 4

4.2.2.13 const char kVertical = 'V'

4.3 RushFinal.cpp File Reference

The driver program for a fast Rush Hour Solver.

```
#include <cstdlib> #include <iostream> #include <iomanip> ×
#include <string> #include <queue> #include <map> #include
"GameState.h"
```

Functions

- int solveRushHour (int numCars, GameState initial, queue< GameState > statesToEvaluate, map< string, int > observedStates)
- int main ()

Variables

- const int kUnsolvable = -8

4.3.1 Detailed Description

The driver program for a fast Rush Hour Solver.

Author

Terence Henriod

Rush Hour: Breadth First Search

Version

Original Code 1.00 (10/29/2013) - T. Henriod

4.3.2 Function Documentation

4.3.2.1 int main ( )

main

The driver of a Rush Hour solver. Reports the number of moves required to solve given rush hour puzzles.

Returns

programSuccess The success/error code of the program's execution.

Precondition

1. There is valid data waiting to be provided to the program.

Postcondition

1. A number of Rush Hour puzzles will have been solved

4.3.2.2 int solveRushHour (int *numCars*, GameState *initial*, queue< GameState > *statesToEvaluate*, map< string, int > *observedStates*)

solveRushHour

Finds the lowest number of moves required to find the solution to a Rush Hour puzzle. If a solution cannot be found, an error code is returned.

Parameters

numCars	The number of cars to be used in a given puzzle.
initial	The initial board for a given problem.
statesToEvaluate	A queue that will be used in a breadth first search to contain every game state that will be considered.
observedStates	A map used to contain an identifying key string that contains vehicle list data used to check to see if a state has already been considered.

Returns

numMoves The minimum number of moves required to solve the given puzzle.

Precondition

1. A valid number of cars that corresponds precisely to the data waiting to be read in is required.
2. statesToEvaluate must be an empty queue.
3. observedStates must be empty as well.
4. The initial state will be representative of the starting condition for the puzzle to be solved.

Postcondition

1. A number of moves corresponding to either the number of moves needed to solve a given puzzle or an error code for an unsolvable puzzle is returned.

1. A breadth first search is implemented.
2. GameStates are added to the queue and their representative strings are added to the queue if they are newly encountered.
3. States are iteratively popped off the queue and checked to see if they are a winning state.
4. If they are not, all possible GameStates based on legal moves are generated and stored in the queue. Their representative strings and the number of moves used up to that point are stored for future reference.
5. Should the queue ever become empty, this indicates that all possible moves have been attempted and the puzzle is unsolvable.

4.3.3 Variable Documentation

4.3.3.1 const int kUnsolvable = -8