lifapc-benj-em

Generated by Doxygen 1.9.1

1 Projet Rush Hour

Le but de ce projet est d'écrire un programme permettant de trouver une solution au jeu *Rush Hour*.

1.1 Installation

1.1.1 Prérequis :

C++ 20
imagemagick
make

```
sudo apt install imagemagick
```

1.2 Compilation

Pour compiler le programme en mode **DEBUG**

```
make debug
```

Pour compiler le programme en mode **RELEASE**

```
make release
```

Pour générer la **documentation**

```
make doxy
```

1.3 Exécution

Pour exécuter le **main** :

./bin/main

Pour générer des pendant l'exécution du main **GIF**

./bin/main –gif

Pour exécuter le main de **test**

./bin/test

1.4 Règles du jeu

Le jeu Rush Hour se joue seul sur une grille carrée de six cases de côté. Sur cette grille sont répartis des véhicules d'une case de largeur, et de deux ou trois cases de longueur. Ces véhicules peuvent être placés horizontalement ou verticalement. Chaque véhicule peut être déplacé en avant ou en arrière, mais pas latéralement, tant qu'il n'entre pas en collision avec un autre véhicule. Le but du jeu est de faire sortir l'un des véhicules par une sortie placée sur le bord du plateau. L'image ci dessous illustre un exemple de partie.

Chaque déplacement de véhicule compte pour un coup, quelle que soit la longueur du déplacement. La qualité de votre solution dépend donc du nombre de coups nécessaires depuis la situation initiale pour faire sortir le véhicule.

1.5 Modélisation

La recherche d'une solution au jeu Rush Hour peut être modélisée sous la forme d'un parcours de graphe. Dans ce graphe, les sommets sont des situations de jeu. Les arêtes sont des coups. Les deux images qui suivent représentent deux situations de jeu, et donc deux sommets du graphe. Il est possible de passer d'une situation à l'autre en déplaçant le long véhicule du haut, elles sont donc reliées par une arête dans le graphe.

Votre première tâche pour ce projet consiste à élaborer une structure de données sous la forme d'une classe pour représenter les situations de jeu, munies de méthodes pour accéder de façon pratique aux situations de jeu adjacentes.

Pour vous aider dans l'élaboration de votre structure de données, vous pourrez utiliser le fait que :

- les véhicules ne sont que de taille deux ou trois
- il n'y a jamais plus de 16 véhicules
- il n'y a toujours qu'un véhicule à sortir

La situation initiale du problème résolu plus haut :

pourra être décrite par `le fichier suivant` :

```
2 5
2 0 2 1
0 0 2 0
0 2 3 0
0 3 3 1
1 3 2 0
1 4 2 1
2 5 2 0
3 0 2 1
4 0 2 0
4 3 2 0
4 4 2 0
4 5 2 0
5 1 2 1
```

La première ligne correspond à la position de la sortie (ligne 2 colonne 5, on commence la numérotation à 0), la seconde ligne est la position du véhicule à sortir (ligne 2, colonne 0, longueur 2, horizontal), les lignes suivantes sont les autres véhicules, toujours avec le format ligne, colonne, longueur, horizontal (1) ou vertical (0). Dans le cas d'un véhicule horizontal, la position donnée est celle de la case la plus à gauche, dans le cas d'un véhicule vertical, la position donnée est celle de la case la plus haute.

Pour favoriser les échanges, vous pouvez munir votre classe d'un constructeur prenant un fichier en paramètre, au format décrit ci-dessus, ainsi que d'une fonction pour exporter votre situation de jeu sous la forme d'un fichier similaire.

1.6 Parcours

Une fois les situations de jeu représentables, il s'agit maintenant d'instancier la situation de jeu initiale, et de parcourir le graphe pour trouver une situation de jeu gagnante, ainsi que les coups permettant de l'atteindre. Idéalement, le nombre de coups à jouer pour atteindre cette situation de jeu gagnante devra être minimal. Dans le cas de l'exemple fourni ci-dessus, le code de votre responsable d'UE a donné une solution en 14 coups. Il est nécessaire de réaliser un parcours de graphe bien choisi. Il n'est pas ici nécessaire de générer tout le graphe, mais seulement de partir de la situation de départ, de lister les situations atteignables en déplaçant des véhicules, et de les ajouter à votre structure de données gérant les situations de jeu encore à traiter, selon le type de parcours choisi.

Les situations de jeu sont donc découvertes petit à petit, attention cependant à faire en sorte que votre exploration n'étudie qu'une fois chaque situation de jeu, et se rendre compte que certaines situations ont déjà été explorées. Sans cette attention, votre exploration risquera de tourner en rond entre des situations de jeu, ou d'en explorer beaucoup trop.

1.7 Élaboration de nouveaux puzzles

Une fois la résolution programmée, et le parcours du graphe compris, consacrez-vous à la création de nouveaux puzzles. Cette fois, il s'agit de fournir une situation de départ qui soit intéressante à jouer. La difficulté du puzzle correspondra au nombre de coups minimal pour le résoudre, et votre but sera ici de trouver des stratégies pour créer les puzzles les plus difficiles possibles.

2 Class Index

2.1 Class List

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

3 File Index

3.1 File List

Here is a list of all files with brief descriptions:

4 Class Documentation

4.1 Coordinate Class Reference

The Coordinate class represents a point in two-dimensional space.

`#include <Coordinate.hpp>`

Public Member Functions

- Coordinate ()
- Coordinate (int _x, int _y)

  Constructor for Coordinate.

- int getX () const

  Gets the x coordinate of the point.
- int getY () const

  Gets the y coordinate of the point.
- int setX (int _x)

  Sets the x coordinate of the point.
- int setY (int _y)

  Sets the y coordinate of the point.

4.1.1 Detailed Description

The Coordinate class represents a point in two-dimensional space.

4.1.2 Constructor & Destructor Documentation

4.1.2.1 Coordinate() [1/2] `Coordinate::Coordinate ( )`

4.1.2.2 Coordinate() [2/2] `Coordinate::Coordinate (`

```
int _x,
int _y )
```

Constructor for Coordinate.

Parameters

↩ _↩ *x*	The x coordinate of the point.
↩ _↩ *y*	The y coordinate of the point.

4.1.3 Member Function Documentation

4.1.3.1 getX() `int Coordinate::getX ( ) const`

Gets the x coordinate of the point.

Returns

The x coordinate of the point.

4.1.3.2 getY() `int Coordinate::getY ( ) const`

Gets the y coordinate of the point.

Returns

The y coordinate of the point.

4.1.3.3 setX() `int Coordinate::setX ( int _x )`

Sets the x coordinate of the point.

Parameters

↩ _↩ *x*	The new x coordinate of the point.

Returns

The new x coordinate of the point.

4.1.3.4 setY() `int Coordinate::setY ( int _y )`

Sets the y coordinate of the point.

Parameters

↩ _↩ *y*	The new y coordinate of the point.

Returns

The new y coordinate of the point.

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

- src/Coordinate.hpp
- src/Coordinate.cpp

4.2 GameState Class Reference

The GameState class represents the state of the game.

`#include <GameState.hpp>`

Public Member Functions

- void writeMapFile (const string &filepath)

  Writes the game map file.
- GameState ()

  Default constructor that builds a random map.
- GameState (const GameState &gameState)

  Copy constructor for the GameState class.
- GameState (const string &filePath)

  Constructor for the GameState class.
- ~GameState ()

  Destructor for the GameState class.
- Coordinate getExit () const

  get the exit coordinate
- std::vector< Vehicule * > getListVehicule () const

  Returns the vector of vehicles in the game.
- Vehicule * getMainVehicule () const

  Returns the main vehicle in the game.
- void setExit (const Coordinate)

  Sets the exit coordinate.
- void addVehicule (Vehicule *toAdd)

  Add a vehicle to the std::vector< Vehicule> listVehicule.*
- void clearVehicule ()

  Delete and remove all vehicles from the std::vector< Vehicule> listVehicule.*
- int getMapSize ()

  Returns the size of the game map.
- void setMainVehicule (Vehicule *main)

  set the main vehicle of the game
- bool playMove (Vehicule *toMove, int distance)

  Plays a move.
- bool victory ()

  Determines whether the player has won the game.
- bool legalMove (const Vehicule *toMove, int distance)

  Determines whether a move is legal.
- bool stayInmap (Vehicule *v, int distance)

  Determines whether a vehicle can stay in the map.
- bool distanceTo (const Vehicule *toMove, vector< Vehicule * > frontVehicule, int deplacement)

  check if the vehicle can move over a certain distance (deplacement)
- void exportMapSvg (string)

  Exports the game map to a SVG file.
- string to_string ()

  Returns a string representation of the game state.
- bool isReachable (Coordinate objective)

  Returns if the objective is reachable.
- vector< Vehicule * > getListVehiculeOnLine (int x)

  Returns a vector of all vehicles on the same line as x.
- vector< Vehicule * > getListVehiculeOnColumn (int y)

  Returns a vector of all vehicles on the same column as y.
- GameState operator= (const GameState &gameState)

  Operator = for gamestate.
- GameState & operator= (GameState &other)

  Operator = for gamestate&.

Friends

- bool operator== (const GameState &game1, const GameState &game2)

  Overloads the equality operator for Gamestate objects.
- bool operator< (const GameState &game1, const GameState &game2)

  Overloads the less than operator for Gamestate objects.

4.2.1 Detailed Description

The GameState class represents the state of the game.

4.2.2 Constructor & Destructor Documentation

4.2.2.1 GameState() [1/3] GameState::GameState ( )

Default constructor that builds a random map.

4.2.2.2 GameState() [2/3] GameState::GameState (const GameState & *gameState*)

Copy constructor for the GameState class.

Parameters

gameState	A const reference to the GameState object to copy.

construteur par copie profonde

4.2.2.3 GameState() [3/3] GameState::GameState (const string & *filePath*)

Constructor for the GameState class.

Parameters

filePath	A string representing the file path to the game map.

4.2.2.4 ~GameState() GameState::~GameState ( )

Destructor for the GameState class.

4.2.3 Member Function Documentation

4.2.3.1 addVehicule() `void GameState::addVehicule ( Vehicule * toAdd )`

Add a vehicle to the std::vector<Vehicule*> listVehicule.

Parameters

toAdd	A pointer to the vehicle to add.

4.2.3.2 clearVehicule() `void GameState::clearVehicule ( )`

Delete and remove all vehicles from the std::vector<Vehicule*> listVehicule.

4.2.3.3 distanceTo() `bool GameState::distanceTo ( const Vehicule * toMove, vector< Vehicule * > frontVehicule, int deplacement )`

check if the vehicle can move over a certain distance (deplacement)

Parameters

toMove	A pointer to the vehicle to move.
frontVehicule	A vector of all vehicles in front of the vehicle to move.
deplacement	An integer representing the distance to move the vehicle.

4.2.3.4 exportMapSvg() `void GameState::exportMapSvg ( string filepath )`

Exports the game map to a SVG file.

Parameters

filepath	A string representing the file path to the game map.

4.2.3.5 getExit() `Coordinate GameState::getExit ( ) const`

get the exit coordinate

4.2.3.6 getListVehicule() `vector< Vehicule * > GameState::getListVehicule ( ) const`

Returns the vector of vehicles in the game.

Returns

A const reference to the vector of vehicles in the game.

4.2.3.7 getListVehiculeOnColumn() `vector< Vehicule * > GameState::getListVehiculeOnColumn ( int y )`

Returns a vector of all vehicles on the same column as y.

Parameters

y	An integer representing the column.

4.2.3.8 getListVehiculeOnLine() `vector< Vehicule * > GameState::getListVehiculeOnLine ( int x )`

Returns a vector of all vehicles on the same line as x.

Parameters

x	An integer representing the line.

4.2.3.9 getMainVehicule() `Vehicule * GameState::getMainVehicule ( ) const`

Returns the main vehicle in the game.

Returns

A const pointer to the main vehicle in the game.

4.2.3.10 getMapSize() `int GameState::getMapSize ( )`

Returns the size of the game map.

Returns

An integer representing the size of the game map.

4.2.3.11 isReachable() `bool GameState::isReachable ( Coordinate objective )`

Returns if the objective is reachable.

Parameters

objective	A Coordinate representing the objective

4.2.3.12 legalMove() `bool GameState::legalMove ( const Vehicule * toMove, int distance )`

Determines whether a move is legal.

Parameters

toMove	A pointer to the vehicle to move.
distance	An integer representing the distance to move the vehicle.

4.2.3.13 operator=() [1/2] `GameState GameState::operator= ( const GameState & gameState )`

Operator = for gamestate.

Parameters

gameState	A const reference to the GameState object to copy.

4.2.3.14 operator=() [2/2] `GameState & GameState::operator= ( GameState & other )`

Operator = for gamestate&.

Parameters

other	A reference to the GameState object to copy.

4.2.3.15 playMove()

```
bool GameState::playMove (
        Vehicule * toMove,
        int distance )
```

Plays a move.

Parameters

toMove	A pointer to the vehicle to move.
distance	An integer representing the distance to move the vehicle.

4.2.3.16 setExit()

```
void GameState::setExit (
        const Coordinate _exit )
```

Sets the exit coordinate.

Parameters

exit	A const reference to the new exit coordinate.

4.2.3.17 setMainVehicule()

```
void GameState::setMainVehicule (
        Vehicule * main )
```

set the main vehicle of the game

Parameters

main	A pointer to the new main vehicle of the game

4.2.3.18 stayInmap()

```
bool GameState::stayInmap (
        Vehicule * v,
        int distance )
```

Determines whether a vehicle can stay in the map.

Parameters

v	A pointer to the vehicle to move.
distance	An integer representing the distance to move the vehicle.

4.2.3.19 to_string() `std::string GameState::to_string ( )`

Returns a string representation of the game state.

4.2.3.20 victory() `bool GameState::victory ( )`

Determines whether the player has won the game.

Returns

A boolean value indicating whether the player has won the game.

4.2.3.21 writeMapFile() `void GameState::writeMapFile ( const string & filepath )`

Writes the game map file.

Parameters

filepath	A string representing the file path to the game map.
listVehicule	A vector of all vehicles in the game.

4.2.4 Friends And Related Function Documentation

4.2.4.1 operator< `bool operator< ( const GameState & game1, const GameState & game2 ) [friend]`

Overloads the less than operator for Gamestate objects.

Parameters

game1	The first Gamestate object to compare.
game2	The second Gamestate object to compare.

Returns

True if the first Gamestate object is less than the second one, false otherwise.

4.2.4.2 operator== `bool operator== ( const GameState & game1, const GameState & game2 ) [friend]`

Overloads the equality operator for Gamestate objects.

Parameters

game1	The first Gamestate object to compare.
game2	The second Gamestate object to compare.

Returns

True if the two Gamestate objects are equal, false otherwise.

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

- src/GameState.hpp
- src/GameState.cpp

4.3 PuzzleGenerator Class Reference

`#include <PuzzleGenerator.hpp>`

Public Member Functions

- PuzzleGenerator ()
- GameState dijkstraGeneration ()

  Generate a puzzle using a derived dijkstra algorithm.
- GameState naiveGeneration ()

  Generate n completely random puzzle and select the hardest one.

4.3.1 Constructor & Destructor Documentation

4.3.1.1 PuzzleGenerator() `PuzzleGenerator::PuzzleGenerator ( )`

4.3.2 Member Function Documentation

4.3.2.1 dijkstraGeneration() GameState PuzzleGenerator::dijkstraGeneration ( )

Generate a puzzle using a derived dijkstra algorithm.

Returns

A base game situation

4.3.2.2 naiveGeneration() GameState PuzzleGenerator::naiveGeneration ( )

Generate n completely random puzzle and select the hardest one.

Returns

A base game situation

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

- src/PuzzleGenerator.hpp
- src/PuzzleGenerator.cpp

4.4 Solver Class Reference

This class contains the BFS algorithm to find the shortest path to the objetive.

`#include <Solver.hpp>`

Public Member Functions

- std::vector< GameState * > BFS (GameState *start, Coordinate objetive=Coordinate(-1,-1))
  Returns a vector of Gamestate with the shortest path to the objetive.
- std::vector< GameState * > BFS_aux (GameState *start, Coordinate objetive, vector< tree_node * > &nodes)
  Auxiliar function for BFS.
- std::vector< GameState * > shortestPath (tree_node *end)
  Returns a vector of Gamestate with the shortest path to the objetive.
- vector< GameState * > neighbours (const GameState *_game)
  Returns a vector of Gamestate with all possible moves.

4.4.1 Detailed Description

This class contains the BFS algorithm to find the shortest path to the objetive.

4.4.2 Member Function Documentation

4.4.2.1 BFS() vector< GameState * > Solver::BFS (GameState * start, Coordinate objetive = Coordinate(-1,-1))

Returns a vector of Gamestate with the shortest path to the objetive.

Parameters

start	The initial GameState.
objetive	The objetive Coordinate (Optionnal (default : -1, -1)).

4.4.2.2 BFS_aux()

```
vector< GameState * > Solver::BFS_aux (
        GameState * start,
        Coordinate objetive,
        vector< tree_node * > & nodes )
```

Auxiliar function for BFS.

Parameters

start	The initial GameState.
objetive	The objetive Coordinate.
nodes	A vector of tree_node∗.

4.4.2.3 neighbours()

```
vector< GameState * > Solver::neighbours (
        const GameState * _game )
```

Returns a vector of Gamestate with all possible moves.

Returns

A vector of Gamestate with all possible moves.

4.4.2.4 shortestPath()

```
vector< GameState * > Solver::shortestPath (
        tree_node * end )
```

Returns a vector of Gamestate with the shortest path to the objetive.

Parameters

end	The last tree_node of the path.

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

- src/Solver.hpp
- src/Solver.cpp

4.5 tree_node Struct Reference

This struct is used to create a tree of GameStates.

`#include <Solver.hpp>`

Public Member Functions

- tree_node (tree_node *p, GameState *c)
- ~tree_node ()

Public Attributes

- tree_node * previous
- GameState * current

4.5.1 Detailed Description

This struct is used to create a tree of GameStates.

Previous is a pointer to the previous GameState. Current is a pointer to the current GameState.

4.5.2 Constructor & Destructor Documentation

4.5.2.1 tree_node()

```
tree_node::tree_node (
        tree_node * p,
        GameState * c ) [inline]
```

4.5.2.2 ~tree_node()

```
tree_node::~tree_node ( ) [inline]
```

4.5.3 Member Data Documentation

4.5.3.1 current

```
GameState* tree_node::current
```

4.5.3.2 previous `tree_node* tree_node::previous`

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

- src/Solver.hpp

4.6 Vehicule Class Reference

The Vehicule class represents a vehicle.

`#include <Vehicule.hpp>`

Public Member Functions

- Vehicule (int _x, int _y, int _size, bool _horizontal)
  Constructor for Vehicule.
- Vehicule (const Vehicule &vehicule)
- int getX () const
  Gets the x coordinate of the vehicle's location.
- int getY () const
  Gets the y coordinate of the vehicle's location.
- int getSize () const
  Gets the size of the vehicle.
- bool getHorizontal () const
  Gets whether the vehicle is horizontal or vertical.
- int setX (int _x)
  Sets the x coordinate of the vehicle's location.
- int setY (int _y)
  Sets the y coordinate of the vehicle's location.

4.6.1 Detailed Description

The Vehicule class represents a vehicle.

Vehicles have a location represented by a Coordinate object, a size, and a direction (horizontal or vertical).

4.6.2 Constructor & Destructor Documentation

4.6.2.1 Vehicule() [1/2]

```
Vehicule::Vehicule (
        int _x,
        int _y,
        int _size,
        bool _horizontal )
```

Constructor for Vehicule.

Parameters

_x	The x coordinate of the vehicle's location.
_y	The y coordinate of the vehicle's location.
_size	The size of the vehicle.
_horizontal	Whether the vehicle is horizontal (true) or vertical (false).

4.6.2.2 Vehicule() [2/2] `Vehicule::Vehicule ( const Vehicule & vehicule )`

4.6.3 Member Function Documentation

4.6.3.1 getHorizontal() `bool Vehicule::getHorizontal ( ) const`

Gets whether the vehicle is horizontal or vertical.

Returns

true if the vehicle is horizontal, false if it is vertical.

4.6.3.2 getSize() `int Vehicule::getSize ( ) const`

Gets the size of the vehicle.

Returns

The size of the vehicle.

4.6.3.3 getX() `int Vehicule::getX ( ) const`

Gets the x coordinate of the vehicle's location.

Returns

The x coordinate of the vehicle's location.

4.6.3.4 getY() `int Vehicule::getY ( ) const`

Gets the y coordinate of the vehicle's location.

Returns

The y coordinate of the vehicle's location.

4.6.3.5 setX() `int Vehicule::setX ( int _x )`

Sets the x coordinate of the vehicle's location.

Parameters

↩ _↩ x	The new x coordinate of the vehicle's location.

Returns

The new x coordinate of the vehicle's location.

4.6.3.6 setY() `int Vehicule::setY ( int _y )`

Sets the y coordinate of the vehicle's location.

Parameters

↩ _↩ y	The new y coordinate of the vehicle's location.

Returns

The new y coordinate of the vehicle's location.

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

- src/Vehicule.hpp
- src/Vehicule.cpp

5 File Documentation

5.1 README.md File Reference

5.2 src/Coordinate.cpp File Reference

This file contains the implementation of the Coordinate class.

```
#include "Coordinate.hpp"
```

5.2.1 Detailed Description

This file contains the implementation of the Coordinate class.

5.3 src/Coordinate.hpp File Reference

This file contains the declaration of the Coordinate class.

Classes

- class Coordinate

  The Coordinate class represents a point in two-dimensional space.

5.3.1 Detailed Description

This file contains the declaration of the Coordinate class.

5.4 src/GameState.cpp File Reference

This file contains the implementation of the GameState class.

```
#include <iostream>
#include <fstream>
#include <sys/stat.h>
#include <filesystem>
#include <algorithm>
#include "GameState.hpp"
#include "Solver.hpp"
```

Functions

- void checkvectorequal (int i, const Vehicule *toMove, vector< Vehicule * > &v)
- void removeToMove (const Vehicule *toMove, vector< Vehicule * > &v)
- bool operator== (const GameState &game1, const GameState &game2)
- bool operator< (const GameState &game1, const GameState &game2)

5.4.1 Detailed Description

This file contains the implementation of the GameState class.

5.4.2 Function Documentation

5.4.2.1 checkvectorequal()

```
void checkvectorequal (
        int i,
        const Vehicule * toMove,
        vector< Vehicule * > & v )
```

5.4.2.2 operator<()

```
bool operator< (
        const GameState & game1,
        const GameState & game2 )
```

Parameters

game1	The first Gamestate object to compare.
game2	The second Gamestate object to compare.

Returns

True if the first Gamestate object is less than the second one, false otherwise.

5.4.2.3 operator==()

```
bool operator== (
        const GameState & game1,
        const GameState & game2 )
```

Parameters

game1	The first Gamestate object to compare.
game2	The second Gamestate object to compare.

Returns

True if the two Gamestate objects are equal, false otherwise.

5.4.2.4 removeToMove()

```
void removeToMove (
        const Vehicule * toMove,
        vector< Vehicule * > & v )
```

5.5 src/GameState.hpp File Reference

This file contains the declaration of the GameState class, which represents the state of the game.

```
#include <vector>
#include <string>
#include "Vehicule.hpp"
#include "Coordinate.hpp"
```

Classes

- class GameState

  The GameState class represents the state of the game.

5.5.1 Detailed Description

This file contains the declaration of the GameState class, which represents the state of the game.

5.6 src/main.cpp File Reference

This file contains the resolution of the initial problem, generate maps, and solve them .

```
#include <iostream>
#include <string>
#include "utilities.hpp"
#include "PuzzleGenerator.hpp"
```

Functions

- void make_gif ()
- void parse_command_line (int argc, char **argv)
- int main (int argc, char **argv)

5.6.1 Detailed Description

This file contains the resolution of the initial problem, generate maps, and solve them .

5.6.2 Function Documentation

5.6.2.1 main()

```
int main (
        int argc,
        char ** argv )
```

5.6.2.2 make_gif()

```
void make_gif ( )
```

5.6.2.3 parse_command_line()

```
void parse_command_line (
        int argc,
        char ** argv )
```

5.7 src/main_test.cpp File Reference

This file contains the main function for the tests.

```
#include <iostream>
#include <string>
#include <cassert>
#include "utilities.hpp"
#include "PuzzleGenerator.hpp"
```

Macros

- #define OPERATOR_INF_TEST
- #define OPERATOR_EQUAL_TEST
- #define NEIGHBOURS_TEST
- #define GAME_TO_STRING_TEST
- #define VICTORY_TEST
- #define BFS_TEST
- #define MAP_GENERATOR_TEST_NAIVE
- #define MAP_GENERATOR_TEST_DIJKSTRA

Functions

- int main ()

5.7.1 Detailed Description

This file contains the main function for the tests.

5.7.2 Macro Definition Documentation

5.7.2.1 BFS_TEST `#define BFS_TEST`

5.7.2.2 GAME_TO_STRING_TEST `#define GAME_TO_STRING_TEST`

5.7.2.3 MAP_GENERATOR_TEST_DIJKSTRA `#define MAP_GENERATOR_TEST_DIJKSTRA`

5.7.2.4 MAP_GENERATOR_TEST_NAIVE `#define MAP_GENERATOR_TEST_NAIVE`

5.7.2.5 NEIGHBOURS_TEST `#define NEIGHBOURS_TEST`

5.7.2.6 OPERATOR_EQUAL_TEST `#define OPERATOR_EQUAL_TEST`

5.7.2.7 OPERATOR_INF_TEST `#define OPERATOR_INF_TEST`

5.7.2.8 VICTORY_TEST `#define VICTORY_TEST`

5.7.3 Function Documentation

5.7.3.1 main() `int main ( )`

5.8 src/PuzzleGenerator.cpp File Reference

This file contains the implementation of the PuzzleGenerator class.

```
#include "PuzzleGenerator.hpp"
#include "utilities.hpp"
```

5.8.1 Detailed Description

This file contains the implementation of the PuzzleGenerator class.

5.9 src/PuzzleGenerator.hpp File Reference

This file contains the declaration of the PuzzleGenerator class.

```
#include <iostream>
#include "Solver.hpp"
```

Classes

- class PuzzleGenerator

5.9.1 Detailed Description

This file contains the declaration of the PuzzleGenerator class.

5.10 src/Solver.cpp File Reference

This file contains the implementation of the Solver class.

```
#include <iostream>
#include <algorithm>
#include "Solver.hpp"
```

5.10.1 Detailed Description

This file contains the implementation of the Solver class.

5.11 src/Solver.hpp File Reference

This file contains the declaration of the Solver class and tree_node struct.

```
#include <vector>
#include <queue>
#include "GameState.hpp"
#include <map>
```

Classes

- struct tree_node

  This struct is used to create a tree of GameStates.
- class Solver

  This class contains the BFS algorithm to find the shortest path to the objetive.

5.11.1 Detailed Description

This file contains the declaration of the Solver class and tree_node struct.

5.12 src/utilities.cpp File Reference

This file contains the implementation of the utilities functions.

```
#include <filesystem>
#include <iostream>
#include "utilities.hpp"
```

Functions

- void write_gamestate (const vector< GameState * > &gamestate, const string &foldername, const string &filename)

  Write the GameState to files (txt and svg format).
- void destroy_vec_gamestate (vector< GameState * > &gamestate)

  Deletes the GameState objects in the vector and clears the vector.
- void printDVector (vector< vector< int >> vec)
- int getRandomNumber (int min, int max)

  Returns a random number between min and max.

5.12.1 Detailed Description

This file contains the implementation of the utilities functions.

5.12.2 Function Documentation

5.12.2.1 destroy_vec_gamestate()

```
void destroy_vec_gamestate (
        vector< GameState * > & gamestate )
```

Deletes the GameState objects in the vector and clears the vector.

Parameters

gamestate	A vector of pointers to GameState objects.

5.12.2.2 getRandomNumber()

```
int getRandomNumber (
        int min,
        int max )
```

Returns a random number between min and max.

5.12.2.3 printDVector()

```
void printDVector (
        vector< vector< int >> vec )
```

5.12.2.4 write_gamestate()

```
void write_gamestate (
        const vector< GameState * > & gamestate,
        const string & foldername = "",
        const string & filename = "" )
```

Write the GameState to files (txt and svg format).

Parameters

gamestate	A vector of pointers to GameState objects.
foldername	The name of the folder in which to write the file(s). Defaults to "".
filename	The name of the file to write. Defaults to "".

5.13 src/utilities.hpp File Reference

This file contains the declaration of the utilities functions.

```
#include "GameState.hpp"
```

Functions

- void write_gamestate (const vector< GameState * > &gamestate, const string &foldername="", const string &filename="")

  Write the GameState to files (txt and svg format).
- void printDVector (vector< vector< int >>)
- void destroy_vec_gamestate (vector< GameState * > &gamestate)

  Deletes the GameState objects in the vector and clears the vector.
- int getRandomNumber (int min, int max)

  Returns a random number between min and max.

5.13.1 Detailed Description

This file contains the declaration of the utilities functions.

5.13.2 Function Documentation

5.13.2.1 destroy_vec_gamestate() `void destroy_vec_gamestate ( vector< GameState * > & gamestate )`

Deletes the GameState objects in the vector and clears the vector.

Parameters

gamestate	A vector of pointers to GameState objects.

5.13.2.2 getRandomNumber() `int getRandomNumber (`

```
int min,
int max )
```

Returns a random number between min and max.

5.13.2.3 printDVector() void printDVector (

```
vector< vector< int >> vec )
```

5.13.2.4 write_gamestate() void write_gamestate (

```
const vector< GameState * > & gamestate,
const string & foldername = "",
const string & filename = "" )
```

Write the GameState to files (txt and svg format).

Parameters

gamestate	A vector of pointers to GameState objects.
foldername	The name of the folder in which to write the file(s). Defaults to "".
filename	The name of the file to write. Defaults to "".

5.14 src/Vehicule.cpp File Reference

This file contains the implementation of the Vehicule class.

```
#include <cstdlib>
#include <iostream>
#include "Vehicule.hpp"
```

5.14.1 Detailed Description

This file contains the implementation of the Vehicule class.

5.15 src/Vehicule.hpp File Reference

This file contains the declaration of the Vehicule class.

```
#include "Coordinate.hpp"
```

Classes

- class Vehicule

  The Vehicule class represents a vehicle.

5.15.1 Detailed Description

This file contains the declaration of the Vehicule class.