Task 6

Generated by Doxygen 1.9.8

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 documented files with brief descriptions:

Chapter 3

Class Documentation

3.1 Path Class Reference

`#include <Path.h>`

Public Member Functions

- Path (const Path &oldPath, const State &newState)

  Constructs a new path by appending a new state to an existing path.
- ~Path ()
- bool isGoalReached () const

  Determines whether the goal state has been reached in the path.
- int getHeuristic () const

  Gets the heuristic value of the current state in the path.
- vector< Path > generatePaths ()

  Expands the current path by generating successor paths.
- int getSize () const

  Gets the size of the path.
- void print () const

  Displays the states in the path.
- bool equals (Path obj)

  Determines whether this path is equal to another path.
- bool isBadPath () const

  Determines if the path is potentially suboptimal or inefficient.

Static Public Member Functions

- static Path getInitPath ()

  Creates a new path with the initial state only.

Friends

- bool operator> (const Path &p1, const Path &p2)

  Compares two paths based on their heuristic values.

3.1.1 Detailed Description

Represents a path consisting of multiple states in a hill-climbing search.

The Path class encapsulates a sequence of states traversed during a hill-climbing search algorithm. It provides methods to manipulate and analyze the path, such as adding new states, checking if the goal state has been reached, computing the heuristic value of the path, generating successor paths, and determining the size of the path.

3.1.2 Constructor & Destructor Documentation

3.1.2.1 Path()

```
Path::Path (
            const Path & oldPath,
            const State & newState )
```

Constructs a new path by appending a new state to an existing path.

Constructor for creating a new path by extending an existing path with a new state.

Parameters

oldPath	The previous path to be extended.
newState	The new state to be added to the path.

This constructor creates a new path by copying the states from the provided old path and then appending the new state to it. It constructs a new instance of the Path class based on the combination of the old path and the new state.

3.1.2.2 ~Path()

```
Path::~Path ( )
```

Destructor.

3.1.3 Member Function Documentation

3.1.3.1 equals()

```
bool Path::equals (
            Path obj )
```

Determines whether this path is equal to another path.

Checks if this path is equal to another path.

Parameters

obj	The path to compare with.

Returns

True if the paths are equal, false otherwise.

3.1.3.2 generatePaths()

```
vector< Path > Path::generatePaths ( )
```

Expands the current path by generating successor paths.

Generates successor paths by adding possible next states to the current path.

Returns

A vector of successor paths generated from the current path.

3.1.3.3 getHeuristic()

```
int Path::getHeuristic ( ) const
```

Gets the heuristic value of the current state in the path.

Retrieves the heuristic value of the current state in the path.

Returns

The heuristic value of the current state.

3.1.3.4 getInitPath()

```
Path Path::getInitPath ( )  [static]
```

Creates a new path with the initial state only.

Generates a path containing only the initial state.

Returns

A path containing the initial state only.

3.1.3.5 getSize()

```
int Path::getSize ( ) const
```

Gets the size of the path.

Retrieves the number of states in the path.

Note

The size = the number of moves + 1.

Returns

The number of states in the path.

3.1.3.6 isBadPath()

```
bool Path::isBadPath ( ) const
```

Determines if the path is potentially suboptimal or inefficient.

Checks if the path exhibits characteristics of a potentially inefficient or suboptimal solution path.

Returns

True if the path is considered potentially inefficient or suboptimal, false otherwise.

This method analyzes various aspects of the path to determine if it exhibits characteristics of inefficiency or suboptimal. It checks if the path length exceeds a threshold and if the heuristic value increases at any point along the path, which may indicate a non-ideal solution path.

3.1.3.7 isGoalReached()

```
bool Path::isGoalReached ( ) const
```

Determines whether the goal state has been reached in the path.

Checks if the goal state has been reached in the path.

Returns

True if the goal state has been reached, false otherwise.

3.1.3.8 print()

```
void Path::print ( ) const
```

Displays the states in the path.

Prints the states in the path.

3.1.4 Friends And Related Symbol Documentation

3.1.4.1 operator>

```
bool operator> (
            const Path & p1,
            const Path & p2 )  [friend]
```

Compares two paths based on their heuristic values.

Overloaded greater than operator for comparing paths based on heuristic values.

Parameters

p1	The first path to compare.
p2	The second path to compare.

Returns

True if the heuristic value of p1 is greater than that of p2, false otherwise.

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

- Path.h
- Path.cpp

3.2 State Class Reference

`#include <State.h>`

Public Member Functions

- State ()
- State (const vector< char > &_state)
- vector< char > getState () const
- int getHeuristic () const
- bool isGoal () const
- vector< State > generateStates ()

  Generates all possible successor states reachable from the current state by moving knights.
- void **print** () const

  Prints the current state of the chessboard.
- bool equals (State state)

  This method determines whether the provided state is identical to the current state.

3.2.1 Detailed Description

The State class represents a configuration of knights on a 4x3 chessboard. It encapsulates the current state of the board along with methods for calculating the heuristic score, generating possible successor states, and performing other operations related to the problem of exchanging knights.

3.2.2 Constructor & Destructor Documentation

3.2.2.1 State() [1/2]

```
State::State ( )  [explicit]
```

Constructor method to initialize the state of the chessboard to its initial configuration. This constructor sets up the initial state of the chessboard with three white knights ("W') at the top row and three black knights ('B') at the bottom row. The empty squares are represented by 'E'.

The initial state of the chessboard:

```
W W W ~

E E E ~

E E E ~

B B B
```

3.2.2.2 State() [2/2]

```
State::State (
            const vector< char > & _state )  [inline], [explicit]
```

Constructor method to initialize the state of the chessboard with a custom configuration.

This constructor sets up the state of the chessboard using the provided vector of characters. Each character represents the occupant of a square on the chessboard ('W' for white knight, 'B' for black knight, and 'E' for an empty square).

Parameters

_state	A vector of characters representing the configuration of the chessboard. 'W' represents a white knight, 'B' represents a black knight, and 'E' represents an empty square.

3.2.3 Member Function Documentation

3.2.3.1 equals()

```
bool State::equals (
            State state )
```

This method determines whether the provided state is identical to the current state.

Parameters

state	The state to compare with the current state.

Returns

True if the states are equal, false otherwise.

3.2.3.2 generateStates()

```
vector< State > State::generateStates ( )
```

Generates all possible successor states reachable from the current state by moving knights.

This method iterates through each knight on the chessboard, generates all possible moves for each knight using the `moveChess` method, and adds the resulting states to the list of generated states. It considers both white and black knights separately and incorporates the `moveChess` method to compute the possible moves for each knight.

Returns

A vector of State objects representing all possible successor states reachable from the current state.

See also

State::moveChess()

3.2.3.3 getHeuristic()

```
int State::getHeuristic ( ) const
```

Getter method to retrieve the heuristic score of the current state.

Returns

The heuristic score of the current state.

3.2.3.4 getState()

```
vector< char > State::getState ( ) const
```

Getter method to retrieve the current state of the chessboard.

Returns

A vector of characters representing the current configuration of the chessboard.

3.2.3.5 isGoal()

```
bool State::isGoal ( ) const
```

Method to check if the current state is the goal state. This method evaluates whether the current state of the chessboard matches the goal state, where the white knights are positioned at the bottom row and the black knights at the top row.

Returns

True if the current state is the goal state, false otherwise.

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

- State.h
- State.cpp

Chapter 4

File Documentation

4.1 Path.h

```
00001 #ifndef INC_07_HILL_CLIMBING_PATH_H
00002 #define INC_07_HILL_CLIMBING_PATH_H
00003
00004 #include <iostream>
00005 #include "State.h"
00006 #include <random>
00007
00008 using namespace std;
00009
00018 class Path {
00019 private:
00023     vector<State> path;
00024
00033     Path();
00034
00035 public:
00036
00049     Path(const Path &oldPath, const State &newState);
00050
00054     ~Path();
00055
00063     [[nodiscard]] bool isGoalReached() const;
00064
00072     [[nodiscard]] int getHeuristic() const;
00073
00081     vector<Path> generatePaths();
00082
00092     [[nodiscard]] int getSize() const;
00093
00101      static Path getInitPath();
00102
00113     friend bool operator>(const Path &p1, const Path &p2);
00114
00120     void print() const;
00121
00122
00132     bool equals(Path obj);
00133
00147     [[nodiscard]] bool isBadPath() const;
00148 };
00149
00150
00151 #endif //INC_07_HILL_CLIMBING_PATH_H
```

4.2 State.h

```
00001 #ifndef INC_07_HILL_CLIMBING_STATE_H
00002 #define INC_07_HILL_CLIMBING_STATE_H
00003
00004 #include <vector>
00005 #include <iostream>
00006 #include <algorithm>
00007
00008
```

```
00009 using namespace std;
00010
00017 class State {
00018 private:
00024     vector<char> state;
00030     int heuristic{};
00031
00053     bool isValidMove(int row, int col, int actualCol, int actualRow);
00054
00065     vector<State> moveChess(int position, vector<char> currentState);
00066
00113     void calculateHeuristic();
00114
00115
00116 public:
00132     explicit State();
00133
00143     explicit State(const vector<char> &_state) : heuristic(0) {
00144         for (char idx: _state) {
00145             state.push_back(idx);
00146         }
00147     }
00148
00154     [[nodiscard]] vector<char> getState() const;
00155
00161     [[nodiscard]] int getHeuristic() const;
00162
00170     [[nodiscard]] bool isGoal() const;
00171
00184     vector<State> generateStates();
00185
00189     void print() const;
00190
00198     bool equals(State state);
00199 };
00200
00201 #endif //INC_07_HILL_CLIMBING_STATE_H
```

Index