srcl_ctrl: planning

Generated by Doxygen 1.8.6

Mon Apr 11 2016 16:28:43

Contents

Chapter 1

Main Page

1. Graph

a. Design

Graph is a type of data structure that can be used to represent pairwise relations between objects. In this library, a graph is modeled as a collection of vertices and edges. The relations between those concepts are shown as follows.

- Graph
  - Vertex 1
    - Edge 1_1
    - Edge 1_2
    - ...
  - Vertex 2
    - Edge 2_1
    - Edge 2_2
    - ...
  - ...
  - Vertex n
    - Edge n_1
    - Edge n_2
    - ...
    - Edge n_m

A minimal implementation of Graph consists of a list of vertices, each of which has an unique ID and a list of edges. For path finding in the graph, we have attributes, such as cost, in the edge and search function, such as A$*$ search can be provided with the graph. These attributes and methods are generic for all graphs.

In different contexts, we usually want to add non-generic attributes to the vertex so that it can be meaningful for the application. For example when we use a graph to represent a square grid, a square cell can be regarded as a vertex, and the connectivities of a cell with its neighbour cells can be represented as edges. In this case, a square cell (vertex) may have attributes such as its location in the grid and its occupancy type (cell filled with obstacle or not). Such attributes can be very different across different applications, thus they are not modeled directly in the "Vertex" data structure. Instead, the "additional information" is packed into a separate object (called a **node** in this design) and we associate a node with a vertex uniquely.

b. Implementation

There are 3 class templates defined: **Graph**, **Vertex**, **Edge**. The use of template enables us to associate different types of "node" to a vertex, without modifying the code of the aforementioned 3 classes. In other words, the Graph, Vertex and Edge all have a "type", which is determined by the type of node we want to associate with the vertex. With the current implementation, the node has to be defined as a class or struct. An unique ID must be assigned to each node before we use them to construct a graph. In the graph data structure, the vertex has the same ID with the node it's associated with. This is for solely for easy indexing to find one with the other.

Here is an example to use the templates.

I. We first define a node type we want to use for constructing the graph.

```
struct ExampleNode{
    ExampleNode(uint64_t id):node_id_(id){}

    const uint64_t node_id_;

    // you can add more attributes here
};
```

II. Then we can create a few objects of class ExampleNode

```
std::vector<ExampleNode*> nodes;

// create nodes, with id from 0 to 3
for(int i = 0; i < 4; i++) {
    nodes.push_back(new ExampleNode(i));
}
```

III. Now use those nodes to construct a graph. Note that the graph is of type ExampleNode in this example.

```
// create a graph of type ExampleNode
Graph<ExampleNode> graph;

graph.AddEdge(nodes[0], nodes[1], 1.0);
graph.AddEdge(nodes[0], nodes[2], 1.5);
graph.AddEdge(nodes[1], nodes[2], 2.0);
graph.AddEdge(nodes[2], nodes[3], 2.5);
```

IV. Now you've got a graph. You can print all edges of this graph in the following way

```
auto all_edges = graph.GetGraphEdges();

for(auto e : all_edges)
    e.PrintEdge();
```

You will get the output

```
Edge: start - 0 , end - 1 , cost - 1
Edge: start - 0 , end - 2 , cost - 1.5
Edge: start - 1 , end - 2 , cost - 2
Edge: start - 2 , end - 3 , cost - 2.5
```

c. Memory Management

When a Graph object goes out of scope, its destructor function will recycle memory allocated for this its vertices and edges. However, **the graph doesn't recycle memoery allocated for the node that each vertex is associated with**. In the square grid example, the graph doesn't assume the square grid also becomes useless when the graph itself is destructed. The **square grid** should be responsible for recycling the memory allocated for its square cells when it becomes of no use. Thus in the above simple example, we will need to do the following operation to free the memory at the end.

```
// delete objects of ExampleNode
for(auto e : nodes)
        delete e;
```

d. Notes

- You may have noticed that when constructing a graph, you don't need to explicitly create objects of "Vertex". By calling member function **AddEdge(src_node, dst_node, cost)** of the graph, vertices are created and associated with the according node internally.

- There are two views of the graph data structure. When constructing the graph (bottom-up view), the nodes are manipulated directly and vertices are handled implicitly. When using the graph (top-down view) for path search, vertices are the the entities you're directly interacting with and the nodes they associate with are of less interest.

- An detailed example of using the graph for path search can be found in "apps/example.cpp". The work flow is shown as follows.

```
// create a graph from square grid
Graph<SquareCell>* graph = GraphBuilder::BuildFromSquareGrid(grid,true);

// specify search start and finish vertex
Vertex<SquareCell>* start_vertex = graph->GetVertexFromID(0);
Vertex<SquareCell>* finish_vertex = graph->GetVertexFromID(1);

// perform A* search and get a vector of Vertics as the search result
std::vector<Vertex<SquareCell>*> path = graph->AStarSearch(start_vertex,finish_vertex);
```

Known Issues

- A∗ search algorithm currently only works with nodes that have attribute "location_". This attribute is used to calculate heuristic cost. A more general method may need to be implemented in the future.

Chapter 2

README

Reference:

Graph

- http://www.geeksforgeeks.org/graph-and-its-representations/
- https://www.topcoder.com/community/data-science/data-science-tutorials/power-up-c-wi
- https://www.topcoder.com/community/data-science/data-science-tutorials/power-up-c-wi
- http://stackoverflow.com/questions/17473753/c11-return-value-optimization-or-move/17
- http://lafstern.org/matt/col1.pdf

Search

- http://homepages.abdn.ac.uk/f.guerin/pages/teaching/CS1013/practicals/a-StarTutorial.htm
- http://www.cppblog.com/mythit/archive/2009/04/19/80492.aspx (Chinese Version)
- http://users.cis.fiu.edu/~weiss/adspc++2/code/
- http://heyes-jones.com/astar.php
- http://stackoverflow.com/questions/11912736/c-a-star-implementation-determining-whet
- http://stackoverflow.com/questions/10394508/which-std-container-to-use-in-a-algorith
- https://github.com/justinhj/astar-algorithm-cpp
- http://code.activestate.com/recipes/577457-a-star-shortest-path-algorithm/
- http://theory.stanford.edu/~amitp/GameProgramming/

C++

- http://stackoverflow.com/questions/642229/why-do-i-need-to-use-typedef-typename-in-g
- http://stackoverflow.com/questions/8584431/why-is-the-keyword-typename-needed-before

Chapter 3

Namespace Index

3.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 4

Class Index

4.1 Class List

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

Chapter 5

File Index

5.1 File List

Here is a list of all files with brief descriptions:

Chapter 6

Namespace Documentation

6.1 srcl_ctrl Namespace Reference

Classes

- struct PriorityQueue

  A simple priority queue structure used as A$*$ open list.
- class AStar

  A$*$ search algorithm.
- struct ExampleNode

  An example node that can be associated with a vertex.
- class Graph

  A graph data structure template.
- class Edge

  An edge data structure template.
- class Vertex

  A vertex data structure template.

Chapter 7

Class Documentation

7.1 srcl_ctrl::AStar< GraphVertexType > Class Template Reference

A∗ search algorithm.

`#include <astar.h>`

Public Member Functions

- AStar ()
- ∼AStar ()
- std::vector< GraphVertexType ∗ > Search (GraphVertexType ∗start, GraphVertexType ∗goal)

Private Member Functions

- double CalcHeuristic (GraphVertexType ∗vertex_a, GraphVertexType ∗vertex_b)

7.1.1 Detailed Description

template<typename GraphVertexType>class srcl_ctrl::AStar< GraphVertexType >

A∗ search algorithm.

7.1.2 Constructor & Destructor Documentation

7.1.2.1 template<typename GraphVertexType> srcl_ctrl::AStar< GraphVertexType >::AStar ( ) `[inline]`

7.1.2.2 template<typename GraphVertexType> srcl_ctrl::AStar< GraphVertexType >::∼AStar ( ) `[inline]`

7.1.3 Member Function Documentation

7.1.3.1 template<typename GraphVertexType> double srcl_ctrl::AStar< GraphVertexType >::CalcHeuristic (GraphVertexType ∗ *vertex_a,* GraphVertexType ∗ *vertex_b*) `[inline]`, `[private]`

7.1.3.2 template<typename GraphVertexType> std::vector<GraphVertexType∗> srcl_ctrl::AStar< GraphVertexType >::Search (GraphVertexType ∗ *start,* GraphVertexType ∗ *goal*) `[inline]`

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

- /home/rdu/Workspace/srcl_robot_suite/srcl_ctrl/planning/src/graph/astar.h

7.2 srcl_ctrl::Edge< EdgeVertexType > Class Template Reference

An edge data structure template.

`#include <vertex.h>`

Public Member Functions

- Edge (EdgeVertexType ∗src=nullptr, EdgeVertexType ∗dst=nullptr, double c=0.0)
- bool operator== (const Edge< EdgeVertexType > other)
- void PrintEdge ()

Public Attributes

- EdgeVertexType ∗ src_
- EdgeVertexType ∗ dst_
- double cost_

7.2.1 Detailed Description

template<typename EdgeVertexType>class srcl_ctrl::Edge< EdgeVertexType >

An edge data structure template.

7.2.2 Constructor & Destructor Documentation

7.2.2.1 template<typename EdgeVertexType> srcl_ctrl::Edge< EdgeVertexType >::Edge (EdgeVertexType ∗ *src* = `nullptr`, EdgeVertexType ∗ *dst* = `nullptr`, double *c* = `0.0`) `[inline]`

7.2.3 Member Function Documentation

7.2.3.1 template<typename EdgeVertexType> bool srcl_ctrl::Edge< EdgeVertexType >::operator== (const Edge< EdgeVertexType > *other*) `[inline]`

7.2.3.2 template<typename EdgeVertexType> void srcl_ctrl::Edge< EdgeVertexType >::PrintEdge ( ) `[inline]`

7.2.4 Member Data Documentation

7.2.4.1 template<typename EdgeVertexType> double srcl_ctrl::Edge< EdgeVertexType >::cost_

7.2.4.2 template<typename EdgeVertexType> EdgeVertexType∗ srcl_ctrl::Edge< EdgeVertexType >::dst_

7.2.4.3 template<typename EdgeVertexType> EdgeVertexType∗ srcl_ctrl::Edge< EdgeVertexType >::src_

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

- /home/rdu/Workspace/srcl_robot_suite/srcl_ctrl/planning/src/graph/vertex.h

7.3 srcl_ctrl::ExampleNode Struct Reference

An example node that can be associated with a vertex.

```
#include <graph.h>
```

Public Member Functions

- ExampleNode (uint64_t id)

Public Attributes

- const uint64_t node_id_

7.3.1 Detailed Description

An example node that can be associated with a vertex.

This node can be either a "struct" or a "class", only need to provide the node_id_ attribute.

7.3.2 Constructor & Destructor Documentation

7.3.2.1 srcl_ctrl::ExampleNode::ExampleNode (uint64_t *id*) `[inline]`

7.3.3 Member Data Documentation

7.3.3.1 const uint64_t srcl_ctrl::ExampleNode::node_id_

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

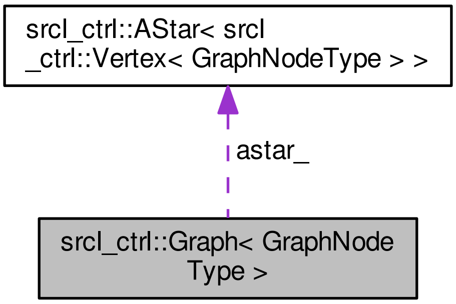
- /home/rdu/Workspace/srcl_robot_suite/srcl_ctrl/planning/src/graph/graph.h

7.4 srcl_ctrl::Graph< GraphNodeType > Class Template Reference

A graph data structure template.

```
#include <graph.h>
```

Collaboration diagram for srcl_ctrl::Graph< GraphNodeType >:



Public Member Functions

- Graph ()

  Graph constructor.
- ∼Graph ()

  Graph destructor.
- void AddEdge (GraphNodeType ∗src_node, GraphNodeType ∗dst_node, double cost)

  This function is used to create a graph by adding edges connecting two nodes.
- std::vector< Vertex
  < GraphNodeType > ∗ > GetGraphVertices ()

  This functions is used to access all vertices of a graph.
- std::vector< Edge< Vertex
  < GraphNodeType > > > GetGraphEdges ()

  This functions is used to access all edges of a graph.
- Vertex< GraphNodeType > ∗ GetVertexFromID (uint64_t vertex_id)

  This function return the vertex with specified id.
- std::vector< Vertex
  < GraphNodeType > ∗ > AStarSearch (Vertex< GraphNodeType > ∗start, Vertex< GraphNodeType > ∗goal)

  Perform A∗ Search and return a path represented by a serious of vertices.

Private Member Functions

- Vertex< GraphNodeType > ∗ GetVertex (GraphNodeType ∗vertex_node)

  This function checks if a vertex already exists in the graph.
- void ResetGraphVertices ()

  This function is used to reset the vertices for a new search.

Private Attributes

- std::map< uint64_t, Vertex
  < GraphNodeType > ∗ > vertex_map_
- AStar< Vertex< GraphNodeType > > astar_

7.4.1 Detailed Description

template<typename GraphNodeType>class srcl_ctrl::Graph< GraphNodeType >

A graph data structure template.

7.4.2 Constructor & Destructor Documentation

7.4.2.1 template<typename GraphNodeType > srcl_ctrl::Graph< GraphNodeType >::Graph ( ) [inline]

Graph constructor.

7.4.2.2 template<typename GraphNodeType > srcl_ctrl::Graph< GraphNodeType >::~Graph ( ) [inline]

Graph destructor.

Graph class is only responsible for the memory recycling of Vertex and Edge objects. The node, such as a quadtree node or a square cell, which each vertex is associated with needs to be recycled separately, for example by the quadtree/square_grid class.

7.4.3 Member Function Documentation

7.4.3.1 template<typename GraphNodeType > void srcl_ctrl::Graph< GraphNodeType >::AddEdge (GraphNodeType * *src_node*, GraphNodeType * *dst_node*, double *cost*) [inline]

This function is used to create a graph by adding edges connecting two nodes.

7.4.3.2 template<typename GraphNodeType > std::vector<Vertex<GraphNodeType>*> srcl_ctrl::Graph< GraphNodeType >::AStarSearch (Vertex< GraphNodeType > * *start*, Vertex< GraphNodeType > * *goal*) [inline]

Perform A* Search and return a path represented by a serious of vertices.

7.4.3.3 template<typename GraphNodeType > std::vector<Edge<Vertex<GraphNodeType> > > srcl_ctrl::Graph< GraphNodeType >::GetGraphEdges ( ) [inline]

This functions is used to access all edges of a graph.

7.4.3.4 template<typename GraphNodeType > std::vector<Vertex<GraphNodeType>*> srcl_ctrl::Graph< GraphNodeType >::GetGraphVertices ( ) [inline]

This functions is used to access all vertices of a graph.

7.4.3.5 template<typename GraphNodeType > Vertex<GraphNodeType>* srcl_ctrl::Graph< GraphNodeType >::GetVertex (GraphNodeType * *vertex_node*) [inline], [private]

This function checks if a vertex already exists in the graph.

If yes, the functions returns the index of the existing vertex, otherwise it creates a new vertex.

7.4.3.6 template< typename GraphNodeType > Vertex< GraphNodeType >* srcl_ctrl::Graph< GraphNodeType >::GetVertexFromID (uint64_t *vertex_id*) [inline]

This function return the vertex with specified id.

7.4.3.7 template< typename GraphNodeType > void srcl_ctrl::Graph< GraphNodeType >::ResetGraphVertices ( ) [inline], [private]

This function is used to reset the vertices for a new search.

7.4.4 Member Data Documentation

7.4.4.1 template< typename GraphNodeType > AStar< Vertex< GraphNodeType > > srcl_ctrl::Graph< GraphNodeType >::astar_ [private]

7.4.4.2 template< typename GraphNodeType > std::map< uint64_t, Vertex< GraphNodeType >*> srcl_ctrl::Graph< GraphNodeType >::vertex_map_ [private]

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

- /home/rdu/Workspace/srcl_robot_suite/srcl_ctrl/planning/src/graph/graph.h

7.5 srcl_ctrl::PriorityQueue< T, Number > Struct Template Reference

A simple priority queue structure used as A∗ open list.

`#include <astar.h>`

Public Types

- typedef std::pair< Number, T > PQElement

Public Member Functions

- bool empty () const
- void put (T item, Number priority)
- T get ()

Public Attributes

- std::priority_queue< PQElement,
  std::vector< PQElement >
  , std::greater< PQElement > > elements

7.5.1 Detailed Description

template<typename T, typename Number = double>struct srcl_ctrl::PriorityQueue< T, Number >

A simple priority queue structure used as A∗ open list.

7.5.2 Member Typedef Documentation

7.5.2.1 template<typename T, typename Number = double> typedef std::pair<Number, T> **srcl_ctrl::PriorityQueue**< T, Number >::**PQElement**

7.5.3 Member Function Documentation

7.5.3.1 template<typename T, typename Number = double> bool **srcl_ctrl::PriorityQueue**< T, Number >::empty ( ) const [inline]

7.5.3.2 template<typename T, typename Number = double> T **srcl_ctrl::PriorityQueue**< T, Number >::get ( ) [inline]

7.5.3.3 template<typename T, typename Number = double> void **srcl_ctrl::PriorityQueue**< T, Number >::put (T *item*, Number *priority*) [inline]

7.5.4 Member Data Documentation

7.5.4.1 template<typename T, typename Number = double> std::priority_queue<**PQElement**, std::vector<**PQElement**>, std::greater<**PQElement**> > **srcl_ctrl::PriorityQueue**< T, Number >::elements

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

- /home/rdu/Workspace/srcl_robot_suite/srcl_ctrl/planning/src/graph/astar.h

7.6 srcl_ctrl::Vertex< VertexNodeType > Class Template Reference

A vertex data structure template.

```
#include <vertex.h>
```

Public Member Functions

- Vertex (VertexNodeType ∗node=nullptr)
- ∼Vertex ()
- bool operator== (const Vertex< VertexNodeType > other)
- void ClearVertexSearchInfo ()
- double GetEdgeCost (Vertex< VertexNodeType > ∗dst_node)

Public Attributes

- VertexNodeType ∗ node_
- uint64_t vertex_id_
- std::vector< Edge< Vertex< VertexNodeType > > > edges_
- bool is_checked_
- bool is_in_openlist_
- double f_astar_
- double g_astar_
- double h_astar_
- Vertex< VertexNodeType > ∗ search_parent_

7.6.1 Detailed Description

template<typename VertexNodeType>class srcl_ctrl::Vertex< VertexNodeType >

A vertex data structure template.

7.6.2 Constructor & Destructor Documentation

7.6.2.1 template<typename VertexNodeType> srcl_ctrl::Vertex< VertexNodeType >::Vertex (VertexNodeType ∗ *node* = `nullptr`) `[inline]`

7.6.2.2 template<typename VertexNodeType> srcl_ctrl::Vertex< VertexNodeType >::∼Vertex ( ) `[inline]`

7.6.3 Member Function Documentation

7.6.3.1 template<typename VertexNodeType> void srcl_ctrl::Vertex< VertexNodeType >::ClearVertexSearchInfo ( ) `[inline]`

7.6.3.2 template<typename VertexNodeType> double srcl_ctrl::Vertex< VertexNodeType >::GetEdgeCost (Vertex< VertexNodeType > ∗ *dst_node*) `[inline]`

7.6.3.3 template<typename VertexNodeType> bool srcl_ctrl::Vertex< VertexNodeType >::operator== (const Vertex< VertexNodeType > *other*) `[inline]`

7.6.4 Member Data Documentation

7.6.4.1 template<typename VertexNodeType> std::vector<Edge<Vertex<VertexNodeType> > > srcl_ctrl::Vertex< VertexNodeType >::edges_

7.6.4.2 template<typename VertexNodeType> double srcl_ctrl::Vertex< VertexNodeType >::f_astar_

7.6.4.3 template<typename VertexNodeType> double srcl_ctrl::Vertex< VertexNodeType >::g_astar_

7.6.4.4 template<typename VertexNodeType> double srcl_ctrl::Vertex< VertexNodeType >::h_astar_

7.6.4.5 template<typename VertexNodeType> bool srcl_ctrl::Vertex< VertexNodeType >::is_checked_

7.6.4.6 template<typename VertexNodeType> bool srcl_ctrl::Vertex< VertexNodeType >::is_in_openlist_

7.6.4.7 template<typename VertexNodeType> VertexNodeType∗ srcl_ctrl::Vertex< VertexNodeType >::node_

7.6.4.8 template<typename VertexNodeType> Vertex<VertexNodeType>∗ srcl_ctrl::Vertex< VertexNodeType >::search_parent_

7.6.4.9 template<typename VertexNodeType> uint64_t srcl_ctrl::Vertex< VertexNodeType >::vertex_id_

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

- /home/rdu/Workspace/srcl_robot_suite/srcl_ctrl/planning/src/graph/vertex.h

Chapter 8

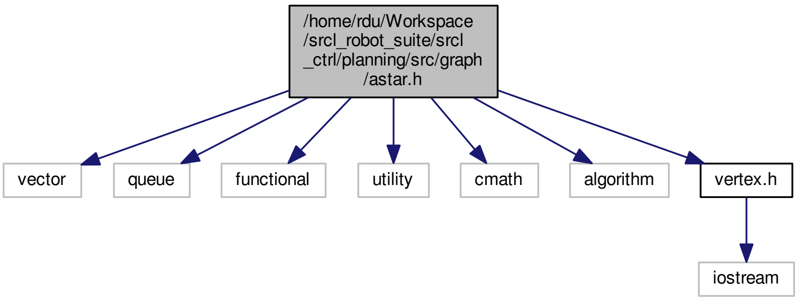
File Documentation

8.1 mainpage.md File Reference

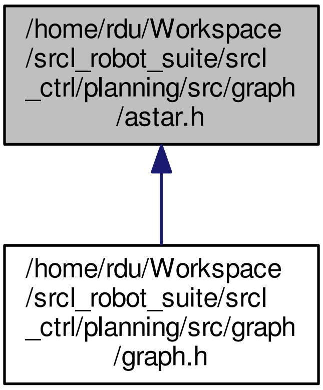
8.2 /home/rdu/Workspace/srcl_robot_suite/srcl_ctrl/planning/src/graph/astar.h File Reference

```
#include <vector>
#include <queue>
#include <functional>
#include <utility>
#include <cmath>
#include <algorithm>
#include <vertex.h>
```

Include dependency graph for astar.h:



This graph shows which files directly or indirectly include this file:


/home/rdu/Workspace /srcl_robot_suite/srcl _ctrl/planning/src/graph /astar.h
/home/rdu/Workspace /srcl_robot_suite/srcl _ctrl/planning/src/graph /graph.h

Classes

- struct srcl_ctrl::PriorityQueue< T, Number >

  A simple priority queue structure used as A∗ open list.

- class srcl_ctrl::AStar< GraphVertexType >

  A∗ search algorithm.

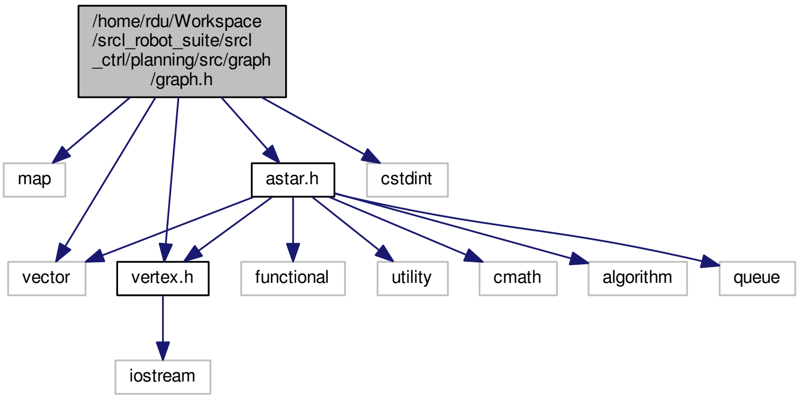
Namespaces

- srcl_ctrl

8.3 /home/rdu/Workspace/srcl_robot_suite/srcl_ctrl/planning/src/graph/graph.h File Reference

```
#include <map>
#include <vector>
#include <cstdint>
#include <vertex.h>
#include <astar.h>
```

Include dependency graph for graph.h:



Classes

- struct srcl_ctrl::ExampleNode

  An example node that can be associated with a vertex.

- class srcl_ctrl::Graph< GraphNodeType >

  A graph data structure template.

Namespaces

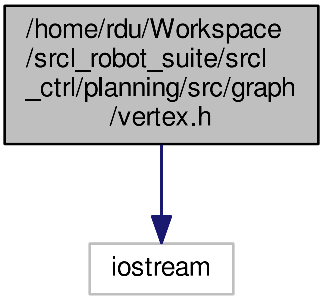
- srcl_ctrl

8.4 /home/rdu/Workspace/srcl_robot_suite/srcl_ctrl/planning/src/graph/README.md File Reference

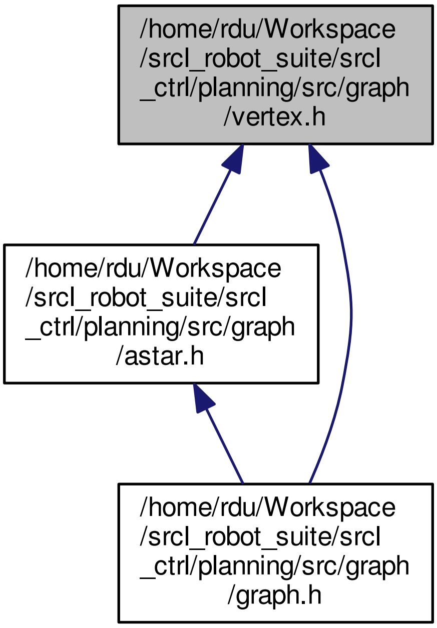
8.5 /home/rdu/Workspace/srcl_robot_suite/srcl_ctrl/planning/src/graph/vertex.h File Reference

```
#include <iostream>
```

Include dependency graph for vertex.h:



This graph shows which files directly or indirectly include this file:



Classes

- class srcl_ctrl::Edge< EdgeVertexType >

  An edge data structure template.
- class srcl_ctrl::Vertex< VertexNodeType >

  A vertex data structure template.

Namespaces

- srcl_ctrl