A* API DOC

A* API DOC

To use A* API, include the **astar.hpp** in your project, write your own version of *class Path* and overload all the function prototypes provided in the hpp file. Since the API serves as an agent, not complete framework, you need to define all the things needed for a problem.

class State

Define your own type of *class State*, which denotes the state of each node. It could be a matrix, a vector or a single number.

class Path

The class *Path* denotes each nodes in the graph. In each *Path* object there is: - A *state* variable, containing the state of current node. - A *achievement* variable, defining the goal state. - ID, used to distinguish each *Path*.

Besides that you should overload several functions.

get_next

Prototype: `std::list<Path<State>> get_next()` Generates next paths of current path. Collect them in `std::list`.

is_goal

Prototype: `bool is_goal()` Check wether current **state** is the same with the goal **state**.

cost_fn

Prototype: `int cost_fn(Path<State> & next)` Cost function for accumulation. Usually defined as:

```
int cost_fn(Path<State> & next){
    return 1;
}
```

cost_left_fn

Prototype: `int cost_left_fn(Path<State> & next)` Calculates the distance between current state and goad state (achievement). In the **Maze** problem this one is defined as the *Euclid distance* or *Manhattan distance* between current position and achievement position.

output()

Prototype: `void output()` Outputs informations of current state.

<

Prototype: `bool operator<(const Path<State> &p)const` Used in arrangement of std::map. You may define that easily by:

```
bool operator<(const Path<State> &p)const  {
        return this->ID < p.ID
}
```

==

Prototype: `bool operator==(const Path<State> &p)const` Checks wether current **state** *equals* the achievement **state**. For example, in the *Maze* problem where the state is a tuple of two elements (x and y that denotes current position), the operator can be defined as :

```
bool operator==(const Path<State> &p)const {
    return this->state.x == p.state.x
                    and
                 this->state.y == p.state.y
}
```

a_star Search Agent

Use the functions below:

init_path

Prototype: `void init_path(Path p)` Use **Path p** as start point and be ready for search. This should be called right after **clear()**.

search

Prototype: `bool search()` Once the start point is set, use this to perform search. The result comes in three possibilities: - Solution found, returns **true**. - Hit depth limit, failed to found. returns **false**. - There is naturally no solution. Returns **false**.

So check the return value of this function to know wether the agent have found the solution or not.

traverse

Prototype: `void traverse()` Calls `output()` on each node on the **solution path**.

get_all_paths

Prototype: `void get_all_paths(list<Path> & container)` Collects each node on the **solution path** into *container*.

clear

Prototype: `void clear()` Clears the last search result. You should always call this one on your agent before next search.

Hack and glory awaits! Tianrui Niu (niwtr)