Function manual

Define RUNNING_DIRS = the directory where you extract "sbpl.tar.gz".
Define LIBRARY_DIRS = RUNNING_DIRS/trunk/sbpl/.

Direct Increment:

The code for direct increment library is located in the directory
"**LIBRARY_DIRS/src/discrete_space_information\template**"

1. HashTable

Hash table is a vector to store visited states. Element is defined as struct "EnvXXXHashEntry_t"

Element 1	Element 2	Element 3	...	Element N
stateID1 x1 y1 theta1 speed1 steer1	stateID2 x2 y2 theta2 speed2 steer2	stateID3 x3 y3 theta3 speed3 steer3	...	stateIDN xN yN thetaN speedN steerN

Useful functions:

EnvXXXHashEntry_t* GetHashEntry(int x, int y, double theta, int speed, double steer)
{
Goes through all the element in the hashtable, find the one with the same (x, y, theta, speed, steer).
Return the pointer to that element.
}

EnvXXXHashEntry_t* CreateNewHashEntry(int x, int y, double theta, int speed, double steer)
{
Insert one element at the end of Hash table.
Return the pointer to the newly added element.
}

2. Initialize environment

This part is about read data from configuration file. The whole process is below, details is in section “useful functions”.

bool EnvironmentXXX::InitializeEnv(const char* sEnvFile, const char* roughnessEnvFile, const vector<sbpl_2Dpt_t>& perimeterptsV)

{
ReadConfiguration(fCfg);
InitializeEnvConfig();
InitializeEnvironment();
ComputeHeuristicValues();
Return 1 if initiation is complete.
}

useful functions:

void EnvironmentXXX::ReadConfiguration(FILE* fCfg)
{
readin map data, roughness map data, vehicle configurations
}

void EnvironmentXXX::InitializeEnvConfig()
{
convert vehicle configurations into cells.
}

void EnvironmentXXX::InitializeEnvironment()
{
init hash table and add start & goal states into hash table.
}

void EnvironmentXXX::ComputeHeuristicValues()
{
precompute useful heuristics.
}

3. Getsuccessors:

"**Get successors**" function is used for forward search planning, the planner begins with start state and visited all possible successors' states, calculate corresponding action cost. Add them to the hashtable. The planners continue working like this until goal state is reached. Details is given below:

void EnvironmentXXX::GetSuccs(int SourceStateID, vector<int>* SuccIDV, vector<int>* CostV)
{

Assign all possible **[newx,newy,newtheta,newspeed,newsteer]** and corresponding **action costs**. ** details of action cost see part 5
Add every single **[new,newy,newtheta,newspeed,newsteer]** into **hash table**.
Pass on Succsessors ID vector **SuccsIDV** to the planner.
Pass on Action cost vector **CostV** to the planner.
}

4. Getpredecessors:

"**Get successors**" function is used for backward search planning, the planner begins with goal state and visited all possible predecessors' states, calculate corresponding action cost. Add them to the hashtable. The planners continue working like this until start state is reached. Details is given below.

void EnvironmentXXX::GetPreds(int TargetStateID, vector<int>* PredIDV, vector<int>* CostV)
{
Assign all possible **[newx,newy,newtheta,newspeed,newsteer]** and corresponding **action costs**. ** details of action cost see part 5
add every single **[new,newy,newtheta,newspeed,newsteer]** into **hash table**.
Pass on Predecessors ID vector **PredIDV** to the planner.
Pass on Action cost vector **CostV** to the planner.

}

5. Action cost calculation:

There are two functions work together to get the action cost.

1."int EnvironmentXXX::actioncost(int oldx, int oldy, double oldtheta, int oldspeed, double oldsteer,int newx, int newy, double newtheta, int newspeed, double newsteer, int dx, int dy)"
2. "int EnvironmentXXX::GetActionCost(int SourceX, int SourceY, int SourceTheta, EnvXXXAction_t* action)"

1. function "**actioncost**" calculates based on roughness, type of trajectory(**shortest, smoothest**) and work out the cost.

i.e. in the case of shortest path, dx & dy means the displacement of action, cost is given by **(dx^2+dy^2)*(maximum roughness of intermediate cells)**

2. function “**GetActionCost**” check validity of cells during the action.

{

Return **infinite cost** if is not valid intermediate cells, there occur vehicle collision during action or future collision problem, i.e. Check 2 times the displacement of action(2*[dx,dy]).

Return **finite cost(the one from “actioncost” function)** if all check is valid.

}