Exercise – Planning

Creating a Planner depends on many factors; is there a known set of steps towards the goal? Are there sub-tasks that need to be completed first? Do we need to find the set of actions to get to the goal ourselves? Each may require a completely different style of planner.

In this exercise we will be constructing a form of **STRIPS** planner, aimed at solving the **Dock-Worker Robot** problem.

Initial State

Goal State

1

2

3

1

2

3

This form of planner uses **States** and **Actions**. A **State** can represent anything from World State to simple Emotional State for A.I. entities. In this problem a State represents a configuration of shipping containers. **Actions** can then be anything as well, but in the case of this specific problem an Action is the movement of one container to a different location.

Actions also have preconditions before they can be executed, and will result in a new world state. In our case the preconditions for an action, for example moving the red container from the 1st column to the 3rd column, is that the red container is on top of the 1st column. The action would result in a new state where the red container is now on top of the 3rd column.

If we were to start with just the initial state, know our intended goal state, and know all potential actions that could occur within our entire problem domain (not just those available from the initial state) then we could construct a graph of all potential states and the actions that connect them.

Once we have the graph that represents the entire domain we can perform a search on it and find all of the actions we need to perform to transition us from the Initial State to the Goal State.

You have been provided with a starter project for this exercise that is available on Portal. It requires the AIE Bootstrap solution to work, and also assumes that you have **already completed the Pathfinding tutorial**. Alternatively you will need some sort of Pathfinding capability that can run **Dijkstra’s Shortest Path** algorithm.

The project uses base classes that are found within a file called **Planner.h**:

namespace Planner {

// a base class for states

class State : public Pathfinding::Node {

public:

State() : id(0) {}

virtual ~State() {}

// a way of identifying each state

unsigned int id;

};

// base class for actions

class Action {

public:

Action() {}

virtual ~Action() {}

virtual bool conditionMet(State\* state) const = 0;

virtual State\* execute(State\* state) const = 0;

};

// a custom pathfinding edge that links with the

// Action that creates the edge link

class ActionLink : public Pathfinding::Edge {

public:

ActionLink(State\* t, Action\* a) : Edge(t, 1), action(a) {}

virtual ~ActionLink() {}

Action\* action;

};

}

There is a State class that inherits from a Pathfinding **Node** so that it can be searched. It also has an id that is used to uniquely identify States that must be correctly updated by derived classes, which we will talk about shortly.

There is also an abstract Action class that tests a State to see if the conditions are met for the Action to execute, and also a method that will execute the action and create a new State from an existing State. This new state is allocated on the heap so will need to be deleted at some point.

Finally there is an **ActionLink** class. This class inherits from Pathfinding **Edge** so that it can be used to connect States together in a Pathfinding Graph, but also tracks which Action would be needed to transition from one State to the next.

*These base classes could be used to solve many different problems, but you will need to implement the steps to build the graph that can then be searched.*

For our Dock-Worker Robot problem you have also been provided with two derived classes, a **DWRState** and **DWRAction**, which inherit from **Planner::State** and **Planner::Action** respectively. They make use of an enumeration for container colours. The **DWRState** contains 3 stacks that can contain the 3 containers, and a **DWRAction** specifies which container it moves from the top of one stack to another:

Look over the classes carefully.

enum eContainer {

RED,

GREEN,

BLUE

};

class DWRState : public Planner::State {

public:

DWRState() {}

DWRState(const DWRState& state) {

stack[0] = state.stack[0];

stack[1] = state.stack[1];

stack[2] = state.stack[2];

id = state.id;

}

virtual ~DWRState() {}

std::deque<eContainer> stack[3];

// converts the state of the containers to a single

// unsigned int, stores it in 'id' and returns it

unsigned int hash();

};

class DWRAction : public Planner::Action {

public:

DWRAction(eContainer c, int s, int e) : colour(c), start(s), end(e) {}

virtual ~DWRAction() {}

virtual bool conditionMet(Planner::State\* state) const {

return ((DWRState\*)state)->stack[start].empty() == false &&

((DWRState\*)state)->stack[start].back() == colour;

}

virtual Planner::State\* execute(Planner::State\* state) const {

if (!conditionMet(state))

return nullptr;

DWRState\* newState = new DWRState(\*(DWRState\*)state);

newState->stack[start].pop\_back();

newState->stack[end].push\_back(colour);

newState->hash();

return newState;

}

eContainer colour;

int start, end;

};

The stacks for the state will be mostly empty. A state will only contain 3 containers at a time, which will be shifted between the stacks when states change. The state also has a **hash()** method. The job of this method is to analyse the stacks and come up with a unique id to represent the state, which it stores within the base class id member variable and returns it. This hash works by treating the containers as a 2-bit combination:

* 00 for no container
* 11 for a red container
* 01 for a green container
* 10 for a blue container

Each stack is then a 6-bit combination, and the total state contains 18-bits, which it stores within the 32-bits of the unsigned int id:

unsigned int DWRState::hash() {

// converts a state to an unsigned int

// each colour is a combination of 2-bits

// red = 11

// green = 01

// blue = 10

// none = 00

// each stack is then 6-bits (3 containers)

// total is 18-bits (whole combination stored in an unsigned int 32-bit)

std::bitset<32> set(0);

for (unsigned int i = 0, j = 0; i < 3; ++i, j = 0) {

for (auto& c : stack[i]) {

if (c == eContainer::RED) {

set[i \* 6 + j \* 2] = true;

set[i \* 6 + j \* 2 + 1] = true;

}

else if (c == eContainer::GREEN) {

set[i \* 6 + j \* 2] = true;

}

else if (c == eContainer::BLUE) {

set[i \* 6 + j \* 2 + 1] = true;

}

++j;

}

}

id = set.to\_ulong();

return id;

}

The **DWRAction’s** **conditionMet()** method simple checks if a certain container is on top of a specified stack. If it is then the action can be executed.

The **execute()** method simply clones the passed in state, then performs the action’s container change on the clone, rehashes the clone and returns it.

This clone might then represent a brand new state, or there may be a matching state that a different action had previously created. We will handle what happens in this case in a moment.

The provided application stores a collection of **DWRAction** within a **std::vector** and stores a collection of **DWRState** within a **std::map**, using the state id as the key in the map. Finally it stores the current state’s id:

unsigned int m\_currentStateID;

// all states and actions

std::map<unsigned int, DWRState\*> m\_states;

std::vector<DWRAction> m\_actions;

During the **startup()** it creates an initial state, pushing containers into the first stack, hashes it and then stores it within the state collection. It then constructs all potential actions. The actions specify a colour, the stack it must be on top of within a state, then the destination stack. There are a total of 18 potential actions that could ever be performed:

// starting state

auto startState = new DWRState();

startState->stack[0].push\_back(eContainer::BLUE);

startState->stack[0].push\_back(eContainer::GREEN);

startState->stack[0].push\_back(eContainer::RED);

startState->hash();

// add to state list

m\_states[startState->id] = startState;

// current state

m\_currentStateID = startState->id;

// create all potential actions

m\_actions.push\_back(DWRAction(eContainer::RED, 0, 1));

m\_actions.push\_back(DWRAction(eContainer::RED, 0, 2));

m\_actions.push\_back(DWRAction(eContainer::RED, 1, 0));

m\_actions.push\_back(DWRAction(eContainer::RED, 1, 2));

m\_actions.push\_back(DWRAction(eContainer::RED, 2, 0));

m\_actions.push\_back(DWRAction(eContainer::RED, 2, 1));

m\_actions.push\_back(DWRAction(eContainer::GREEN, 0, 1));

m\_actions.push\_back(DWRAction(eContainer::GREEN, 0, 2));

m\_actions.push\_back(DWRAction(eContainer::GREEN, 1, 0));

m\_actions.push\_back(DWRAction(eContainer::GREEN, 1, 2));

m\_actions.push\_back(DWRAction(eContainer::GREEN, 2, 0));

m\_actions.push\_back(DWRAction(eContainer::GREEN, 2, 1));

m\_actions.push\_back(DWRAction(eContainer::BLUE, 0, 1));

m\_actions.push\_back(DWRAction(eContainer::BLUE, 0, 2));

m\_actions.push\_back(DWRAction(eContainer::BLUE, 1, 0));

m\_actions.push\_back(DWRAction(eContainer::BLUE, 1, 2));

m\_actions.push\_back(DWRAction(eContainer::BLUE, 2, 0));

m\_actions.push\_back(DWRAction(eContainer::BLUE, 2, 1));

The application is also set up to draw the current state, and displays the total number of states that exist within the collection. There are a total of 60 potential container configurations.

And lastly, before we begin implementing our actual planning, during the application’s update it is set up to randomly select an action that could currently be performed on the current state and then performs the action. If this results in a new state it is added to the collection and set as the current state. If it results in a state that already is contained within the collection then it specifies the original as the current state and deletes the newly created state:

// every second it performs an action

static float timer = 0;

timer += deltaTime;

if (timer >= 1) {

timer -= 1;

// find all actions that we can perform

std::vector<DWRAction> potentialActions;

for (auto& action : m\_actions) {

if (action.conditionMet(m\_states[m\_currentStateID]))

potentialActions.push\_back(action);

}

// pick a random action and execute it

if (potentialActions.empty() == false) {

auto id = rand() % potentialActions.size();

auto& action = potentialActions[id];

auto state = action.execute(m\_states[m\_currentStateID]);

m\_currentStateID = state->id;

// add new state to list if it doesn't exist, or delete it

if (m\_states.find(state->id) == m\_states.end())

m\_states[state->id] = (DWRState\*)state;

else

delete state;

}

}

Now that we have gone through the existing project it is time for you to find the set of actions needed to get from the initial starting state to a goal state!

Creating the Domain Graph:

Before we can search for the lowest set of actions to get us to a goal state we need to create a graph from all of the potential states and actions.

We only know the initial state so far, and all actions that could potentially be run on any state, but with that information we can construct a graph by starting with the initial state and finding the actions whose pre-conditions have been met. We then use those actions to create links to new states, then repeat the process for all of the new states create, and repeat for all of their potential actions, so on and so forth.

This process is exactly like a depth-first graph search, except that we are building the graph as we go. The pseudocode for the process is:

Implementing this in our application would then go within **startup()** after the intial state and actions have been created:

// build state graph

std::list<Planner::State\*> toProcess;

toProcess.push\_back(startState);

while (toProcess.empty() == false) {

// remove a state

auto state = toProcess.back();

toProcess.pop\_back();

// get potential actions for the state

for (auto& action : m\_actions) {

if (action.conditionMet(state)) {

// create new state

auto newState = action.execute(state);

// check if new state already exists

auto iter = m\_states.find(newState->id);

if (iter == m\_states.end()) {

// if it doesn't then add it to collection

m\_states[newState->id] = (DWRState\*)newState;

// create a link from current state to new state

state->edges.push\_back(

new Planner::ActionLink(newState, &action));

// add new state to be processed

toProcess.push\_back(newState);

} else {

// state already exists

delete newState;

// create link from current to existing state

state->edges.push\_back(

new Planner::ActionLink(iter->second, &action));

}

}

}

}

openList : list

add initialState to openList

while openList not empty

currentState = openList pop

for all valid actions for currentState

newState = action.execute( currentState )

if newState doesn’t already exist in stateCollection

add newState to stateCollection

add newState to openList

add link from currentState to newState with cost of 1

else

delete newState

add link from currentState to existingState with cost of 1

Once that process has been completed we will have a collection of all potential states, and they will have their **Pathfinding::Edge** links filled in with valid connections that also include the action required to transition from one state to another.

So now it’s onto finding the set of actions that could get us to a goal state.

Finding the Goal State:

So we need a goal state’s ID, and in our case this is our goal state:

Goal State

1

2

3

Its hash id is **122880**, so that is our goal within our **std::map** of states.

To find a path to it we can perform a simple **Dijkstra’s Shortest Path** search right after we’ve built the graph:

// find shortest path to goal

std::list<Pathfinding::Node\*> path;

Pathfinding::Search::dijkstra(startState, m\_states[122880], path);

But the problem with our current search is that it gives us each of the states between our initial state and our goal state, but we instead want the series of actions that will perform these transitions. The actions are stored within the edge links!

We can store the actions we need to perform within a **std::list** within our application, then during the **update()** we can execute the first action after 1 second, remove it, then 1 second later execute the next action, and so on. This would replace the current update process that selects a random potential action to execute.

// stored as a member variable within the application class

std::list<DWRAction\*> m\_pathActions;

// replaces random action execution within update()

if (m\_pathActions.empty() == false) {

auto action = m\_pathActions.front();

m\_pathActions.pop\_front();

auto state = action->execute(m\_states[m\_currentStateID]);

m\_currentStateID = state->id;

delete state;

}

But we still need to populate **m\_pathActions**.

This is fairly easy as we simply take the first state from the found path, then check its edges against the next state to find the edge link that connects the two states. Once it is found we add the relevant action to our **m\_pathActions** list and then move on to the next state and loop the process.

The following would be executed at the end of **startup()** after the graph has been assembled:

// get the actions that are needed

auto state = path.front();

path.pop\_front();

while (path.empty() == false) {

// find which action from 'state' gets us to the front of the path

for (auto& e : state->edges) {

if (e->target == path.front()) {

m\_pathActions.push\_back(

(DWRAction\*)((Planner::ActionLink\*)e)->action);

break;

}

}

state = path.front();

path.pop\_front();

}

And with all steps successfully implemented you would now have a functional basic STRIPS implementation that can solve the Dock-Worker Robot problem and find the shortest series of actions needed to achieve a goal state!

Challenges:

A problem that is very similar to the Dock-Worker Robot is the **Towers of Hanoi**. With the Towers of Hanoi the **blocks can only go on top of blocks that are larger** than themselves.

Attempt to adapt your Dock-Worker Robot solution to solve the Towers of Hanoi.

This will require your Actions to have more pre-conditions, and pre-conditions about the target stack as well to ensure that a large block cannot be placed on top of a smaller block.

Initial State

Goal State

1

2

3

1

2

3