CE318/CE818: High-level Games Development

Lecture 9: Game AI: Decision Making

Diego Perez

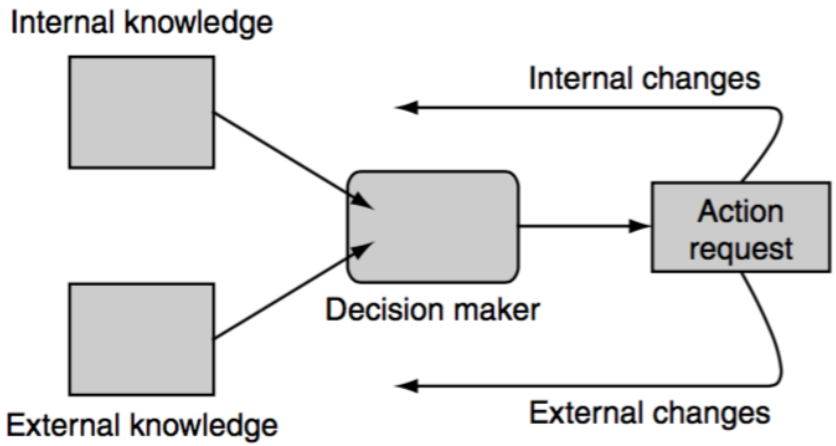
dperez@essex.ac.uk
Office 3A.527

2016/17

Outline

The Decision Making Problem

Decision making: the ability of a character to decide what to do.



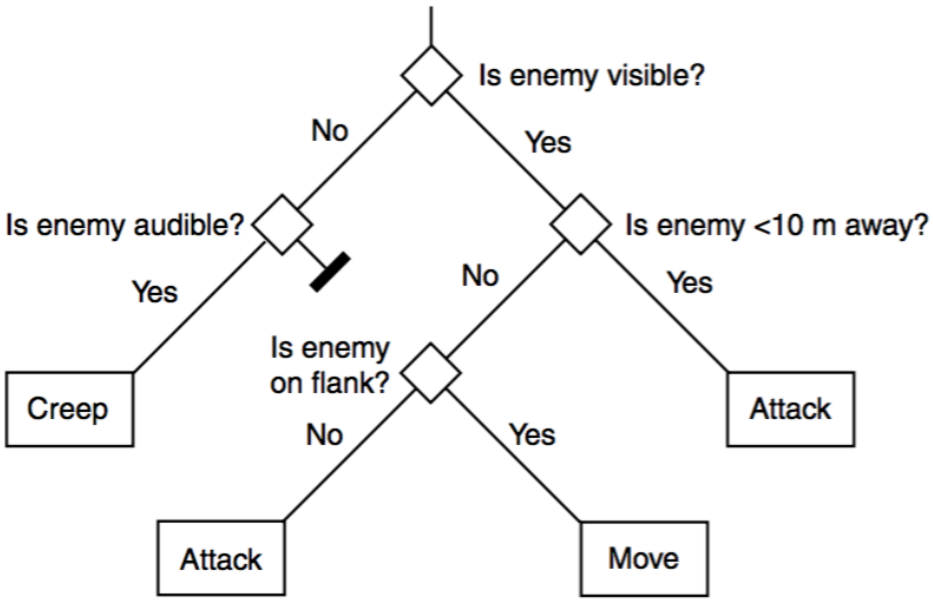
- The character processes a set of information that can be used to generate an action to be carried out.
- Input of Decision Maker: Information the character possesses.
  - External Knowledge: Information from the game environment (position of other entities, level layout, direction of noise, etc.).
  - Internal Knowledge: Character's internal state (health, goals, past actions, etc.).
- Output of Decision Maker: Action request.
  - External Changes: Movement, animations.
  - Internal Changes: Beliefs, change in goals.

Outline

Decision Trees

Decision Trees are simple structures, quite fast and easy to implement and understand. The concept is simple: given a set of input information, decide what's the best action to execute.

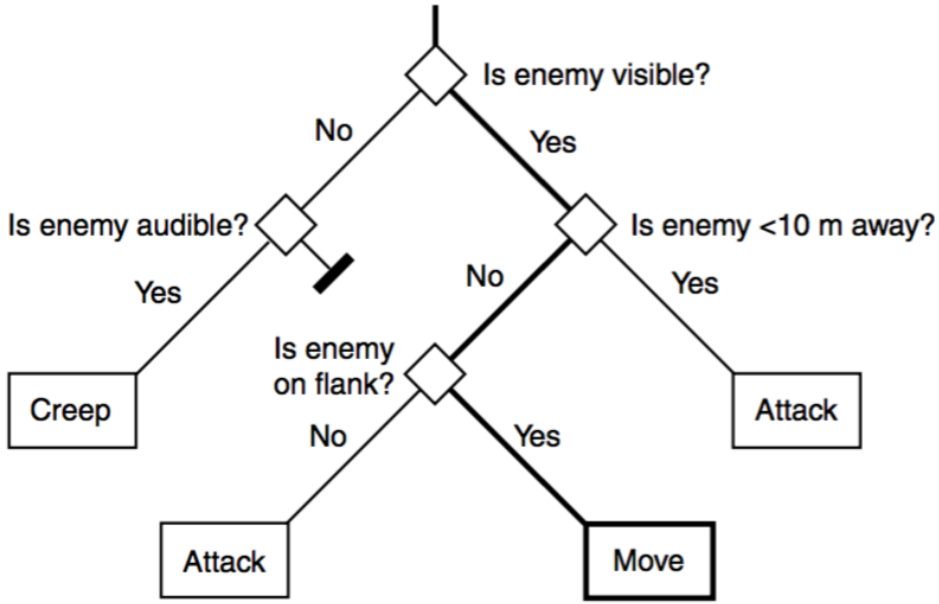
A decision tree is made of decision **nodes**, starting from a first decision at its root. Each decision has at least two possible options.



Decisions at each node are made based on the internal knowledge of the agent, or queried to a centralized manager object that provides this information.

Decision Trees

The algorithm starts at the first decision (root) and continues making choices at each decision node until the decision process has no more decisions to consider.



- At the each leaf of the tree an action is attached.
- Then, the action is carried out automatically.
- Note that the same action can be placed in multiple leaves of the tree.

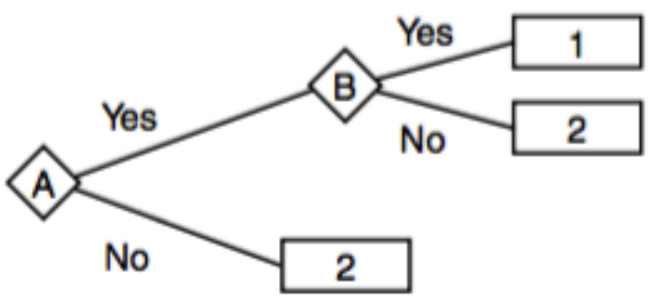
Decision Trees

Decisions in a tree are simple:

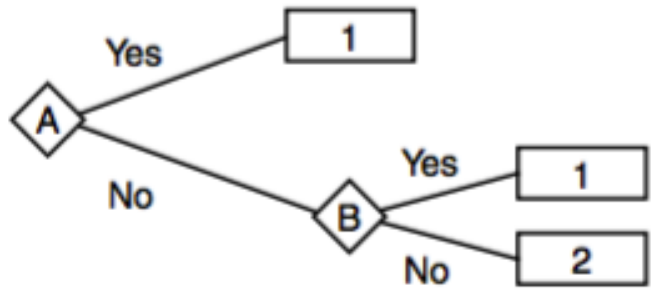
- Typically, they check a *single* numeric value, an enumeration, or a *single* boolean condition.
- They can be given access to (more) complex operations, such as calculating distances, visibility checks, pathfinding, etc.

Decision can be put into series to reflect **AND** and **OR** clauses:

If A AND B then action 1, otherwise action 2

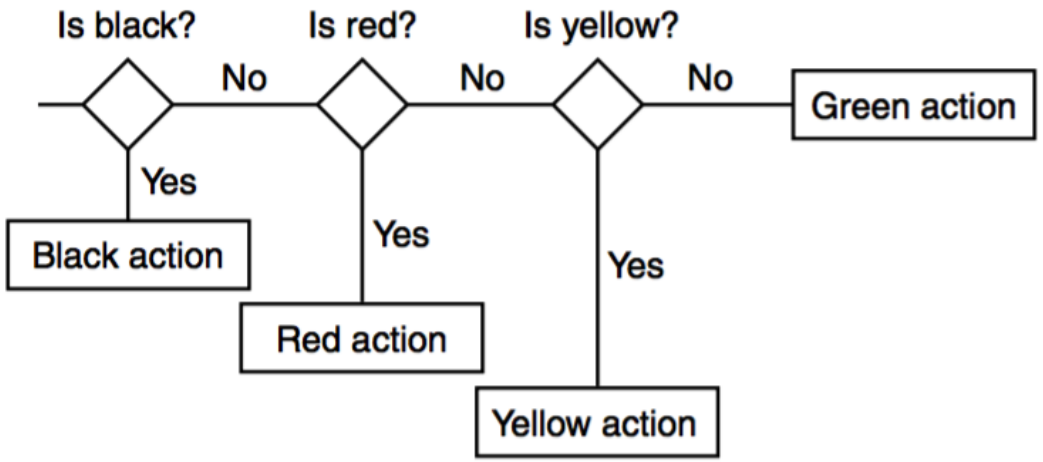


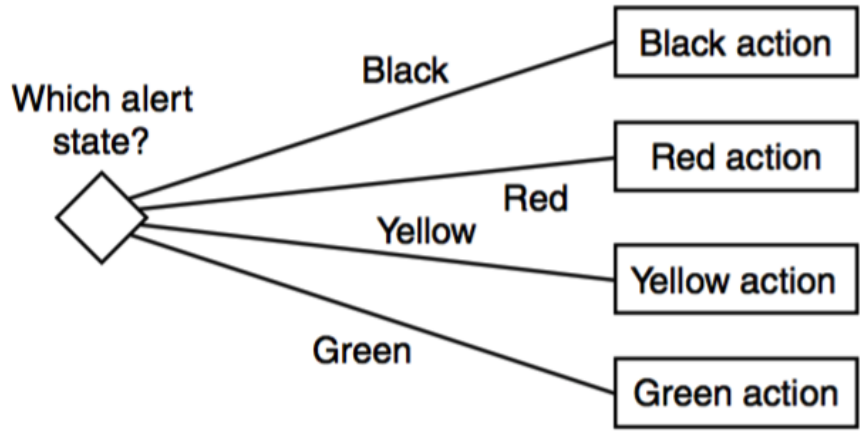
If A OR B then action 1, otherwise action 2



Decision Trees

Binary vs. N-ary decision trees:



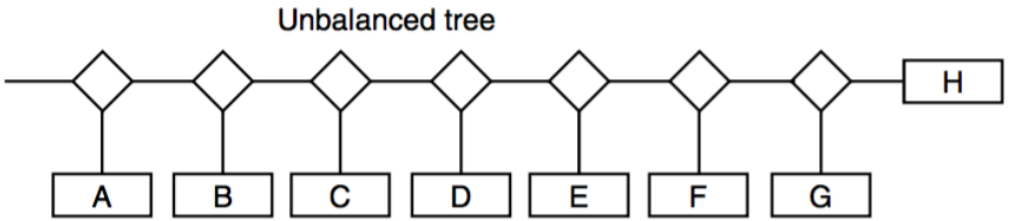


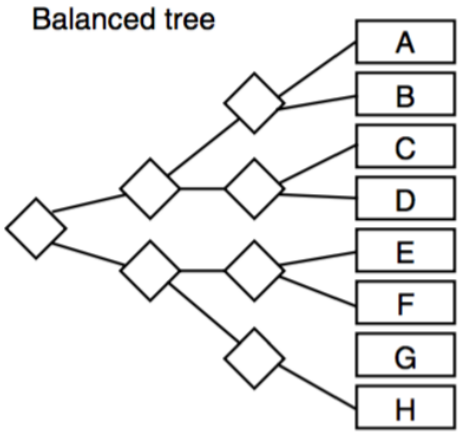
The N-ary tree requires less comparisons: it's more efficient.

But, binary trees are more common. Efficiency gain is lost at the code level (sequence of *if* statements). Also, binary decision trees are more easily optimized.

Decision Trees: Balancing

A tree is **balanced** if it has, approximately, the same number of leaves on each branch. A balanced tree guarantees that selecting an action (reaching a leave node) of the tree is performed in $O(\log_2(n))$ time, where n is the number of nodes in the tree.



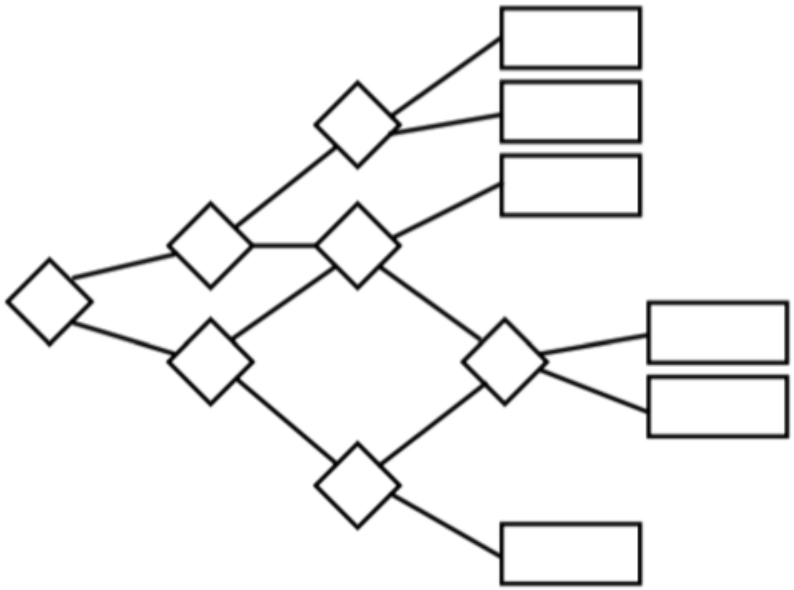


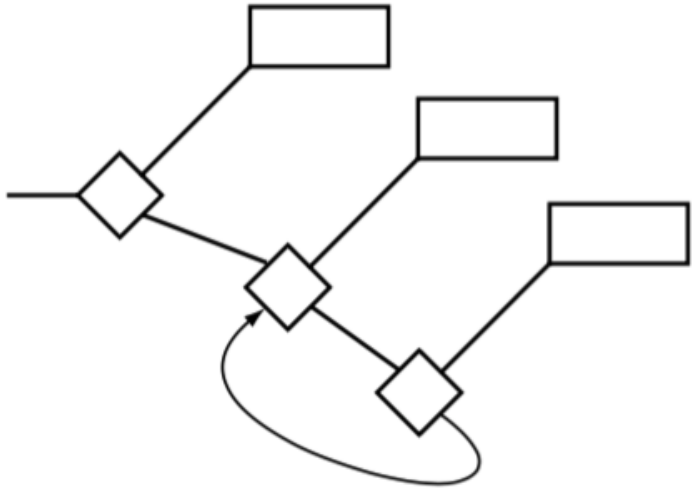
At worst, a totally unbalanced tree can return an action in $O(n)$ decisions.

However, the optimal tree structure depends on the decision nodes. Most commonly used checks should be placed close to the root node, and the most expensive decisions should be located closer to the leaves.

Decision Trees: Variations

A **Directed Acyclic Graph** (DAG) can be created by allowing certain nodes in the tree to be reached by different branches:



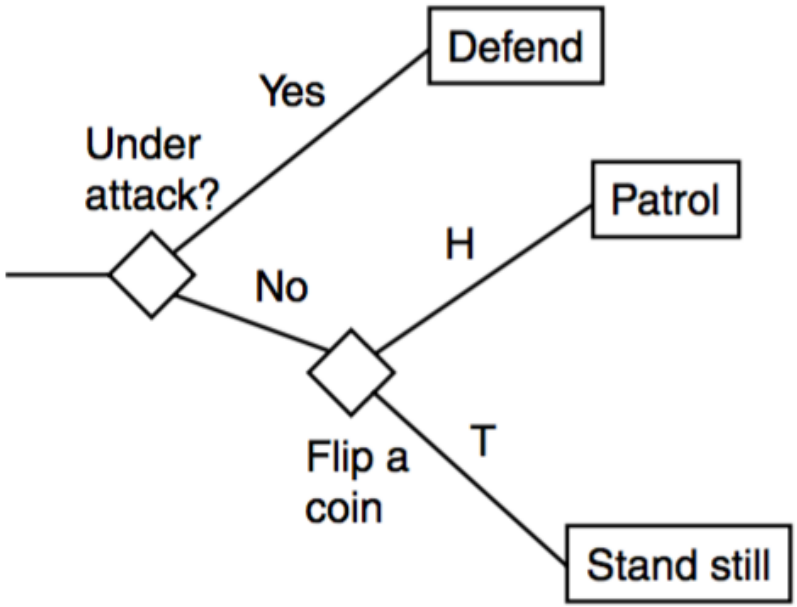


The tree on the left is a valid DAG.

The tree on the right, however, is an **invalid** tree (it's not acyclic), and it can produce infinite loops.

Decision Trees: Variations

A **Random Decision Tree** is a decision tree that allows variation in the behaviour by introducing a random element.



Q? What happens if this decision tree is executed at every frame?

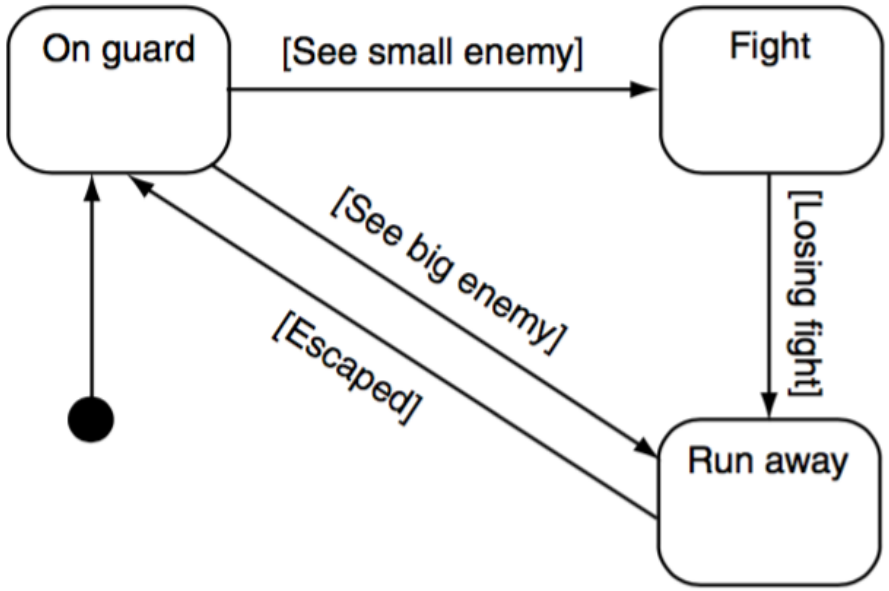
Outline

Finite State Machines

A better way to take account for the world outside the agent **and** its internal state is a Finite State Machine (FSM). An FSM is composed by:

- A finite set of states. As long as the FSM stays in the same state, the agent that is controlled by it will keep performing the same action.
- One of this state is considered to be the *initial* state.
- A finite set of transitions from one state to another. Transitions are governed by conditions that can be triggered by internal or external events.

An example:



FSM Implementation Example (1/6)

We'll see an implementation of an FSM for character behaviours. This implementation is taken from `http://wiki.unity3d.com/index.php?title=Finite_State_Machine`, which is based on chapter 3.1 of *Game Programming Gems 1* by *Eric Dybsend*.

There are four components:

- **Transition enum:** labels to the transitions that can be triggered.

```
1 public enum Transition
2 {
3     NullTransition = 0, // A non-existing transition (don't
          delete).
4 }
```

- **StateID enum:** IDs of the states of the FSM.

```
1 public enum StateID
2 {
3     NullStateID = 0, // A non-existing State (don't delete).
4 }
```

- **FSMState class:** Stores pairs transition-state indicating the relationships between origin and destination states, and the transition that links them.
- **FSMSystem class:** The FSM class the character uses to determine the behaviour, including a list of the possible states.

FSM Implementation (2/6) Example - FSM Class

```
1  public abstract class FSMState {
2     protected Dictionary<Transition, StateID> map = new (...)
3     protected StateID stateID;
4     public StateID ID { get { return stateID; } }
5
6     public void AddTransition(Transition trans, StateID id) {
7       // [+] Checks for null transitions or states, or duplicates.
8        map.Add(trans, id);
9     }
10
11    public void DeleteTransition(Transition trans) {
12       // [+] Check for NullTransition and existence of the transition.
13       map.Remove(trans);
14    }
15
16    public StateID GetOutputState(Transition trans) {
17       if (map.ContainsKey(trans))
18          return map[trans]; // The next state, given a transition.
19       return StateID.NullStateID;
20    }
21
22    // Used to set up the State condition before entering it.
23    public virtual void DoBeforeEntering() { }
24
25    // Used to finalize the State condition before leaving it.
26    public virtual void DoBeforeLeaving() { }
27
28    // Decides if the state should transition to another on its list.
29    public abstract void Reason(GameObject player, GameObject npc);
30
31    // Behavior of the NPC in this state.
32    public abstract void Act(GameObject player, GameObject npc);
33 }
```

FSM Implementation (3/6) Example - FSMSystem I

```
1  public class FSMSystem {
2     private List<FSMState> states;
3     private StateID currentStateID;
4     public StateID CurrentStateID { get { return currentStateID; } }
5     private FSMState currentState;
6     public FSMState CurrentState { get { return currentState; } }
7
8     public FSMSystem(){
9        states = new List<FSMState>();
10    }
11
12    //Places new states inside the FSM.
13    public void AddState(FSMState s){
14       // [+] Check for Null reference before deleting
15       states.Add(s);
16       if (states.Count == 1) {
17          currentState = s; //First added is initial state.
18          currentStateID = s.ID;
19       }
20    }
21
22    // This method delete a state from the FSM List if it exists
23    public void DeleteState(StateID id) {
24       // [+] Check for NullState before deleting
25       foreach (FSMState state in states)  {
26          if (state.ID == id) {
27             states.Remove(state);
28             return;
29          }
30       }
31    }
32    // ...
```

FSM Implementation (4/6) Example - FSMSystem II

```
1     // Changes the state the FSM is in based on
2     // the current state and the transition passed.
3     public void PerformTransition(Transition trans){
4        // [+] Check for NullTransition before changing the current
             state
5
6        // Check if the currentState has the transition passed as
             argument
7        StateID id = currentState.GetOutputState(trans);
8        currentStateID = id;
9        foreach (FSMState state in states){
10          if (state.ID == currentStateID){
11             // Do post processing before setting the new one
12             currentState.DoBeforeLeaving();
13             currentState = state;
14
15             // Do pre processing before it can reason or act
16             currentState.DoBeforeEntering();
17             break;
18          }
19       }
20    } // END PerformTransition() function.
21 } //END class FSMSystem
```

FSM Implementation (5/6) Usage I

```
1  public class NPCControl : MonoBehaviour {
2      public GameObject player;
3      public Transform[] path;
4      private FSMSystem fsm;
5
6      public void Start(){
7        MakeFSM();
8      }
9
10     public void Update(){
11       fsm.CurrentState.Reason(player, gameObject);
12       fsm.CurrentState.Act(player, gameObject);
13     }
14
15     //From FollowPath, if SawPlayer transition is fired -> ChasePlayer
16     //From ChasePlayerState, if LostPlayer trans. fired -> FollowPath
17     private void MakeFSM(){
18       FollowPathState follow = new FollowPathState(path);
19       follow.AddTransition(Transition.SawPlayer,StateID.ChasingPlayer);
20
21       ChasePlayerState chase = new ChasePlayerState();
22       chase.AddTransition(Transition.LostPlayer,StateID.FollowingPath);
23
24       fsm = new FSMSystem(); //Two states: FollowPath and ChasePlayer
25       fsm.AddState(follow);
26       fsm.AddState(chase);
27     }
28
29     public void Transit(Transition t) {
30        fsm.PerformTransition(t);
31     }
32 }
```

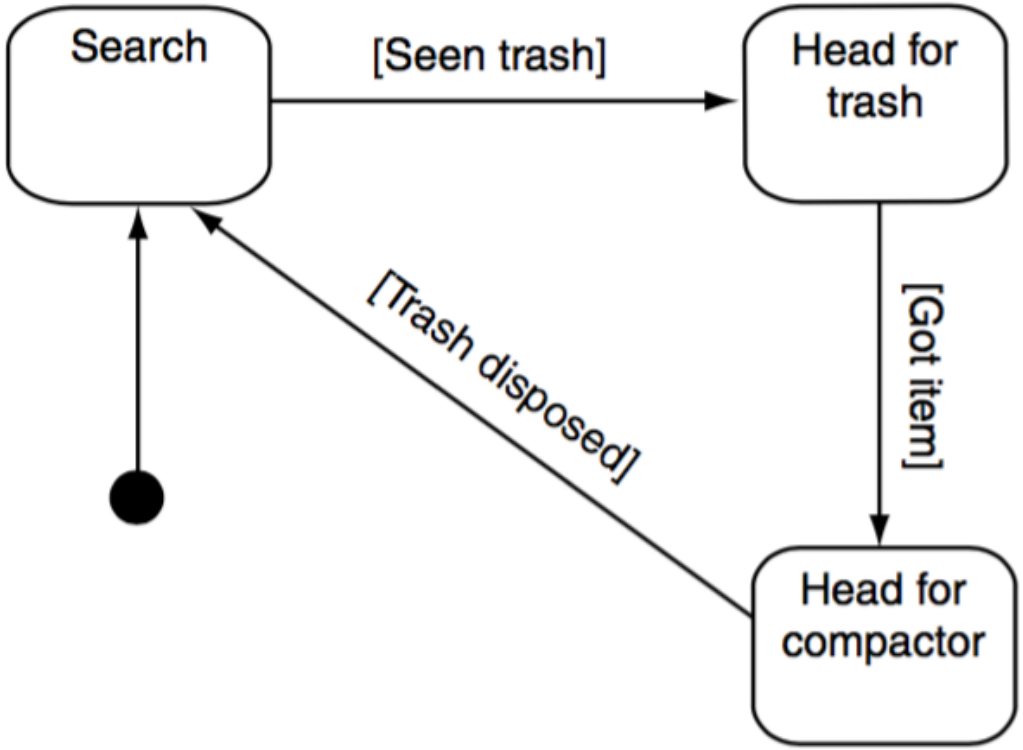
FSM Implementation (6/6) Usage II

```
1  public class FollowPathState : FSMState{
2
3      public FollowPathState(Transform[] wp) {
4          stateID = StateID.FollowingPath;
5      }
6
7      // Decision to transit to another state:  If the Player passes less
8      // than 15 meters away in front, transit to Transition.SawPlayer
9      // npc.GetComponent<NPCControl>().Transit(Transition.SawPlayer);
10     public override void Reason(GameObject player, GameObject npc){...}
11
12     // Act: Follow the path of waypoints for patrolling.
13     public override void Act(GameObject player, GameObject npc){...}
14 }
```

```
1  public class ChasePlayerState : FSMState{
2
3      public ChasePlayerState(Transform[] wp) {
4          stateID = StateID.ChasingPlayer;
5      }
6
7      // Decision to transit to another state: If player gone 30+ meters
8      // away from the NPC, fire LostPlayer transition
9      // npc.GetComponent<NPCControl>().Transit(Transition.LostPlayer);
10     public override void Reason(GameObject player, GameObject npc){...}
11
12     // Act: Follow the path of waypoints towards the player.
13     public override void Act(GameObject player, GameObject npc){...}
14 }
```

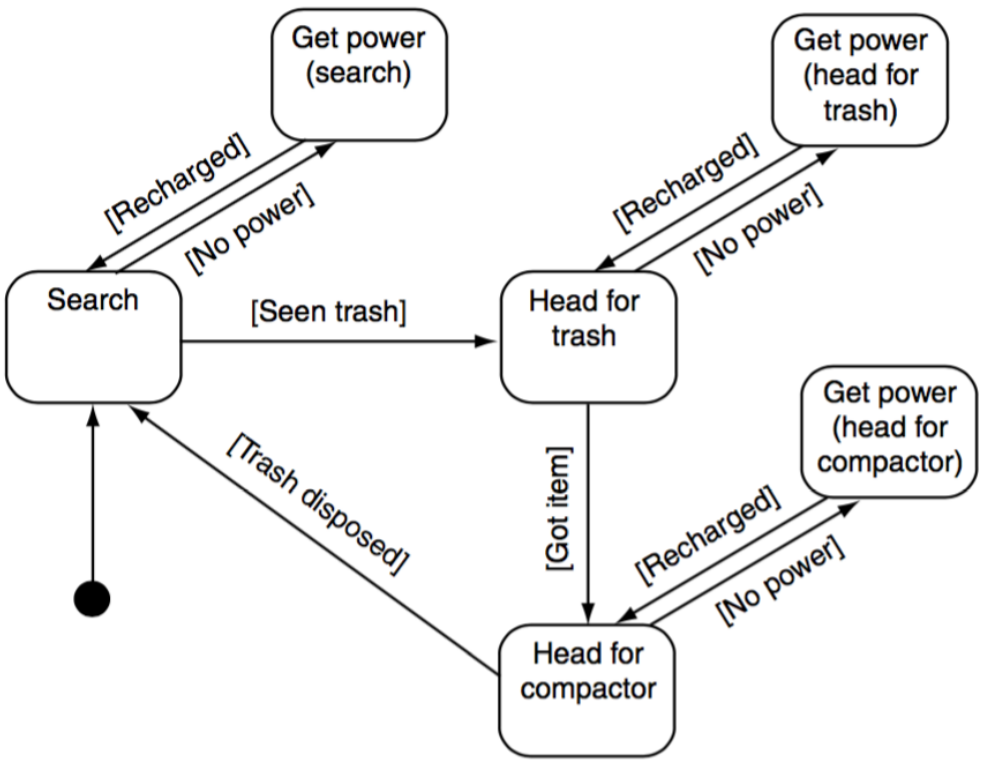
Hierarchical State Machines

FSM transitions are nice to indicate that a change of state should be made. All these state share the same **context**. For example, a cleaning robot that moves around a level with a given goal (cleaning). In *normal circumstances*, the FSM that controls the robot operates it to perform cleaning procedures.



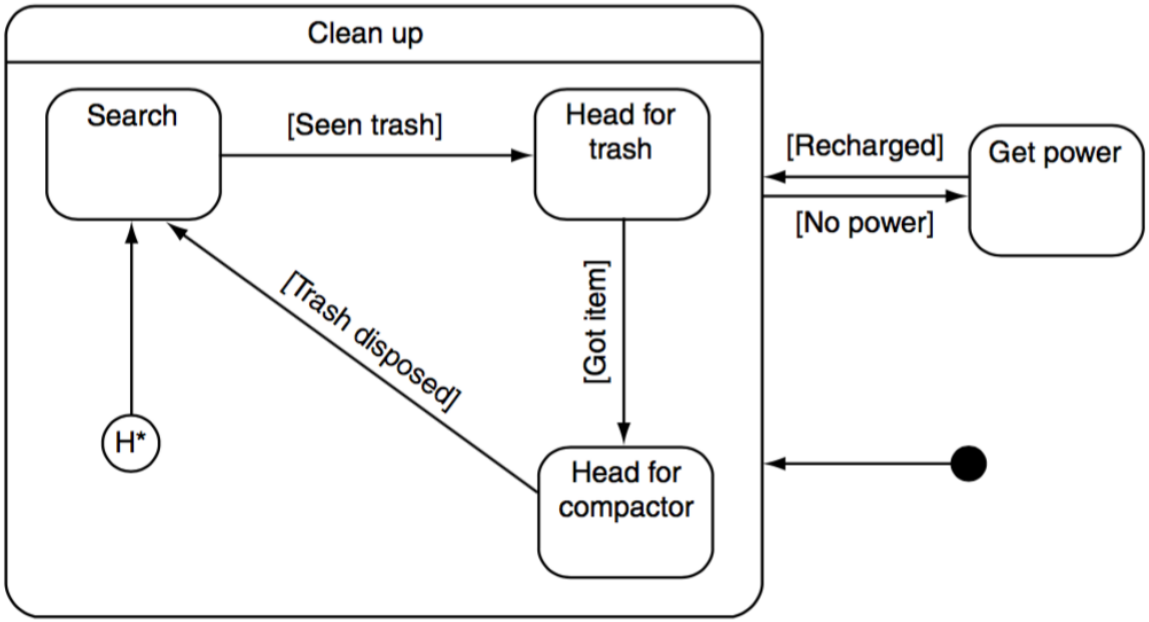
Hierarchical State Machines

However, external or internal circumstances can change the context of the agent. For instance, the robot may start running out of battery, or there could be an emergency. Taking each one of these changes of context into account may end up **doubling** the number of states in the FSM.



Hierarchical State Machines

A better solution is a **Hierarchical Finite State Machine** (HFSM). This machine groups states that belong to the same context in a *higher-level* state:



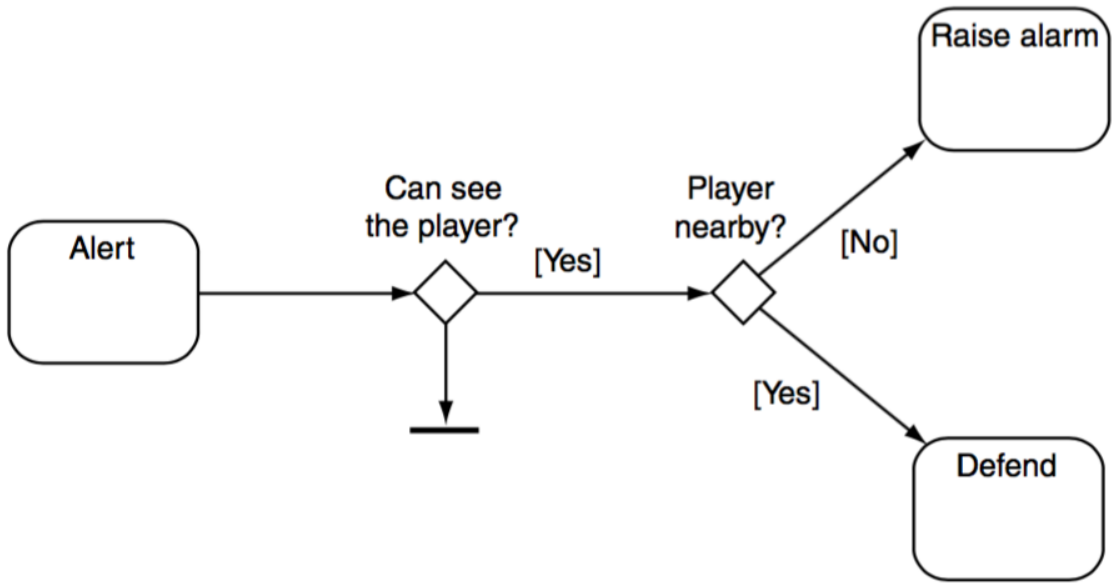
- There is an starting *high-level* state (indicated with a black dot).
- There is a starting state for each *high-level* state (H*).
- Transitions are restrained[1] within each *high-level* state.
- Transitions between *high-level* states are triggered from any state in the origin.

[1] Or not: particular states can have output transitions too!

Decision Trees and FSMs

It is possible to combine both FSMs and Decision Trees in a single structure:

- Transitions are replaced with decision trees.
- Actions from the decision trees are states.



Outline

Behaviour Trees

Behaviour trees have become very popular in recent years (starting 2004) as a more flexible, scalable and intuitive way to encode complex NPC behaviours.

One of the first popular games to use BTs was Halo 2.

Some of the advantages of BTs are as follows:

- Can incorporate numerous concerns such as path finding and planning
- Modular and scalable
- Easy to develop, even for non-technical developers
- Can use GUIs for easy creation and manipulation of BTs

Tasks can be composed into sub-trees for more complex actions and multiple tasks can define specific behaviours.

For this topic, we will use the notation and examples from Millington and Funge (ch. 5).

A Task

A task is essentially an activity that, given some CPU time, returns a value.

- Simplest case: returns success or failure (boolean)
- Often desirable to return a more complex outcome (enumeration, range)

Task should be broken down into smaller complete (independent) actions.

A basic BT consists of the following 3 **nodes** (elements):

- **Conditions**: test some property of the game
- **Actions**: alter the state of the game; they usually succeed
- **Composites**: collections of child tasks (conditions, actions, composites)

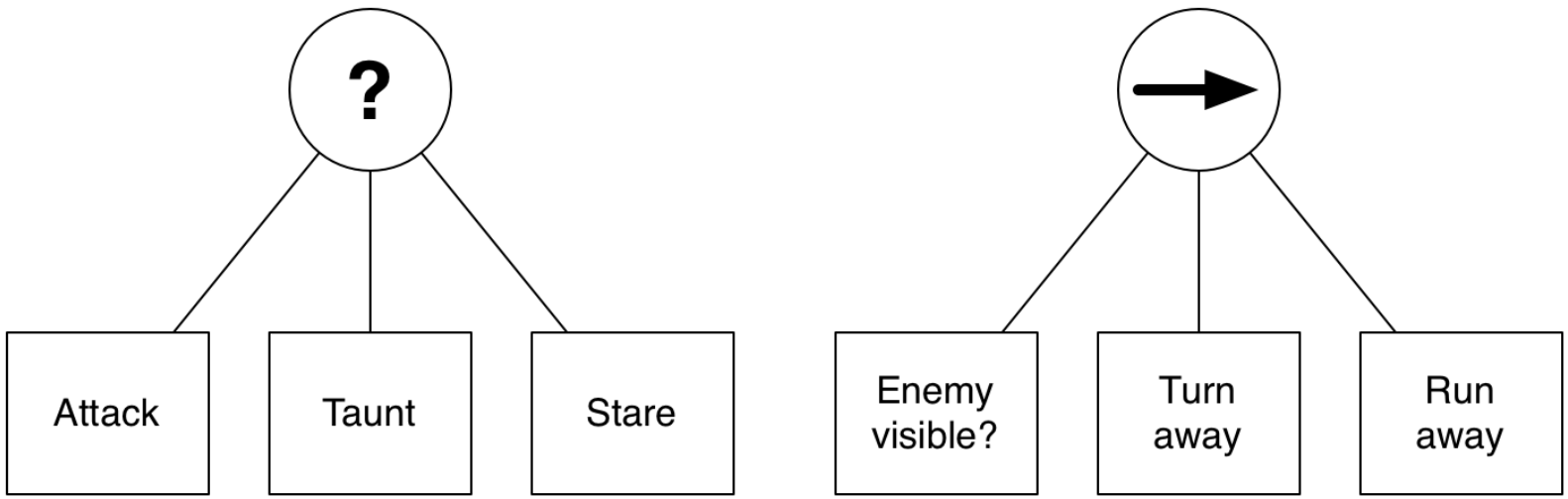
Conditions and actions sit at the leaf nodes of the tree. They are preceded by composites. Actions can be anything, including playing animations etc.

Composite tasks (always consider child behaviours in sequence):

- **Selector**: returns success as soon as one child behaviour succeeds
- **Sequence**: returns success only if **all** child behaviours succeeds

Selectors and Sequences

Two simple examples of selectors and sequences:



Q What do Selector and Sequence correspond to in term of logical operands?

Order of Actions

We can use the order of actions to imply priority:

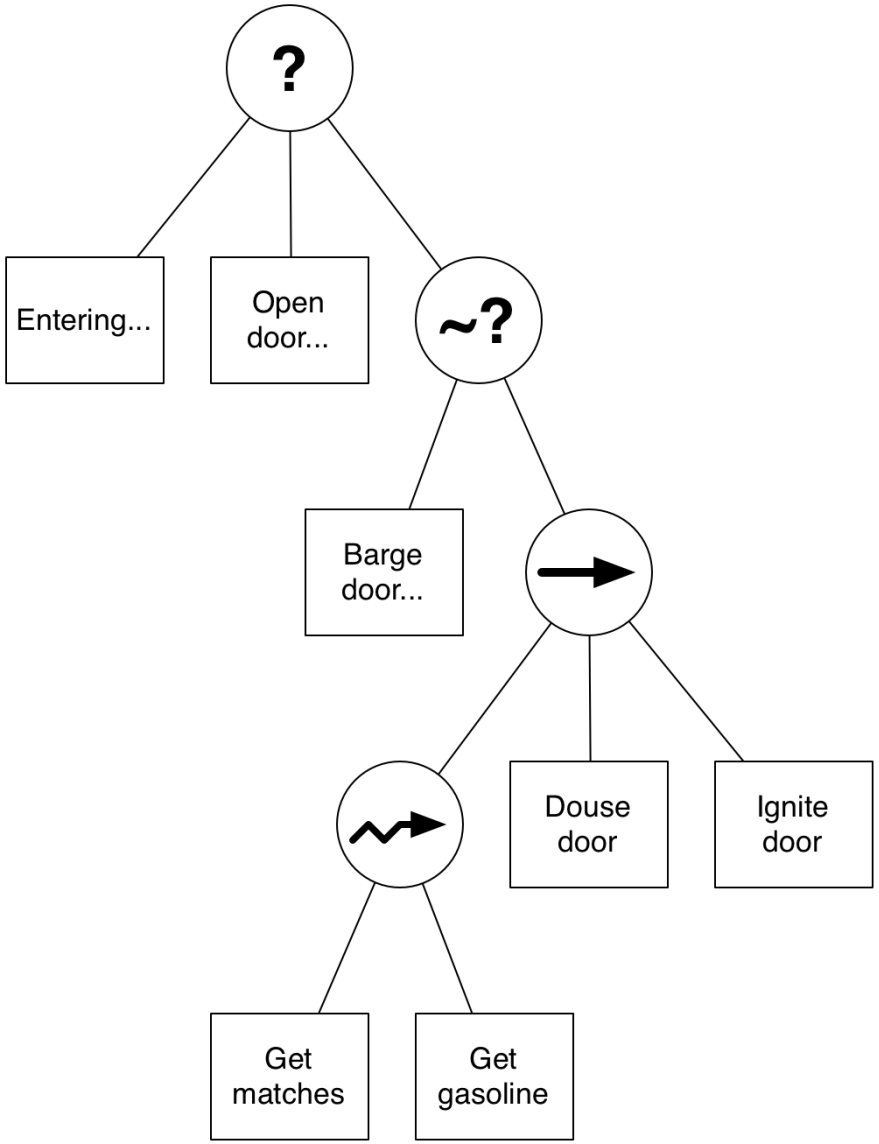
- Child actions are always executed in the order they were defined
- Often a fixed order is necessary (some actions are prerequisites of others)
- Sometimes, however, this leads to predictable (and boring) behaviour

Imagine you need to obtains items A, B and C to carry out an action. If the items are independent of one another, one obtains a more diverse behaviour if these items are always collected in a different order.

We want to have **partial ordering**: some strict order mixed with some random order. We thus use two new operators, random Selector and random Sequence.

Note: you can see from this that the design of behaviours does not only have to consider functionality (i.e., getting the job done) but also gameplay experience.

Partial Order Tree

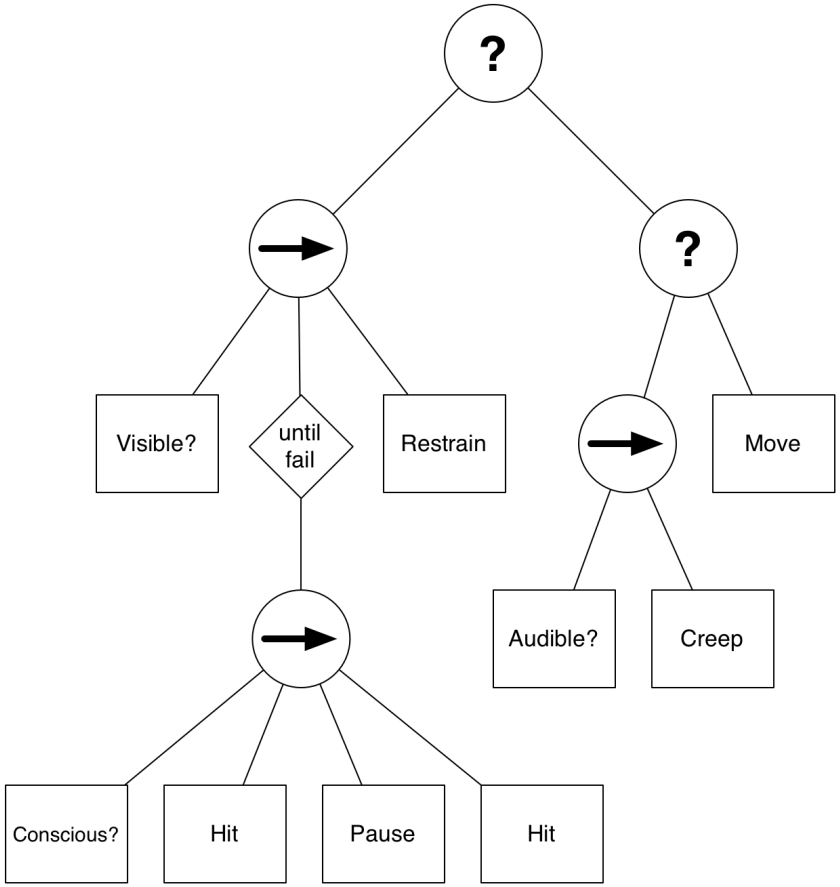


Decorators

Decorators add extended functionality to an existing task. A decorator has a single child whose behaviour it modifies. A popular usage is **filters**: if and how a task may run.

For instance, we can repeatedly execute a task until some goal has been achieved.

Decorators can also be used to invert the return value of an action.



Resource Management using Decorators

Individual parts of a BT often need to access the same resource (e.g., limited number of pathfinding instances, sounds). We can:

- Hard-code a resource query into the behaviour
- Create a condition task to perform the test using a Sequence
- Use a Decorator to guard the resource

Decorators can be used in a general approach such that the designer does not need to be aware of the low level details of the game's resource management.

We can use *semaphores*: a mechanism to ensure a limited resource is not over-subscribed. A decorator is subsequently associated with a semaphore which in turn is associated with the resource.

Can use a semaphore factor with unique name identifies – these can be used by both programmers and behaviour designers.

Concurrency and Timing

We need to take the time it takes to execute an action into account. If we execute each BT in its own thread, we can:

- Wait for some time for actions to finish (put the thread to sleeping)
- Timeout current tasks and return outcome (e.g., failure)
- Don't execute an action for some time after its last execution

Most importantly, taking care of concurrency allows for **parallel** tasks.

Sequence, Selector and Parallel form the backbone of most BTs.

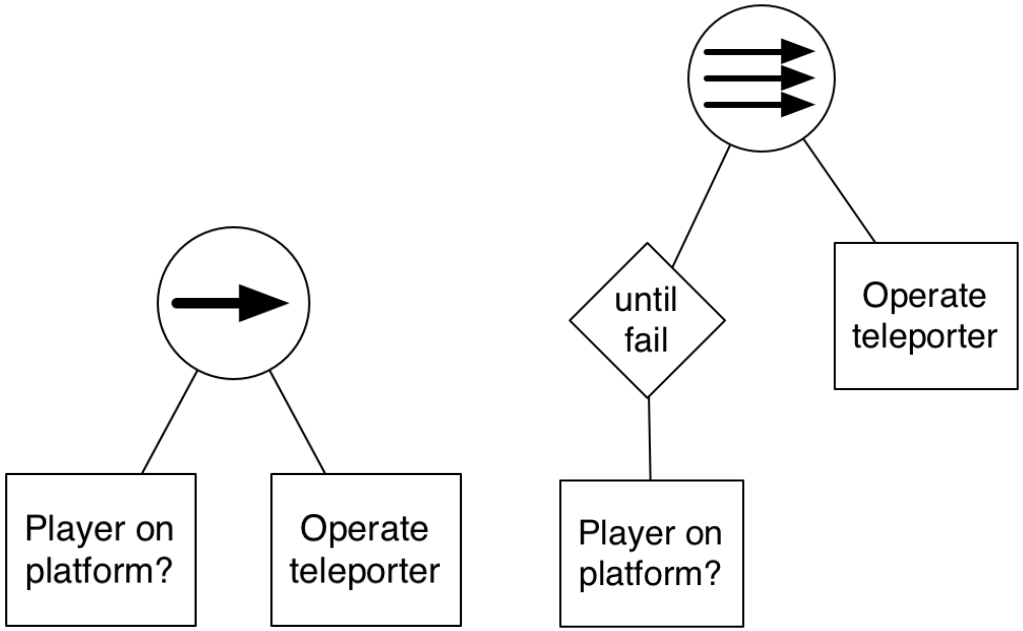
Parallel is similar to sequence but runs all tasks simultaneously. Can also implement this as a Selector. Different policies can be established:

- Terminate once one of the children fails.
- Terminate once one of the children succeeds.
- Different combinations of events.

Use of Parallel

Uses of parallel:

- Execute multiple actions simultaneously such as falling and shouting
- Control a group of characters: combine individual and group behaviours
- We can also use parallel for continuous condition checking



Q What is the difference between these trees?

Data and Reuse of Behaviours

Data: We often need to refer to (shared) data to carry out the actions of a behaviour. For instance, the action "open door" needs to refer to a particular door (and room).

The best approach is to decouple the data from the tree (rather that, say, use arguments to the actions). The central external data structure is called a **blackboard**. It can store any kind of data and responds to any type of request. This allows to still write independent tasks but which may communicate with one another. It is common for sub-trees to have their own blackboards.

Reuse of Behaviours: It is apparent that BTs, if designed properly, can be quite modular and hence it makes sense to re-use whole or partial trees. Actions and conditions can be parameterised and data can be exchanged via the blackboard. This allows one to re-use trees if agents require identical behaviours. This can be done in numerous different ways:

- Use a cloning operation
- Create a behaviour tree library
- Use a dictionary and reference names to retrieve trees/sub-trees

Behaviour Trees: Sample Implementation (in Java)

BehaviourTree class:

```
1  public class BehaviorTree {
2
3      //Root of the tree
4      private BTRootNode rootNode;
5
6      //Root of the tree
7      private BTNode currentNode;
8
9      //Agent that holds this behavior tree
10     private Object agent;
11
12     //Execution tick
13     private long tick;
14
15     public void execute(){
16         if(currentNode.getParent() == null) {
17             //We are at the root, execute from here.
18             rootNode.step();
19         } else {
20             currentNode.notifyResult();
21         }
22         tick++;
23     }
24
25     public void reset()  {
26         currentNode = rootNode;
27         tick = 0;
28         rootNode.reset();
29     }
30 }
```

BT: Sample Implementation (Node)

BTNode class (1/2):

```
1  public abstract class BTNode {
2
3      protected int nodeStatus; //Status of this node
4
5      protected Vector<BTNode> children; //Children of this node
6
7      protected BTNode parent;    //Parent of this node
8
9      protected int curNode;    //Current child node
10
11     protected int nodeCount;    //Child node count
12
13     protected BehaviorTree tree; //Reference to the BT that owns this
14
15     protected long lastTick; //Last tick when this node was executed.
16
17     public BTNode() {
18         nodeStatus = BTConstants.NODE\_STATUS\_IDLE;
19         //...
20     }
21
22      //Adds a new node to the list of children, in the given index
23     public void add(BTNode node, int index) {
24         node.setParent(this);
25         children.add(index, node);
26         nodeCount++;
27     }
```

BT: Sample Implementation (Node)

BTNode class (2/2):

```
1     //Function to notify the parent about my result.
2     public abstract void update(int nodeStatus);
3
4     public void notifyResult()  {
5         if((nodeStatus == BTConstants.NODE_STATUS_FAILURE) ||
6            (nodeStatus == BTConstants.NODE_STATUS_SUCCESS))   {
7             int nodeStatusResult = nodeStatus;
8             nodeStatus = BTConstants.NODE_STATUS_IDLE;
9             parent.update(nodeStatusResult);
10        } else {
11            //This will be executed if an action returns
12            //something different than SUCCESS or FAILURE (action
                  executed again)
13            this.step();
14        }
15    }
16
17    //Function to execute this node.
18    public void step() {
19        tree.setCurrentNode(this);
20        lastTick = tree.getCurTick();
21    }
22
23    public abstract void resetNode();
24    public void reset()  {
25        this.resetNode();
26        for(int i = 0; i < nodeCount; ++i)  {
27            children.get(i).reset();
28        }
29    }
```

BT: Sample Implementation (Root node)

BTConstants and BTRootNode class:

```
1  public interface BTConstants {
2      //Behavior tree node states
3      public static int NODE_STATUS_IDLE = 0;
4      public static int NODE_STATUS_EXECUTING = 1;
5      public static int NODE_STATUS_SUCCESS = 2;
6      public static int NODE_STATUS_FAILURE = 3;
7  }
```

```
1  public class BTRootNode extends BTNode{
2
3      BTRootNode() { super(); }
4
5      public void update(int nodeStatus) {
6          //In this case update current node
7          tree.setCurrentNode(this);
8
9          long lastTick = tree.getCurTick();
10         if(lastTick == lastTick)
11             //Whole tree evaluated in 1 cycle, but no action found
12             return;
13
14         step(); //No action was executed this cycle: re-run the tree.
15     }
16
17     public void step() {
18         super.step();
19         curNode = 0;
20         children.get(curNode).step();
21     }
22 }
```

BT: Sample Implementation (Selector node)

A Selector Node:

```
1  public class BTSelectNode extends BTNode{
2
3      BTSelectNode(BTNode parentNode)  { super(parentNode); }
4
5      public void update(int nodeStatus) {
6
7          if(nodeStatus != BTConstants.NODE_STATUS_EXECUTING)
8              ++curNode;
9
10         if(nodeStatus == BTConstants.NODE_STATUS_FAILURE)  {
11             if(curNode < nodeCount)
12                 children.get(curNode).step();
13             else {
14                 curNode = 0;
15                 nodeStatus = BTConstants.NODE_STATUS_FAILURE;
16                 parent.update(nodeStatus);
17             }
18         }
19         else if(nodeStatus == BTConstants.NODE_STATUS_SUCCESS) {
20             curNode = 0;
21             nodeStatus = BTConstants.NODE_STATUS_SUCCESS;
22             parent.update(nodeStatus);
23         }
24     }
25
26     public void step() {
27         super.step();
28         curNode = 0;
29         children.get(curNode).step();
30     }
31 }
```

BT: Sample Implementation (Sequence node)

A Sequence Node:

```
1  public class BTSequenceNode extends BTNode {
2      BTSequenceNode(BTNode parentNode)   { super(parentNode);  }
3
4      public void update(int nodeStatus)
5      {
6          if(nodeStatus != BTConstants.NODE_STATUS_EXECUTING)
7              ++curNode;
8
9          if(nodeStatus == BTConstants.NODE_STATUS_SUCCESS) {
10             if(curNode < nodeCount)
11                 children.get(curNode).step();
12             else {
13                 curNode = 0;
14                 nodeStatus = BTConstants.NODE_STATUS_SUCCESS;
15                 parent.update(nodeStatus);
16             }
17         }
18         else if(nodeStatus == BTConstants.NODE_STATUS_FAILURE){
19             curNode = 0;
20             nodeStatus = BTConstants.NODE_STATUS_FAILURE;
21             parent.update(nodeStatus);
22         }
23     }
24
25     public void step() {
26         super.step();
27         children.get(curNode).step();
28     }
29 }
```

BT: Sample Implementation (Parallel node)

A Parallel Node:

```
1  public class BTParallelNode extends BTNode {
2      private int nodesToFail, nodesToSucceed; //Min to fail / succeed
3      private int curNodesFailed, curNodesSucceeded;
4      //...
5      public void update(int nodeStatus)  {
6
7          if(nodeStatus == BTConstants.NODE_STATUS_FAILURE)   {
8              curNodesFailed++;
9              if(curNodesFailed == nodesToFail) {
10                 nodeStatus = BTConstants.NODE_STATUS_FAILURE;
11                 parent.update(nodeStatus);
12             }
13         }else if(nodeStatus == BTConstants.NODE_STATUS_SUCCESS)  {
14             curNodesSucceeded++;
15             if(curNodesSucceeded == nodesToSucceed)  {
16                 nodeStatus = BTConstants.NODE_STATUS_SUCCESS;
17                 parent.update(nodeStatus);
18             }
19         }
20     }
21     public void step()  {
22         super.step();
23         nodeStatus = BTConstants.NODE_STATUS_EXECUTING;
24         curNode = 0;
25         //A call to BTParallelNode.update() will change 'nodeStatus'
26         while(curNode < nodeCount && nodeStatus == BTConstants.
               NODE_STATUS_EXECUTING) {
27             children.get(curNode).step();
28             curNode++;
29         }
30     }
```

BT: Sample Implementation (Non filter)

A *Non* or *Negation* Filter:

```
1  public class BTNonFilter extends BTNode{
2
3      BTNonFilter(BTNode parentNode)  {
4          super(parentNode);
5      }
6
7      //Function to notify the parent about my result.
8      public void update(int nodeStatus)  {
9          ++curNode;
10
11         if(nodeStatus == BTConstants.NODE_STATUS_FAILURE) {
12             curNode = 0;
13             nodeStatus = BTConstants.NODE_STATUS_SUCCESS;
14             parent.update(nodeStatus);
15         }
16         else if(nodeStatus == BTConstants.NODE_STATUS_SUCCESS)  {
17             curNode = 0;
18             nodeStatus = BTConstants.NODE_STATUS_FAILURE;
19             parent.update(nodeStatus);
20         }
21     }
22
23     public void step()  {
24         super.step();
25         curNode = 0;
26         children.get(curNode).step();
27     }
28 }
```

BT: Sample Implementation (Loop filter)

A *Loop* Filter:

```
1  public class BTLoopFilter extends BTNode{
2
3      private int times;
4      private int limit;
5
6      BTLoopFilter(BTNode parentNode, int limit) {
7          super(parentNode);
8          times = 0;
9          limit = limit;
10     }
11
12     public void update(int nodeStatus) {
13         //Does not matter, just execute it again until loop counter
               expires
14         if(times == limit)  {
15             nodeStatus = BTConstants.NODE_STATUS_SUCCESS;
16             parent.update(nodeStatus);
17         } else {
18             times++;
19             curNode = 0;
20             children.get(curNode).step();
21         }
22     }
23
24     public void step() {
25         super.step();
26         times=1;
27         curNode = 0;
28         children.get(curNode).step();
29     }
30 }
```

BT: Sample Implementation (UntilFails filter)

An *UntilFails* filter:

```
1  public class BTUntilFails extends BTNode{
2
3      BTUntilFails(BTNode parentNode)  {
4        super(parentNode);
5      }
6
7      public void update(int nodeStatus) {
8          if(nodeStatus == BTConstants.NODE_STATUS_FAILURE)  {
9              //FAIL: This node fails.
10             nodeStatus = BTConstants.NODE_STATUS_FAILURE;
11             parent.update(nodeStatus);
12         }
13         else if(nodeStatus == BTConstants.NODE_STATUS_SUCCESS) {
14             //Execute again.
15             curNode = 0;
16             children.get(curNode).step();
17         }
18     }
19
20     public void step()  {
21         super.step();
22         curNode = 0;
23         children.get(curNode).step();
24     }
25 }
```

BT: Sample Implementation (Action/Condition)

Example of *Condition* and *Action* nodes:

```
1  public class IsItemUp extends BTNode{
2
3      public IsItemUp(BTNode parent)  { super(parent);  }
4
5      public void step() {
6          super.step();
7          Object agent = tree.getAgent();
8          nodeStatus = BTConstants.NODE_STATUS_FAILURE;
9
10         if(agent.isItem(agent.X-3,agent.X-1, agent.Y,agent.Y)) {
11            nodeStatus = BTConstants.NODE_STATUS_SUCCESS;
12         }
13         parent.update(nodeStatus);
14     }
15 }
```

```
1  public class Fire extends BTNode{
2
3      public Fire(BTNode parent)  {  super(parent); }
4
5      public void step() {
6          super.step();
7          Object agent = tree.getAgent();
8
9          //Set the ACTION that I want to do here.
10         agent.setAction(Agent.KEY_FIRE,true);
11         nodeStatus = BTConstants.NODE_STATUS_SUCCESS;
12         //parent.update(nodeStatus); //No need! Just change in status.
13     }
14 }
```

Outline

Lab Preview

Second part of Animations and AI Lab Assignment, including assessment.