DM842

Computer Game Programming: AI

Lecture 6

Decision Making

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

Outline

Outline

Open Goal Pathfinding

- check if a node is a goal
- heuristics need to report the distance to the nearest goal.
  This is problematic and handled by decision making (selecting a goal).

Dynamic Pathfinding

- environment is changing in unpredictable ways or its information is incomplete.
- replan each time new information is collected
- replan only the part that has changed $\rightsquigarrow$ D* but requires a lot of storage space for, eg, storing path estimates and the parents of nodes in the open list

Memory-Bounded Search

- Try to reduce memory needs
- Take advantage of heuristic to improve performance
  - Iterative-deepening A* (IDA*)
  - SMA*

Iterative Deepening A*

- IDA*
  - Idea from classical Uniformed Iterative Deepening
    depth-first search where the max depth is iteratively increased
  - skip open and closed list
  - depth-first search with cutoff on the f-cost
  - cutoff set on the smallest f-cost of nodes that exceeded the threshold at the previous iteration
  - very simple to implement but less efficient
  - is the "best" variant for goal-oriented action planning in decision making

Properties of IDA*

Complete	Yes
Time complexity	Still exponential
Space complexity	linear
Optimal	Yes. Also optimal in the absence of monotonicity

Simple Memory-Bounded A*

Use all available memory

- Follow A* algorithm and fill memory with new expanded nodes
- If new node does not fit
  - remove stored node with worst f-value
  - propagate f-value of removed node to parent
- SMA* will regenerate a subtree only when it is needed
  the path through subtree is unknown, but cost is known

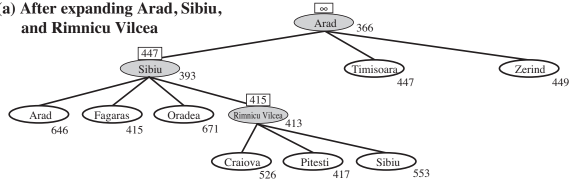
Properties of SMA*

Complete	yes, if there is enough memory for the shortest solution path
Time	same as A* if enough memory to store the tree
Space	use available memory
Optimal	yes, if enough memory to store the best solution path

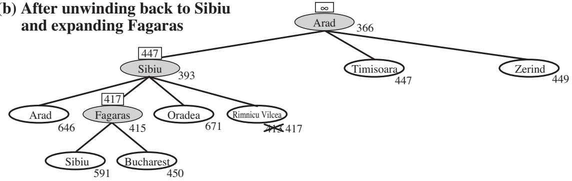
In practice, often better than A* and IDA* trade-off between time and space requirements

Recursive Best First Search

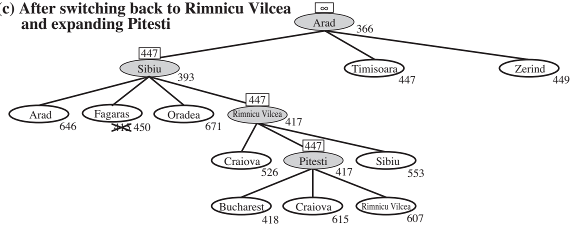
(a) After expanding Arad, Sibiu, and Rimnicu Vilcea



(b) After unwinding back to Sibiu and expanding Fagaras



(c) After switching back to Rimnicu Vilcea and expanding Pitesti



Recursive Best First Search

```
function RECURSIVE-BEST-FIRST-SEARCH($problem$) returns a solution, or failure
  return RBFS($problem$, MAKE-NODE($problem$.INITIAL-STATE), $\infty$)

function RBFS($problem$, $node$, $f\_limit$) returns a solution, or failure and a new $f$-cost limit
  if $problem$.GOAL-TEST($node$.STATE) then return SOLUTION($node$)
  $successors \leftarrow$ [ ]
  for each $action$ in $problem$.ACTIONS($node$.STATE) do
     add CHILD-NODE($problem$, $node$, $action$) into $successors$
  if $successors$ is empty then return $failure$, $\infty$
  for each $s$ in $successors$ do /* update $f$ with value from previous search, if any */
     $s.f \leftarrow \max(s.g + s.h,\ node.f))$
  loop do
     $best \leftarrow$ the lowest $f$-value node in $successors$
     if $best.f > f\_limit$ then return $failure$, $best.f$
     $alternative \leftarrow$ the second-lowest $f$-value among $successors$
     $result, best.f \leftarrow$ RBFS($problem$, $best$, min($f\_limit$, $alternative$))
     if $result \neq failure$ then return $result$
```

Other Issues

Interruptible Pathfinding

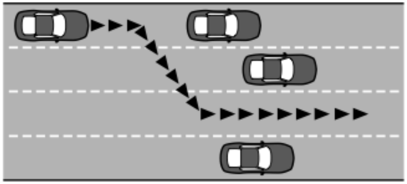
- rendering needs to run every 1/60 or 1/30 of a second ($= 0.6ms$)
- A* algorithm can be easily stopped and resumed.
- data required to resume are all contained in the open and closed lists.

In Real Time Strategy games: possible many requests to pathfinding at the same time

- serial $\rightsquigarrow$ problems for time, parallel $\rightsquigarrow$ problems for space
- central pool of pathfinding + path finding queue (FIFO).
- information from previous pathfinding runs could be useful to be stored above all valid for hierarchical pathfinding)

Continuous Pathfinding

Vehicle pathfinding: eg, police car pursuing a criminal
Split down by placing a node every few yards along the road
path = a period of time in a sequence of adjacent lanes.



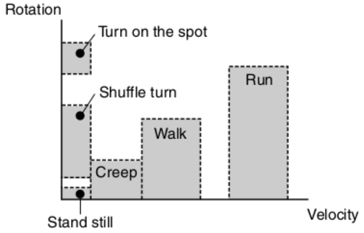
But cars are moving. Depending on the speed the gap may be there or not.

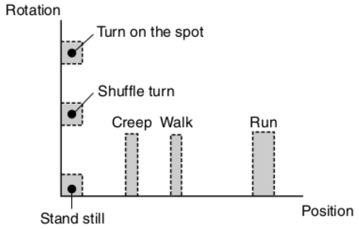
- A* in a graph where nodes represent states rather than positions
- a node has two elements: a position and a time.
- an edge exists between two nodes if the end node can be reached from the start node and if the time it takes to reach the node is correct.
- two different nodes may represent the same position

- graph created dynamically: connections, so they are built from scratch when the outgoing connections are requested from the graph.
- retrieving the out-going connections from a node is a very time-consuming process $\rightsquigarrow$ avoid A* versions that need recalculations
- It should be used for only small sections of planning.
  Eg, plan a route for only the next 100 yards or so. The remainder of the route planned on intersection-by-intersection basis.
  The pathfinding system that drove the car was hierarchical, with the continuous planner being the lowest level of the hierarchy.

Movement Planning

- If characters are highly constrained, then the steering behaviors might not produce sensible results. Eg: urban driving.
- Chars have, eg, walk animation, run animation, or sprint animation
- Animations need specific conditions for being believable
- plan sequence of animations to reach a large scale maneuver



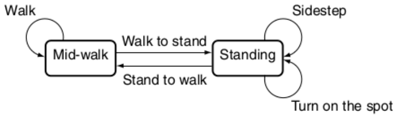


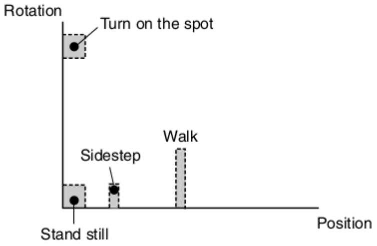
- Movement planning uses a graph representation. Each node of the graph represents both the position and the state of the character at that point, ie, the velocity vector, that determines the set of allowable animations that can follow
- Connections in the graph represent valid animations; lead to nodes representing the char after the animation
- route returned consists of a set of animations
- If the velocities and positions are continuous, then infinite number of possible connections. Heuristic only returns the best successor nodes for addition to the open list.
- similarly to continuous pathfinding, graph is generated on the fly and recomputations in A*are avoided.

Example

Walking bipedal character
Animations: walk, stand to walk, walk to stand, sidestep, and turn on the spot.
They can be applied to a range of movement distances
Positions: Each animation starts or ends from one of two positions: mid-walk or standing still.
Some positions in the environment are forbidden

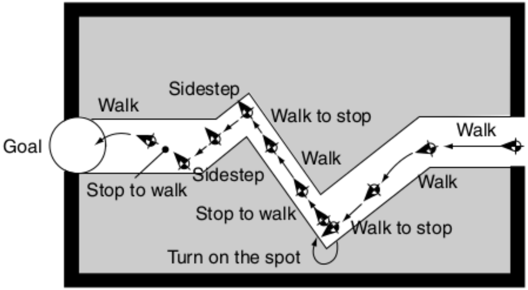
State machine: positions ≡ states and transitions ≡ animations.





Goal: range of positions with no orientation.

Result from A*:

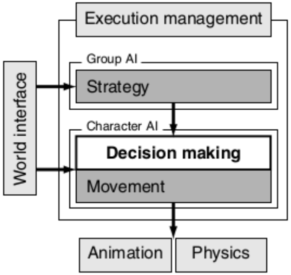


Outline

Decision Making

Decision Making: ability of a character to decide what to do.
We saw already how to carry out that decision (movement, animation, ...).
From animation control to complex strategic and tactical AI.

- state machines,
- decision trees
- rule-based systems
- fuzzy logic
- neural networks



Input internal and external knowledge
Output action

Knowledge representation:

- External knowledge identical for all algorithms
  Message passing system.
  Eg, `danger` is a constant at the character. Every new object in toolchain needs to define when to send message `danger` and the character will react.

- Internal knowledge algorithm dependent

- Actions:
  Objects notify which actions they are capable of by means of flags.
  For goal oriented behavior, every action has a list of goals that will be achieved
  Alternatively, actions as objects with associated data such as state of world after action, animations, etc. Actions are then associated to objects.

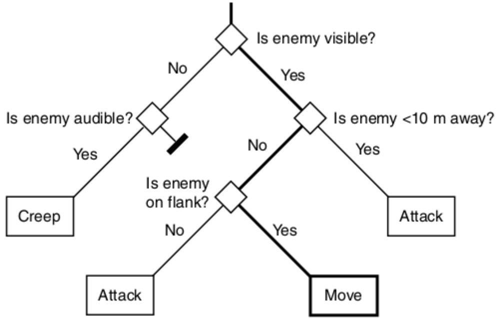
The Toolchain

- AI-related elements of a complete toolchain
- Custom-designed level editing tools to be reused over all the games
- data driven or object oriented. Each object in the game world has a set of data associated with it that controls behavior
  Eg, data type “to be avoided” / “to be collected”.
- Different characters require different decision making logic and behavior
- Allowing level designers to have access to the AI of characters they are placing without a programmer requires specialist AI design tools.
  Eg: AI-Implant and SimBionic provide a palette of AI behaviour to combine
- Actions selected by level designer are mostly steering behaviors.
  They are put together by the graphical definition of finite state machines
- Debugging at run time
- SDK that allows new functionality to be implemented in the form of plug-in tools.

Outline

Decision Trees

- Tree made up of connected decision points.
- Each choice is made based on the character's knowledge.
- At each leaf of the tree an action is attached
- Typically binary tree (multibranches are equivalent) but more generally directed acyclic graph (DAG).



Data Type	Decisions
Boolean	Value is true
Enumeration (i.e., a set of values,only one of which might be allowable)	Matches one of a given set of values
Numeric value (either integer or floating point)	Value is within a given range
3D Vector	Vector has a length within a given range (this can be used to check the distance between the character and an enemy, for example)

Combinations of decisions are obtained by the structure of the tree. Eg: AND, OR

Decision trees can express any function of the input attributes.
E.g., for Boolean functions, truth table row   path to leaf

Execution time depends on decisions
Eg, checking if any enemy is visible may involve complex ray casting sight checks through the level geometry.

Implementation

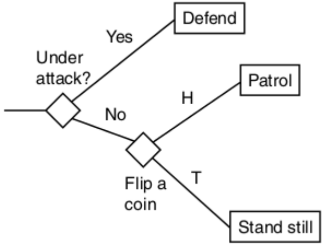
A simple tree can be implemented initially, and then as the AI is tested in the game, additional decisions can be added.

```
class DecisionTreeNode:
  def makeDecision() # Recursion though the
        tree

class Action: #interfacing virtual functions
  def makeDecision():
  return this

class Decision (DecisionTreeNode):
  trueNode # pointer to a node
  falseNode
  testValue # pointer to data for the test
  def getBranch() # carries out the test
  def makeDecision() # Recursion

class FloatDecision (Decision):
  minValue
  maxValue
  def getBranch():
  if maxValue >= testValue >= minValue:
    return trueNode
  else:
    return falseNode
```

```
class MultiDecision (DecisionTreeNode):
  daughterNodes
  testValue

  def getBranch():
    return daughterNodes[testValue]

  def makeDecision():
    branch = getBranch()
    return branch.makeDecision()
```

Random Decision Trees

- Some element of random behavior choice adds unpredictability, interest, and variation
- Requires some care if the choice is made at every frame to yield stable behavior $\rightsquigarrow$ keep track of last decision



```
struct RandomDecision (Decision):
  lastFrame = −1
  lastDecision = false
  def test():
    if frame() > lastFrame + 1: # old
      # Make a new decision
      lastDecision = randomBoolean()
    lastFrame = frame() # curr. frame num.
    return lastDecision
```

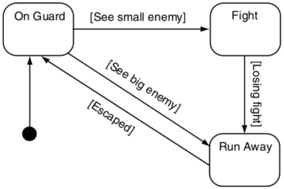
- Add a time-out information, so the agent changes behavior occasionally.

Outline

Finite State Machines

An FSM is an algorithm used for parsing text, eg, tokenize the input code into symbols that can be interpreted by the compiler.

- States: actions or behaviors. Chars are in exactly one of them at any time.
- Transitions: a set of associated conditions, if they are met the char changes state
- Initial state for the first frame the state machine is run



In a decision tree, the same set of decisions is always used, and any action can be reached through the tree.

In a state machine, only transitions from the current state are considered, so not every action can be reached.

General State Machines

- set of possible states
- current state
- set of transitions
- at each iteration (normally each frame), the state machine's update function is called.
- checks if any transition from the current state is triggered
- the first transition that is triggered is scheduled to fire
  (some actions related to transition are executed)

Implementation

```
class StateMachine:
  states # list of states for the machine
  initialState
  currentState = initialState
  def update(): # checks and applies
    triggeredTransition = None
    for transition in currentState.getTransitions():
      if transition.isTriggered():
         triggeredTransition = transition
         break
    if triggeredTransition:
      targetState = triggeredTransition.getTargetState()
      actions = currentState.getExitAction()
      actions += triggeredTransition.getAction()
      actions += targetState.getEntryAction()
      currentState = targetState
      return actions
    else: return currentState.getAction()
```

```
class MyFSM:
  enum State:
    PATROL
    DEFEND
    SLEEP
  myState # holds current state

  # transition by polling (asking for information explicitly)
  def update():
    if myState == PATROL:
      if canSeePlayer(): myState = DEFEND # access to game state data
      if tired(): myState = SLEEP # access to game state data
    elif myState == DEFEND:
      if not canSeePlayer(): myState = PATROL
    elif myState == SLEEP:
      if not tired(): myState = PATROL

  # transition in an event-based approach (waiting to be told information)
  def notifyNoiseHeard(volume):
    if myState == SLEEP and volume > 10:
      myState = DEFEND

  def getAction():
    if myState == PATROL: return PatrolAction
    elif myState == DEFEND: return DefendAction
    elif myState == SLEEP: return SleepAction
```

State machines implemented like this can often get large and code unclear

```
class State:
  def getAction()
  def getEntryAction()
  def getExitAction()
  def getTransitions()
```

```
class Transition:
  actions
  def getAction(): return actions
  targetState
  def getTargetState(): return targetState
  condition
  def isTriggered(): return condition.test()
```

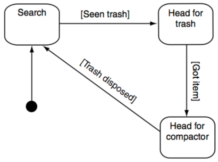
Often defined in a data file and read into the game at runtime.
Do not allow to compose questions easily. Requires condition interface.

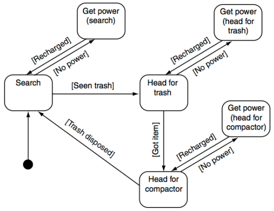
```
class Condition:
  def test()

class FloatCondition (Condition):
  minValue
  maxValue
  testValue # ptr to game data
  def test():
    return minValue <= testValue
          <= maxValue
```

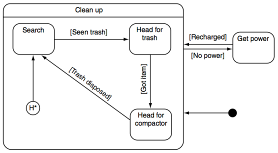
```
class AndCondition (Condition):
  conditionA
  conditionB
  def test():
    return conditionA.test() and conditionB.test()

class NotCondition (Condition):
  condition
  def test(): return not condition.test()

class OrCondition (Condition):
  conditionA
  conditionB
  def test():
    return conditionA.test() or conditionB.test()
```

Hierarchical State Machines

- Alarm mechanism: something that interrupts normal behavior to respond to something important.
- Representing this in a state machine leads to a doubling in the number of states.
- Instead: each alarm mechanism has its own state machine, along with the original behavior.



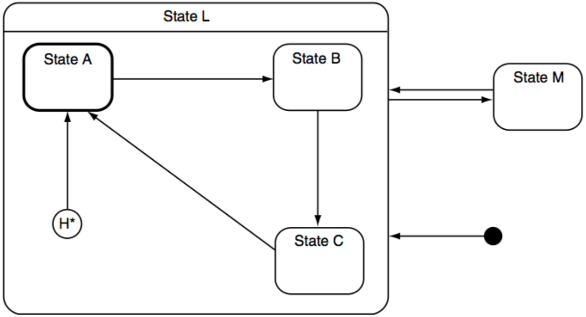




We can add transitions between layers of machines

Implementation

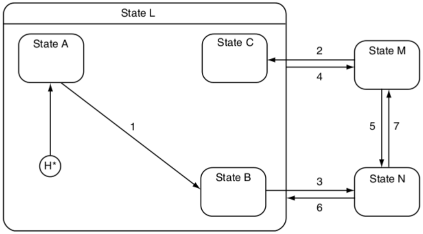
In a hierarchical state machine, each state can be a complete state machine in its own right $\rightsquigarrow$ recursive algorithm



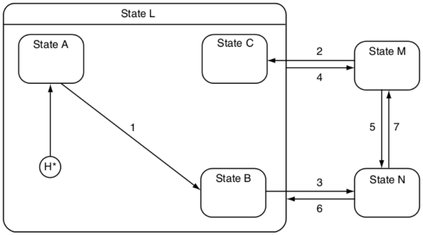
A triggered transition may be: (i) to another state at current level, (ii) to a state higher up, or (iii) to a lower state

Example

- start in State L
- from H* transition to A
  `update` = [L-active, A-entry]
  current State [L, A]



- top-level state machine no valid transitions
  state machine L: current state [A], triggered transition 1 ⇝ stay at current level, transition to B, `update` = [A-exit, 1-actions, B-entry]
  top-level state machine accepts and adds L-active. current State [L, B].
- top level machine: triggered transition 4
  transition to State M, `update` = [L-exit, 4-actions, M-entry].
  current State is [M]. (state machine L still keeps State B)
- top level machine: triggered transition 5
  transition to State N, `update` = [M-exit, 5-actions, N-entry]. current State N
- top level machine: triggered transition 6
  transitions to State L, `update` = [N-exit, 6-actions, L-entry].
  state machine L has current state still State [L, B] ⇝ no B-entry action



- top-level state machine no transition; State [L, B] triggered transition 3.
  top-level state machine no triggers state machine L: B, transition has one level up
  update: B-exit
  top-level machine: transition to State N; `update` += [L-exit, 3-actions, N-entry]
- State N $\rightarrow$ transition 7 $\rightarrow$ State M
  ...
- top level machine: triggered transition 2. top-level state machine: transition down $\rightsquigarrow$ updateDown. state machine L: `update` = C-enter
  top-level state machine changes from State M to State L, `update` += [M-exit, L-entry, 2-actions]

Implementation

```
class HSMBase:
  struct UpdateResult:
  actions
  transition
  level
  def getAction(): return []
  def update():
    UpdateResult result
    result.actions = getAction()
    result.transition = None
    result.level = 0
    return result
  def getStates()

class State (HSMBase):
  def getStates():
    return [this]
  def getAction()
  def getEntryAction()
  def getExitAction()
  def getTransitions()

class Transition:
  def getLevel()
  def isTriggered()
  def getTargetState()
  def getAction()
```

```
class HierarchicalStateMachine (HSMBase):
  states # List of states at this level
  initialState # when no current state
  currentState = initialState
  def getStates():
    if currentState: return currentState.getStates()
    else: return []
  def update(): ...
  def updateDown(state, level): ...

class SubMachineState (State,HierarchicStateMachine):
  def getAction(): return State::getAction()
  def update(): return HierarchicalStateMachine::update()
  def getStates():
    if currentState:
      return [this] + currentState.getStates()
    else:
      return [this]
```

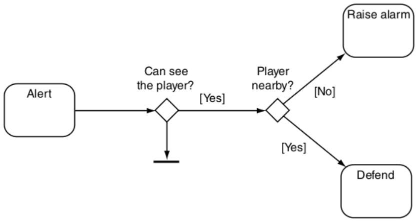
```
class HierarchicalStateMachine (HSMBase):
  states # List of states at this level
  initialState # when no current state
  currentState = initialState
  def getStates():
    if currentState: return currentState.getStates()
    else: return []
  def update():
    if not currentState:
      currentState = initialState
      return currentState.getEntryAction()
    triggeredTransition = None
    for transition in currentState.getTransitions():
      if transition.isTriggered():
        triggeredTransition = transition
        break
    if triggeredTransition:
      result = UpdateResult()
      result.actions = []
      result.transition = triggeredTransition
      result.level = triggeredTransition.getLevel()
    else:
      result = currentState.update() # rcrs.
```

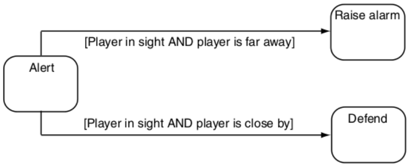
```
if result.transition:
  if result.level == 0: # Its on this level: honor it
    targetState = result.transition.getTargetState()
    result.actions += currentState.getExitAction()
    result.actions += result.transition.getAction()
    result.actions += targetState.getEntryAction()
    currentState = targetState
    result.actions += getAction()
    result.transition = None # so nobody else does it
  else if result.level > 0: # it is for a higher level
    result.actions += currentState.getExitAction()
    currentState = None
    result.level −= 1
  else: # It needs to be passed down
    targetState = result.transition.getTargetState()
    targetMachine = targetState.parent
    result.actions += result.transition.getAction()
    result.actions += targetMachine.updateDown(targetState, −result.level) # recursion
    result.transition = None # so nobody else does it
else: # no transition
  result.action += getAction()
return result
```

```
def updateDown(state, level):
  if level > 0: # continue recursing
    actions = parent.updateDown(this, level−1)
  else: actions = []
  if currentState:
    actions += currentState.getExitAction()
  currentState = state # move to the new state
  actions += state.getEntryAction()
  return actions
```

Combining DT and SM

Decision trees can be used to implement more complex transitions



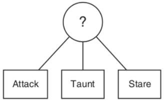

Raise alarm
[Player in sight AND player is far away]
Alert
[Player in sight AND player is close by]
Defend

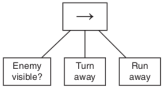
Outline

Behavior Trees

- synthesis of: Hierarchical State Machines, Scheduling, Planning, and Action Execution.
- state: task composed of sub-trees
- tasks are Conditions, Actions, Composites
- tasks return `true, false, error, need more time`
- Actions: animation, character movement, change the internal state of the character, play audio samples, engage the player in dialog, pathfinding.
- Conditions are logical conditions
- behavior trees are coupled with a graphical user interface (GUI) to edit the trees.

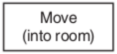
- Both Conditions and Actions sit at the leaf nodes of the tree. Branches are made up of Composite nodes.
- Composites: two main types: Selector and Sequence
- Both run each of their child behaviors in turn and decide whether to continue through its children or to stop according to the returned value.
- Selector returns immediately with a success when one of its children succeeds. As long as children are failing, it keeps on trying. If no children left, returns failure. (used to choose the first of a set of possible actions that is successful) Eg: a character wanting to reach safety.
- Sequence returns immediately with a failure when one of its children fails. As long as children are succeeding, it keeps on trying. If no children left, returns success. (series of tasks that need to be undertaken)

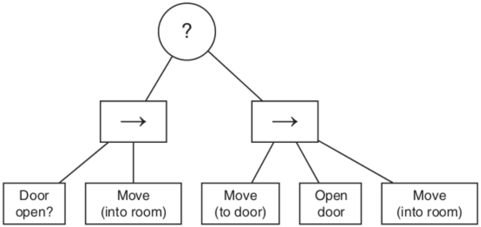


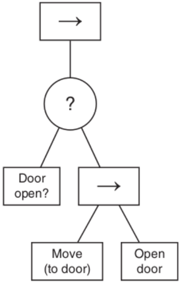


Developing Behaviour Trees

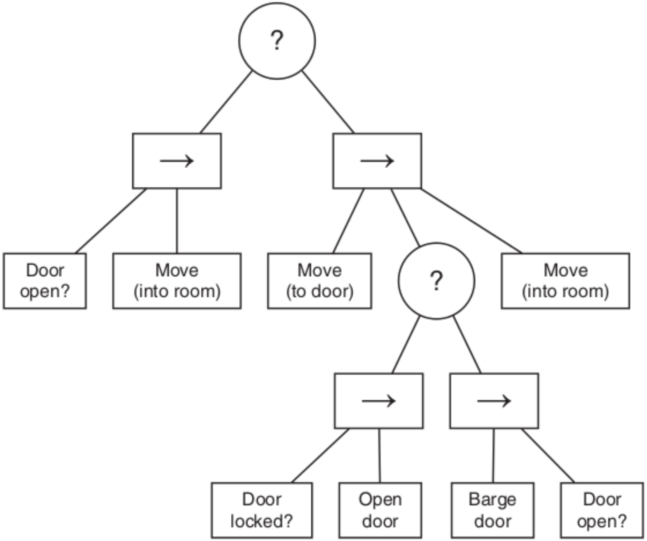
- get something very simple to work initially







Condition task in a Sequence acts like an IF-statement.
If the Sequence is placed within a Selector, then it acts like an IF-ELSE-statement


?
→
→
Door open?
Move (into room)
Move (to door)
?
Move (into room)
→
→
Door locked?
Open door
Barge door
Door open?

- behaviour trees implement a sort of reactive planning. Selectors allow the character to try things, and fall back to other behaviors if they fail. (look ahead only via actions)
- depth-first search
- could be written as state machines or decision trees but more complicated

Implementation

```
class Task:
  children
  def run() # true/false


class Selector (Task):
  def run():
    for c in children:
      if c.run():
        return True
    return False


class Sequence (Task):
  def run():
    for c in children:
      if not c.run():
        return False
    return True
```

```
class EnemyNear (Task):
  def run():
    if distanceToEnemy < 10:
      return True
    return False



class PlayAnimation (Task):
  animation_id
  speed
  def Attack(animation_id, loop=False, speed
        =1.0):
    this.animation = animation
    this.speed = speed
  def run():
    if animationEngine.ready(): # resource
          checking
      animationEngine.play(animation, speed)
      return True
    return False
```

Non-Deterministic Composite Tasks

- In some cases, always trying the same things in the same order can lead to predictable AIs.
- Selectors: eg, if alternative ways to enter the door, no relevant the order
- Sequences: eg, collect components, no relevant the order "partial-order" constraints in the AI literature.
- Some parts may be strictly ordered, and others can be processed in any order.

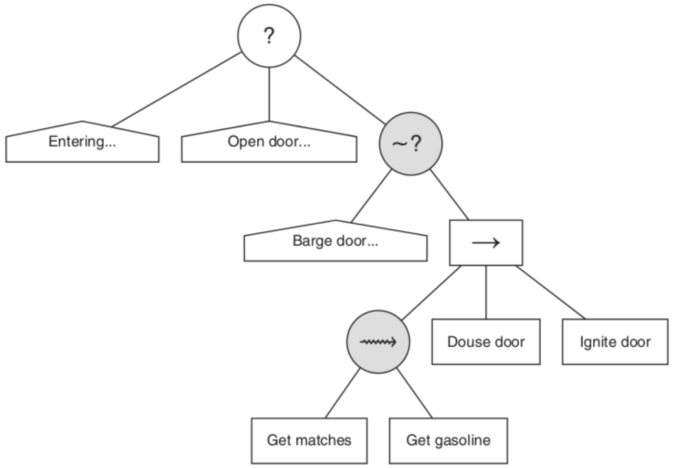
```
class NonDeterministicSelector (Task):
  children
  def run():
    shuffled = random.shuffle(children)
    for child in shuffled:
      if child.run(): break
    return result
```

```
class NonDeterministicSequence (Task):
  children
  def run():
    shuffled = random.shuffle(children)
    for child in shuffled:
      if not child.run(): break
    return result
```

Shuffle

by Richard Durstenfeld in 1964 in *Communications of the ACM*, volume 7, issue 7, as "Algorithm 235: Random permutation", and by Donald E. Knuth in volume 2 of his book *The Art of Computer Programming* as "Algorithm P" but originally by Fisher and Yates.

```
def shuffle(original):
  list = original.copy()
  n = list.length
  while n > 1:
    k = random.integer_less_than(n)
    n--;
    tmp = list[k], list[k] = list[n], list[n] = tmp
  return list
```


?
Entering...
Open door...
~?
Barge door...
→
⇝
Douse door
Ignite door
Get matches
Get gasoline

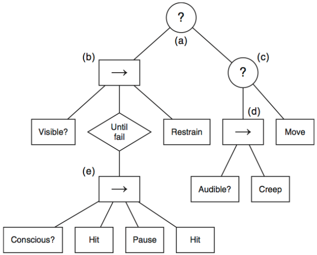
Decorators

- The decorator pattern is a class that wraps another class, modifying its behavior (from object-oriented software engineering).
- Composite that has one single child task and modifies its behavior in some way.

Like filters that:

- limit the number of times a task can be run (eg, does not insist with some action)
- keep running a task until it fails
- negation

Combination:



```
ex = Selector(
            Sequence(Visible, UntilFail(Sequence(Conscious,Hit,Pause,Hit)), Restrain),
            Selector(Sequence(Audible,Creep),Move)
           )
```

Resume