Tufts
UNIVERSITY

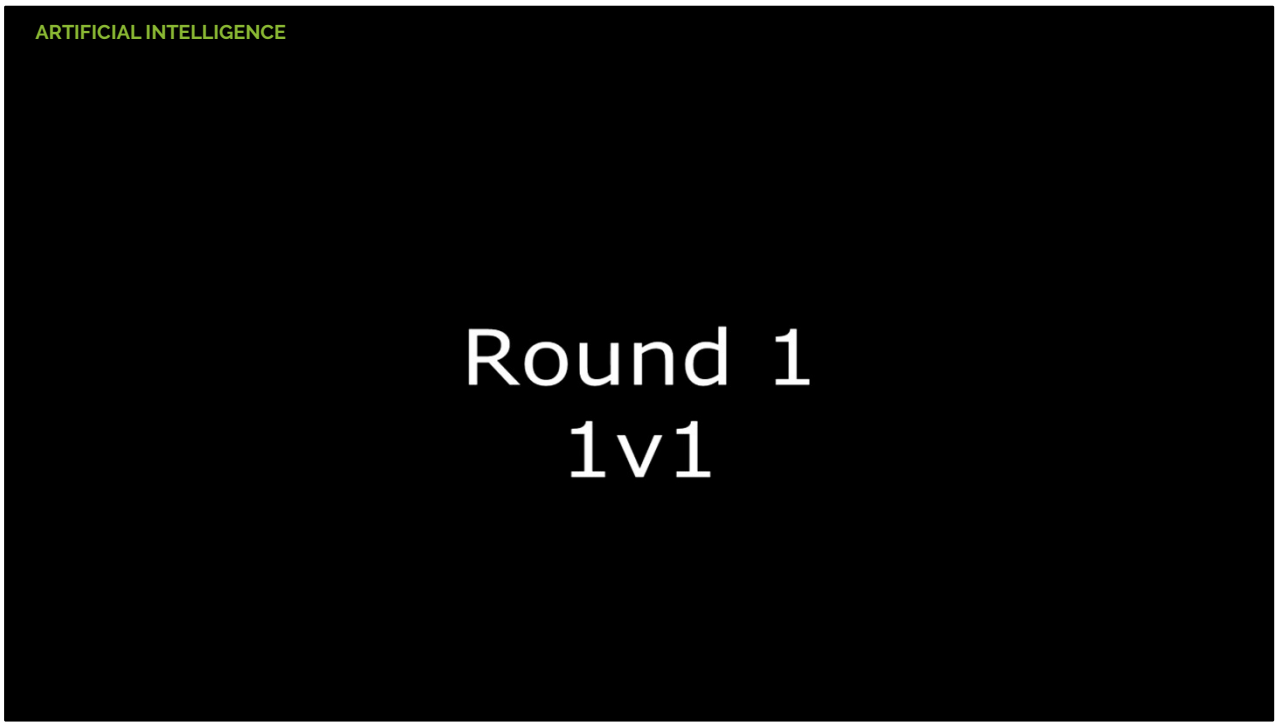
BEHAVIOR TREES

Fabrizio Santini | COMP 131A

VERSION 1.1

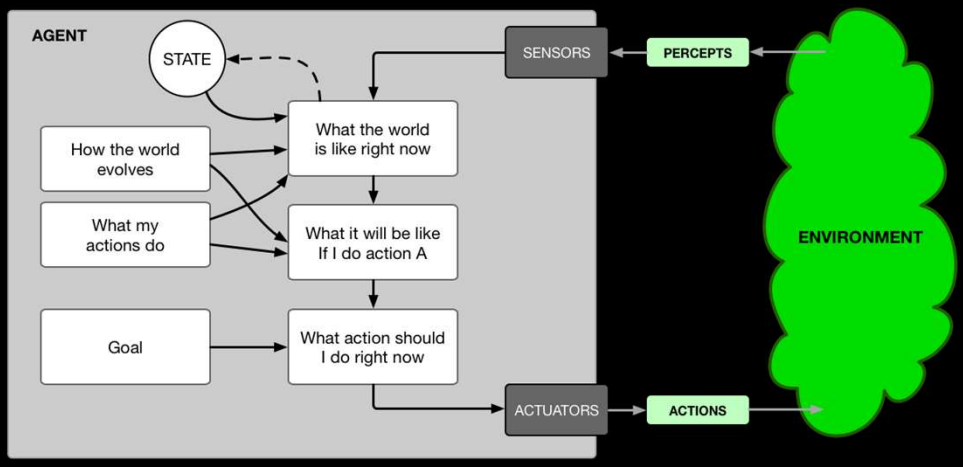
TODAY ON AI

- Reactive planning and Behavior Trees
- What's a Behavior Tree?
- What can a Behavior Tree do?
- Behavior Tree elements
- Behavior Tree features
- Design patterns
- Questions?


ARTIFICIAL INTELLIGENCE
Round 1
1v1

Reactive planning and Behavior Trees

SECTION 01


AGENT
STATE
SENSORS
PERCEPTS
What the world is like right now
How the world evolves
What my actions do
What it will be like If I do action A
Goal
What action should I do right now
ACTUATORS
ACTIONS
ENVIRONMENT

Wikipedia defines a Behavior Tree as: **Behavior trees are a formal, graphical modeling language used in Systems and Software Engineering**. Behavior trees employ a well-defined notation to unambiguously represent natural language requirements for large-scale software-integrated systems.

- Developed by **R. G. Dromey** with some key concepts published in **2001**
- Used to describe: large-scale systems, embedded systems, role-based access control, biological systems, etc.

R. G. Dromey, Genetic Software Engineering - Simplifying Design Using Requirements Integration, IEEE Working Conference on Complex and Dynamic Systems Architecture, Brisbane, Dec 2001.

- **Reactive planning** denotes a class of algorithms for **action selection** by autonomous agents
- **Reactive planning** is different from **classical planning**:
  - They are time-bound so they can quickly deal dynamic and unpredictable environments
  - They compute just one next action in every instant, based on the current context

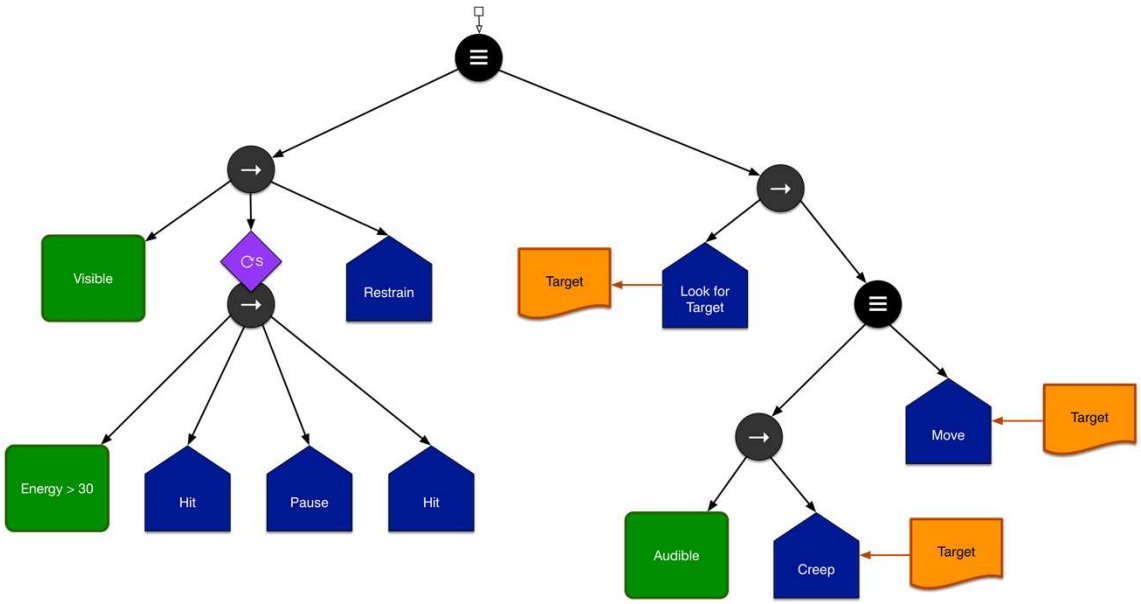
- requirements to avoid “short-term memory overload” and natural language ambiguities
-

- In 2004 and 2005, **Halo 2** and **Façade** AI designers adopted a similar graphical representation and name for a different formalism
- They wanted to synthesize a number of techniques and algorithms in one manageable tool:
  - Finite State Machines
  - Hierarchical Finite State Machines
  - Scheduling
  - Search
  - Resource Conflict Resolution

- requirements to avoid "short-term memory overload" and natural language ambiguities
-

Finite State Machines are not good for Game AIs

- While finite state machines are **reasonably intuitive** for simple cases, as they become more **complex** they are hard to keep goal-oriented
- As the number of states increases, the transitions between states become exponentially complex
- Hierarchical state machines can help, but many of the same issues remain


Visible
C's
Restrain
Energy > 30
Hit
Pause
Hit
Target
Look for Target
Move
Target
Audible
Creep
Target

Behavior Tree elements

SECTION 02

- Behavior trees organize their **nodes** into a **tree** or, more generally, **directed acyclic graph (DAG)**
- **Tasks**, **conditions**, **composites**, and **decorators** are the basic elements of a behavior tree. They are specific to the application field:

ELEMENT	REPRESENTATION	RESULT
A **task** alters the state of the system		SUCCEEDED, FAILED, or RUNNING
A **condition** tests some property of the system		SUCCEEDED, or FAILED
A **composite** aggregates tasks and conditions		SUCCEEDED, FAILED, or RUNNING
A **decorator** alters the basic behavior of the tree-node it is associated with		SUCCEEDED, FAILED, or RUNNING

- The most common composites are:
  - **Sequence**: Children are evaluated in order (left to right). It fails as soon as one of the children fails, otherwise it succeeds
  - **Selection**: Children are evaluated in order (left to right). It fails if all children have failed, otherwise it succeeds
  - **Priority**: Like selection, but the children are evaluated in order of priority
  - **Random sequence**: Like sequence, but the children are evaluated in random order
  - **Random selection**: Like selection but the children are evaluated in random order

- The most common decorators are:
  - **Logical negation**: It executes the attached node and then it negates its result

  - **Until Fail**: It executes the attached node until it fails

  - **Resource semaphore**: It resolves conflicts between nodes associated with the same resource

  - **Timer**: It executes the attached node for a specific amount of time

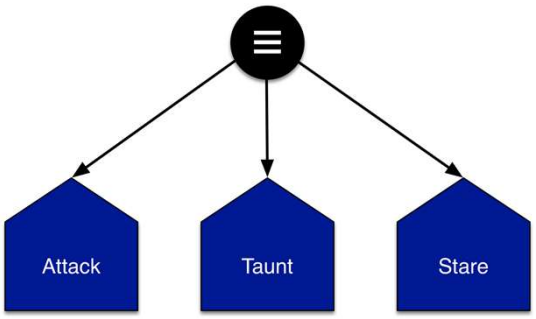
- Real Behavior Tree implementations normally have a inter-task communication mechanism called **blackboards**.

- The **blackboard** implementation is one big design choice:
  - **One** blackboard for the whole tree
  - **Sub-tree** private blackboards
  - All the above

- The simplest implementation of a blackboard is a **hash-table** or **dictionary**:
  - The **key** is the variable name
  - The **value** is the variable value

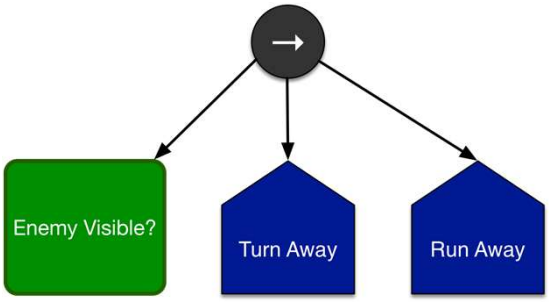
Data stored in the blackboard can have a simple nature (records, or class containers). They can also make use of the full power of the language specific RTTI (if available).

Try all the children until one succeeds.

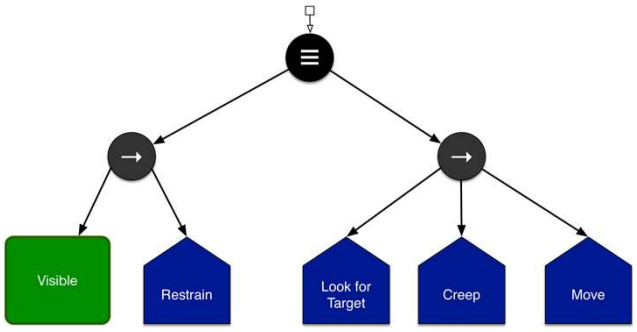


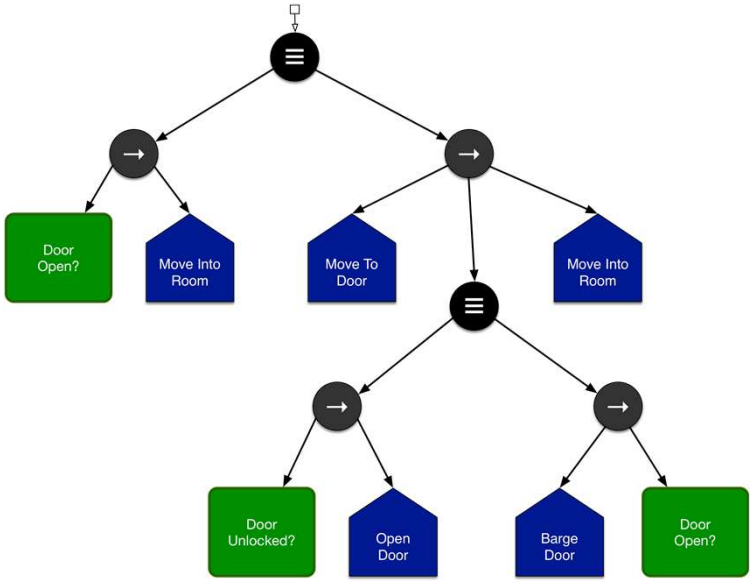

```
class Selector(Node)
  function run()
    for child in children
      if child.run()
        return TRUE
    return FALSE
```

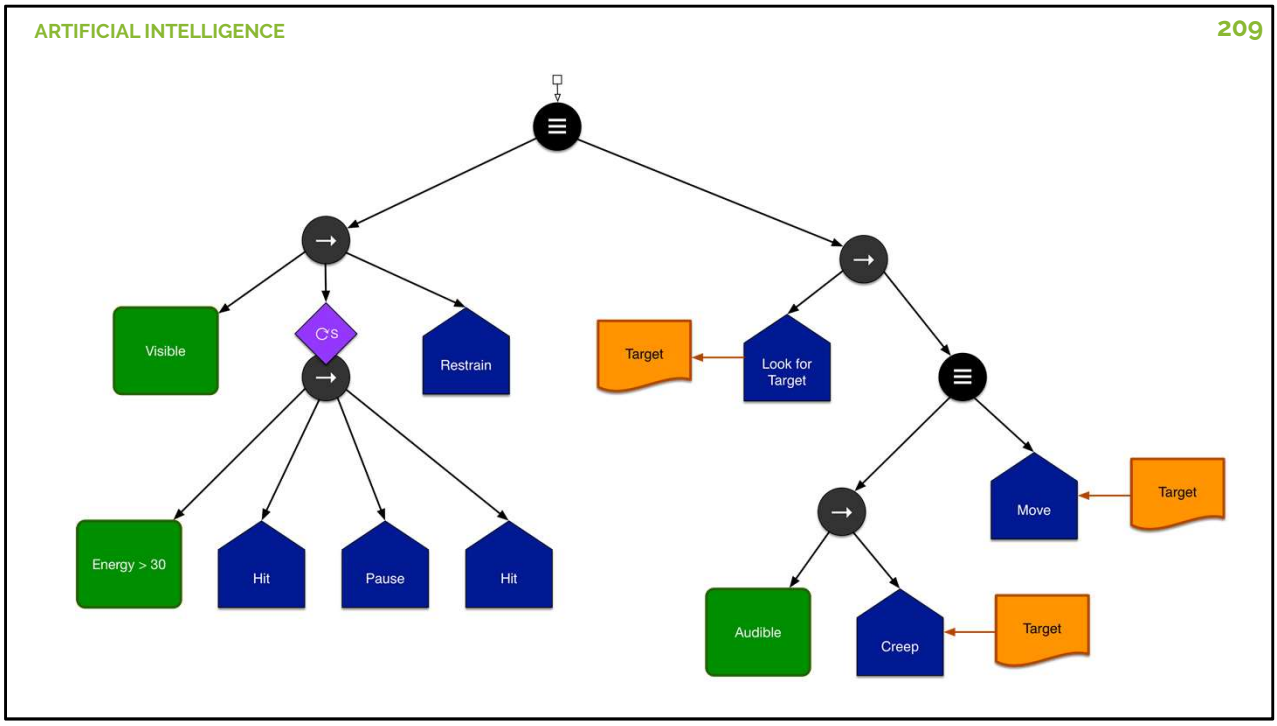
Execute all the children sequentially, succeeding if all succeed.




```
1 class Sequence(Node)
2   function run()
3     for child in children
4       if not child.run()
5         return FALSE
6     return TRUE
```


Visible
Restrain
Look for Target
Creep
Move


Door Open?
Move Into Room
Move To Door
Move Into Room
Door Unlocked?
Open Door
Barge Door
Door Open?


Visible
C's
Restrain
Target
Look for Target
Energy > 30
Hit
Pause
Hit
Move
Target
Audible
Creep
Target

Behavior Tree features

SECTION 03

BTs have a number of highly desirable features:

- The basic components are reusable
- Designers have full control
- AI can be goal directed and autonomous
- AI can respond to events
- The knowledge base is easy to read and debug
- The knowledge base is easy to maintain

Full control: micromanage behaviors – perfect designer supervision

Goal directed: Easier to separate goals from behaviors

BT features
Improvements over HFSM

BTs also have a number of improvements over hierarchical finite state machines:

- The history of the state transitions is clear
- Easy to build sequences
- Easy to add new behaviors without rewiring

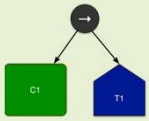
No recursion to confuse things

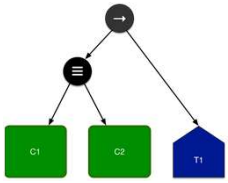
- It is hard to come up with a minimal set of tasks, conditions, and decorators
- Less important criticism:
  - Slow to react to changes in strategy
  - Do not implement a full search in the search space, rather a so-called “reactive search”

Design patterns

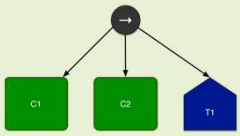
SECTION 04

TASKS	IF-THEN-ELSE	RULE NAME
	**return** C1	`Empty`
	**return** T1	`Always`

	**if** C1:
	  return T1
	else
	  return failed
		`Conditional execution`

	**if** C1 | C2:
	  return T1
	else
	  return failed
		`Conditional execution OR`

TASKS	IF-THEN-ELSE	RULE NAME

	**if** C1 & C2:
	  return T1
	else
	  return failed
		`Conditional Execution AND`

	**return not** C1	`Negation`

	**if not** Ti1.has_expired():
	  return T1
	else
	  return succeeded
		`Timer`

	**if** T1 == succeeded
	  return running
	else
	  return succeeded
		`Until fail`

QUESTIONS ?