CMSC 198I:

Guest Lecture:

Artificial Intelligence in Games

Guest Lecturer:

Dave Mount

Overview

- AI Basics and Agents
- Finite-State Machines
- Path Planning
- Group Motion and Flocking

Artificial Intelligence in Computer Games

What is Artificial Intelligence?
- "the study and design of intelligent agents"

Roles of AI in Games: Complex behaviors not specified by player
- **Non-player Opponents**: Realistic attack behavior
- **Non-player Teammates**: Coordinated supportive behavior
- **Support and Autonomous Characters**: Crowd/flock behavior

What AI is not:
- **Determined by physical laws**: E.g., path of a tennis ball
- **Purely random**: E.g., which block falls next in Tetris
- **Direct response to user inputs**: E.g., shoot gun when key pressed

Agents: The Ideal Opponent

Provide a challenging (but flawed) opponent:
- Should be **beatable** (but not too easily)
- Should be **entertaining** and fun

Not too challenging:
- Should **not be superhuman** in accuracy, precision, sensing, ...

No unintended weaknesses:
- No "**golden path**" to defeating opponent every time
- Must not fail miserably or look **dumb** (e.g., getting lost)

Not be too predictable:
- Through **randomness**
- Through **multiple**, fine-grained **responses**
- Through **adaptation** and learning

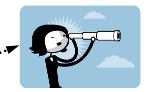
Agents: What are They?

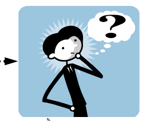
Definition:

An **autonomous agent** is a system situated within and a part of an environment that **senses** that environment and **acts** on it, over time, in pursuit of its **own agenda** and so as to effect what it senses in the future

Structure of Agent Action:

- **Sensing**: perceive features of the environment
- **Thinking**: decide what action to take to achieve its goals, given the current situation and its knowledge
- **Acting**: doing things in the world





Overview

- AI Basics and Agents
- Finite-State Machines
- Path Planning
- Group Motion and Flocking

Finite-state machines (FSMs)

Finite-State Machines:

- A (finite) set of possible **states** that the agent can be in
- Connected by **transitions** that are triggered by a **events** or **changes** in the world or user input
- Normally represented as a **directed graph**, with the edges labeled with the transition event

Ubiquitous:

- Almost **all** computer game AI systems support FSMs

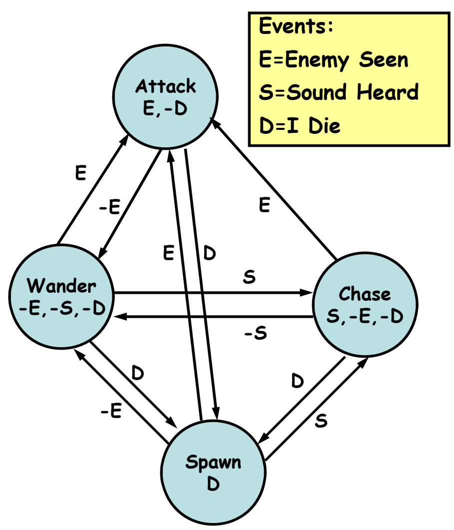
Zombie-Agent Example

Types of behavior to capture:

- If you don't see an enemy, **wander** randomly
- When see enemy, **attack**
- When hear an enemy, **chase** enemy
- On dying, **re-spawn**

Possible variations/additions:

- If health is low and seeing an enemy, **retreat**



Efficient FSM Implementation

Basic Implementation:

- Compile into an **array** indexed by [state-name, event]
- **Transition**: state-name$_{i+1}$ ← FSM[state-name$_i$, event]
- Switch based on state-name to call execution logic

Nondeterministic FSMs: (To reduce predictability)

- Have array of **possible transitions** for every (state-name, event) pair
- Choose one at **random**, based on transition probabilities

Overview

- AI Basics and Agents
- Finite-State Machines
- Path Planning
- Group Motion and Flocking

Motion Planning

Motion planning: Move an agent from one position to another

- Avoiding (fixed) **obstacles**
- Avoiding other **moving objects**
- **Goals**: Smooth, compact, natural motion. Robustness

Formulation: Given a **start point** s and a **destination point** t, find a **minimum cost path** from s to t that **avoids obstacles**

Cost: Distance, travel time, …Travel time may depend on terrain, for instance

Dynamic environments: Obstacles may appear or be removed

Limited knowledge: No map; just what sensors provide

Call of Duty 3

Generic Path-Finding Algorithm

Generic search algorithm: From source s to destination t

- **Initialization**:
  - Mark s as discovered, and mark all other nodes as undiscovered
  - dist[s] ← 0, and dist[u] ← ∞ for all other nodes
  - Create a priority queue containing only s
- Repeat until reaching t:
  - **Extract**: Remove the node u of highest priority from the priority queue
  - **Process**: For each unfinished neighbor v of u:
    - dist[v] ← min(dist[v], dist[u] + weight(u, v))
    - Mark v as discovered and add to/update priority queue
  - **Finish**: Mark u as finished

Algorithm invariant: At any time there are three types of nodes:

Undiscovered: Haven't seen this node yet

Discovered: A neighbor has seen you. You are waiting in the queue

Finished: You have completed processing. dist value will not change

Search Algorithms

Making the generic algorithm concrete:

- How are **priorities** assigned to the discovered but unfinished nodes?
- The choice of priority affects the **correctness** and **efficiency** of the search algorithm

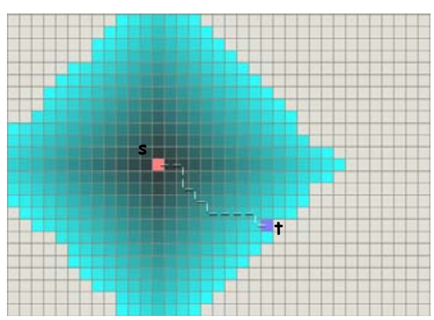
Common Search Algorithms:

- Breadth-First Search and Dijkstra's Algorithm
- Best-first (Local Greedy) Search
- A* Search

Dijkstra's Algorithm

Dijkstra's Algorithm:

- The **priority** of node u is the **distance estimate**, dist[u]
- The unvisited node u with the smallest **dist[u]** is processed next
- Under the assumption that edge weights are non-negative, this **correctly** finds the shortest path
- The order in which vertices are visited is **independent** of the location of the **destination** node



Breadth-First Search:

- When there are **no edge weights**, can replace the priority queue with a simple **FIFO queue**

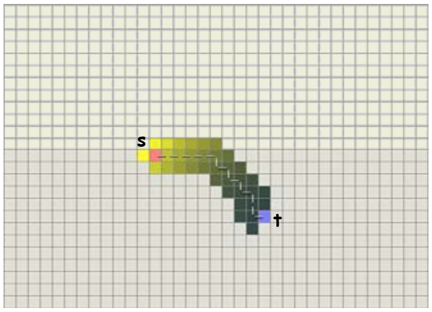
Best-First Search: A Greedy Heuristic

Best-First Search:

- The priority of a node is its **Euclidean distance** from t
- Next node to process is the discovered node that is **closest** to t
- This heads straight to t, and so is potentially much **faster** than Dijkstra's algorithm or Breadth-First Search

Bad news:

- This heuristic may **fail** when obstacles are present

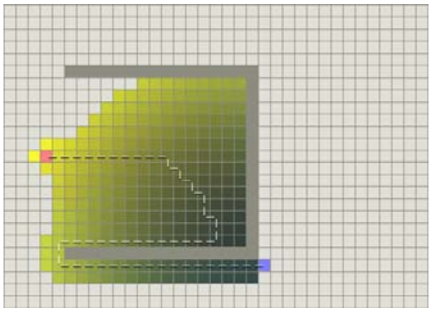


Best-First Search: Getting Caught

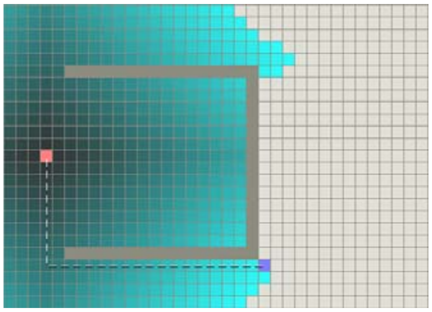
Best-First Search vs. Dijkstra:

- When obstacles are present, best-first search may get **trapped**, and will need to **backtrack**. The result is a suboptimal path
- Dijkstra's algorithm visits more nodes, but gets the **correct** path

Best-first Search



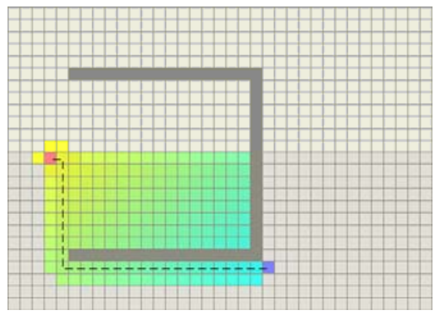
Dijkstra's Algorithm



A* Algorithm

A* Algorithm:

- Combines the **best aspects** of Best-First Search and Dijkstra
- The priority of a node is the **sum** of two functions:
  $f(u) = g(u) + h(u)$, where:
  - $g(u) = dist[u]$ (current **distant estimate** to u)
  - $h(u) = ||u\ t||$ (**Euclidean distance** from u to t)
- Dijkstra is a special case where $h(u) = 0$
- Best-first is a special case where $g(u) = 0$



Key:

- h(u) should be a **lower bound** on the remaining distance to t
- If h(u) is a good estimate, A* is superior to Dijkstra in practice

Overview

- AI Basics and Agents
- Finite-State Machines
- Path Planning
- Group Motion and Flocking

Group Motion and Flocking

Motion of multiple agents: Objectives:

- **Collision avoidance**.
- **Cohesion**:
  - Tendency to stick together
  - **Examples**: School of fish, flock of birds, herd of cattle
- **Common goal**?
  - Yes → Group seeks a common objective: A squadron of soldiers
  - No → Goals are independent: People walking on a street.

Emergent behavior:

- Define **simple rules** on individuals based on purely local information
- Each rule induces a **force**. Total force affects acceleration
- Interesting **global behavior** naturally emerges as a result

Flocking Rules

Boids: (virtual birds)

- Term coined by Craig Reynolds (1986)
- Behavior determined by the following rules:

Separation:

Avoid collisions with nearby boids

E.g., each boid generates a **repulsive potential field** of limited radius

Alignment:

Align flight direction with neighboring boids

Cohesion:

Attraction to centroid (center of mass) of the flock

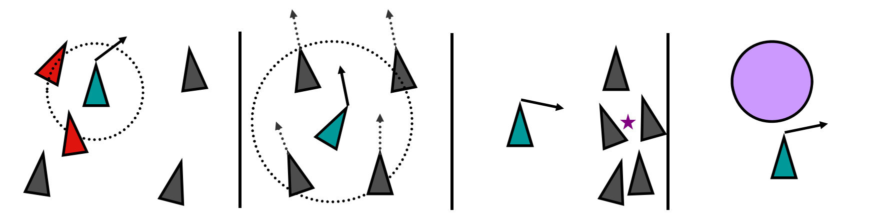
E.g., flock centroid generates an **attractive potential field**

Avoidance:

Avoid obstacles in the environment

E.g., each obstacles generates a **repulsive potential field**

Flocking Rules



Separation: Steer away from neighbors that are too close.

Alignment: Steer towards average heading.

Cohesion: Steer towards flock centroid.

Avoidance: Steer away from obstacles.

Combining Commands

Steering Rule:
- Steering rules act as **forces**; alter acceleration
- Set a **maximum acceleration** to avoid **jerky** behavior

Tuning Behavior:
- Assign a **weight** to each of the flocking rules
- Avoidance should be **high**
- Cohesion should be **lower**

Option 1: Apply rules in **decreasing weight order**, until max acceleration is reached

Responsive: Ensures that high priority forces are applied

Option 2: Take **weighted sum** and truncate to max acceleration

Fair: Ensures that all forces affect final motion

Demo: http://blog.soulwire.co.uk/laboratory/flash/as3-flocking-steering-behaviors

Flocking Evaluation

Advantages:

Simple: Easy to implement

Emergent Behavior: Complex behavior can emerge from simple rules

Flexible: Many varieties of behavior from a small set of different rules and varying parameter settings

Disadvantages:

Tuning: Can be difficult to set parameters to achieve desired result

Local Minima: Groups can get trapped in since there is no backtracking

Summary

- AI Basics and Agents
- Finite-State Machines
- Path Planning
- Group Motion and Flocking