

# AI Tutorial 2: Path-Finding and Crowd Management



# New Concepts

- ▶ Graph Search
  - ▶ The Game Environment as a Graph
  - ▶ Minimum-Cost Path-Finding through  $A^*$
  - ▶ Following the Path
- 
- ▶ Crowd Management
  - ▶ Approaches
  - ▶ Strengths
  - ▶ Issues

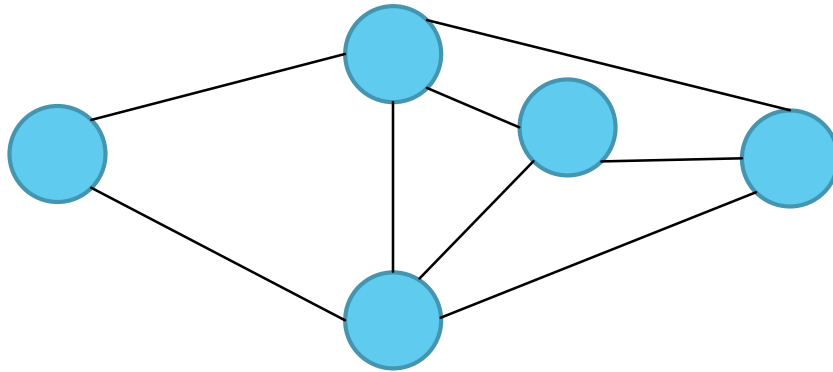
# Graph Search

# So, what is graph search?

- ▶ Unsurprisingly, it's the searching of a graph
- ▶ What's a graph?
- ▶ That, detective, is the right question.

# So, what is graph search?

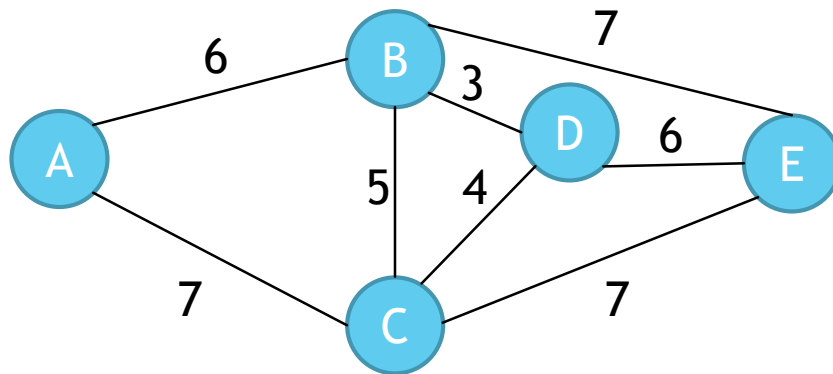
- ▶ A graph is a means of representing data through a series of vertices (points) and edges (connections)



- ▶ This graph has five vertices and eight edges

# So, what is graph search?

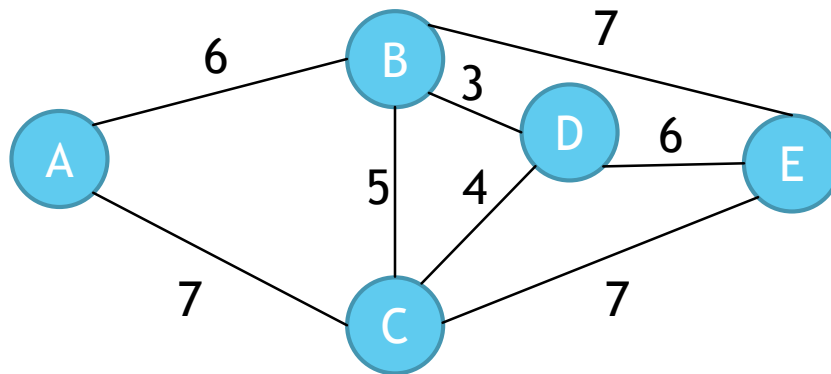
- ▶ It's easy to see how this abstract concept maps (pun intended) neatly to representation of an environment



- ▶ We add labels to the vertices, and costs to the edges.
- ▶ The the cost of moving from B to E is...?

# So, what is graph search?

- ▶ While it's good to know the cost of each route we can take to get from B to E, normally we just want the shortest route.



- ▶ That's the focus of today's tutorial - the means in which we can determine that, and how we can leverage graphs as a concept to support our AI

# Why would we use it in Games?

- ▶ Consider what we discussed yesterday
- ▶ Imagine complex scenarios
- ▶ Shortcomings of a naïve FSM approach
- ▶ The most elegant solution is one we can apply with a minimum of edge cases



# When would we use it in Games?

- ▶ A few examples are...
- ▶ The ghosts eyes heading back to base after Pac-Man has eaten them.
- ▶ The bad guys in Deus Ex or Metal Gear Solid heading toward the place where an alarm has been tripped, or the player has been spotted.
- ▶ Yorda running to the player when called for by Ico.
- ▶ The golden thread showing where to go next in Fable.
- ▶ The route plotted on the map to the desired destination in Red Dead Redemption

# A\* as a Path-Finding Algorithm

- ▶ In this tutorial, we focus on the heuristic path-finding algorithm A\*.
- ▶ A\* is an optimisation of the Dijkstra graph search algorithm
- ▶ Commonly employed in industry
- ▶ Not just for path planning

# A\* as a Path-Finding Algorithm

- ▶ There are three main steps in providing path-finding technology for AI agents in a game
- ▶ Represent the environment as a graph of small navigable units or nodes.
- ▶ Find a route connecting a series of these nodes from the starting point to the target location.
- ▶ Move the AI agent along that route convincingly.

# Representation of the Environment

# What do we need to know about our node?

- ▶ This is a very context-laden question
- ▶ It varies depending on how you've opted to structure your graph geography
- ▶ The essential elements are, in the simple case:
  - ▶ Unique Node ID
  - ▶ Connections
  - ▶ Position
  - ▶ Something which lets us know if the node is passable

# What do we need to know about our node?

- ▶ A more realistic node structure would resemble:

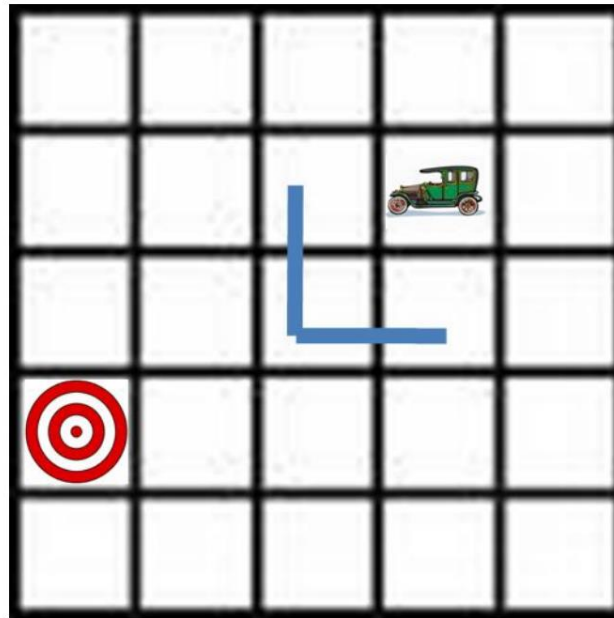
```
struct mapNode {  
    int nodeID;  
    int[] connectedNodes;  
    Vector3 position;  
    int terrainType;  
    int coverType;  
    bool passable;  
    bool isOnFire;  
    bool isFullOfDeathOhTheHumanity  
    ...  
};
```

# What do we need to know about our node?

- ▶ The reason there's no universal list is that the problem of what needs storing is contextual
- ▶ If you can algorithmically determine your connections, you don't need to maintain a list of connected nodes
- ▶ If you're using a list of connected nodes, you don't necessarily need a 'passable' Boolean, you can just delete impassable connections
  - ▶ Difference between representing roads between junctions as edges, and geographical representation of grids

# Our example scenario

- ▶ Consider the diagram right.
- ▶ This is a geographical division of the environment
- ▶ The car seeks to navigate from its location to the target





# Our example scenario

- ▶ In this example, we represent our environment in the simplest manner possible
- ▶ Booleans determine if a node is passable
- ▶ All connections can be computed algorithmically, as opposite

<i>Rows 5</i>
<i>Columns 5</i>
<i>0 0 0 0 0</i>
<i>0 0 1 A 0</i>
<i>0 0 1 1 0</i>
<i>B 0 0 0 0</i>
<i>0 0 0 0 0</i>

# The A\* Algorithm

# A\* Origins

- ▶ An extension of the Dijkstra search algorithm
- ▶ Enhances Dijkstra by adding a ‘best first’ heuristic
- ▶ Means that, in an ideal world, we’re only ever considering nodes which might feasibly feature in the final path
- ▶ Dijkstra, by contrast, requires us to search the entire graph

# A\* Underpinnings and Characteristics

- ▶ A\* is built upon the idea of an  $f$ -value which is computed for a node under consideration - the lower the  $f$ -value, the earlier we will consider the node.

$$f = g + h$$

- ▶ Where  $g$  is the cost taken to reach the node we're considering
- ▶  $h$  is the **heuristic** estimated minimum cost to reach the goal from the considered node, and thus
- ▶  $f$  is the estimate of the node's total cost; the cheaper the node, the earlier it is considered as a candidate

# A\* Underpinnings and Characteristics

- ▶ A\* maintains two lists throughout an execution, these are:
- ▶ The Open List. This is a list of nodes of which the algorithm is aware - think of it as nodes that have been looked at, but not explored yet. This is a priority queue, sorted such that the first considered element has lowest  $f$ -value
- ▶ The Closed List. This is a list of nodes which have been explored, accompanied by their parent nodes. All nodes on the final path will be in the Closed List, but not all nodes in the Closed List will be on the final path.
- ▶ Generating the path tracks from the goal, via parents.

# A\* Algorithm

**procedure** INITIALIZATION( $A, B$ , Open List, Closed List)

Let  $A$  be the Start Node

Let  $B$  be the Goal Node

Compute  $g$  and  $h$  for  $A$

Assign  $f$  to  $A$ , where  $f = g + h$

Add  $A$  to the Open List

**end procedure**

▷  $g$  will normally be 0, and  $h$  will depend on your heuristic

▷ At this point,  $A$  will be the only node on the Open List

# A\* Algorithm

**procedure** A\* SEARCH LOOP(Open List, Closed List, Node Graph,  $B$ )

Let  $P$  be the best node on the Open List

**while** Open List is not *empty* **do**

**if**  $P = B$  **then**

    Pop  $P$  off the Open List

    Push  $P$  onto the Closed List

    Go to Generate Path

**else**

    Let  $n$  be the number of nodes connected to  $P$

**for**  $i = 0; i = n; i++$  **do**

      Let  $Q$  be the  $i^{th}$  Node connected to  $P$

      Compute  $g$  and  $h$  for  $Q$

      Assign  $f$  to  $Q$ , where  $f = g + h$

**if**  $Q$  is on the Open List and  $f(Q)_{new} \geq f(Q)_{old}$  **then**

        Do nothing

**else if**  $Q$  is on the Open List and  $f(Q)_{new} < f(Q)_{old}$  **then**

        Set parent of  $Q$  on the Open List to  $P$

**else**

        Add  $Q$  to the Open List with parent  $P$

**end if**

**end for**

    Pop  $P$  off the Open List

    Push  $P$  onto the Closed List

    Let  $P$  be the new best node on the Open List

**end if**

**end while**

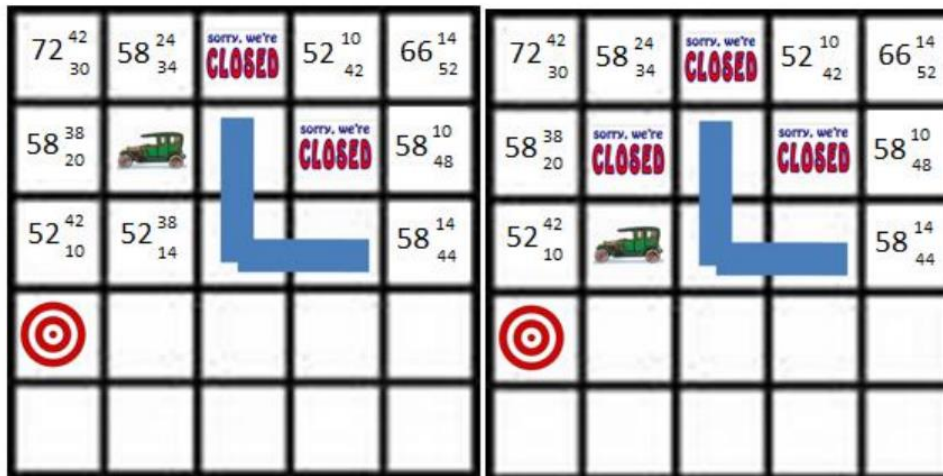
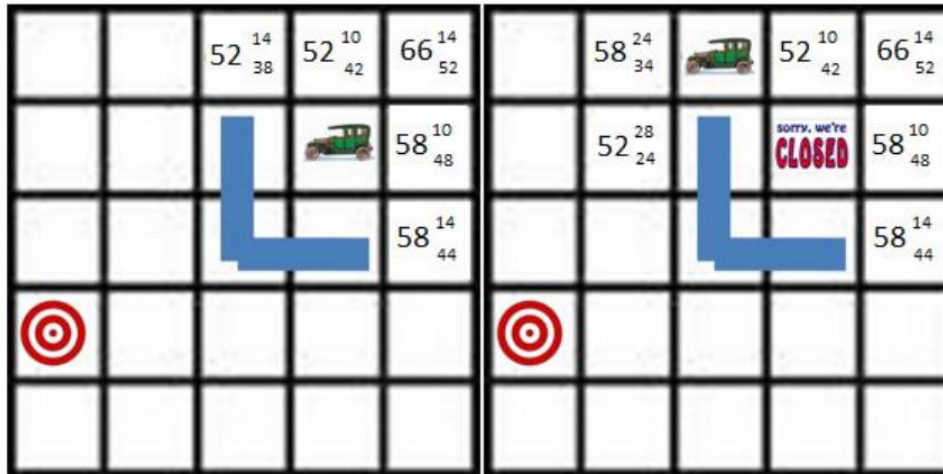
**end procedure**

# A\* Algorithm

```
procedure GENERATE PATH(Closed List, Path,  $A$ ,  $B$ )  
  Let  $R = B$   
  while  $R \neq A$  do  
    Add  $R$  to Path  
    Let  $S$  be the parent of  $R$   
     $R = S$   
  end while  
  Add  $A$  to Path  
end procedure
```



# A\* Algorithm Applied to the Scenario



# A\* Algorithm Addenda

- ▶ Often in high performance simulations, integer math is employed
- ▶ In our example, 10 is used for horizontal/vertical movement, and 14 for diagonal
- ▶ This is acceptable because the heuristic is an underestimate
- ▶ If the heuristic is an overestimate, we LOSE OUR CORRECTNESS CONDITION

# Following the Path

# Well, we have a route. What now?

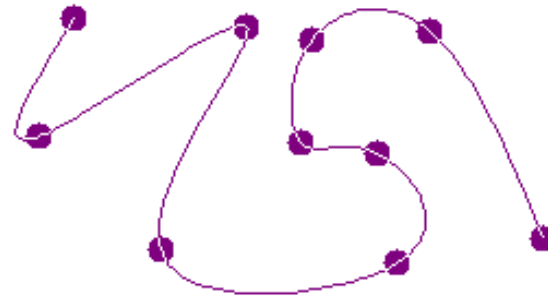
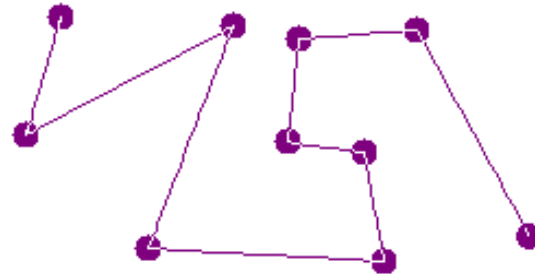
- ▶ Agents have to follow the path
- ▶ If we simply move them from the centre of one grid square/location of one node to the next, then turn and repeat, it will look robotic
- ▶ Appropriate for certain situations - like a robot
- ▶ Not appropriate for many scenarios in which we'll employ path-finding

# Following the Path Believably

- ▶ Simple method:
- ▶ Take the computed path and use this to generate a set of target points
- ▶ When the agent gets close enough to a target waypoint, it becomes influenced by the next waypoint, changing direction gently
- ▶ This smooths out corners

# Following the Path Believably

- ▶ More complex method:
- ▶ Use waypoints to generate a spline
- ▶ A spline is a curved line connecting a number of waypoints
- ▶ There are several ways to parameterise the creation of a spline
- ▶ Depending on parameterisation, the spline might not touch all waypoints, but it should follow the general shape



# Following the Path Believably

- ▶ As we smooth out our path, we might introduce obstacles which our node graph avoids
- ▶ First, probably a flaw with the way we're determining our node locations - we create the universe, we should know if the gap between two obstacles is so slender that splining about two points will make us collide
- ▶ Second, can address this using collision detection/response or, generally more appropriately, some form of FSM.

# Weaknesses of Graph Search

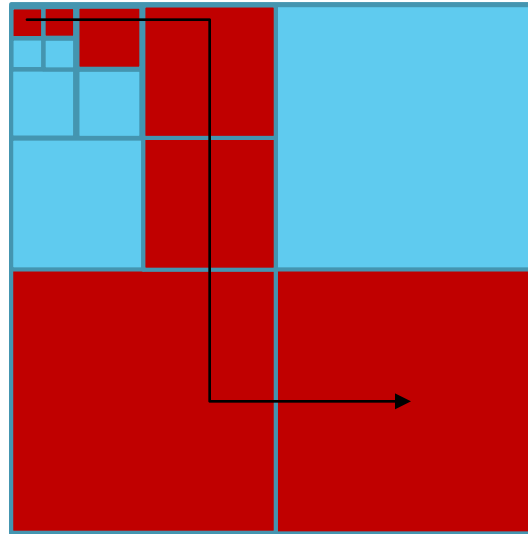


# Path-Finding is Expensive

- ▶ Representing the environment as a graph has a high memory footprint
- ▶ Computing a route through that graph has a large processor cost - larger without the heuristic,  $O(n \log n)$  vs  $O(n^2)$
- ▶ We need to be circumspect in its application (and the complexity of the scenarios to which it's applied)
- ▶ A\* only valuable if the heuristic is actually useful - consider a maze, this can be a poor scenario

# Path-Finding is Expensive

- ▶ Hierarchical path planning can be employed to reduce costs
- ▶ We can often substitute FSMs in for path-planning in simple pathing scenarios.
- ▶ A Goomba doesn't need A\*



# The Heuristic

- ▶ If using  $A^*$ , as opposed to Dijkstra, our heuristic guides us on the basis of which nodes it believes are most likely to lead to the goal (the closest nodes to the goal)
- ▶ The optimality condition here requires that the heuristic be a minimum possible cost.
- ▶ When given the choice between overestimating and underestimating, underestimate to guarantee optimality
- ▶ Or we'll explore more expensive nodes first, and potentially overlook cheaper options

# Crowd Navigation Overview

# What is it?

- ▶ The navigation of groups of entities
- ▶ Could be large crowds with individual AI
- ▶ Could be squad-size collectives with shared decision making
- ▶ Could be large crowds made up of squads, or squads made up of individuals with independent AI, etc., etc.

# Why do we care?

- ▶ We just covered graph search!
- ▶ Why don't we just use that?
- ▶ Complexity - can be insoluble
- ▶ Expense - even if soluble (decoupled) can be expensive
- ▶ Many navigation scenarios don't need explicit planning

# How do we go about it?

- ▶ Largely solutions are based on adapting other algorithms
- ▶ Borrow principles from physics (attraction, repulsion)
- ▶ Borrow principles from nature
- ▶ Important part: the algorithm should be scalable, meaning it should not exponentially increase in time as agents are added

# Moving as a “Fixed” Group



# Fixed Group Motion

- ▶ Appropriate for small groups - RPGs, squad-based RTS
- ▶ Agents need to move intelligently on the abstract level - navigate from point A to point B - but not necessarily on the agent level
- ▶ Computationally cheap and relatively simple for simple environments

# The Approach

- ▶ Define arrangement of entities (this will likely be an array of coordinates, one point per entity, relative to a centre-point). We shall call this a 'squad'.
- ▶ Using the centre-point as your starting position, perform an A\* search to determine a path to the squad's destination.
- ▶ Doesn't need to be A\* - if using some other algorithm to determine the squad's general direction, can use that instead

# The Approach

- ▶ Begin moving the centre-point of the squad along this route. This motion will, by definition, change the absolute coordinates of the entity position array - but their relative coordinates will remain the same.
- ▶ Apply a force to each member of the squad, in the direction of the updated location of its specific point.
- ▶ When the centre-point reaches its destination, after a given time period (long enough for entities to reach their own points relative to the centre-point), cease applying the force (to avoid oscillation).

# Fixed Group Motion

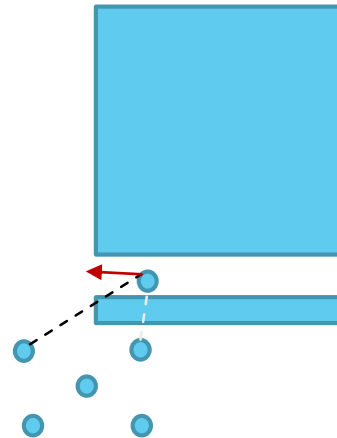


# Fixed Group Motion

- ▶ Efficient
- ▶ Plugs neatly into physics
- ▶ Scalable
- ▶ Can combine with flocking for large crowds of cohesive squads

# Fixed Group Motion

- ▶ VULNERABLE IN COMPLEX ENVIRONMENTS
  - ▶ Trapped entities
  - ▶ Dumb entities
- ▶ This can be offset through combination of Line-of-Sight checks against Nearest Neighbours



# Embedded Data

# Embedded Data

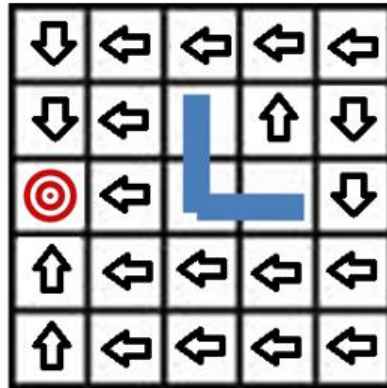
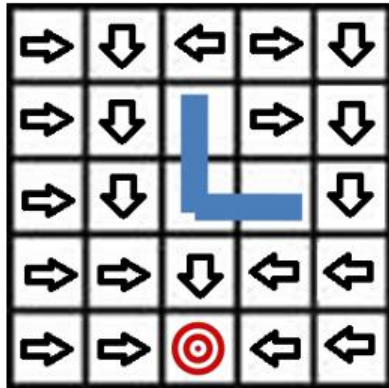
- ▶ Leverages data which can be computed at compile time
- ▶ Data can also be computed during loading
- ▶ Can be valuable even in systems with only a few agents
  - pre-computation saves framerate
- ▶ Limited utility in highly dynamic environments
  - ▶ Normally needs to be supplemented or recomputed in such cases



# The Approach

- ▶ Divide map into a grid/mesh, appropriate to geography
- ▶ Solve Dijkstra/A\* from every node, to every known goal node (VERY expensive in uncontrolled situations, but can be precomputed in many cases)
- ▶ Store the first 'direction' for each node, with respect to each goal; create a map of these 'first directions' for every goal node
- ▶ Entities reference this map when navigating, and flow in a direction defined by the grid/mesh region they occupy

# Example



# Notes

- ▶ Efficient at run-time
- ▶ Versatile - CAN account for deformable terrain, but not optimally
- ▶ Only appropriate for systems with known goal-points - can be generated at run-time, however, during level set-up. Not appropriate for generalised run-time generation.
- ▶ Bolts onto many other approaches (e.g. flocking, squads)
- ▶ Quick way to determine individual agent paths without computing A\* again

# Flocking

# Flocking

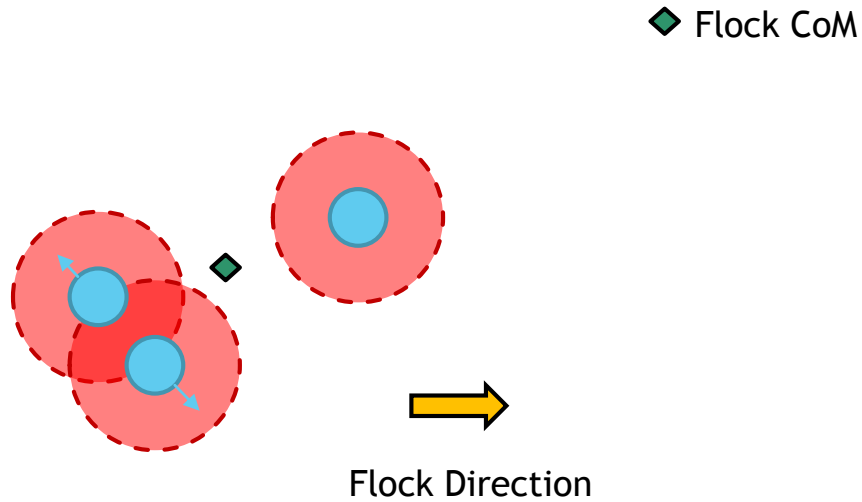
- ▶ Very cheap and efficient algorithm
- ▶ Three elements to consider
- ▶ Can model huge crowds - hundreds of thousands on modern hardware
- ▶ Easily scales and subdivides

# Flocking

- ▶ Three elements:
- ▶ Separation: How far an entity is from its nearest entities, to avoid crowding each other. This is important, as without this consideration eventually all Boids will wind up occupying the same position (or clustered into a massive mess of collision checks, if they have physical presence). This is a short-range repulsion.
- ▶ Alignment: The average direction of the flock is calculated (vector sum, then normalised).
- ▶ Cohesion: Steering towards the average position of the flock (mean positional vector sum). This is a long-range attraction

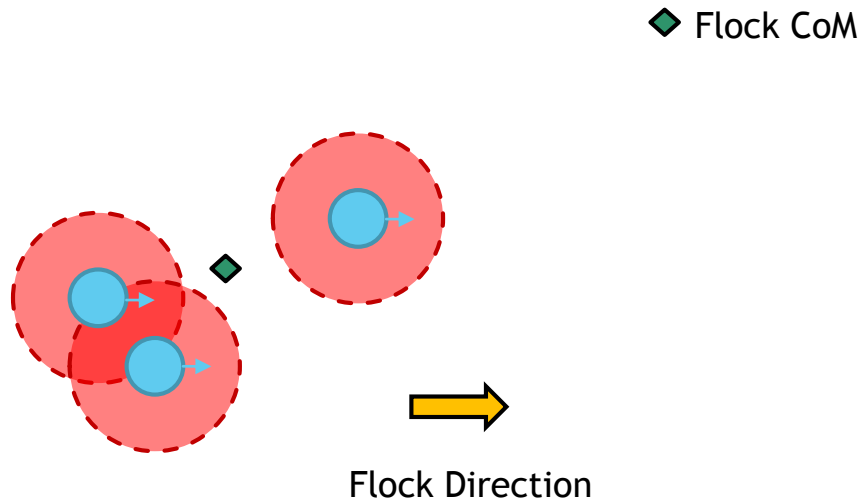
# Flocking

## ► Separation:



# Flocking

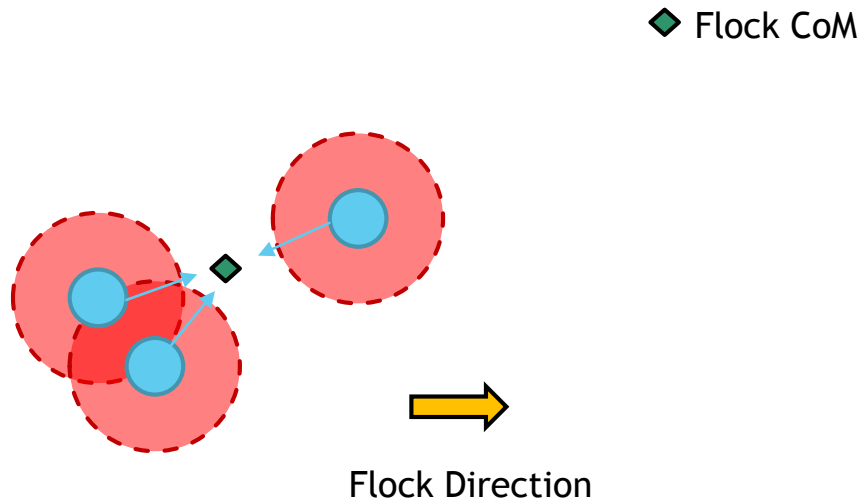
## ► Alignment:



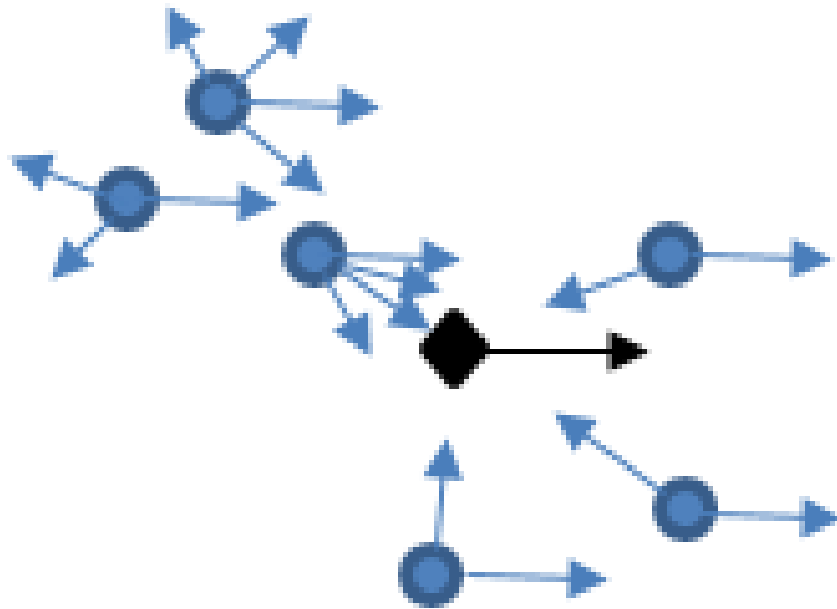


# Flocking

## ► Cohesion:



# Flocking



# Flocking

- ▶ Weighting factors can be applied to vary the importance of these three elements.
- ▶ Range of separation boundary can be varied - more or less cohesive
- ▶ Alignment can be taken from other algorithms ( $A^*$ , embedded data)
- ▶ Splining works for Boid alignment; only needs performing once, as boids naturally curve based on other factors over time

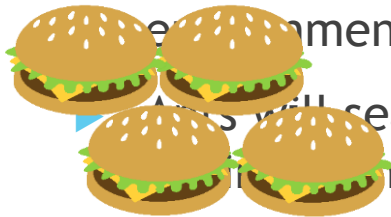
# Nature-Inspired Algorithms

# Adapting Strategies Found in the Real World

- ▶ Many algorithms which come under ‘traditional’ research AI domain have roots in biology
- ▶ Genetic Algorithms
- ▶ Neural Networks
- ▶ Behaviourism vs. Cognitivism
- ▶ Boids, in many ways
- ▶ Here, we discuss natural strategies for multi-agent navigation

# Ant Colonies

- ▶ An ant colony is comprised of a great many ants (our entities)
- ▶ At the beginning of our simulation, ants have no knowledge of their environment.



- ▶ Ants will search the environment for food

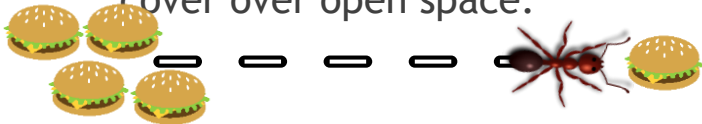


# Ant Colonies

- ▶ Upon finding food, they will return to the colony and leave a pheromone trail down the path they travelled



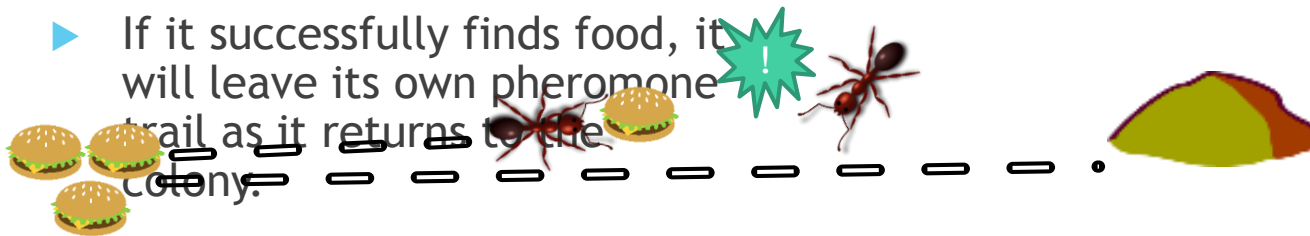
- ▶ This can be considered analogous to a weighting factor in intelligent path planning - like favouring cover over open space.



# Ant Colonies

- ▶ If another ant, while searching for food, crosses the pheromone trail it is likely to try and follow it, rather than continue wandering.

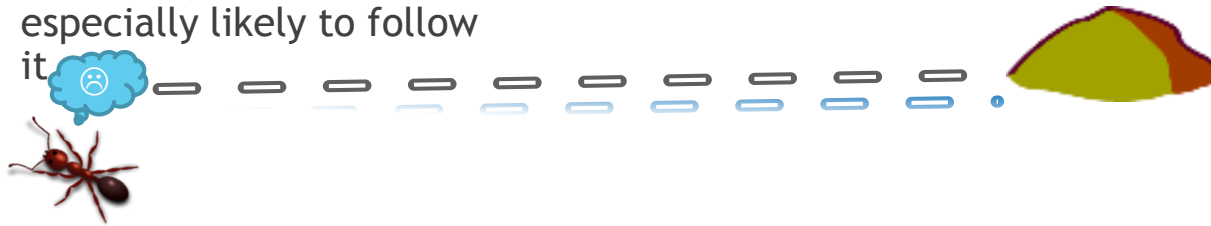
- ▶ If it successfully finds food, it will leave its own pheromone trail as it returns to the colony.





# Ant Colonies

- ▶ Over time, pheromone trails will naturally decay
- ▶ If a source of food is emptied, ants which follow the trail will not renew it
- ▶ Eventually will not be especially likely to follow it



# Ant Colonies

- ▶ Only applicable, really, in scenarios where strategic element matters
- ▶ Without a strategic element, this isn't suitable for many game-related path-finding scenarios, for crowds or otherwise
- ▶ Can be bolted on to other approaches, or vice versa, in order to swarm multiple agents believably
- ▶ Other natural world approaches provide similar behaviours, from similar bases.

# Key Reflections

# What should we take from this regarding Crowds?

- ▶ Generally, these algorithms have more in common with physics than AI.
- ▶ Consider flocking - where is the intelligence?
- ▶ The idea is normally “what direction should I head this frame”, not “which nodes are on the route from my present location to my goal location?”
- ▶ Usually, the individual agent has no concept of goal location
- ▶ Excellent example of the overlap between various areas within game engineering

# Summary

- ▶ Introduced Graph Search
  - ▶ Discussed how we might represent our environment as a graph
  - ▶ Explored heuristic graph search through the A\* algorithm
  - ▶ Discussed how graph search might be employed in other scenarios
  - ▶ Considered the weaknesses of graph search
- 
- ▶ Introduced crowd navigation
  - ▶ Discussed several approaches to the problem
  - ▶ Considered the influence of nature on path-finding
  - ▶ Considered the difference between path-finding and strategy - comes back to the 'AI engineering vs Design' aspect