Enemy Behaviors and Pathfinding

Negative Motivation

-As players attempt to reach their objectives, they are kept back or deterred by obstacles and enemies

- Obstacles could be anything from spikes, to a bottomless pit, to the terrain itself
- Obstacles can also follow a predetermined path - A sawblade navigating between two points
- Enemies are sometimes little more than obstacles
- They generally act with slightly more “agency” than spikes, often reacting to the player’s presence in some manner
    - Uninformed - Repeating some action, continuing movement along a path, etc
    - Informed - Such as approaching the player, attacking them, or some combination of the two
- In complex game environments with obstructions, path finding becomes a necessity to avoid easily exploited inconsistencies in enemy logic

Attack the Player!

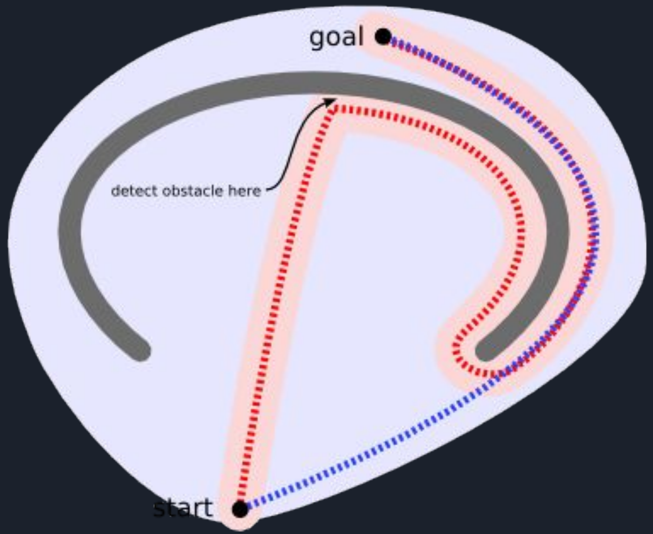
- Enemies may present a challenge to the player through their design, placement, or behaviors intended to deal with specific circumstances
  - Spikes may be placed in areas deemed likely for an average player to touch, requiring a particular movement pattern to avoid damage/death
  - Similarly, “dumb” enemies like goombas or koopas from Super Mario can be challenging due to their placement within a world
- Enemies intended to pursue the player may make due with simple movement mechanics, others will need more complicated logic that directly uses the player’s current location
- Involves use of “path finding” to identify a route to the objective following navigation rules

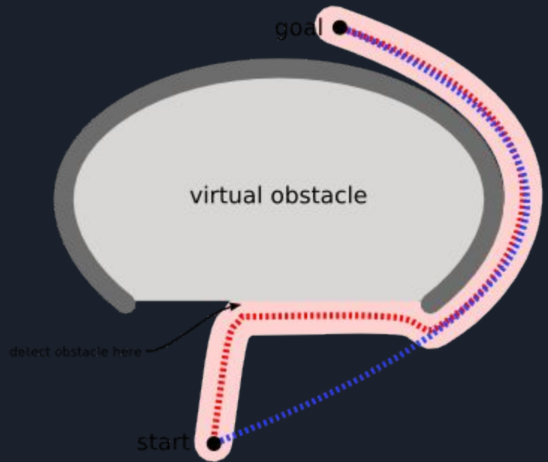
Simple vs Ideal Path

-Naive best first search - moves towards the goal, adapting to obstacles without any foresight (red)

-Ideal - finds and follows shortest path from start to goal (blue)

-Naive approach can be strengthened by identifying “dead ends” and other obstacles that will never be part of an ideal “through path” where A and B are both outside of the “dead end”



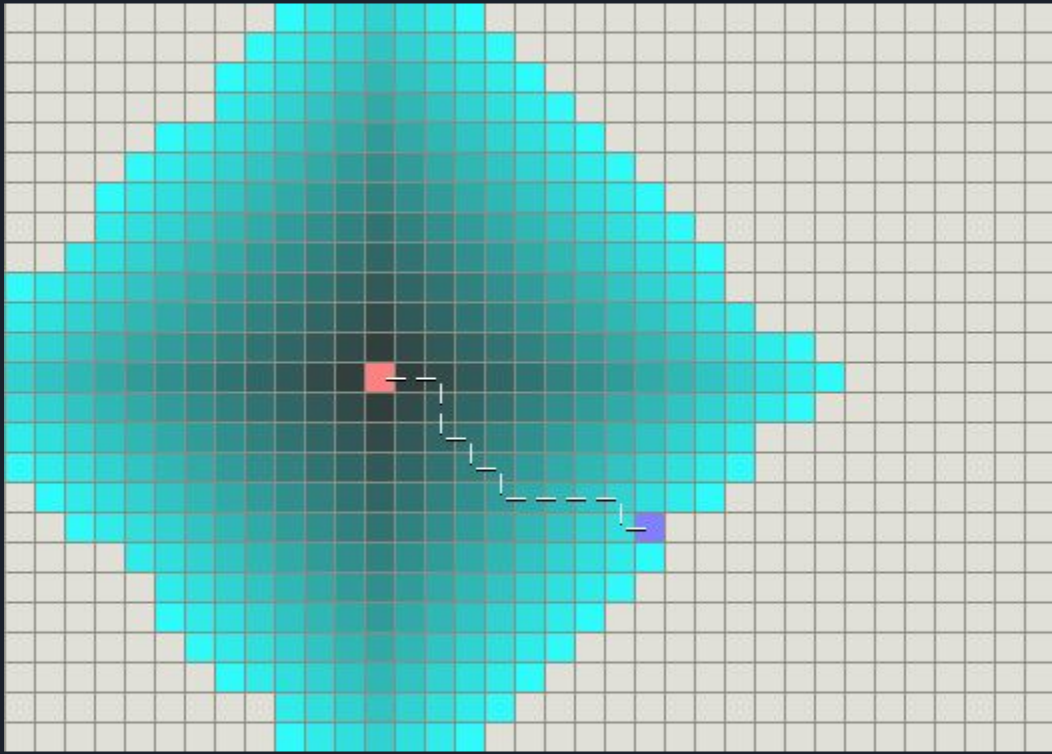


Dijkstra’s Algorithm

-Algorithm that determines the minimum cost to move from the origin to any given point

-Capable of tracking/reproducing the path taken

-As with other distance calculation/pathfinding, this works with connected graphs, but is more easily shown using grids

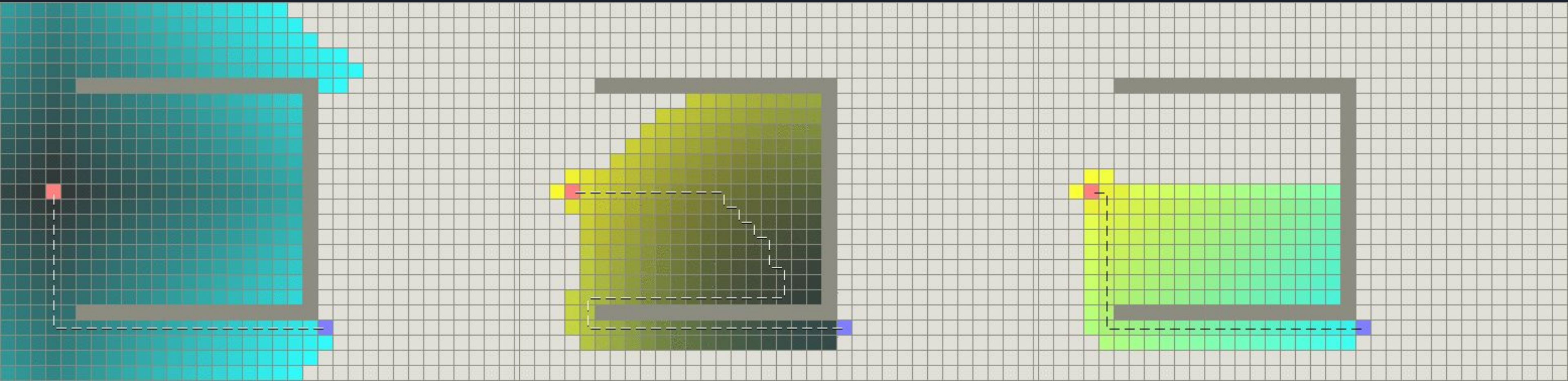


Astar

-Dijkstra's Algorithm finds good results, but is too unfocused

-Naive Best Case First searches are too blind/poor at handling unexpected obstacles

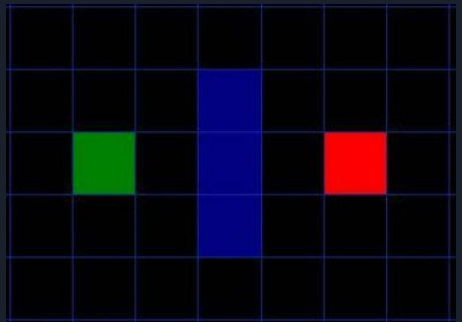
Can we get the best of both worlds? We sure can

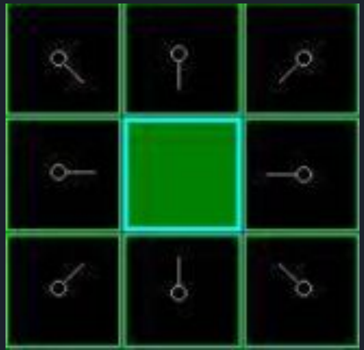


Astar

Astar is capable of finding a shortest path to the destination despite obstacles

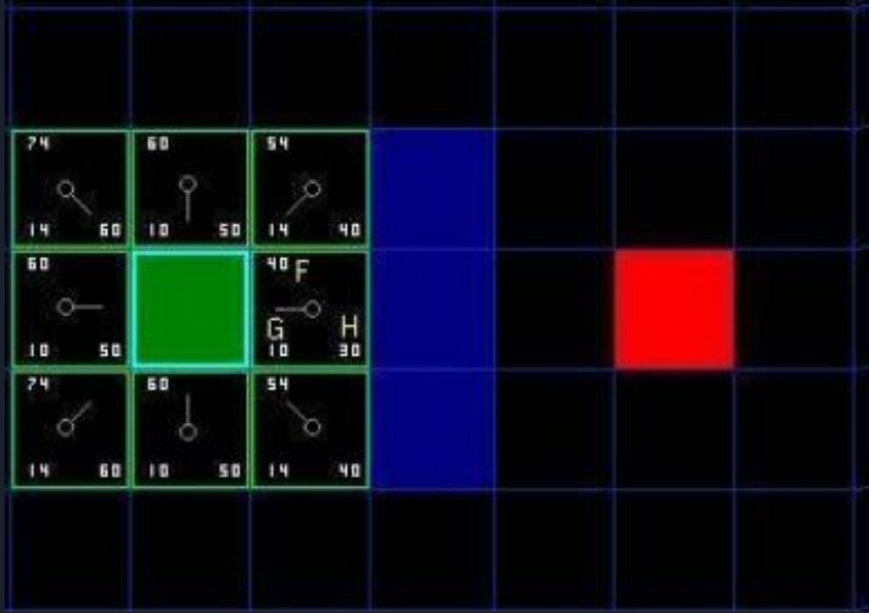
- Cells selected for expansion based on finding a value F
  - F = G + H
  - G = Cost to reach designated node from start
  - H = Estimated/heuristic cost to reach destination from designated node
- As we move out from the starting point, G increases, forcing our algorithm to prefer “shorter” (cheaper) paths
- As we move towards the destination, H decreases, encouraging paths that are closer to (eventually reaching) our destination
- Nodes keep track of the direction they were reached from the start, which is what allows for the path as a whole to be reproduced





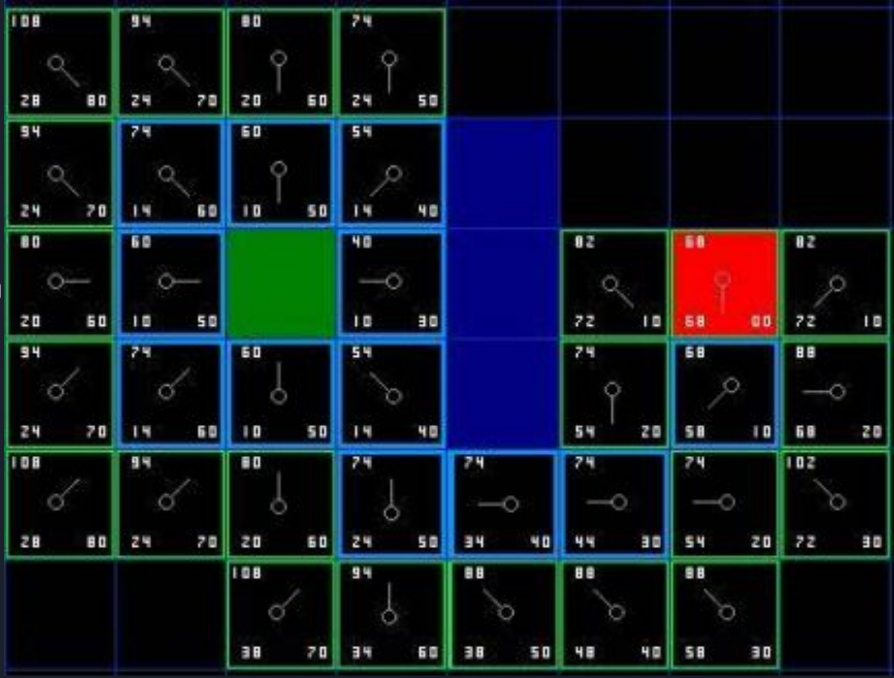
Astar

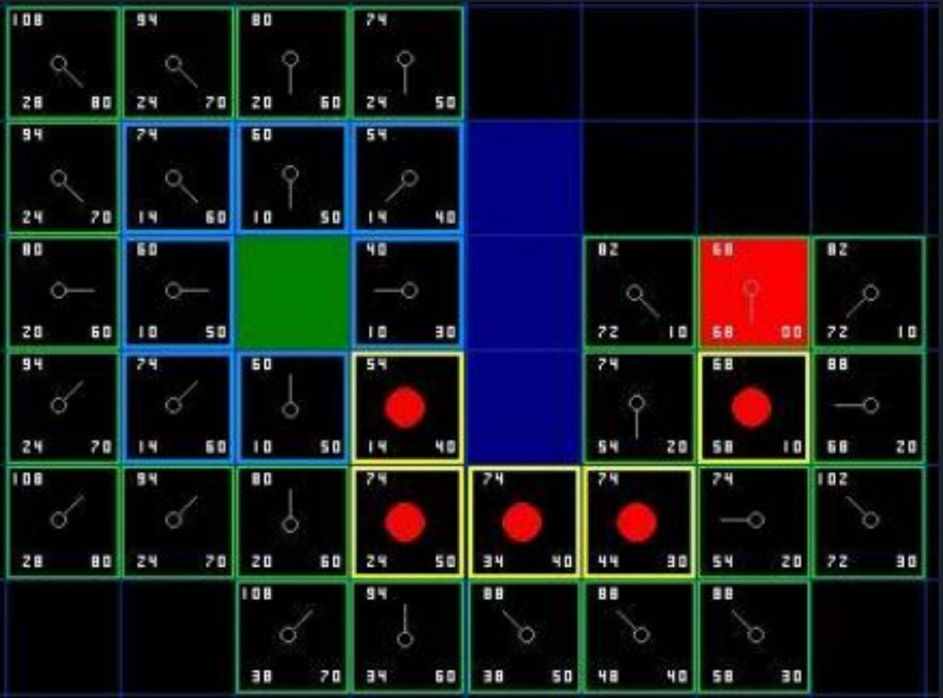
- F (top) = G (bottom left) + H (bottom right)
  - G = 10 for movement along faces
  - G = 14 for movement across corners
  - H = Estimated distance to destination
    - Here, we use Manhattan Distance (10 * (vertical distance + horizontal distance))
    - Other distance calculations can be used, and will have an impact on the chosen path
- Lowest F value is selected for expansion
  - In this case, dead end at F = 40!



Astar

-As list of Open Squares is processed, gradually the next most expensive node is expanded, until a node expands into the destination





Astar - Algorithm

// A* Search Algorithm
1. Initialize the open list
2. Initialize the closed list
   put the starting node on the open list (you can leave its f at zero)
3. while the open list is not empty
   a) find the node with the least f on the open list, call it "q"
   b) pop q off the open list
   c) generate q's 8 successors and set their parents to q
   d) for each successor
   e) push q on the closed list
   end (while loop)

i) if successor is the goal, stop search
ii) else, compute both g and h for successor
    successor.g = q.g + distance between successor and q
    successor.h = distance from goal to successor (This can be done using many ways, we will discuss three heuristics- Manhattan, Diagonal and Euclidean Heuristics)

    successor.f = successor.g + successor.h
iii) if a node with the same position as successor is in the OPEN list which has a lower f than successor, skip this successor
iv) if a node with the same position as successor is in the CLOSED list which has a lower f than successor, skip this successor
    otherwise, add the node to the open list
end (for loop)

Even More Methods

Navmesh - Generates polygons from traversable terrain. When moving to a point, path is chosen based on the navmesh

Node Based System - Destination selects nearest node, travel then deals with traveling to nearest node/distance before closer logic takes over

Visibility Graph - Tracks which corners can be seen by objects. Can behave similarly to node based system for pathfinding, can also be used for different goals (object loading/unloading)

References and Resources

- https://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html
  - Conceptual, effective at comparing and contrasting multiple approaches
- https://csis.pace.edu/~benjamin/teaching/cs627/webfiles/Astar.pdf
  - Solid explanation of motivations behind astar, mechanics and process
- https://www.geeksforgeeks.org/a-search-algorithm/
  - Provides simple representation of algorithm for a-star, and identifies different distance methods