Algorithms

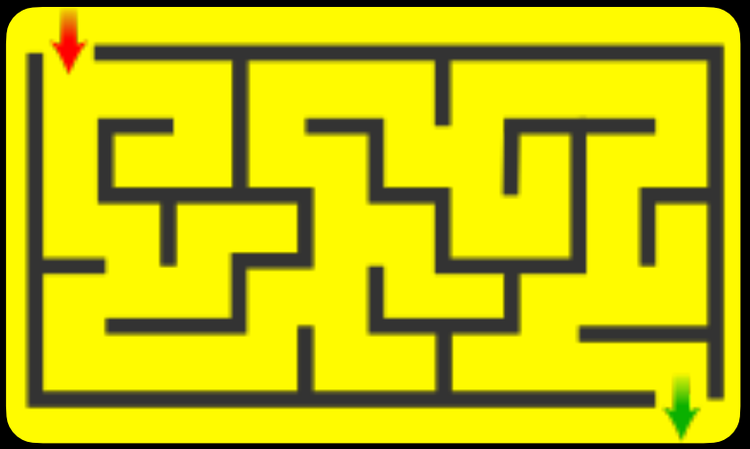
(for Game Design)

Session 4: Navigational Algorithms

Last time

- Mazes
- Maze Creation Algorithms
- Maze Solution Algorithms

Classifications

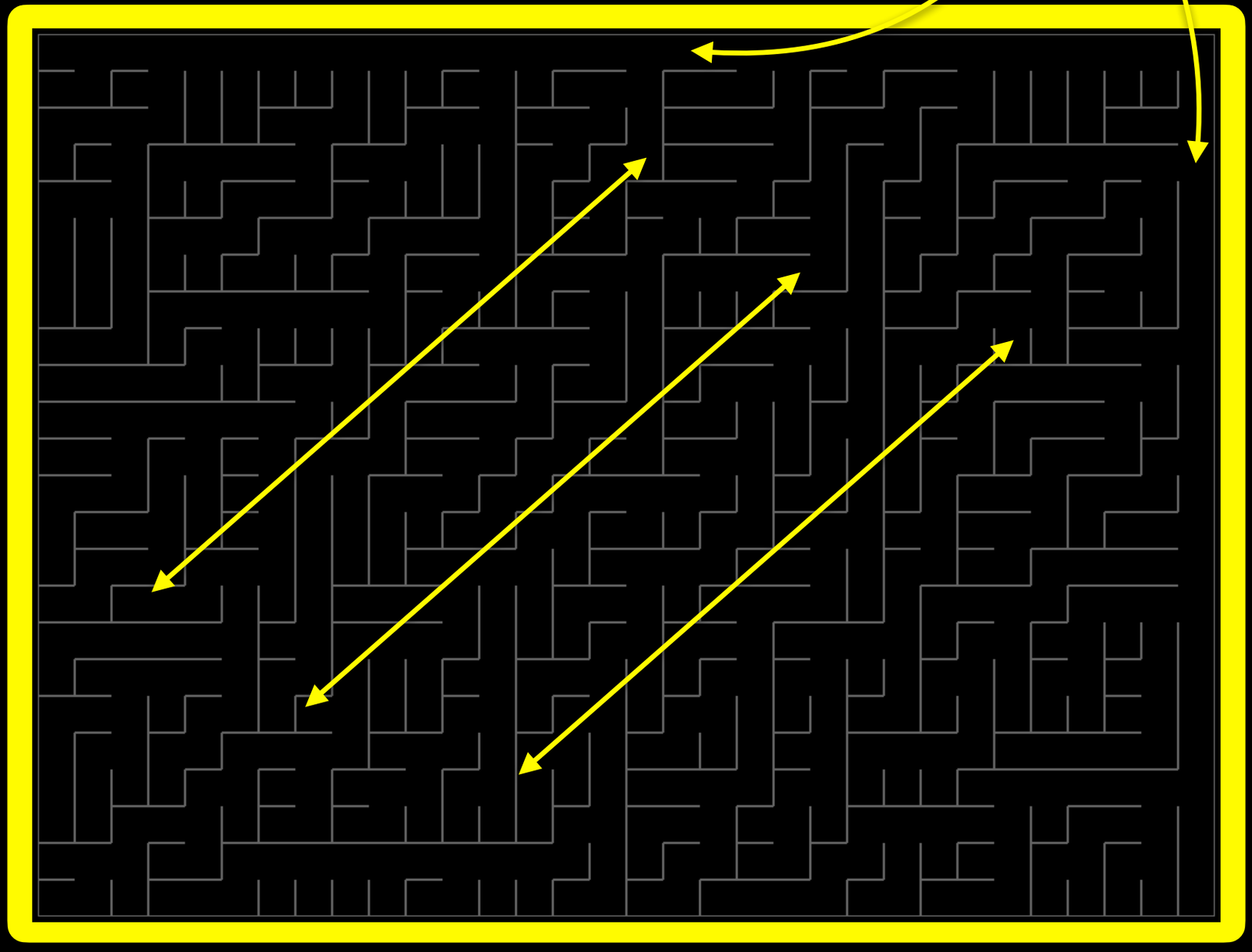


- **Perfect** maze: Has one and only one path between any two cells -- thus plenty of branching, but no loops
  - **Cell**: position in the maze, the smallest geographical element of the maze
- **Unicursal** maze: Has no branches, so there is just a single path from beginning to end

Result: Binary Tree

Top-row and Right-column are always clear

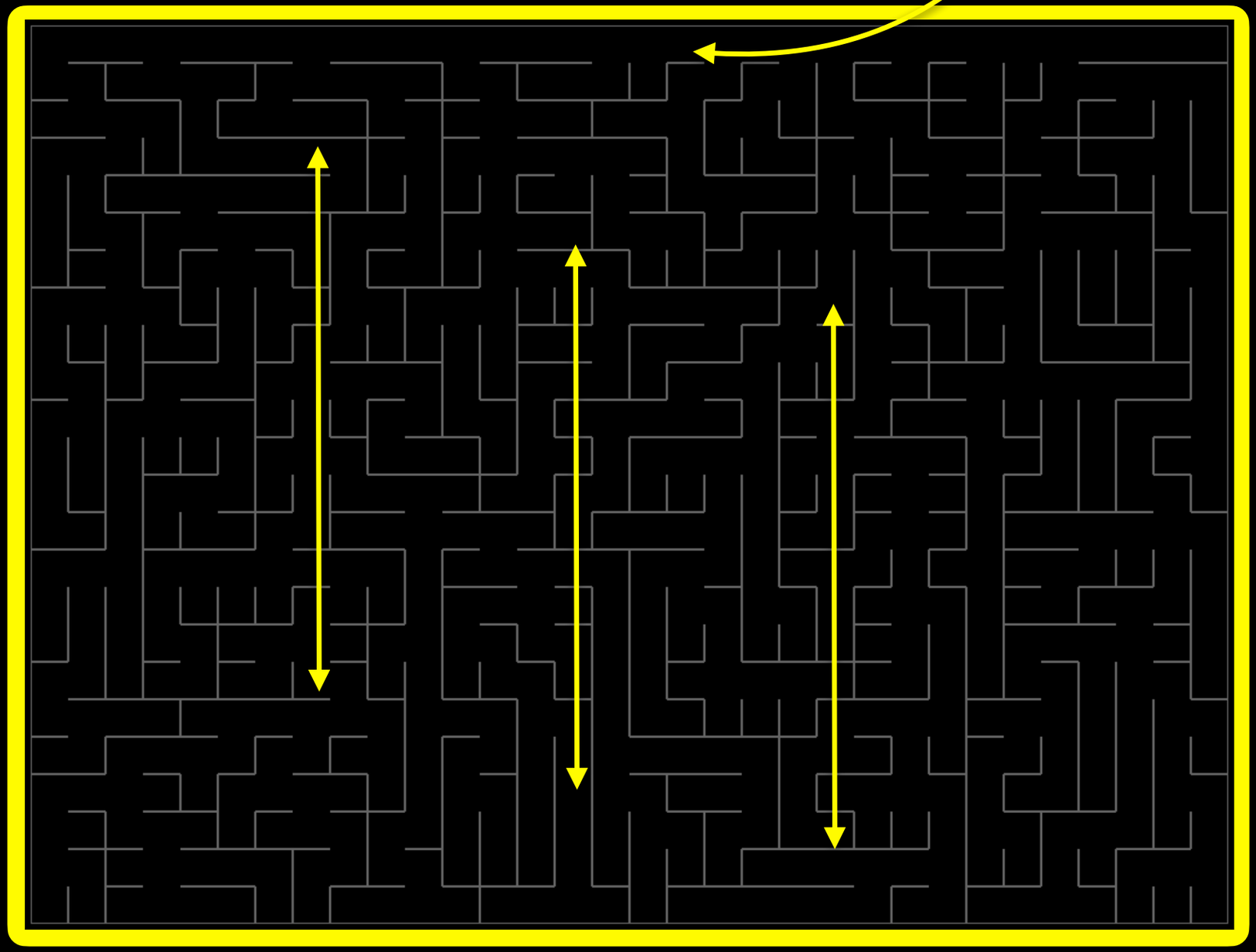
Bias



Result: Sidewinder

Top-row is always clear

Bias



Aldous-Broder

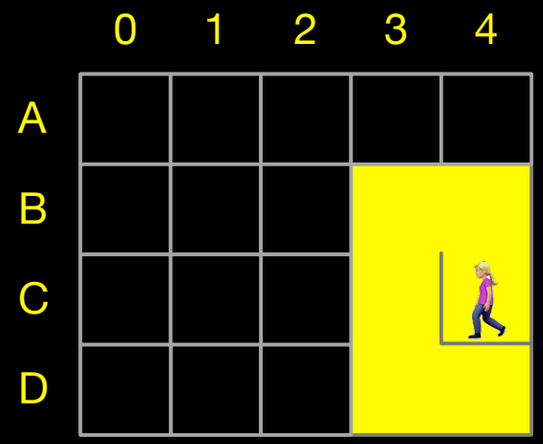
- Discovered simultaneously by two professors with those names
- A **random walk** algorithm: step from cell to random neighbor to create random paths

Wilson's Algorithm

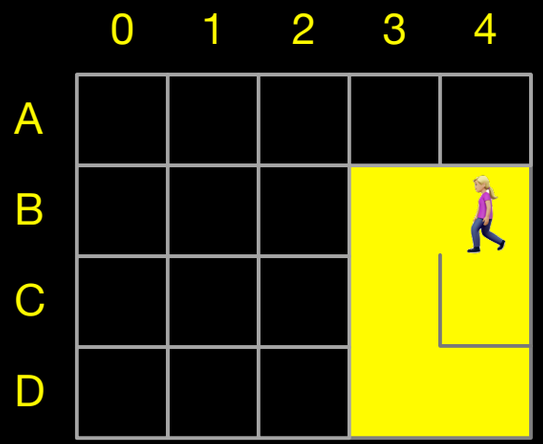
- Also a random walk algorithm
- Also creates mazes with no bias
- Also has inefficient performance characteristics

The Algorithm

- Anytime there is no unvisited neighbor, back up along your path until you do have one, which you pick



stack=[D4,D3,C3,B3,
B4,C4]



stack=[D4,D3,C3,B3,B4]

Dijkstra's Algorithm

- A very famous graph algorithm, named after a very famous Dutch computer scientist
- Used in network routing algorithms, map navigation, robot movement planning, and many others
- We will start off with a simplified version that works well for our mazes (a "breadth-first search")
    - In sessions to come, we will revisit it several times

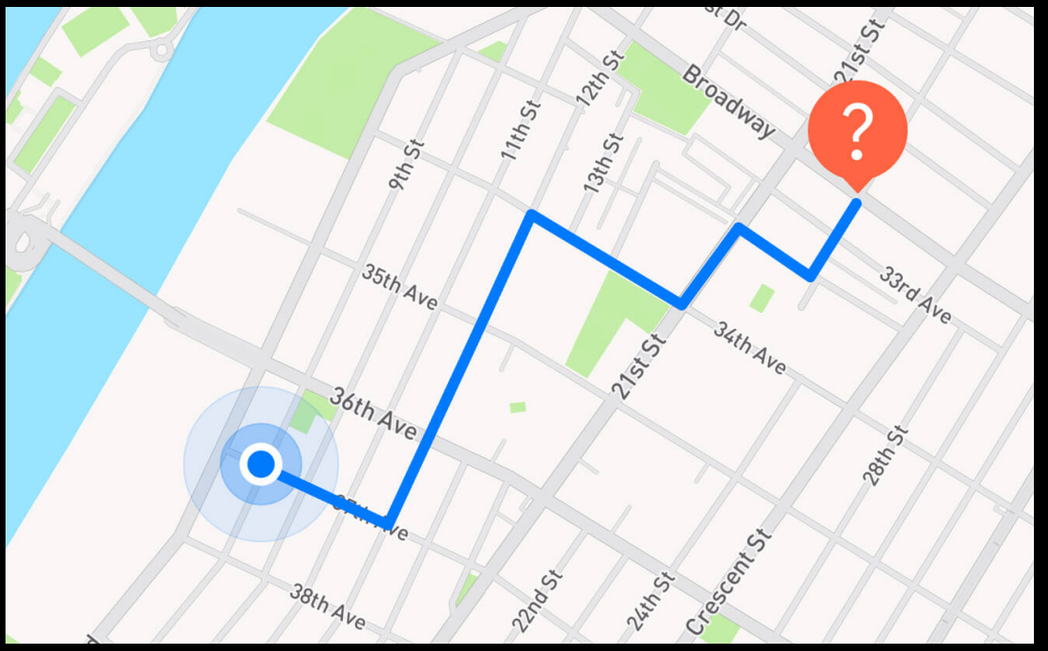
Today

- Navigation
- Breadth First Search
- Dijkstra's Algorithm
- Greedy Best First Search
- A*
- Navmeshes

Image credit: Midjourney

navigation

Navigation



- The process of finding a path through the world
  - Often, but not always, about geography

Control

- Often, Navigation is combined with Control
  - Especially in games, where Control means AI
    - Pay attention to "State Machine"
  - Our textbook does this

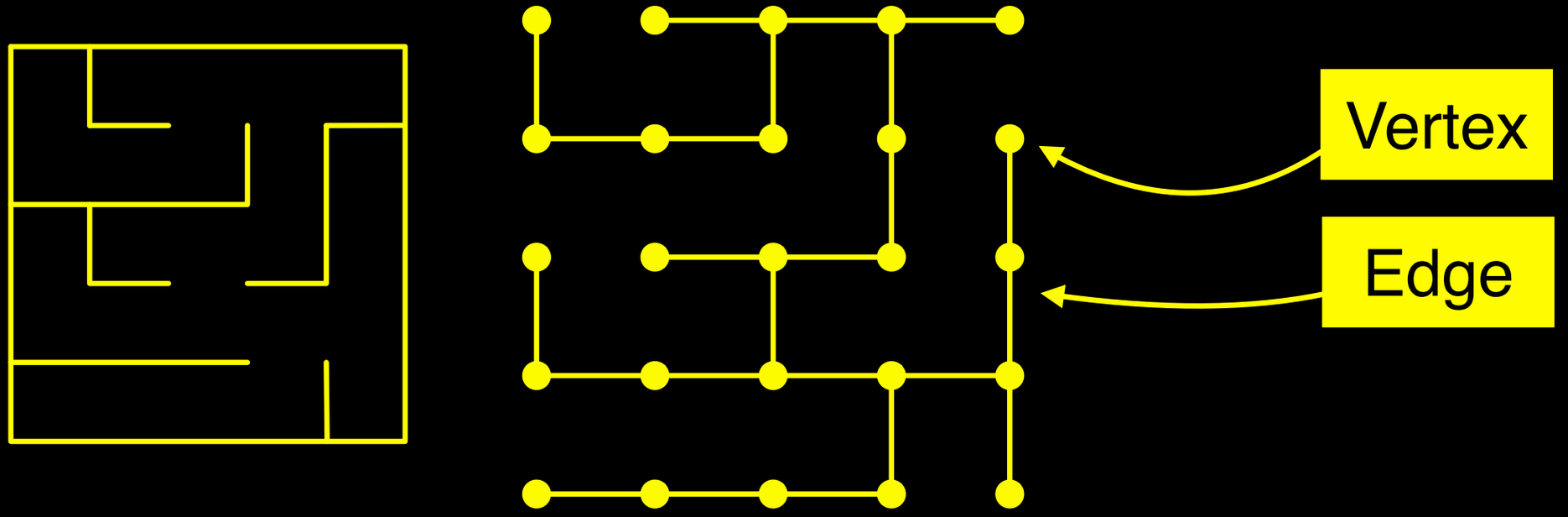
The World

- Consists of locations / ways to get between them
- Mazes → Cells
- Lots of games → Tiles
- In general, a **Graph**

Graph = Set(V) + Set(E)

- **Vertices**: Locations in the world
  - Points, nodes, rooms, cells, towns, planets
  - Annotate Vertex with info about the place
- **Edges**: Connect two Vertices, represent a way to get from one to the other
  - Hallways, tunnels, roads, hyperspace links
  - Annotate Edge with info about the connection (like travel costs)

Mazes are Graphs



- Each cell in a maze ➙ vertex in a graph
- Linked cells ➙ edge in graph

General Graphs

- We will show lots of tiled worlds, mazes today
- But, the algorithms work on any graph

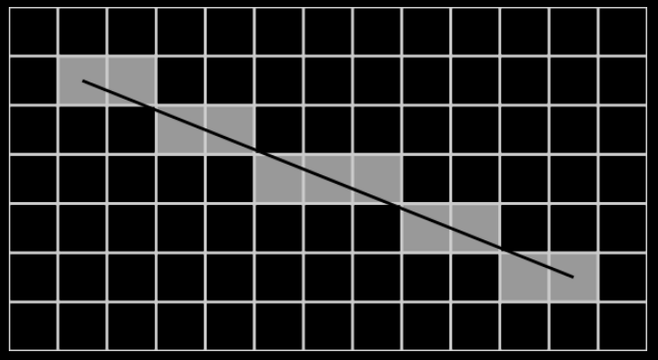
Example: Renowned Explorers

- Uses irregular convex polygons

Aside: Game Sizes

- How big should each vertex be?
  - i.e. how much real space should each vertex represent?
- Very interesting blog post about sizes
  - http://www.zarkonnen.com/grids_in_games

Aside: Line Drawing



- **Bresenham's Algorithm**: Classic (1962), low CPU powered (i.e. fast)
- **Linear interpolation** (lerp): Easier to understand
- Orthogonal steps: Modification to either of above such that no diagonal steps are taken
  - No "light leak" through a wall

Back to Navigation

- Navigation: Finding a path from one vertex (V_{start}) to another (V_{end})
- Path: Set of edges (E_0, E_1, ..., E_n)
  - Connect V_{start} to V_1 to V_2 ... to V_{end}
- Generally so Sum(E_0.cost, E_1.cost, ... E_n.cost) is minimized

Classic Graph Search

- One source, one destination
  - Example: How does my AI monster move towards the player
  - **Greedy Best First Search**
  - **A***

- One source, all destinations (or all source, one destination)
  - Ex: Is it possible to move all my monsters towards the player
  - Ex: Tower defense game, "Flow field"
  - Ex: In which tiles can I place treasure, such that players coming from the entrance can get it
  - **Breadth First Search**: unweighted edges
  - **Dijkstra's Algorithm**: weighted edges, cost > 0
  - Bellman-Ford: weighted edges, cost can be negative

- All sources, all destinations
  - Floyd-Warshall Algorithm
  - Johnson's Algorithm
    - A mixture of Dijkstra's and Bellman-Ford
- Note: All these algorithms (from the past three slides) all work on any general graph
  - Simplification: Tiled maps ➙ grid graphs, constant weights, like mazes

breadth first search

Breadth First Search

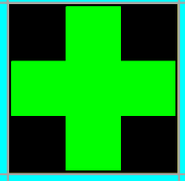
- We've seen this before
  - Flood fill using a frontier (FIFO queue)
- Continue until frontier is empty
  - Pick the first location from the frontier
  - Mark it as "visited"
  - Add unvisited neighbors to the frontier

```
frontier = [ start ]
visited = {}
visited[start] = True

while not frontier.empty():
    current = frontier.pop(0)
    for neighbor in current.neighbors():
        if not visited[neighbor]:
            frontier.append(neighbor)
            visited[neighbor] = True
```

FIFO. Take from front

FIFO. Put on back

Some Pictures

F	56	51	45	39	33	27	20	12	21
57		46	40	34	28	22	13	5	14
53	47	41	35	29	23	15	6	1	7
48	42	36	30	24	16	8	2		4
54	49	43	37	31	25	17	9	3	10
58	55	50	44	38	32	26	18	11	19

Aftermath

- The point of BFS is to discover if you can reach some destination cell
- Test is easy -- just look it up in the visited dictionary

```
def can_reach(some_cell):
    return some_cell in visited
```

pathfinding with BFS

Pathfinding

- BFS gives a True/False result about whether a destination cell is reachable from the starting cell
- To get the path, need to save a bit more data
  - Usually, we want the shortest path

Modified Algorithm

- For every location we visit, keep track of which neighbor we came from
- I'll change the name of visited to show this

```
frontier = [ start ]
came_from = {}
came_from[start] = None

while not frontier.empty():
    current = frontier.pop(0)
    for neighbor in current.neighbors():
        if neighbor not in came_from:
            frontier.append(neighbor)
            came_from[neighbor] = current
```

Previously, `visited`

Same stuff

Same stuff

Same stuff

This line is new

Pathfinding Results

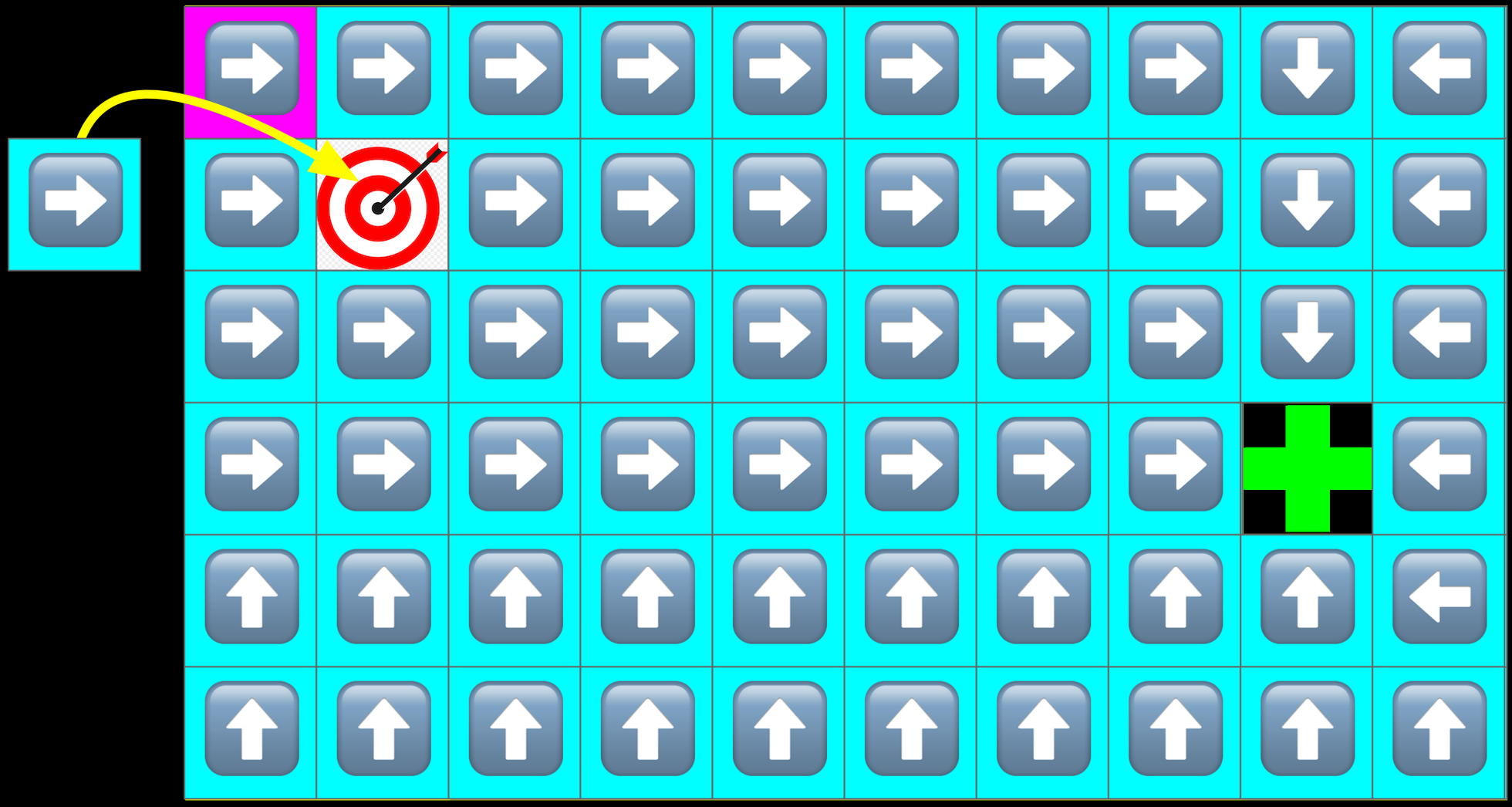
- Notice that the result is not a path!
  - More like breadcrumbs left on a trail by Hansel and Gretel
- To use the breadcrumbs, follow the arrows backwards from the target

```
def get_path(start, destination, came_from):
    current = destination
    path = []

    while current != start:
        path.insert(0, current)
        current = came_from[current]

    path.insert(0, start) #optional
    return path
```

Put the vertex in the front of the list

With this, the list includes start and destination

Pathfinding Pictures



A Few Notes

- Technicality: A path is a sequence of edges in a graph
  - Our path is a sequence of cells (vertices)
  - Equally usable
- The `came_from` dictionary can be used to find a path from ANY cell, not just the target cell
- This is a **flow field**. Very useful for many games, especially tower defense games

early exit

Early Exit

- As shown, algorithm only completes when the entire graph has been explored
- Algorithm can easily be modified to quit when the target is reached

F	56	51	45	39	33	27	20	12	21
57		46	40	34	28	22	13	5	14
53	47	41	35	29	23	15	6	1	7
48	42	36	30	24	16	8	2		4
54	49	43	37	31	25	17	9	3	10
58	55	50	44	38	32	26	18	11	19

		F	45	39	33	27	20	12	21
		46	40	34	28	22	13	5	14
	F	41	35	29	23	15	6	1	7
F	42	36	30	24	16	8	2		4
	F	43	37	31	25	17	9	3	10
		F	44	38	32	26	18	11	19

```
target = some_cell
found_target = False
frontier = [ start ]
came_from = {}
came_from[start] = None

while not frontier.empty():
    current = frontier.pop(0)
    if current == target:
        found_target = True
        break
    for neighbor in current.neighbors():
        if neighbor not in came_from:
            frontier.append(neighbor)
            came_from[neighbor] = current
```

This lets you know, after the end of this code, if the while loop stopped because the target was found or not

obstacles

Obstacles

- Typically, you won't be navigating across big caves with nothing in the way
- Mazes had lots of obstacles, walls between unlinked cells
- Navigation algorithms can easily work with obstacles
  - Else, they wouldn't be very useful

Representation

- Goal: Ensure obstacles never get put into the frontier
- Pure graph: obstacles are represented by a lack of edge between two vertices
  - No edge ➙ can't get there from here
- Tiles: Don't list obstacles as neighbors
  - Or, have a list of obstacle cells

Pictures

F	47	44		30	26	21	15	22	27
48		41		28	23	16	10		
46	42	38		24	17	11	5	1	6
43	39	35		18	12	7	2		4
40	36	33			19	13		3	8
37	34	32	31	29	25	20		9	14

```
obstacles = [ ] #list of cells that are walls/stuff
frontier = [ start ]
came_from = {}
came_from[start] = None

while not frontier.empty():
    current = frontier.pop(0)
    for neighbor in current.neighbors():
        if neighbor not in came_from and
           neighbor not in obstacles:
            frontier.append(neighbor)
            came_from[neighbor] = current
```

Very easy extension to our BFS code

Check for obstacle first

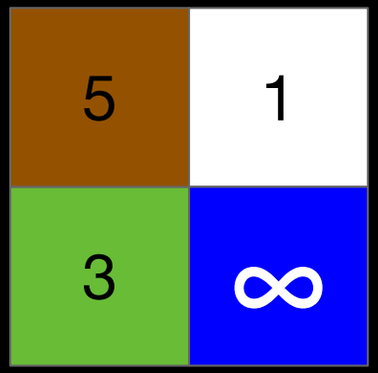
Dijkstra's

Dijkstra's Algorithm

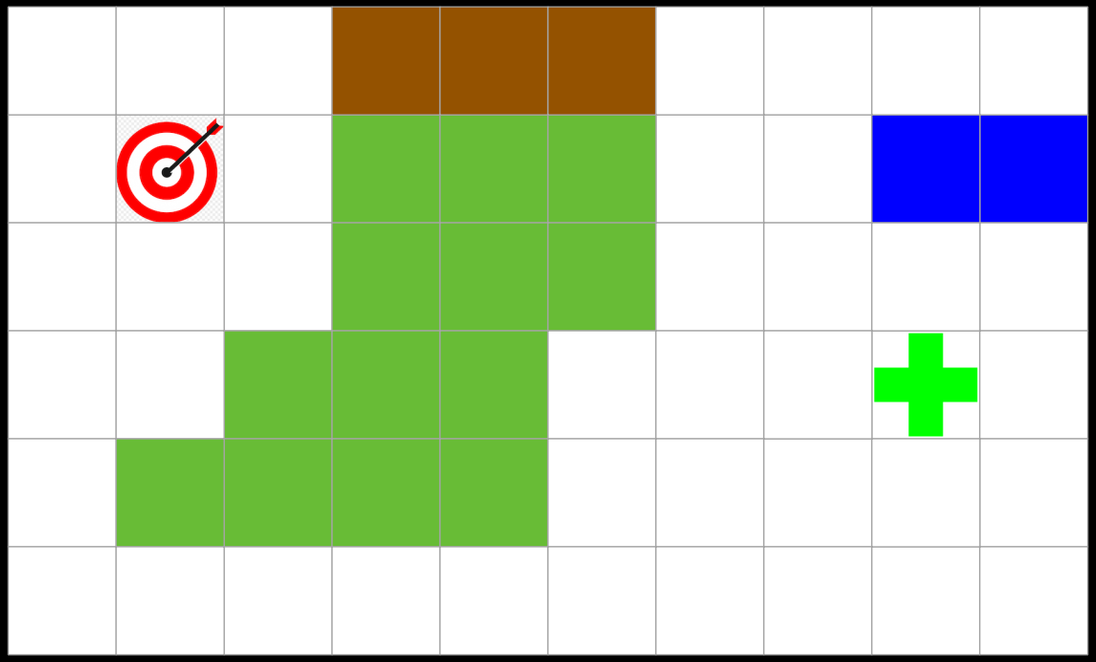
- Calculates the minimum cost path from one vertex to all other vertices
- Implies that edges have costs
  - Terrain costs
  - Does it cost more to go through a door? Or to cross a river?

- Dijkstra's is a "single source, all destinations" algorithm
- The idea:
  - Mark each vertex with the lowest cost to get there
  - Put neighbors into the frontier with additional cost of the edge
  - There may be multiple paths to a particular vertex (which means it will be in the frontier multiple times)
  - Take out the lowest-cost vertex from the vertex
    - This means there can never be a lower cost path discovered to that vertex in the future

Pictures

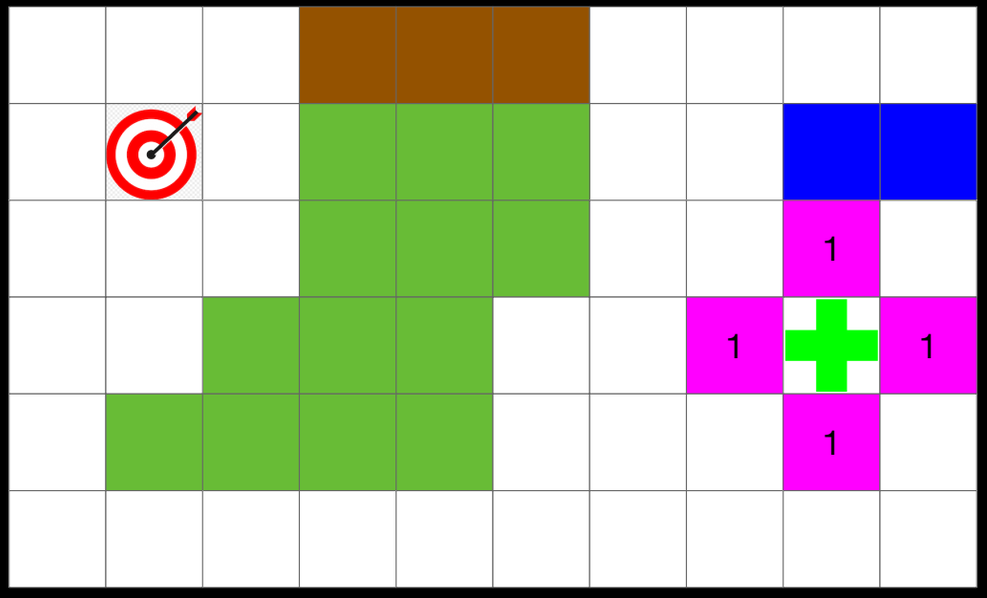


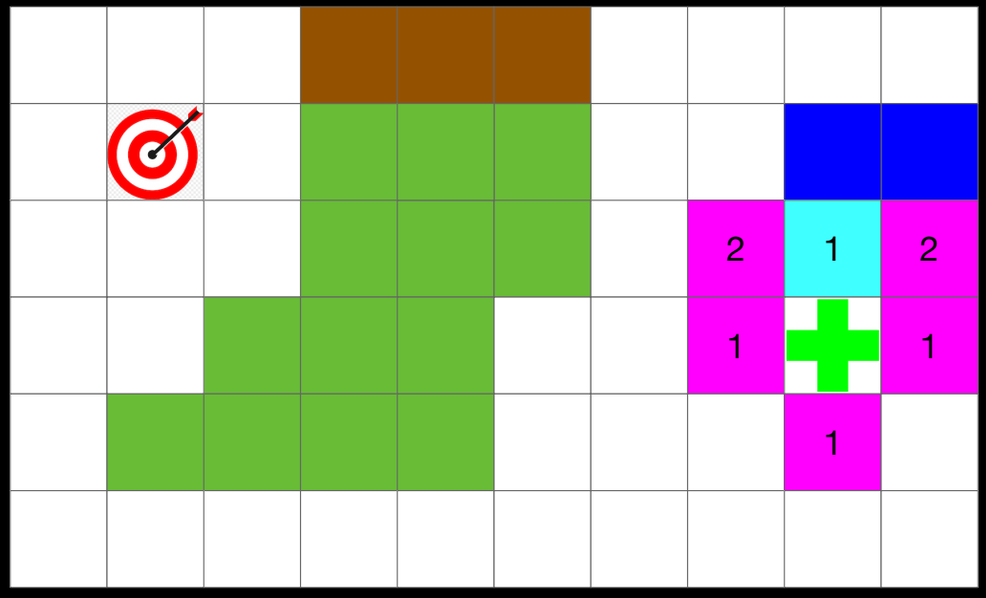
- Let's imagine that our world has less obstacles, but also some forests (movement cost of 3)
  - And some mountains (movement cost of 5)



More Pictures

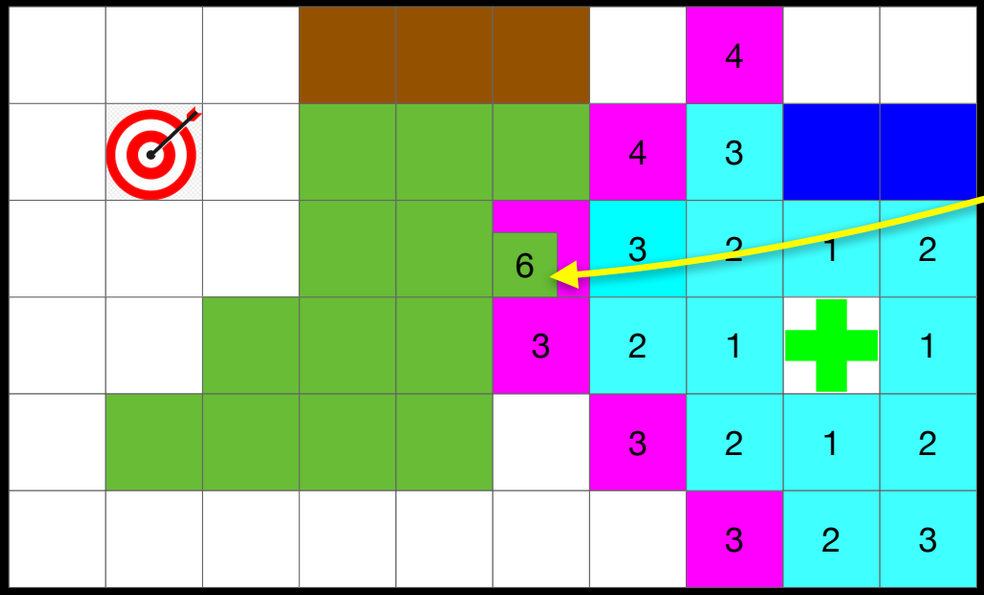
- In the beginning, it looks exactly like we've seen before
- But! We are adding one because that is the terrain cost, not because we always add one





Even More Pictures

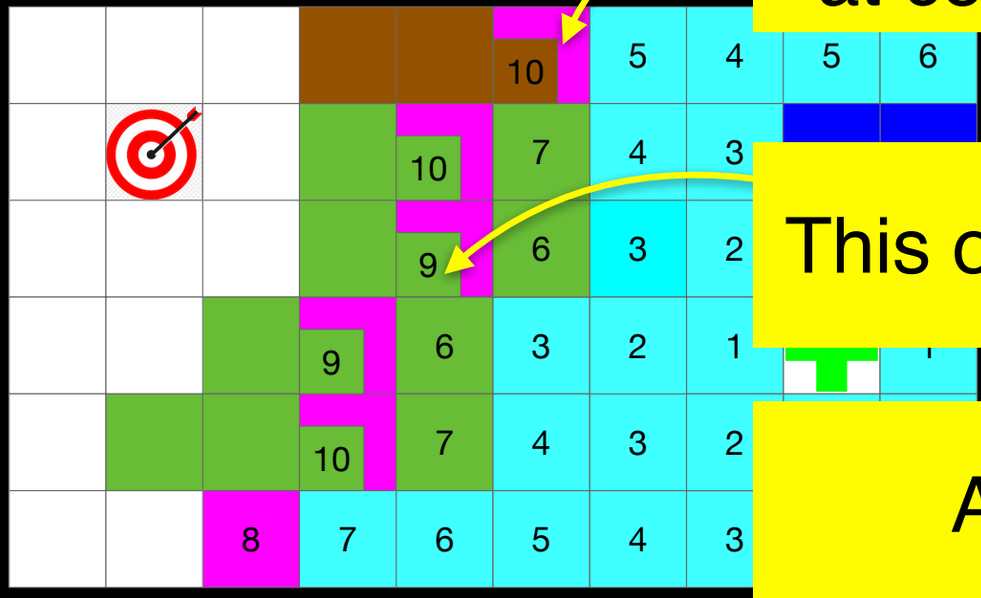
- When the frontier extends to the forest, we add three to the current vertex cost, not one



Notice! Cost of 6, not 4

… Pictures

- Some locations get put into the frontier multiple times, as multiple paths to that location are discovered



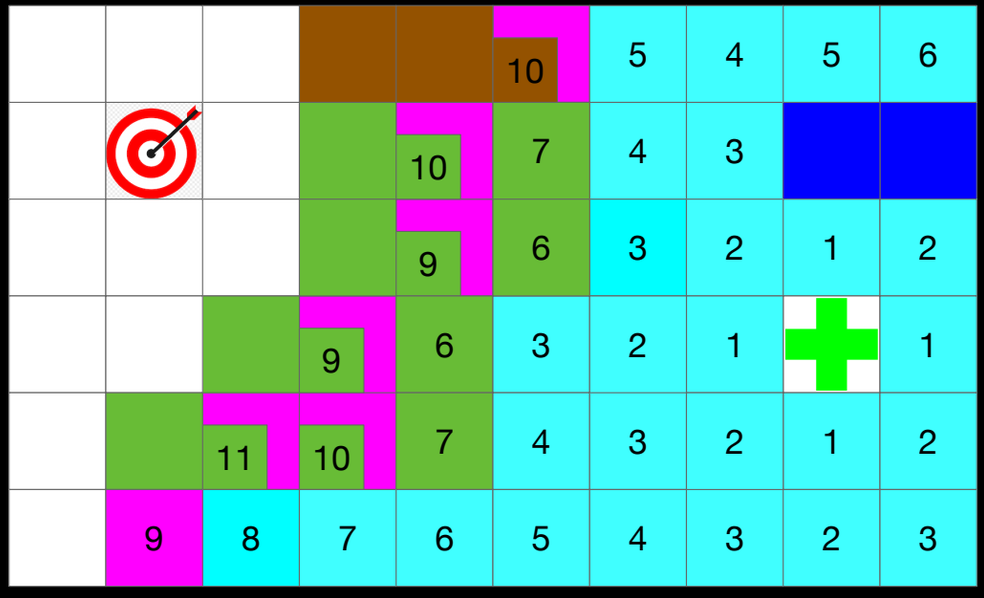
This cell is in frontier at cost of 10 and 12

This one in at 9 and 9

Any others?

Last Pictures

- The path around the forest is pretty close to the cost of the path through the forest
- Here's the final situation -- least cost is 15



Implementation

- The algorithm relies on easily finding the lowest cost of all the cells in the frontier
- Efficiently: use a **Priority Queue** (also called a min-heap)
- In Python, the `heapq` module is part of the standard library
  - Uses a list, but managed as a priority queue

```
import heapq
frontier = [ ]
heapq.heappush(frontier, (0, start, None))
   #tuple of distance, cell, came from cell
came_from = {start : None} #Completed knowledge

while not frontier.empty():
    cost, current, from = heapq.heappop(frontier)
    if current in came_from:
        continue
    came_from[current] = from

    #early exit code could be here

    for neighbor in current.neighbors():
        if neighbor not in came_from:
            new_cost = cost + neighbor.cost
            heapq.heappush(frontier,
                           (new_cost, neighbor, current))
```

Tuples kept in sorted order, so distance must be first

Multiple records are likely, so ignore longer routes

greedy best first search

Efficiency!

- BFS and Dijkstra's spread out to search in all directions
  - If there is only a single goal, this is inefficient
  - Just head towards the goal
- But, if we knew which step was the best way to the goal, we'd already know the entire path

Heuristic

- Can we use a heuristic function to guide our search?
- Heuristic is a "rule of thumb" or a loosely defined guideline ➙ imperfect

```
def heuristic(ax, ay, bx, by):
    # Manhattan distance on a square grid
    return abs(ax - bx) + abs(ay - by)
```

- Manhattan distance make sense for a grid of squares ➙ other situations, it might not

Heuristic Search?

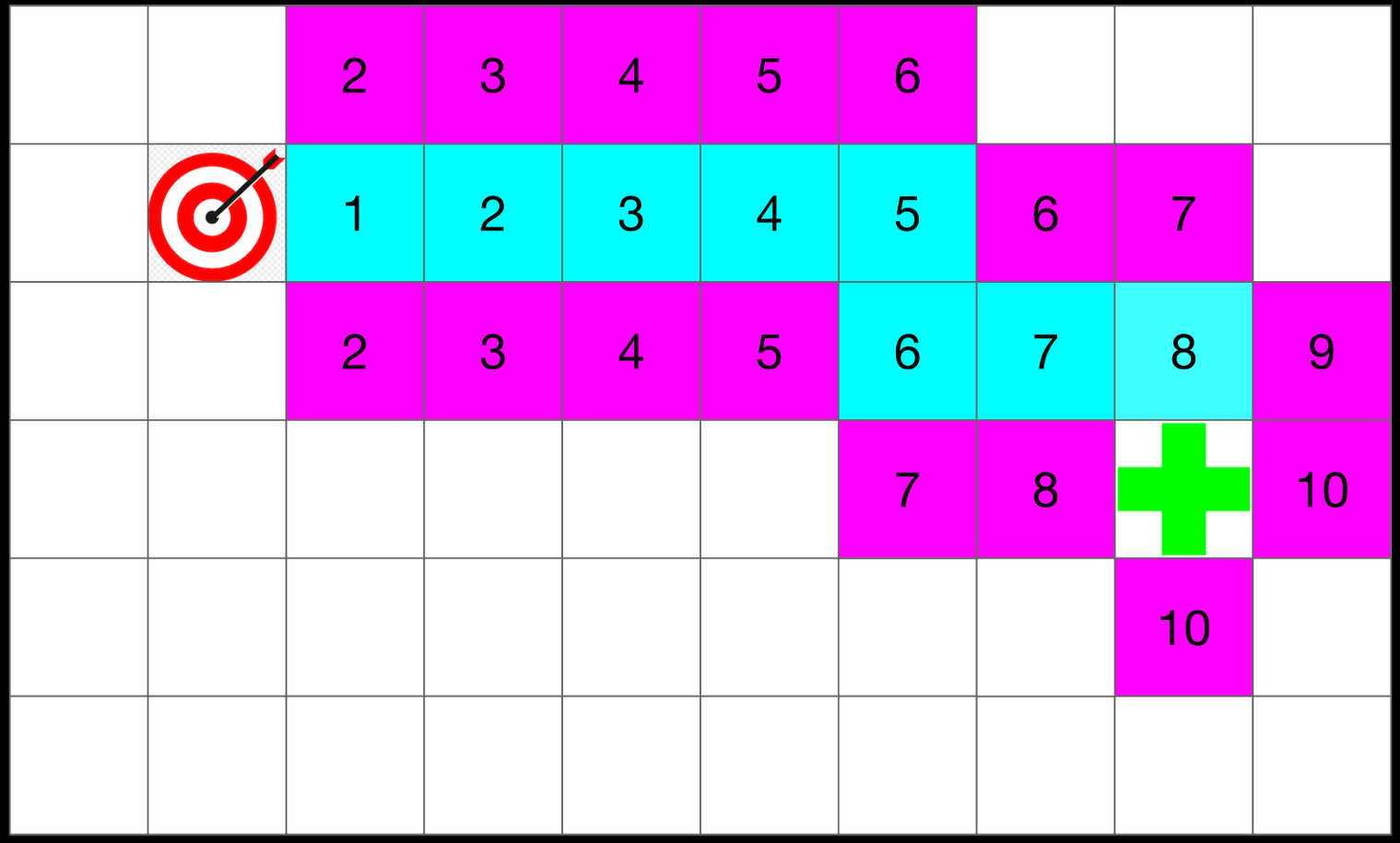
- To use a heuristic is fairly simple
- Keep the frontier / loop structure
- Each time around the loop, take the cell from the frontier with the lowest heuristic distance
  - i.e. the one we think is closest to the goal

```
import heapq
frontier = [ ] #tuple of h_dist, cell, came_from
heapq.heappush(frontier, (0, start, None))
came_from = {}

while not frontier.empty():
    _, current, from = heapq.heappop(frontier)
    if current in came_from:
        continue
    came_from[current] = from

    #early exit here

    for neighbor in current.neighbors():
        if neighbor not in came_from:
            distance = heuristic(goal.x, goal.y,
                                 neighbor.x, neighbor.y)
            heapq.heappush(frontier,
                           (distance, neighbor, current))
```

Using the heuristic value just for sorting order

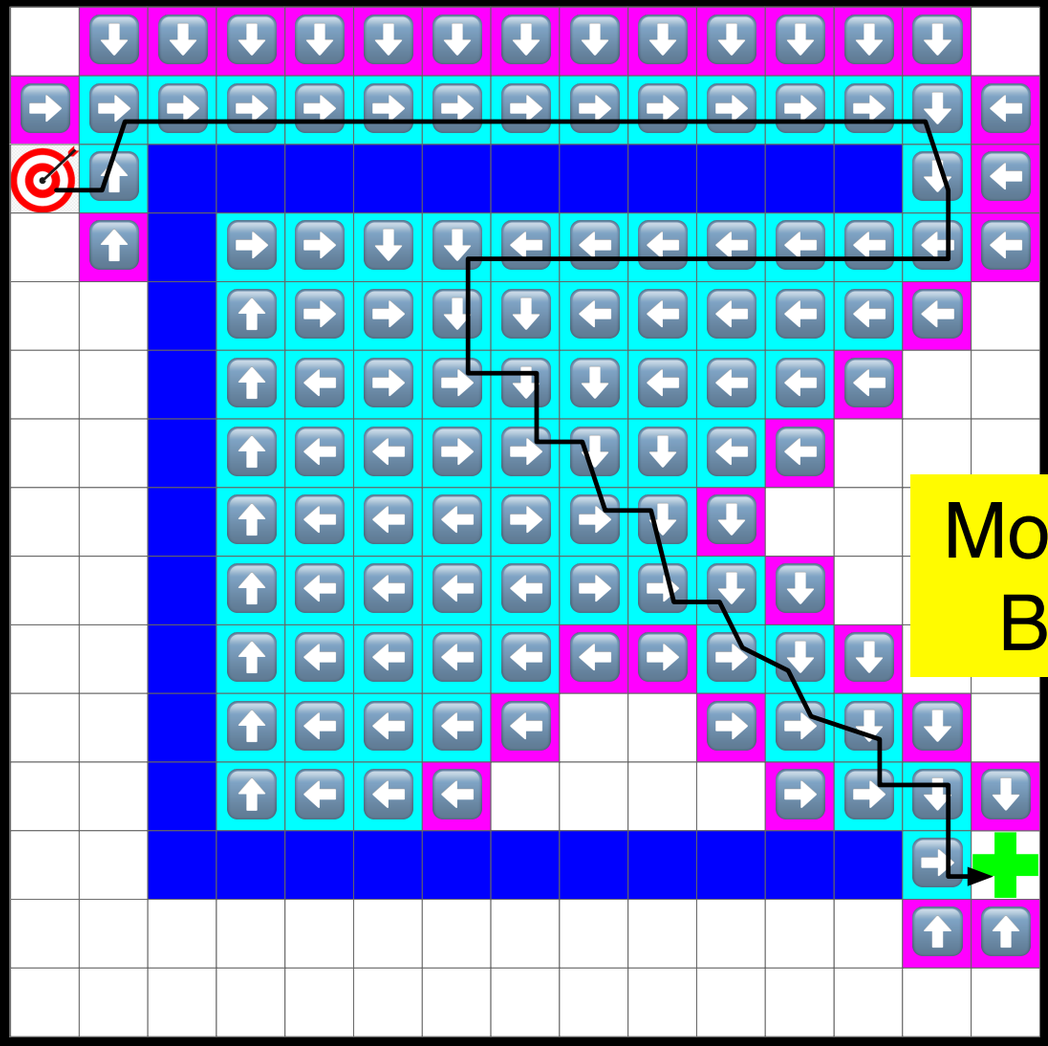
Pictures

- Compare to BFS, which took 52 steps to find the target



Sounds Fantastic!

Yeah, but ...



More efficient than BFS / Dijkstra's

Resulting path is worse than BFS / Dijkstra's

A*

So Far

- Dijkstra's Algorithm
  - Resulting path is perfect
  - Inefficient -- searches everywhere
- Greedy Best First Search
  - Efficient
  - Resulting path is not good

Would a combination be better?

Tradeoffs?

A* Algorithm

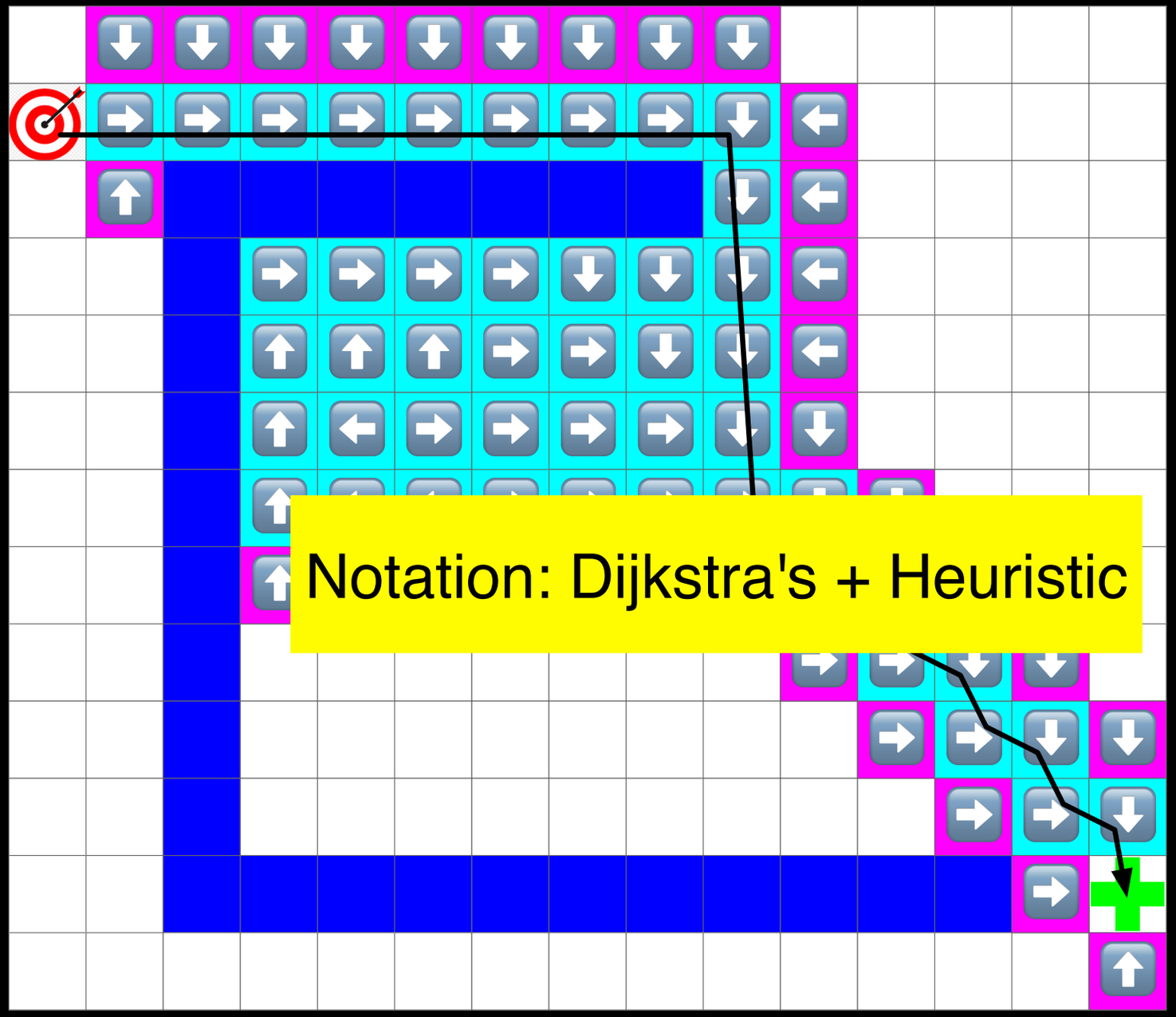
- A graph search algorithm
  - Developed at SRI in 1962
  - Developed for robotics
- A best-of-both worlds algorithm, combining Dijkstra's and GBFS

```
import heapq
frontier = [ ] #tuple of priority, cell, Dijkstra cost, came_from
heapq.heappush(frontier, (0, start, 0, None))
came_from = {}

while not frontier.empty():
    priority, current, d_cost, from = heapq.heappop(frontier)
    if current in came_from:
        continue
    came_from[current] = from

    #early exit here

    for neighbor in current.neighbors():
       if neighbor not in came_from:
            cost = d_cost + neighbor.cost
            priority = cost + heuristic(goal.x, goal.y,
                                neighbor.x, neighbor.y)
            heapq.heappush(frontier,
                           (priority, neighbor, d_cost, current))
```

Some comments

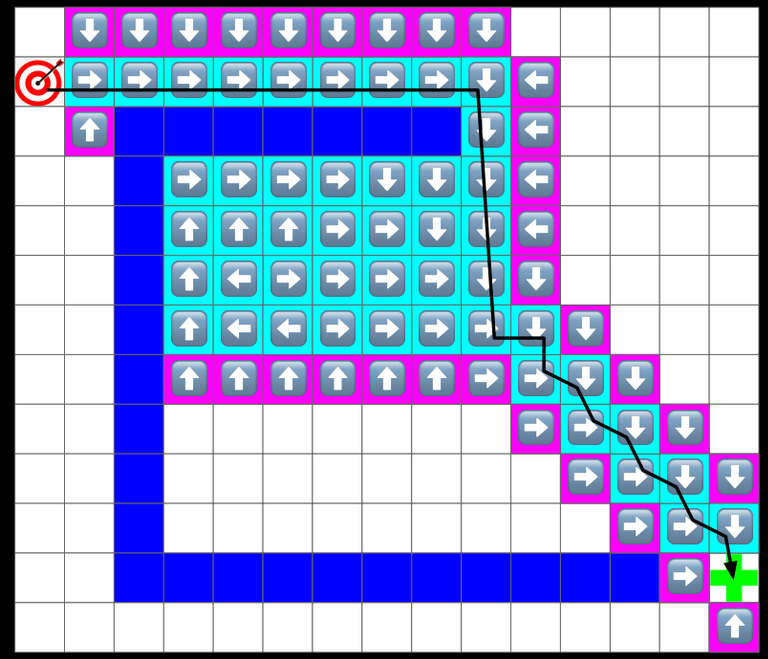
- If GBFS would have found the right path, A* will explore exactly the same area
- But, if GBFS → longer path, then A* is restrained by Dijkstra distance and keeps to the right path
- Dijkstra's algorithm is simply A* with a constant heuristic value
- A* uses the heuristic to re-order the search so it finds the goal sooner

Pictures



A* Results

- Search area doesn't look much different from GBFS, does it?
- But, resulting path is much better



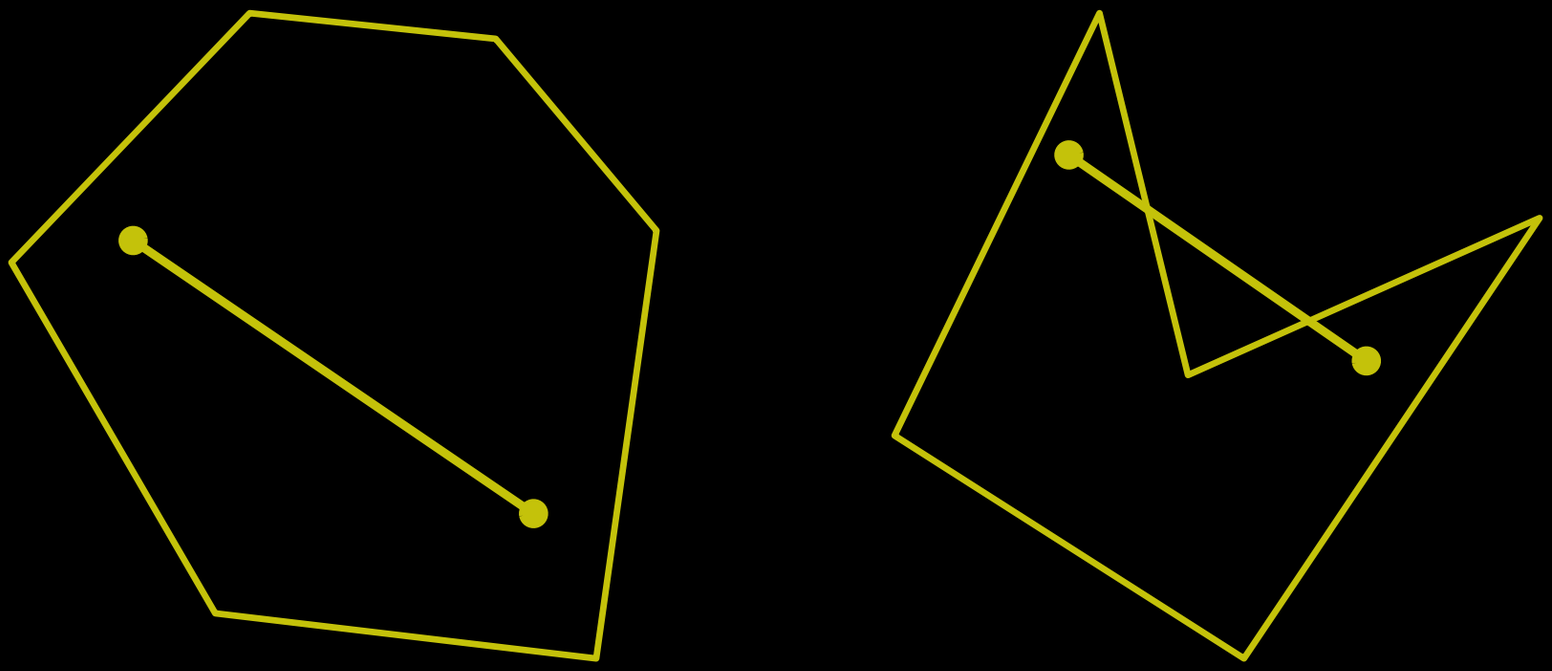
navigation meshes

A* at Scale

- As your space gets bigger and the obstacles more complex, A* (even though it is great) can take too long
  - Even worse, if you have lots of entities doing pathfinding
- Navigation meshes (navmesh) comes from robotics research, 1986

Convex Polygons

- A polygon where **any** line segment connecting two points on the interior or boundaries ...
- ... contains only points on the interior or boundaries



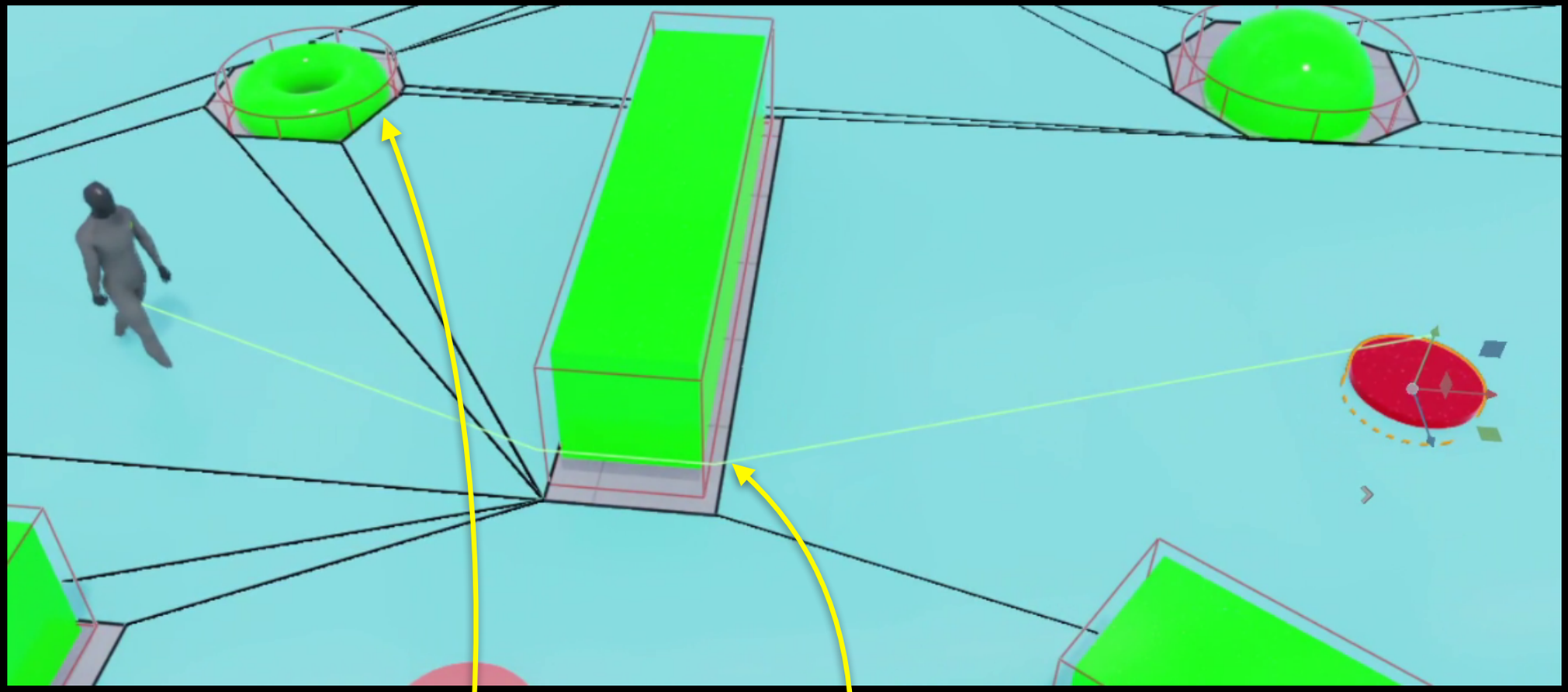
Properties

- Convex polygons have lots of interesting properties that make them useful
- Easy to navigate between any two points in the polygon
  - Just go in a straight line
  - Guarantees you will remain within the polygon

Navmesh

- A collection of convex polygons that define all freely-traversable areas in your game
  - i.e. every space where an agent (player, monster, etc) can legally walk
- All obstacles exist outside of the navmesh
- All polygons are connected by sides where traversal is allowed

Example



Notice that obstacles are not included -- they are just a hole in the mesh

Image credit: NVIDIA Omniverse Documentation

Navigation in the Mesh

1. If the target is in the same polygon as the agent
    - Just go in a straight line
2. If the target is not in the same mesh
    - The connection of polygons form a graph
    - Use A* to determine which polygons to travel through to get to the target's polygon
    - Then use rule #1

Added Benefits

- Each polygon must be 2-dimensional
- But, the collection of polygons doesn't have to be
  - Exploit to model platforms, mezzanines, stairs, different levels, etc

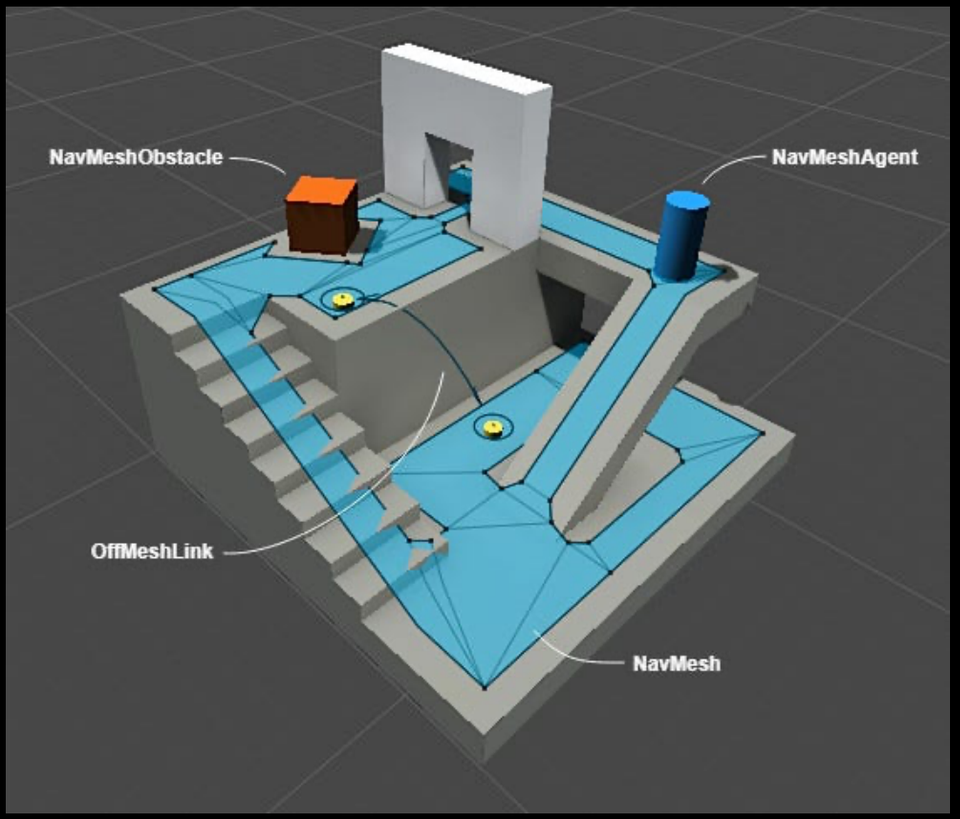
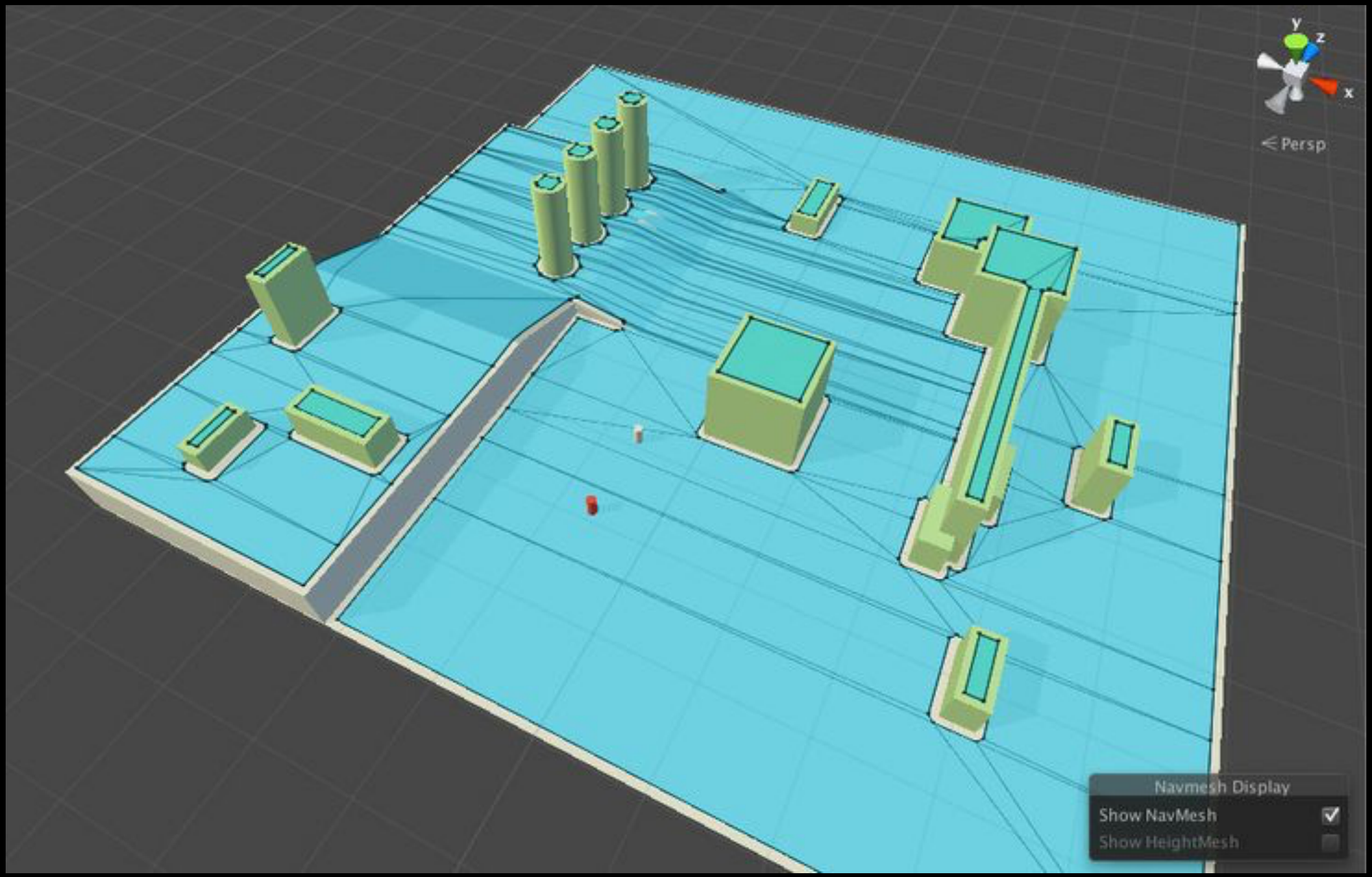


Image credit: Unity


y
z
x
Persp
Navmesh Display
Show NavMesh
Show HeightMesh

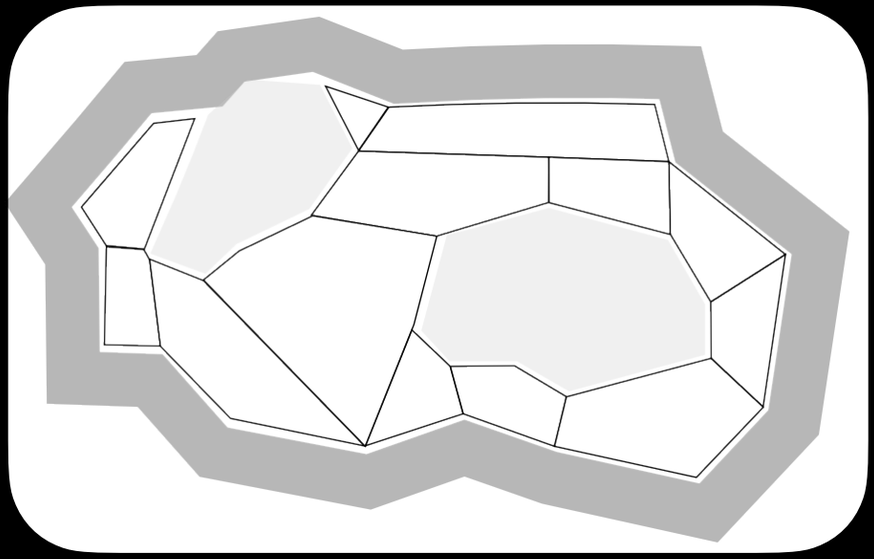
funnel algorithm

Funnel Algorithm

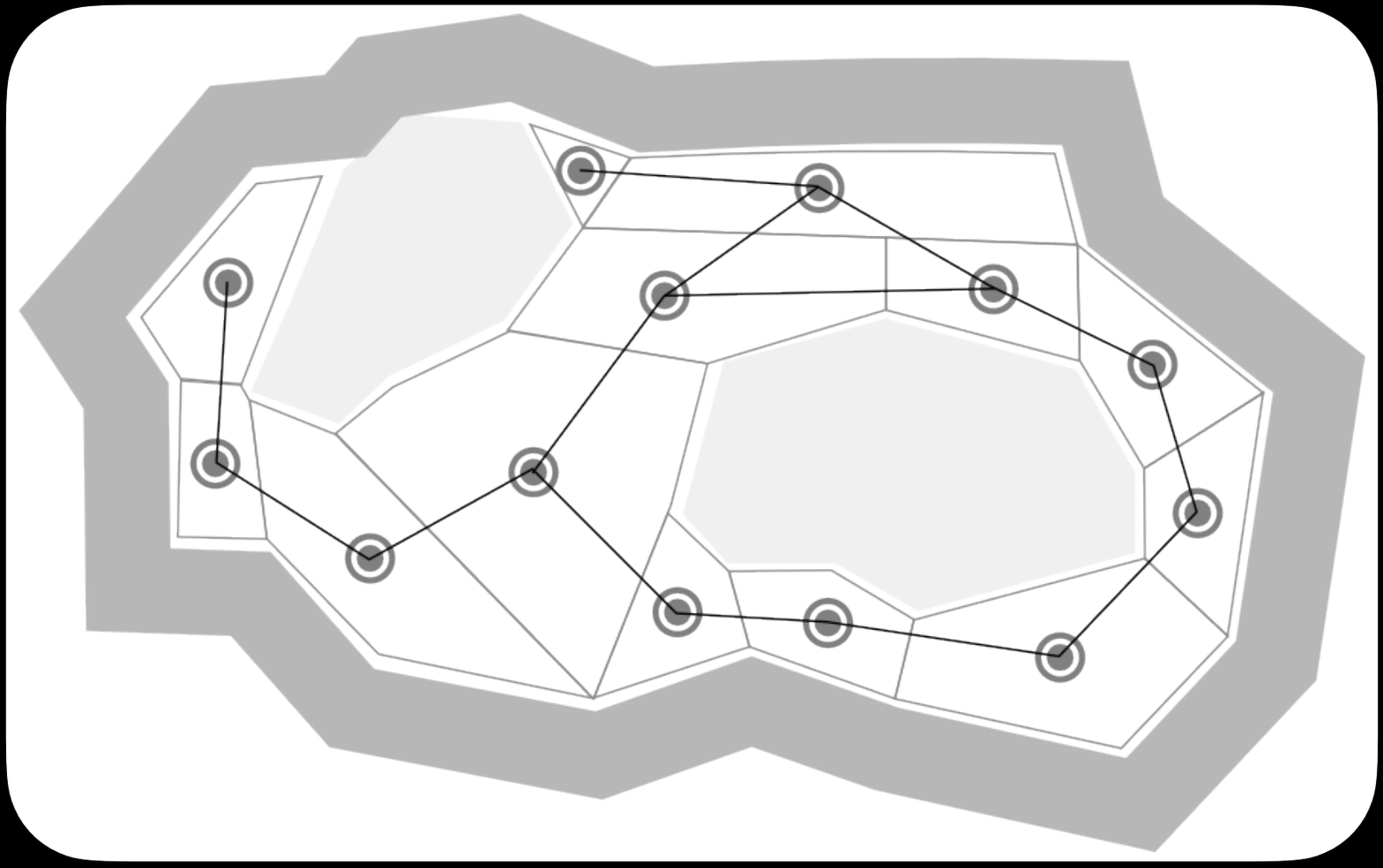
- The navmesh is an abstraction of the complete geography of your game
- Navigation calculations are simplified
- ... but, the resulting path is also simplified and may not be realistic
- Funnel algorithm: improving the resulting path

Funnel Algorithm in Pictures

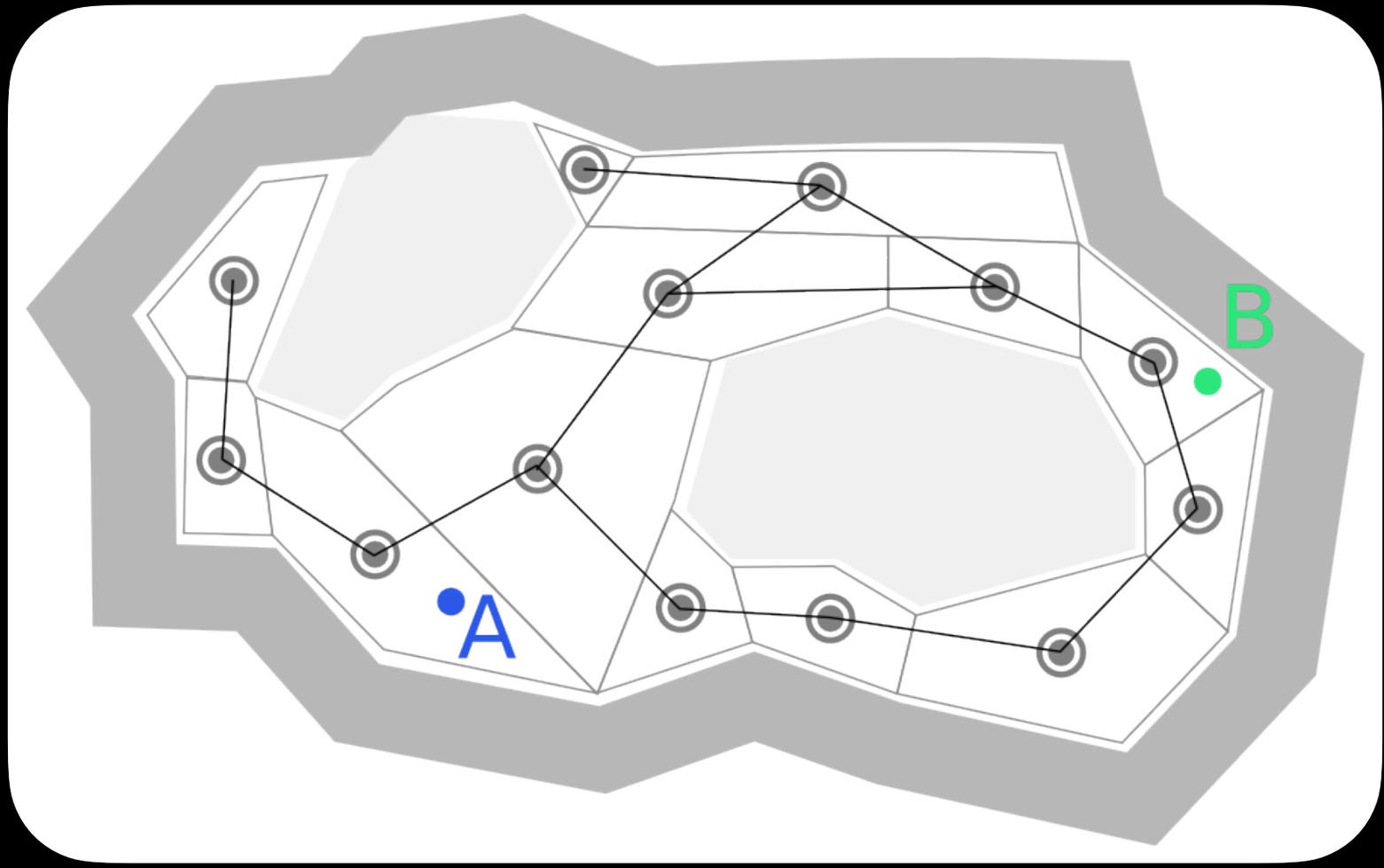
- I took this example from jceipek.com/Olin-Coding-Tutorials/pathing.html
- Imagine an island with two hills (prohibited movement) and the navigable land broken into 13 convex polygons



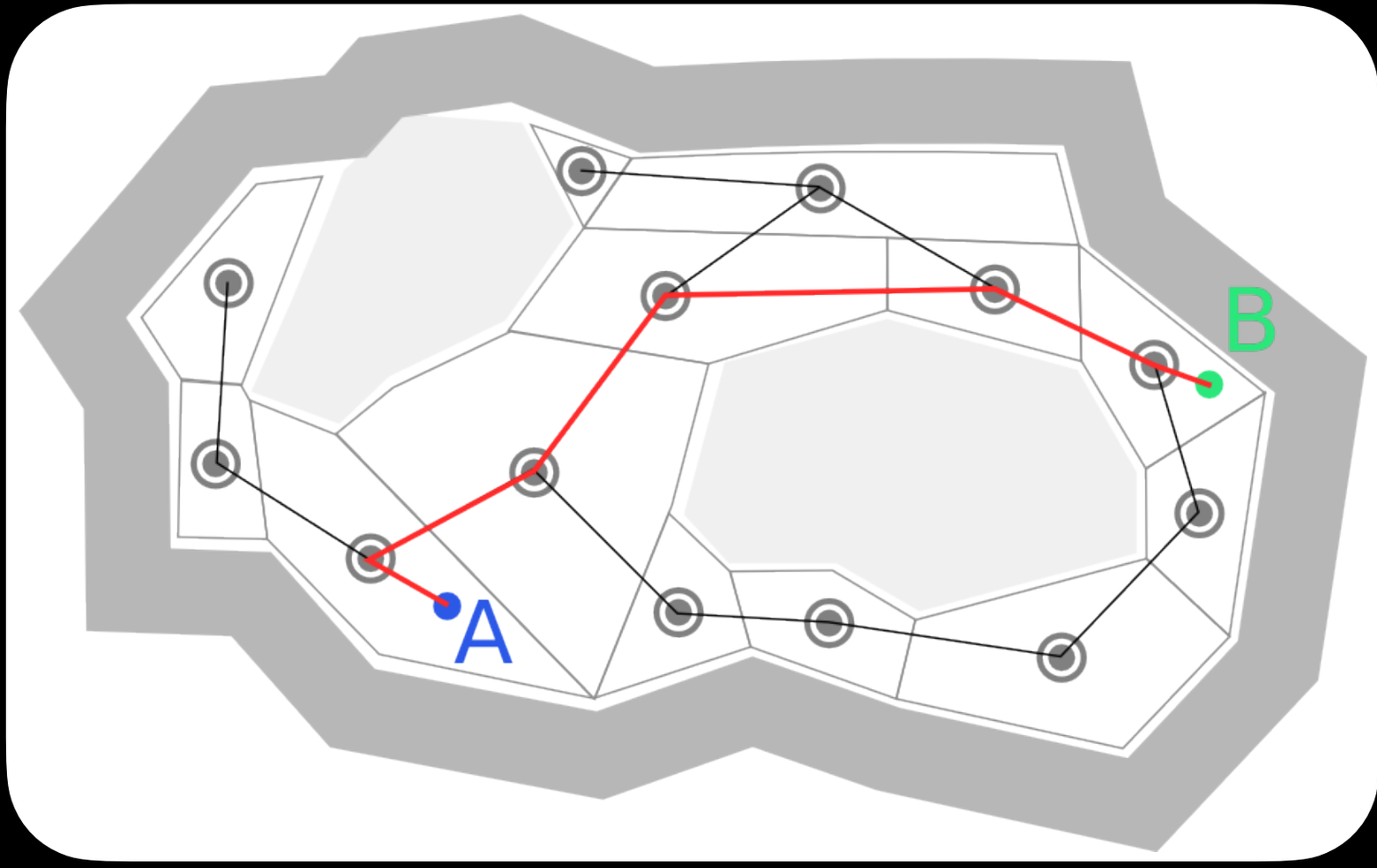
- The navmesh is a graph, so let's put a vertex in each polygon and draw edges between adjacent polygons



- Travel between A and B involves multiple convex polygons
  - Must use Rule #2 for navigation (not straight line)



- A* tells us to go along the path of these polygons
- But, it isn't smoothed



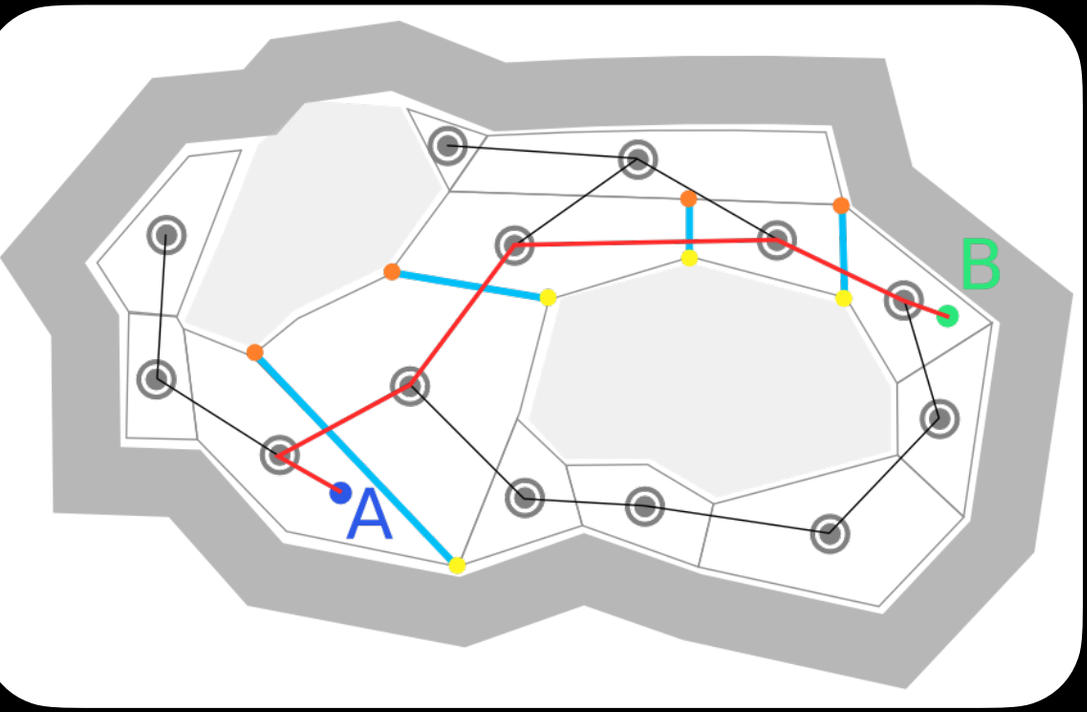
The Algorithm

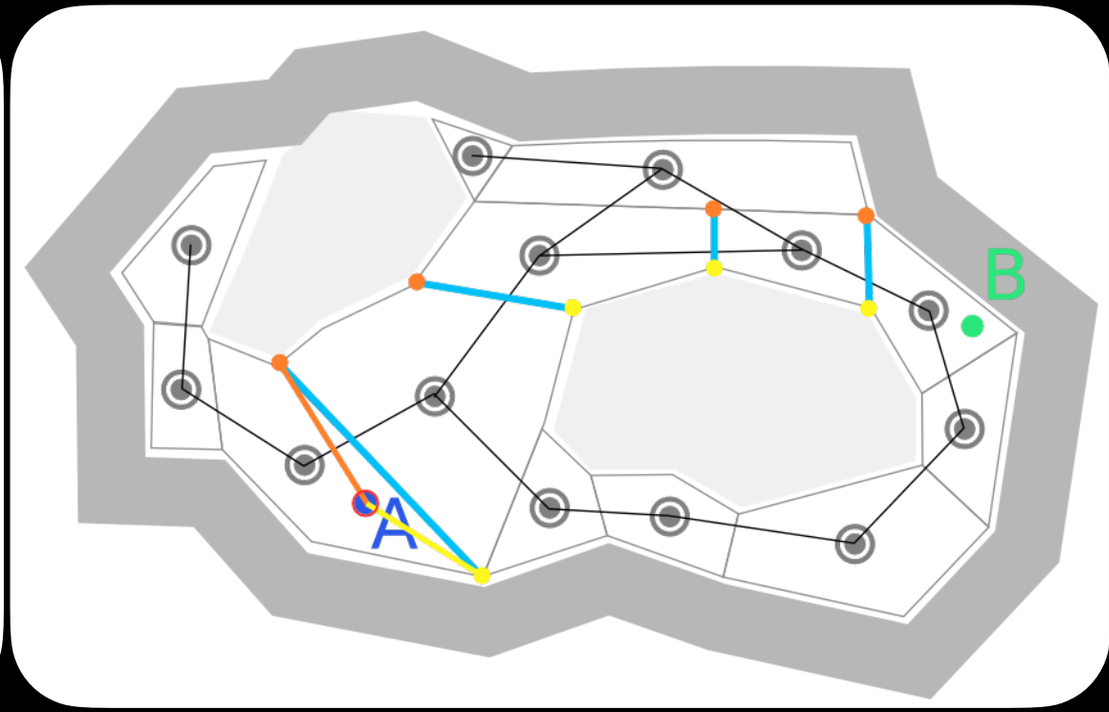
- Start at A (not the vertex in A's polygon)
- Consider the two edges of the common line segment between polygons (called a "portal")
  - Remember, A* told you which is the next polygon to move into
- Create a "funnel" with three points
  - The apex (point A), left portal end, right portal end

- Now, consider the second portal
- Move the left point of the funnel to the left edge of this second portal
  - If the update would widen the funnel, then ignore the update
  - Remember: you're still looking for a straight line to travel from the apex, so you can't widen or you might need to go through a corner
- Repeat with the right of the funnel

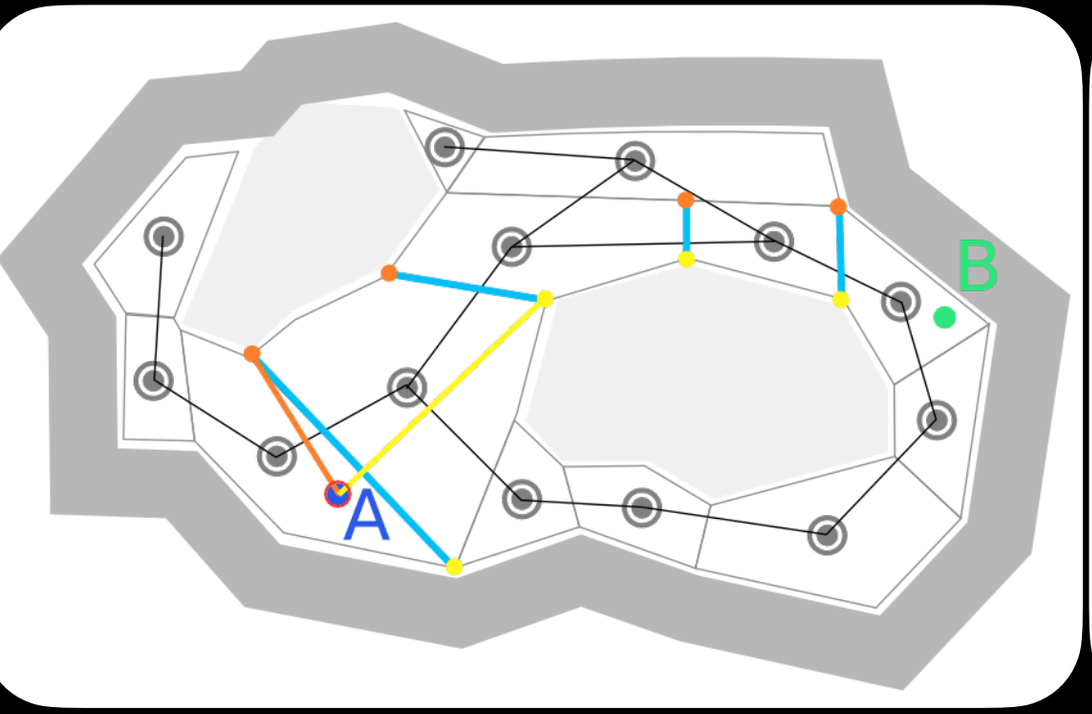
- If the update would crosses the sides of the funnel (left-over-right or right-over-left)
  - Move the apex to the last point on that side of the funnel and start over
  - Keep track of where the apex has been, as that will be your path

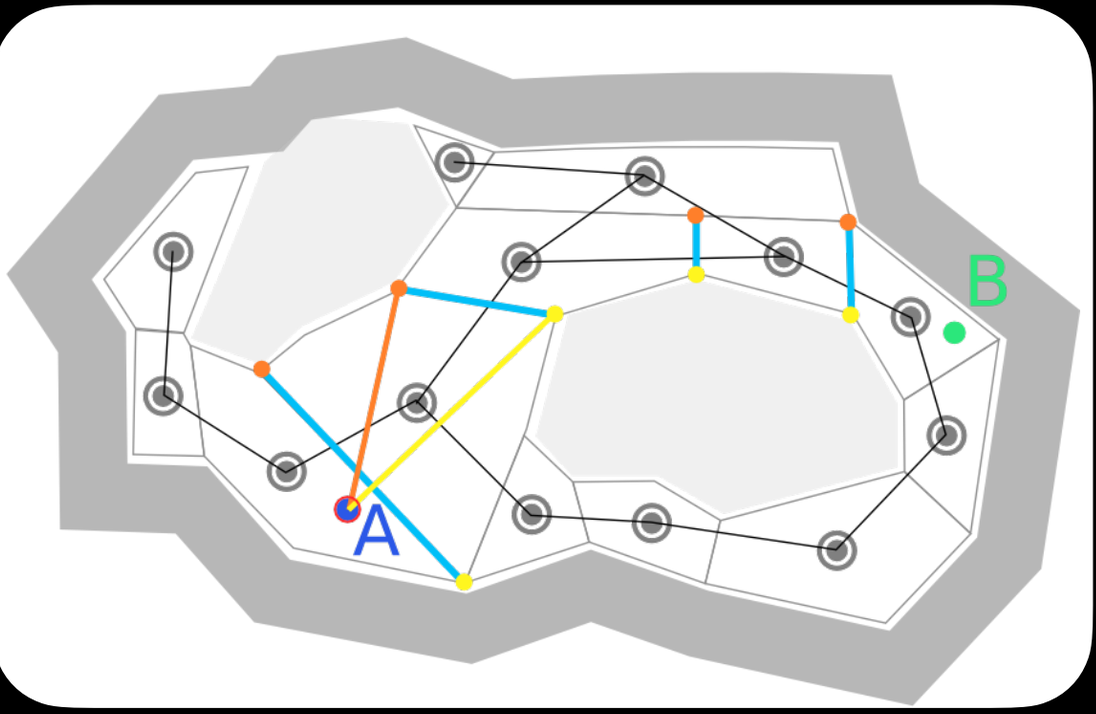
Pictures



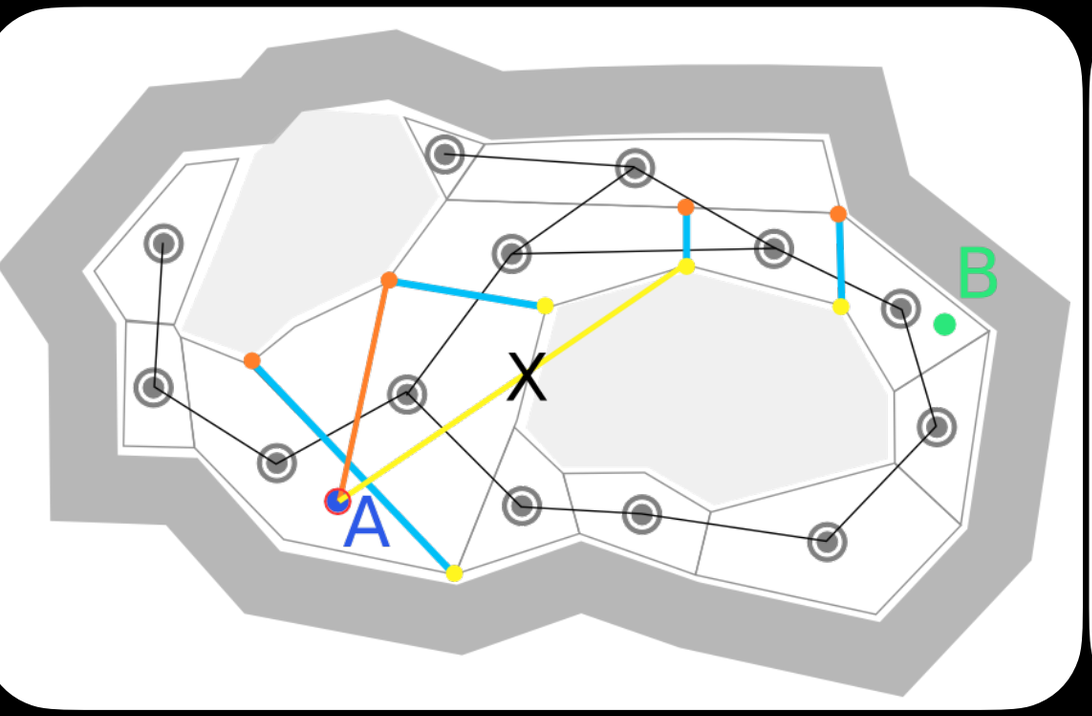


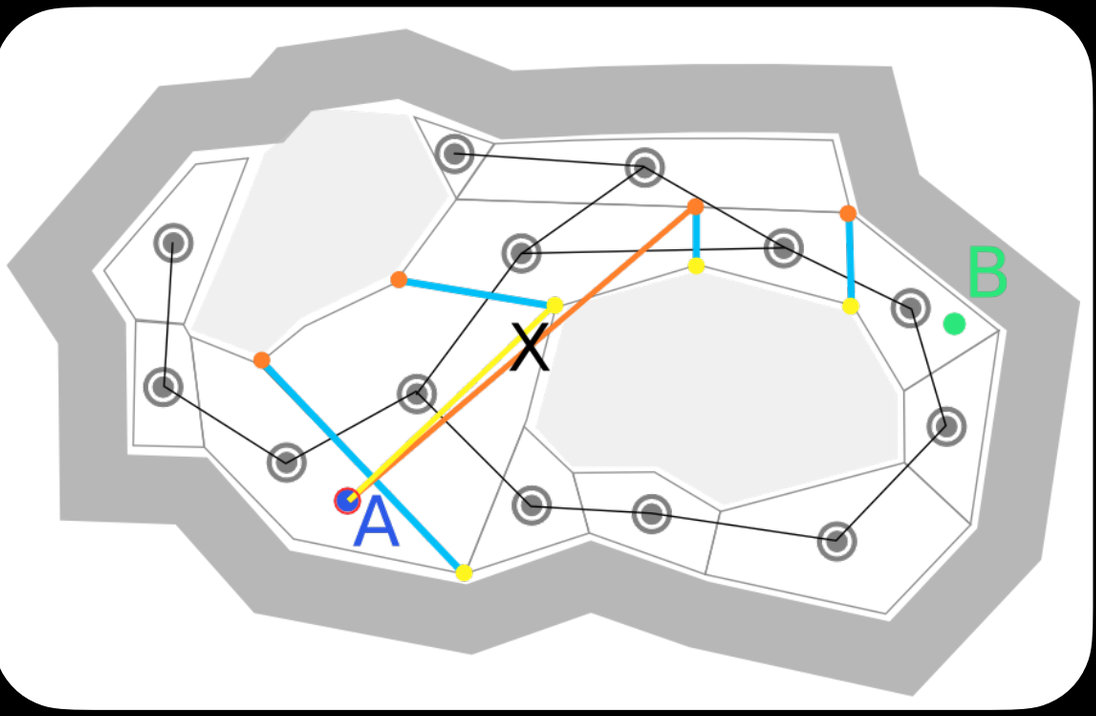
Pictures



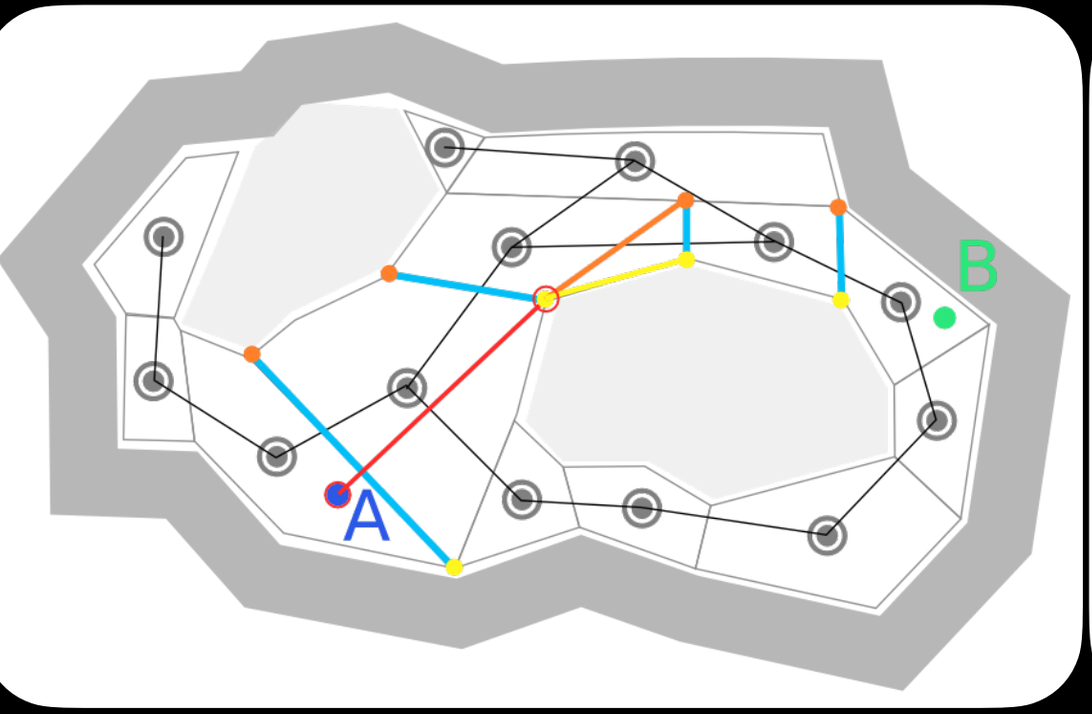


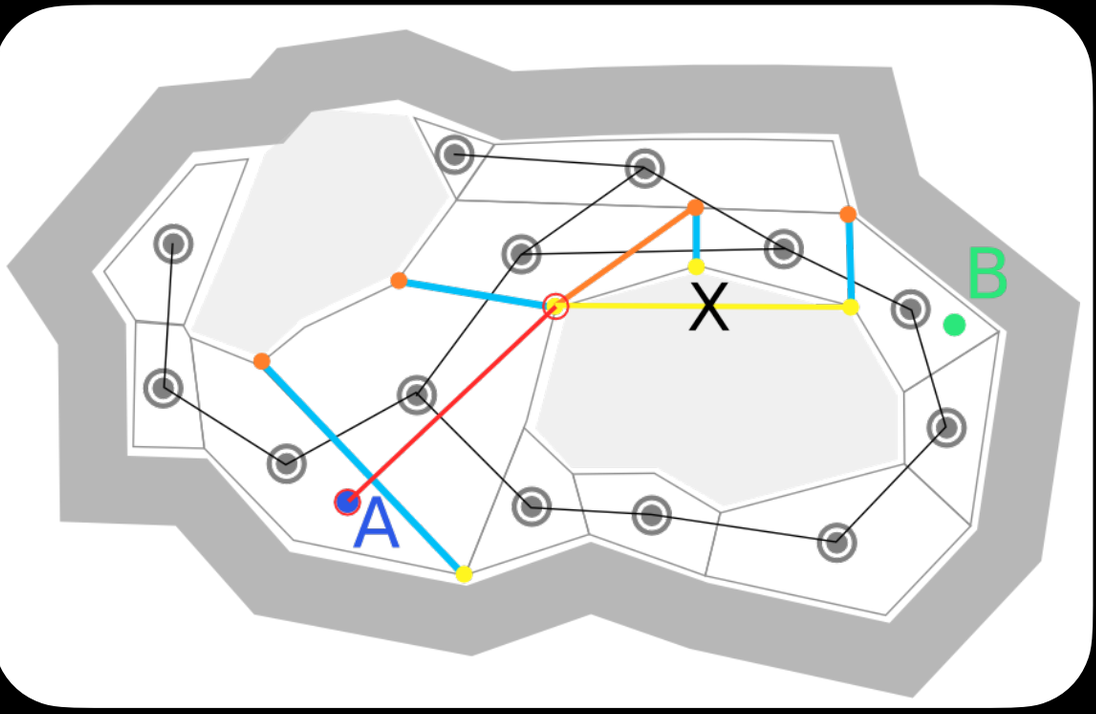
Pictures



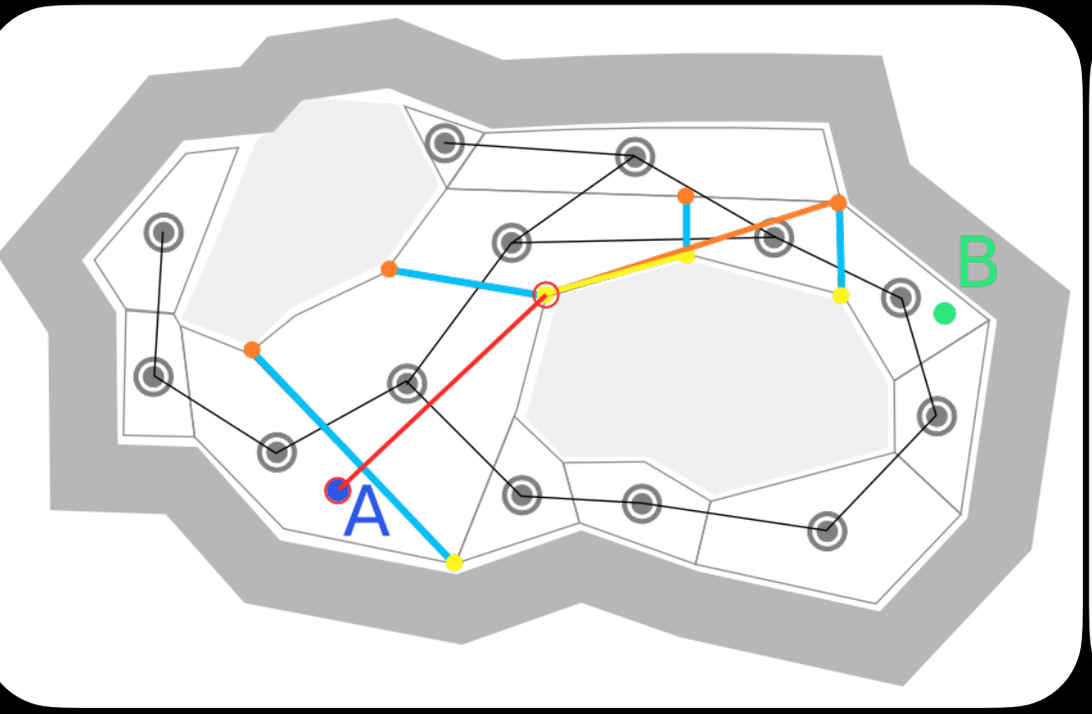


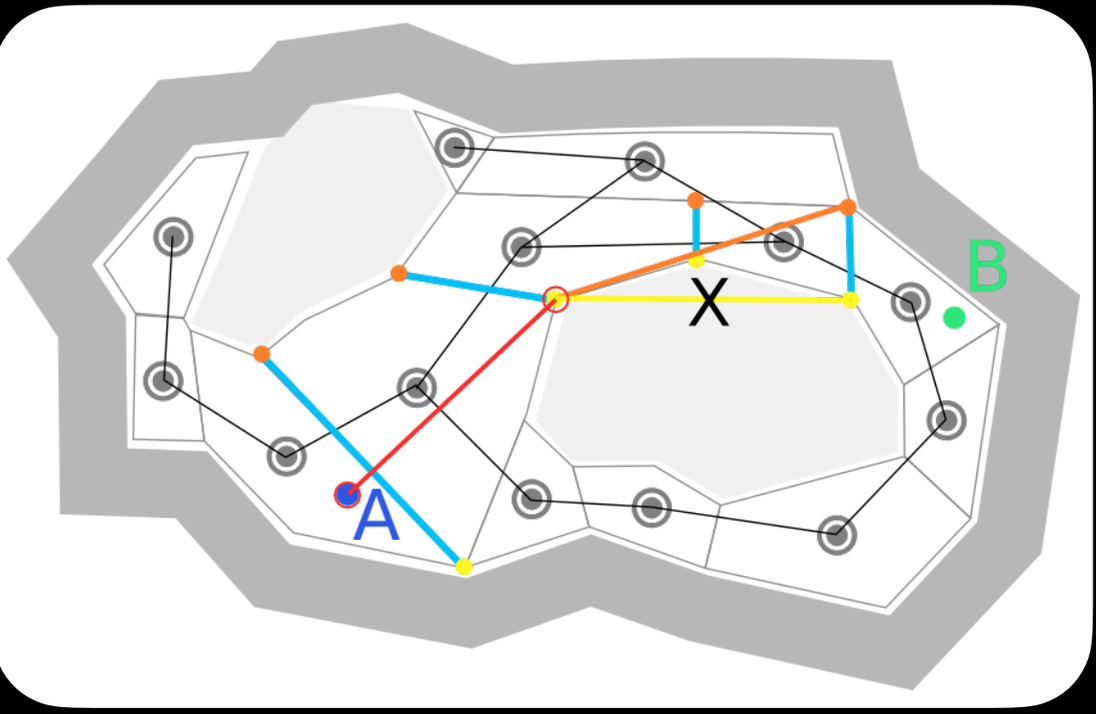
Pictures



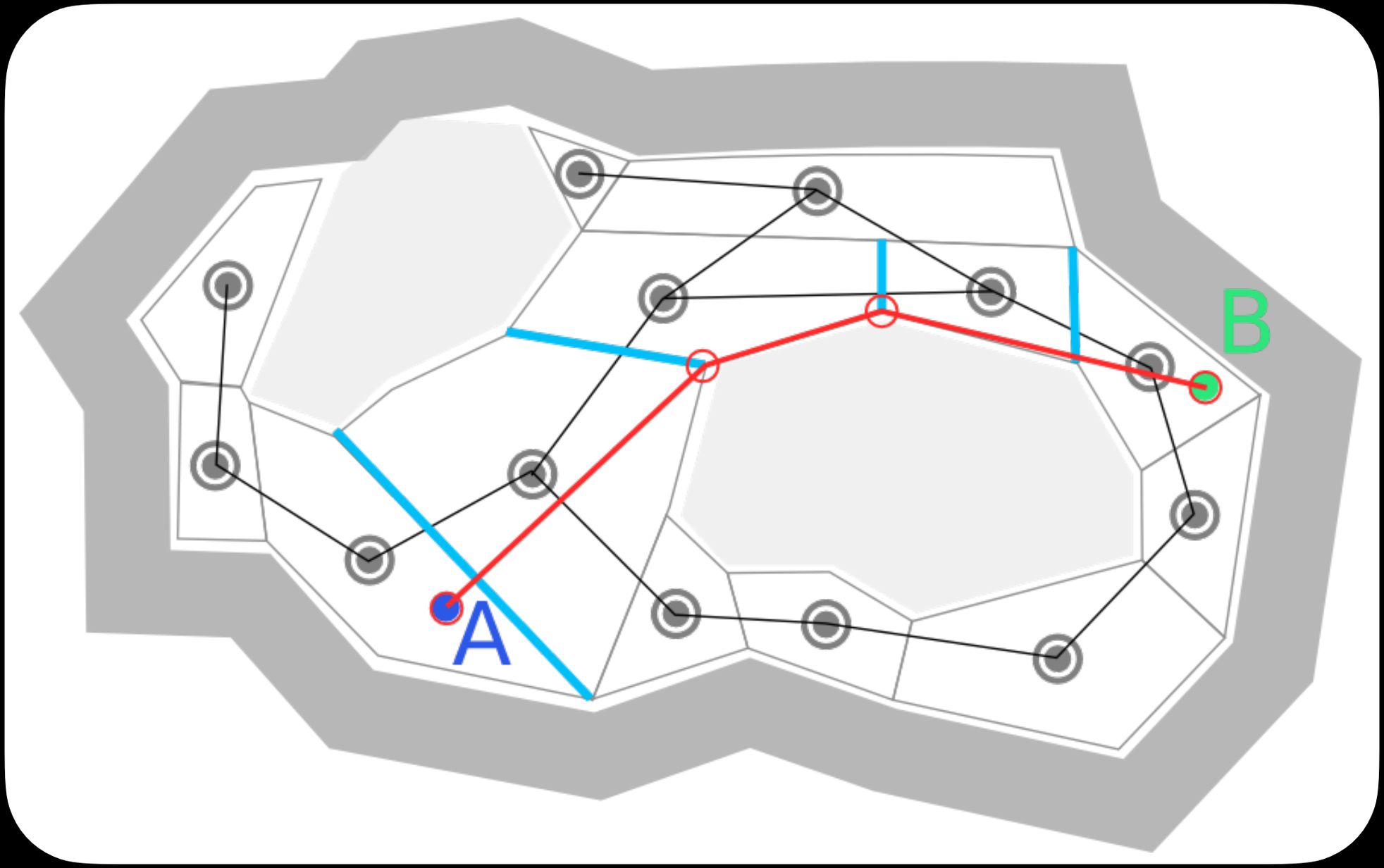


Pictures





Pictures



What did you learn today?

- Breadth First Search: Flooding the entire grid
- Pathfinding: Keeping track of where you came from
- Early exit: If you have a target, you can stop when you get to it.
- Dijkstra's Algorithm (again): Least cost path from source to every cell. But inefficient
- Greedy Best First Search (Heuristic): Very efficient, except if obstacles are in the way
- A* Search (Heuristic): a mix of Dijkstra's with a heuristic. Best of both worlds
- Navigation Meshes: When A* would be too expensive to run, we can abstract it.
- There are a lot of Navigation algorithms (and there are many more than we've discussed).