CS354R

DR SARAH ABRAHAM

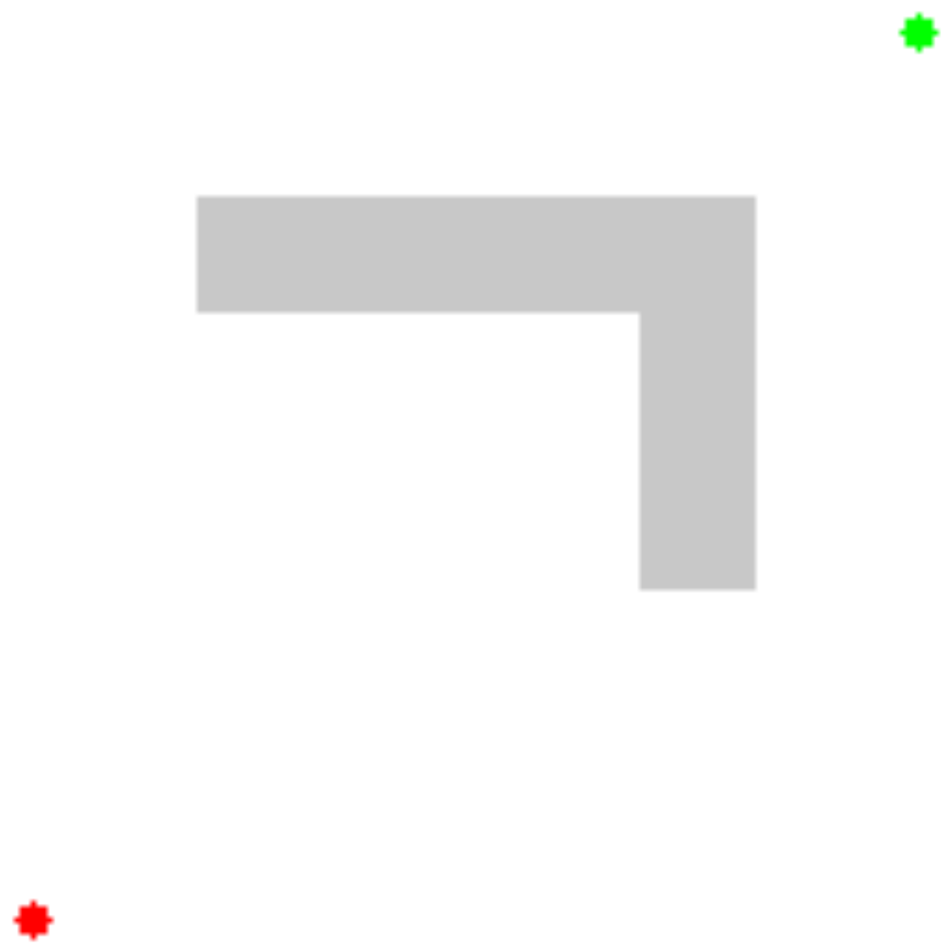
A* HEURISTICS

A* SEARCH

- *f(n)*: The current best estimate for the best path through a node: *f(n)=g(n)+h(n)*
  - *g(n)*: current known best cost for getting **to** a node from the start point
  - *h(n)*: current estimate for how much more it will cost to get **from** a node to the goal
- Optimality and efficiency depends on *h(n)*

A* IN ACTION

- Empty circle are in open set
- Fills circles are in closed set
    - Color indicates distance from start
- Line is set of nodes with lowest cost from start to goal



HEURISTICS

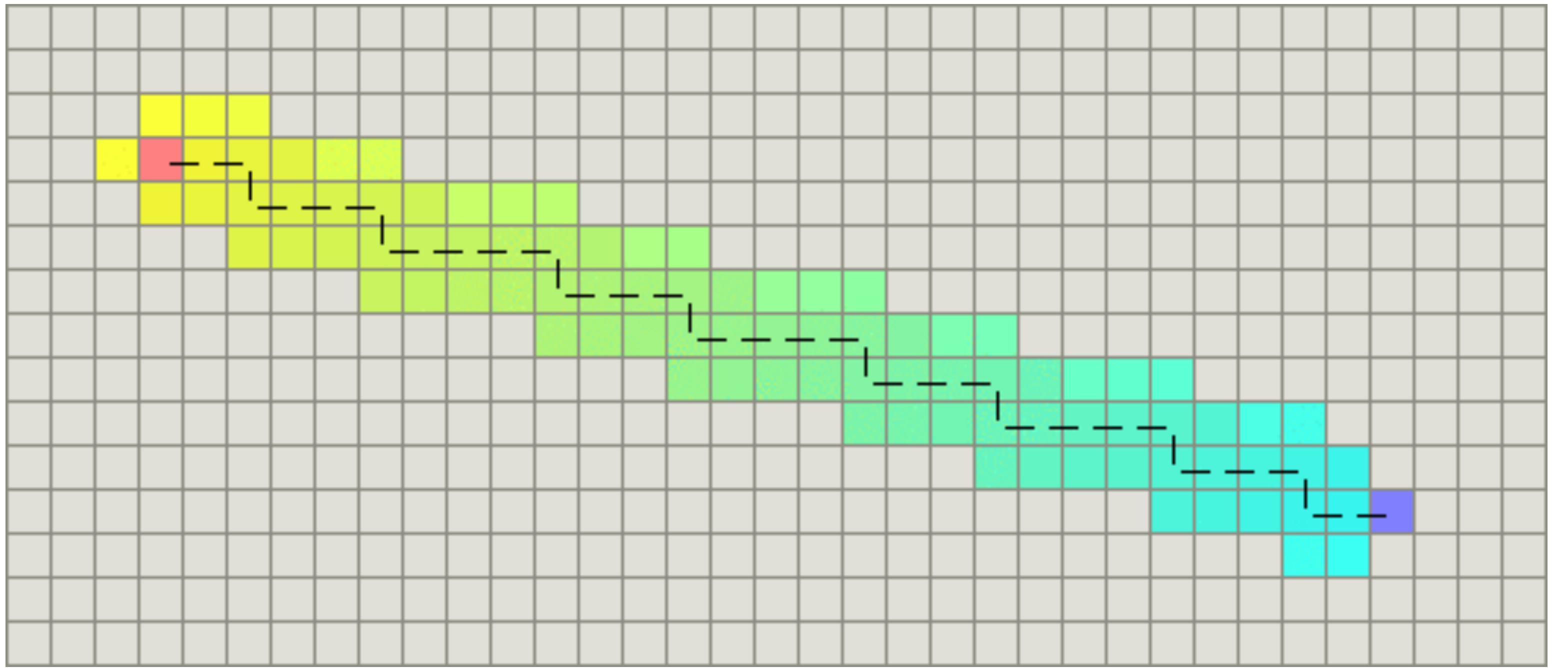
- For A* to be optimal, heuristic must be lower or equal to the true cost
    - Property of admissible path-finding algorithms
- The *f(n)* function must monotonically increase along any path out of the start node
    - True for almost any admissible heuristic (triangle inequality)
- The lower *h(n)*, the more nodes A* must expand
    - A* considers nodes with lower cost first
    - If *h(n)* matches the cost, will only expand best path
- Can combine heuristics if they provide different estimates:
    - $h(n) = max(h1(n),h2(n),h3(n),\ldots)$

DISCUSS

- What are some potential heuristics for A*?

MANHATTAN DISTANCE

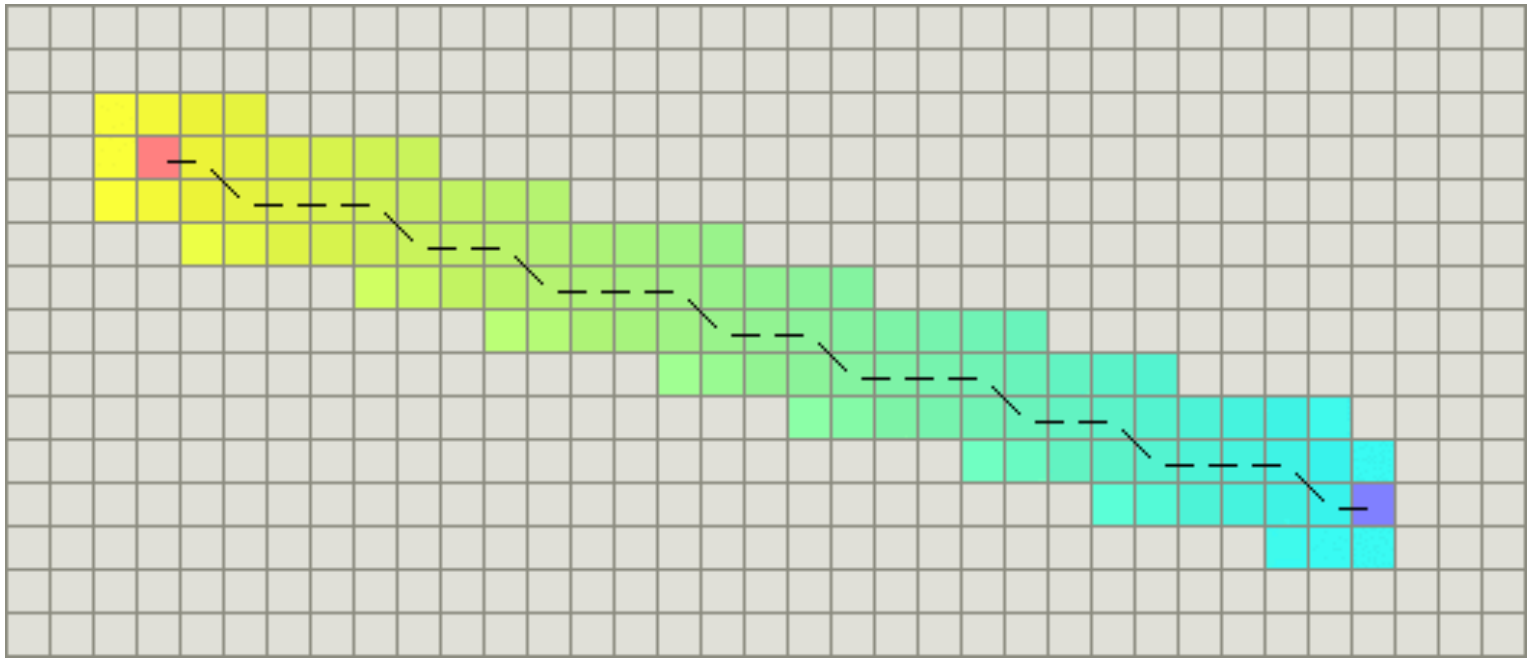
- Distance on strictly horizontal/vertical path
- Used on grids that allow 4 directions of movement
- Adaptable to hexagonal grids
- Find minimum cost D for moving to neighboring cell
- Heuristic is D * (dx + dy) where dx and dy are distance from node to goal on x and y axis



(http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html)

DIAGONAL DISTANCE

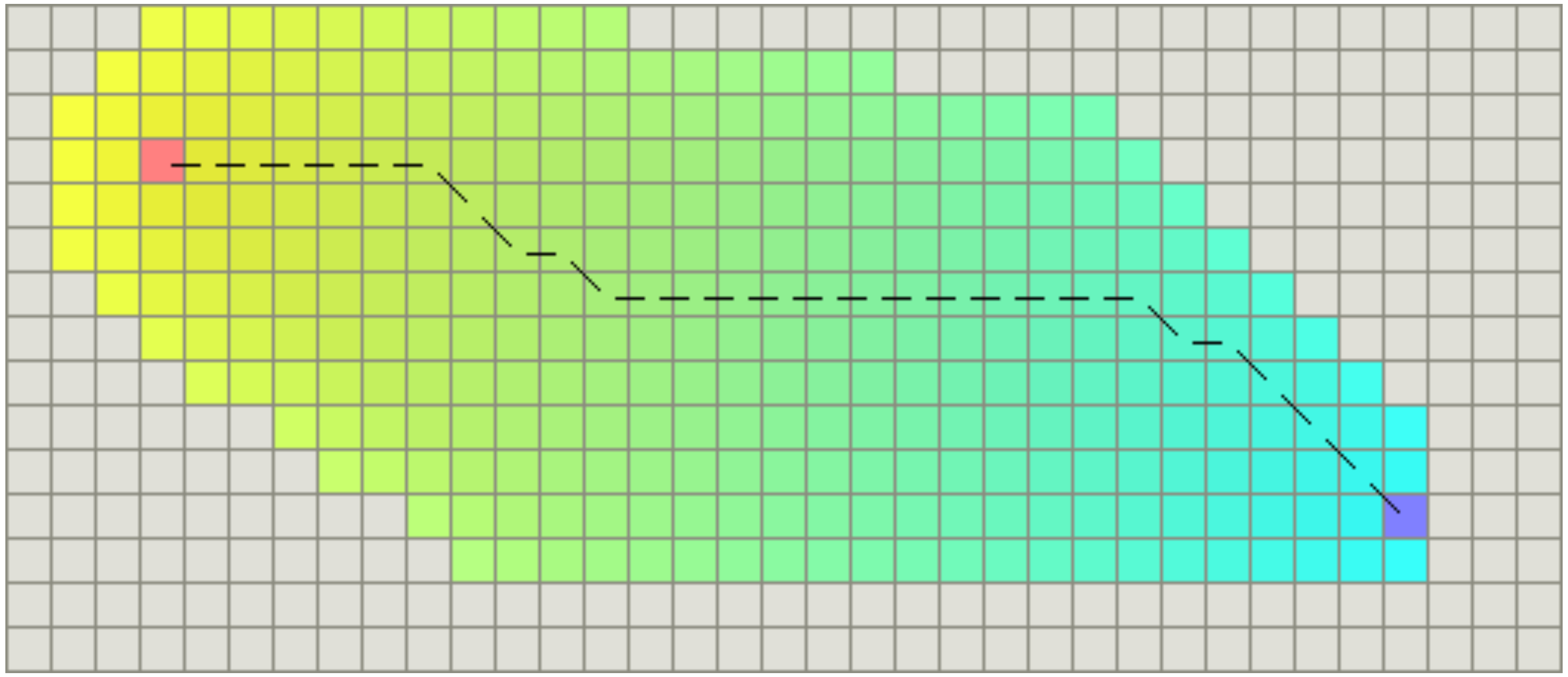
- Used on grids that allow 8 directions of movement
  - D is cost in cardinal directions
  - D2 is cost in ordinal directions
- Heuristic is D * (dx + dy) + (D2 - 2 * D) * min(dx, dy)
  - Cost of steps that cannot use diagonal plus cost of diagonal steps minus non-diagonal steps it avoids



(http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html)

EUCLIDEAN DISTANCE

- Used when units can move at any angle
- Heuristic is straight-line distance
  - D * sqrt(dx * dx + dy * dy)
- Shorter than Manhattan or diagonal distance
  - Will expand more nodes



(http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html)

A* PROBLEMS

- Discrete Search
  - Must have simple paths to connect waypoints
  - Typically use straight segments
  - Have to be able to compute cost
  - Must know that the object will not hit obstacles
- Unnatural Path Shape
    - Infinitely sharp corners
    - Jagged paths across grids
- Low Efficiency
  - Finding paths in complex environments can be expensive

DISCUSS

- How can we handle the jagged, unnatural paths A* might produce?

PATH STRAIGHTENING

- Straight paths typically look more plausible than jagged paths, particularly through open spaces
- Option 1: After the path is generated, look ahead from each waypoint to farthest unobstructed waypoint on the path
  - Replaces many segments with one straight path
  - Add more connections in waypoint graph (increases cost)
- Option 2: Bias the search toward straight paths
  - Segment cost increases if it requires turning a corner
  - Reduced efficiency when straight, unsuccessful paths are preferred

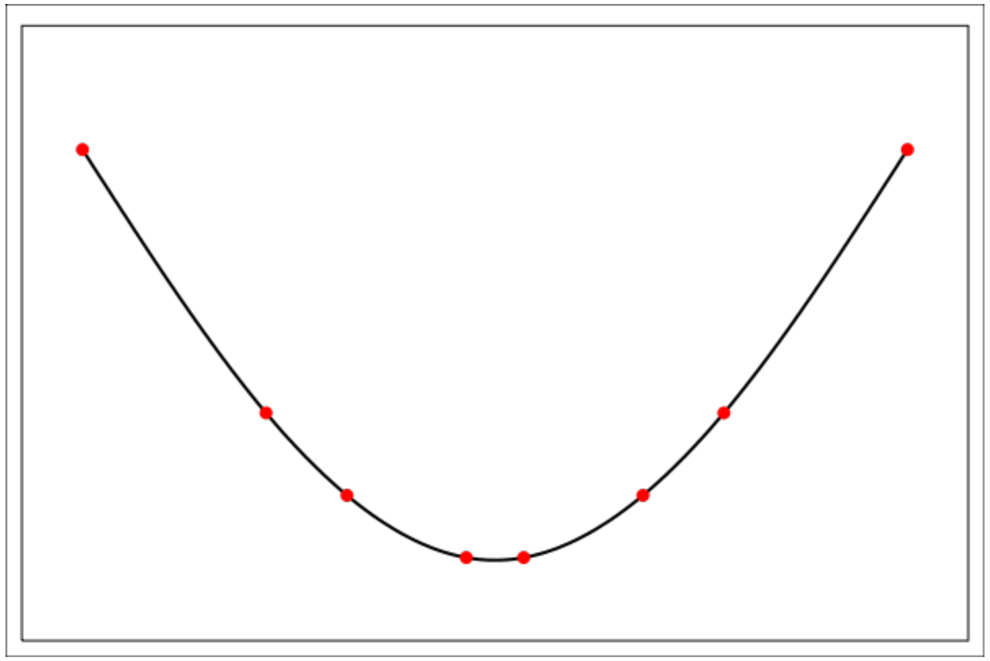
SMOOTHING WHILE FOLLOWING

- Rather than smooth out the path, smooth out the agent’s motion along it
- Typically, the agent’s position linearly interpolates between the waypoints
- Two primary choices to smooth the motion
  - Change the interpolation scheme
  - “Chase the point”

DIFFERENT INTERPOLATION SCHEMES

- View the task as moving a point (the agent) along a curve fitted through the waypoints
- We can now apply classic interpolation techniques to smooth the path such as splines
- Interpolating splines:
    - The curve passes through every waypoint
    - Specify the directions at the interpolated points
- Bezier or B-splines:
    - May not pass through the points
    - Only approximates them

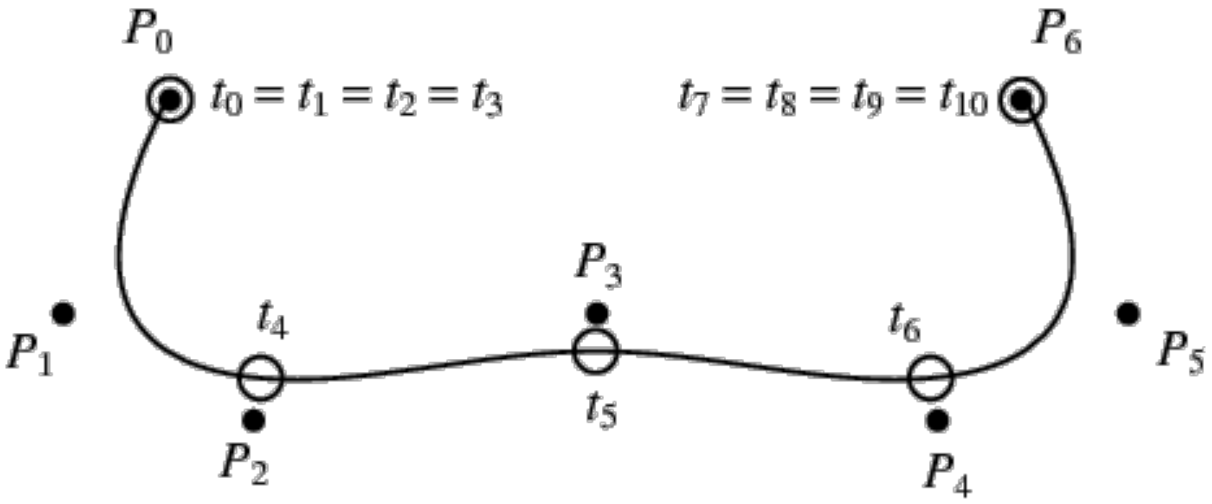
INTERPOLATION SCHEMES



(Wikipedia)

Cubic Interpolation

B-Spline

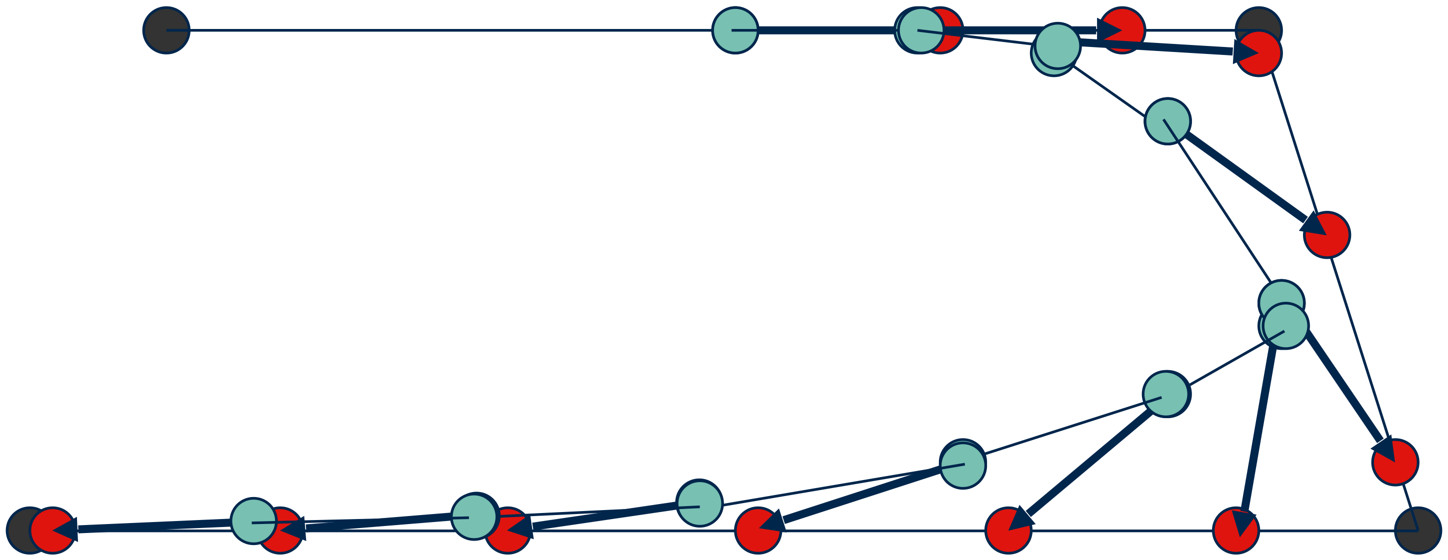


(Wolfram Mathworld)

CHASE THE POINT

- Instead of tracking along the path, agent chases a target point moving along the path
- Start with the target on the path ahead of the agent
- At each step:
    - Move the target along the path using linear interpolation
    - Move the agent toward the point location, keeping it a constant distance away or moving the agent at the same speed
- Works best for driving or flying games

CHASE THE POINT DEMO



IMPROVING A* EFFICIENCY

- Recall, A* is the most efficient optimal algorithm for a given heuristic
- Improving efficiency, therefore, means relaxing optimality
- Basic strategy: Use more information about the environment
    - Inadmissible heuristics use intuitions about which paths are likely to be better
    - Bias toward getting close to the goal ahead of exploring early unpromising paths

INADMISSIBLE HEURISTICS

- A* still gives an answer with inadmissible heuristics
  - Won’t be optimal (may not explore a node on the optimal path because its estimated cost is too high)
- Inadmissible heuristics may be much faster
  - Ignore “unpromising” paths earlier in the search
  - But not always faster (initially promising paths may be dead ends)

INADMISSIBLE EXAMPLE

- Multiply an admissible heuristic by a constant factor
- What does this do?
    - The frontier in A* consists of nodes that have roughly equal estimated total cost: *f* = *cost_so_far* + *estimated_to_go*
    - Consider two nodes on the frontier: one with *f* = *1+5*, another with *f* = *5+1*
    - Originally, A* would have expanded these at about the same time
    - If we multiply the estimate h(n) by 2, we get: *f* = *1+10* and *f* = *5+2*
    - So now, A* will expand the node that is closer to the goal long before the one that is further from the goal

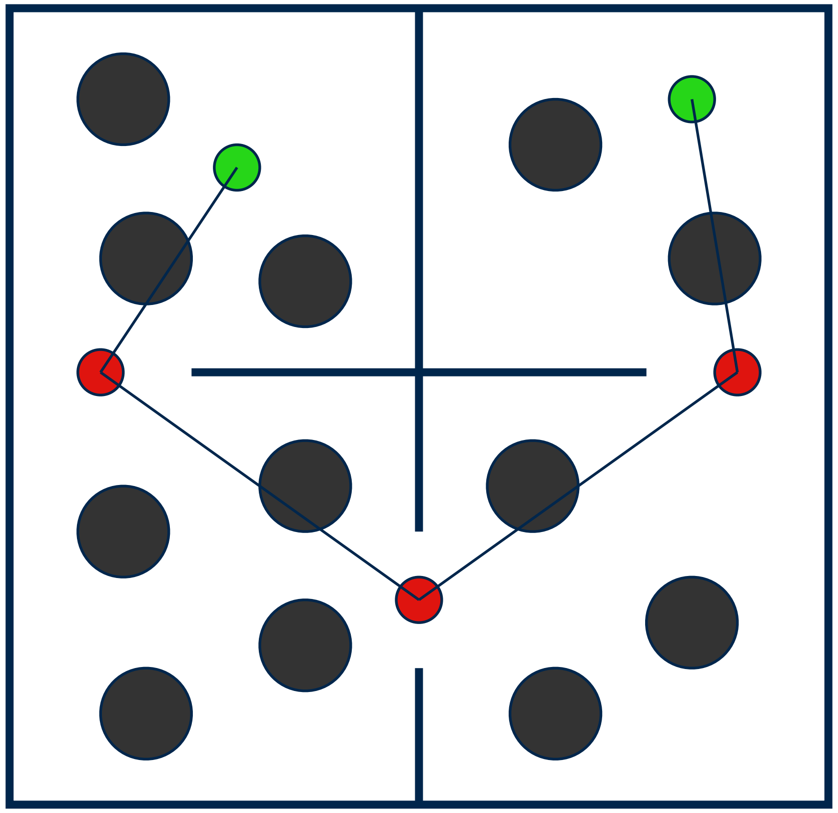
HIERARCHICAL PLANNING

- Many planning problems can be thought of hierarchically
  - To pass this class, I have to do the projects
  - To do the projects, I need to go to class, review the material, and start early
  - To go to class, I need to get to GDC
- Path planning is no exception:
  - To go from my current location to slay the dragon, I first need to know which rooms I will pass through
  - Then I need to know how to pass through each room, around the furniture, and so on

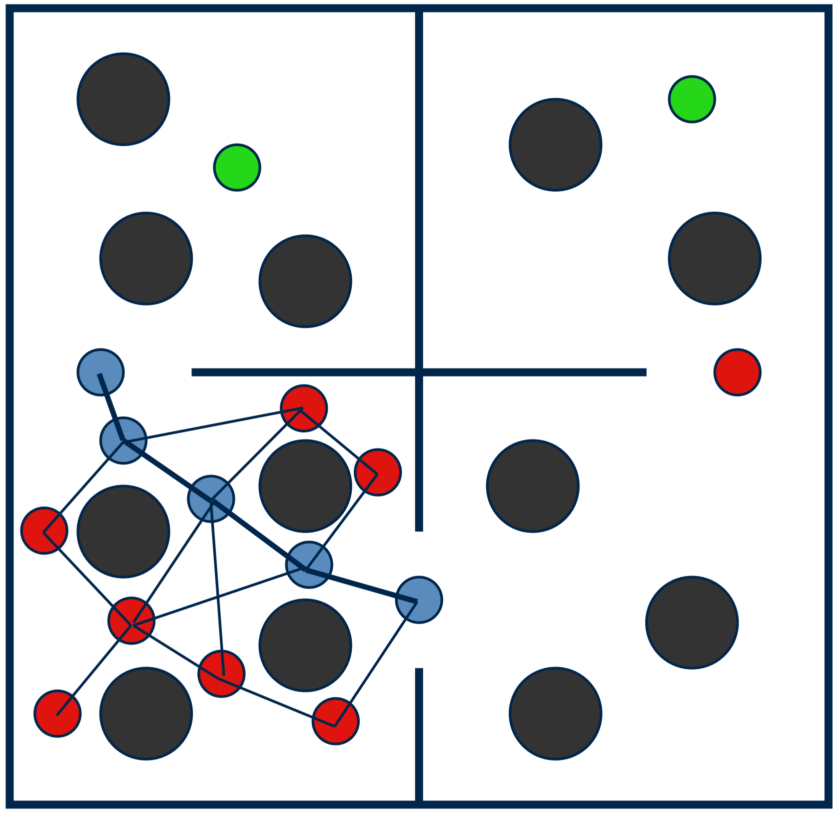
DOING HIERARCHICAL PLANNING

- Define a waypoint graph for the top of the hierarchy
    - e.g. Graph with waypoints in doorways (the centers)
    - Nodes linked if a clear path exists between them (not necessarily straight)
- For each edge in that graph, define another waypoint graph
    - Tells agents how to get between doorway in a room
    - Nodes from top level also in this graph
- First plan on the top level (returns a list of rooms to traverse)
- For each room on the list, plan a path across it
    - Delays low level planning until required

HIERARCHICAL PLANNING EXAMPLE



Plan this first



Then plan each room
(second room shown)

HIERARCHICAL PLANNING ADVANTAGES

- Search is typically cheaper
    - Initial search restricts the number of nodes considered in latter searches
- Well-suited to partial planning
    - Only plan each piece of path when it's required
    - Averages out cost of path over time avoiding lag when movement command issued
    - Path more adaptable to dynamic changes in the environment

HIERARCHICAL PLANNING ISSUES

- Result not optimal
  - No information about actual cost of low level is used at top level
- Top level plan locks in nodes that may be poor choices
  - Number of nodes at the top level restricted for efficiency
  - Cannot include all options available to a full planner
- Solution is to allow lower levels to override higher level
- Textbook example: Plan 2 lower level stages at a time
  - Plan from current doorway, through next doorway, to doorway after
  - After reaching the next doorway, drop the second half of the path and start again

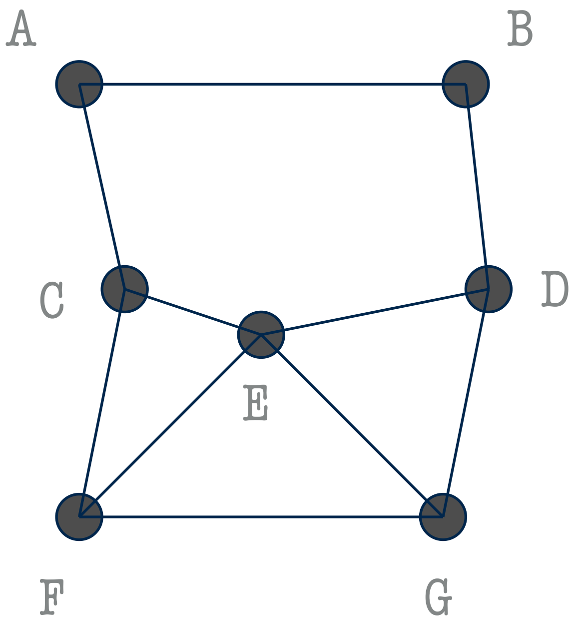
PRE-PLANNING

- If the set of waypoints is fixed and obstacles don't move, the shortest path between any two never changes
- If it doesn't change, compute it ahead of time
- This can be done with all-pairs shortest paths algorithms
  - Dijkstra's algorithm run for each start point, or special purpose all-pairs algorithms
- How to store the paths?

STORING ALL-PAIRS PATHS

- Trivial solution is to store the shortest path to every other node in every node
- A better way:
    - If there is a shortest path from A to B: A-B
        - Every shortest path that goes through A on the way to B must use A-B
    - This holds for any source node: the **next step** from any node on the way to B **does not** depend on how you got to that node
    - Only store the next step out of each node for each possible destination

EXAMPLE



And I want to go to:

If I'm at:

	A	B	C	D	E	F	G
A	-	A-B	A-C	A-B	A-C	A-C	A-C
B	B-A	-	B-A	B-D	B-D	B-D	B-D
C	C-A	C-A	-	C-E	C-E	C-F	C-E
D	D-B	D-B	D-E	-	D-E	D-E	D-G
E	E-C	E-D	E-C	E-D	-	E-F	E-G
F	F-C	F-E	F-C	F-E	F-E	-	F-G
G	G-E	G-D	G-E	G-D	G-E	G-F	-

To get from A to G:

+ A-C

+ C-E

+ E-G