Design of Human Interface Game Software

AI

- Path Finding
- Hierarchical Path Planning
- Dynamic Path Planning

Path Finding

- Very common problem in games:
  - In FPS: How does the AI get from room to room?
  - In RTS: User clicks on units, tells them to go somewhere. How do they get there? How do they avoid each other?
  - Chase games, sports games, …
- Very expensive part of games
  - Lots of techniques that offer quality, robustness, speed trade-offs

Path Finding Problem

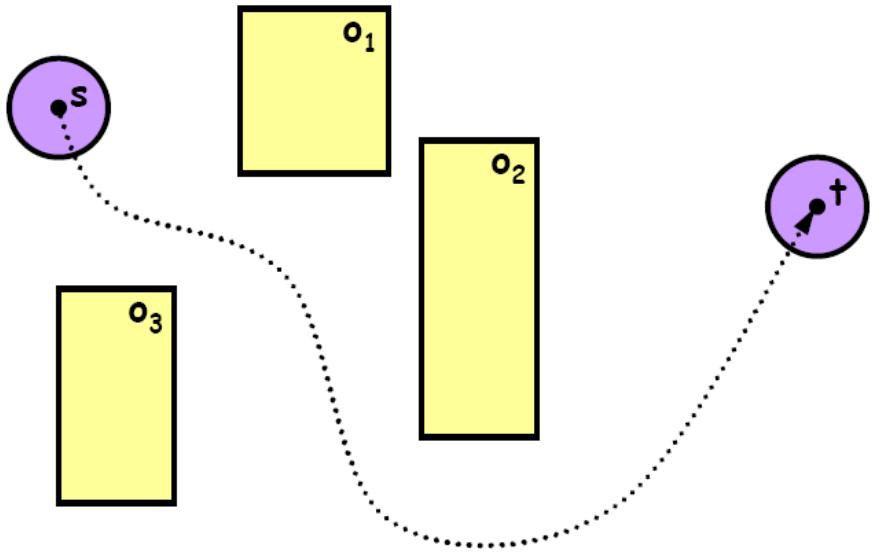
- Problem Statement (Academic): Given a start point, A, and a goal point, B, find a path from A to B that is clear
  - Minimize a cost: distance, travel time, …
    - Travel time depends on terrain, for instance
  - May be complicated by dynamic changes: paths being blocked or removed
  - May be complicated by unknowns – don't have complete information
- Problem Statement (Games): Find a reasonable path that gets the object from A to B
  - Reasonable may not be optimal – not shortest, for instance
  - It may be OK to pass through things sometimes
  - It may be OK to make mistakes and have to backtrack

Search or Optimization?

- Path planning (also called route-finding) can be phrased as a search problem:
  - Find a path to the goal B that minimizes Cost(path)
  - There are a wealth of ways to solve search problems, and we will look at some of them
- Path planning is also an optimization problem:
  - Minimize Cost(path) subject to the constraint that path joins A and B
    - State space is paths joining A and B, kind of messy
  - There are a wealth of ways to solve optimization problems
- The difference is mainly one of terminology: different communities (AI vs. Optimization)
  - But, search is normally on a discrete state space

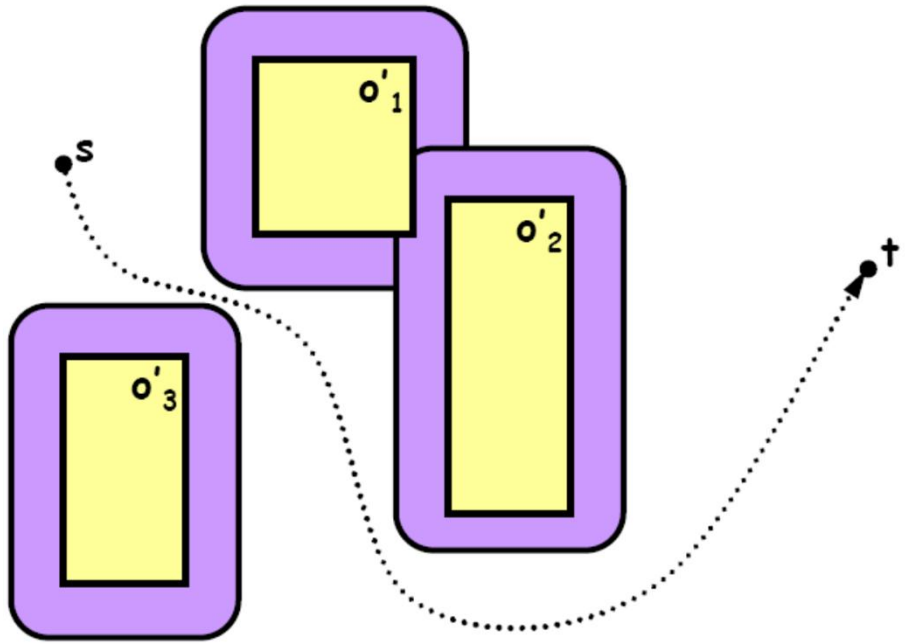
Configuration Space

- Point motion planning
  - Much easier to plan the motion of a point than a geometric object
  - Can we reduce object motion planning to point motion planning?
- Simple example
  - We want to plan the motion of a circular disk of radius r from s to t



Configuration Space

- Equivalent problem
  - Grow each obstacle by radius r
  - Shrink the disk down to a point
  - Plan the motion of a point from s to t



Brief Overview of Techniques

- Discrete algorithms: BFS, Greedy search, A*, ...
- Potential fields:
  - Put a “force field” around obstacles, and follow the “potential valleys”
- Pre-computed plans with dynamic re-planning
  - pre-compute answer and modify as required
- Special algorithms for special cases:
  - E.g. Given a fixed start point, fast ways to find paths around polygonal obstacles

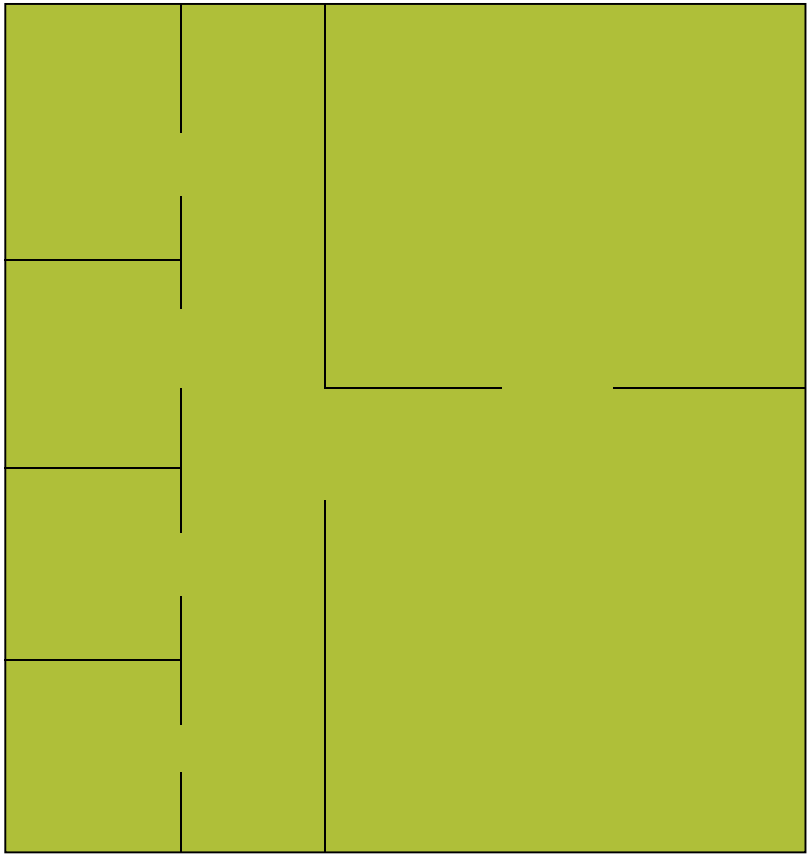
Graph-Based Algorithms

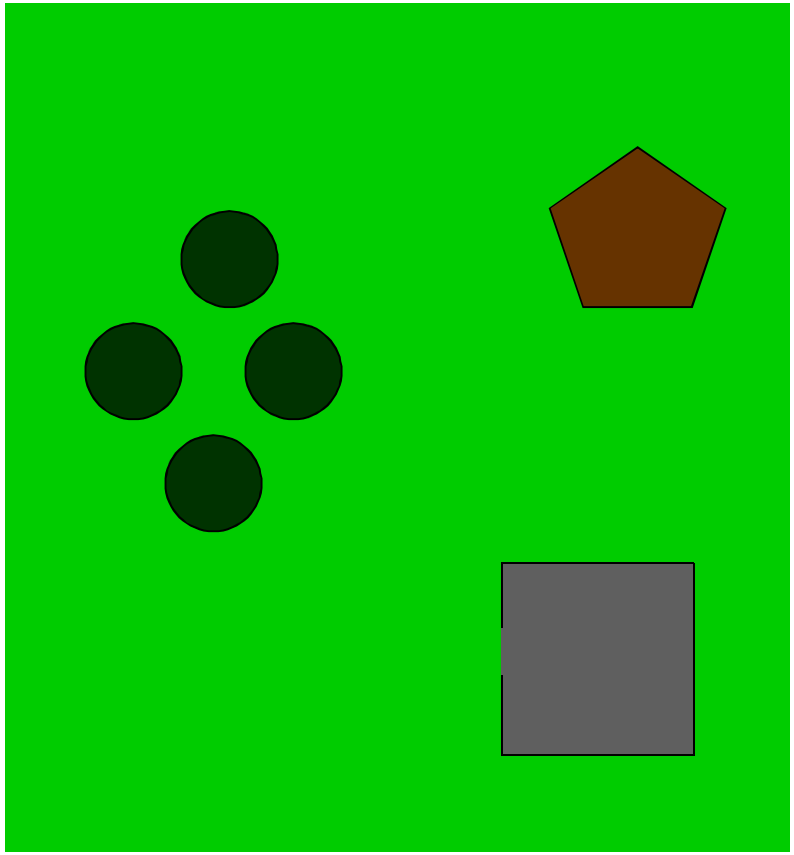
- Ideally, path planning is point to point (any point in the world to any other, through any unoccupied point)
- But, the search space is complex (space of arbitrary curves)
- The solution is to discretize the search space
  - Restrict the start and goal points to a finite set
  - Restrict the paths to be on lines (or other simple curves) that join points
- Form a graph: Nodes are points, edges join nodes that can be reached along a single curve segment
  - Search for paths on the graph

Waypoints

- The discrete set of points you choose are called *waypoints*
- Where do you put the waypoints?
  - There are many possibilities
- How do you find out if there is a simple path between them?
  - Depends on what paths you are willing to accept – almost always assume straight lines
- The answers to these questions depend very much on the type of game you are developing
  - The environment: open fields, enclosed rooms, etc...
  - The style of game: covert hunting, open warfare, friendly romp, ...

Where Would You Put Waypoints?





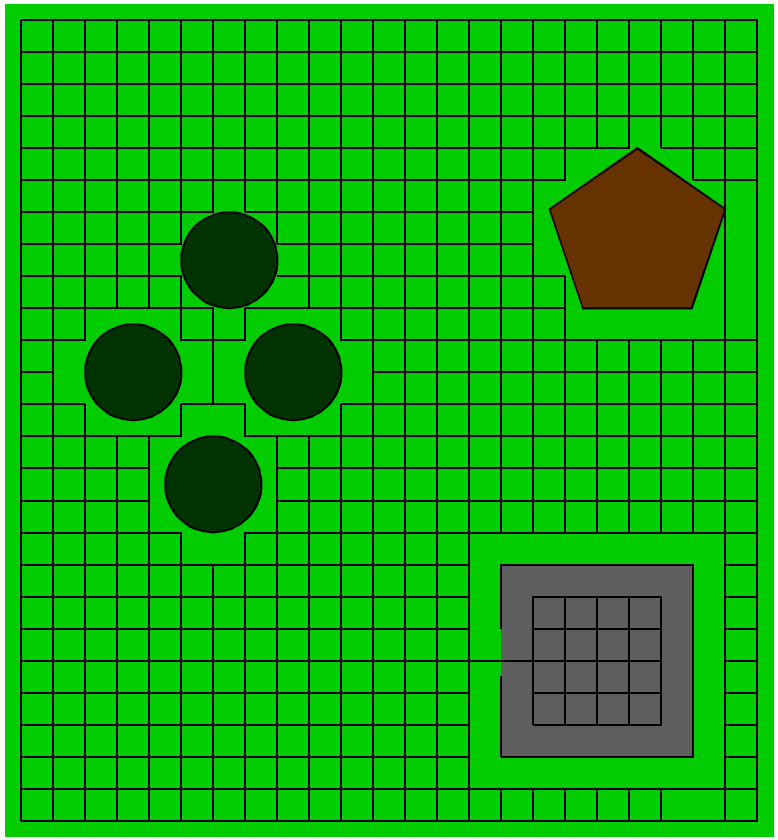
Waypoints By Hand

- Place waypoints by hand as part of level design
  - Best control, most time consuming
- Many heuristics for good places:
  - In doorways, because characters have to go through doors and straight lines joining rooms always go through doors
  - Along walls, for characters seeking cover
  - At other discontinuities in the environments (edges of rivers, for example)
  - At corners, because shortest paths go through corners
- Good waypoints can make the AI seem smarter
  - More natural, varied motion
  - Automatic obstacle avoidance

Waypoints By Grid

- Place a grid over the world, and put a waypoint at every gridpoint that is open
  - Automated method, and maybe even implicit in the environment
- Joining waypoints
  - Edge/world intersection test
  - Normally only allow moves to immediate neighbors (4-neighbors, 8-neighbors)
- Pros & Cons
  - Easy to implement
  - Insensitive to variations in resolution
  - Problems with tight spaces

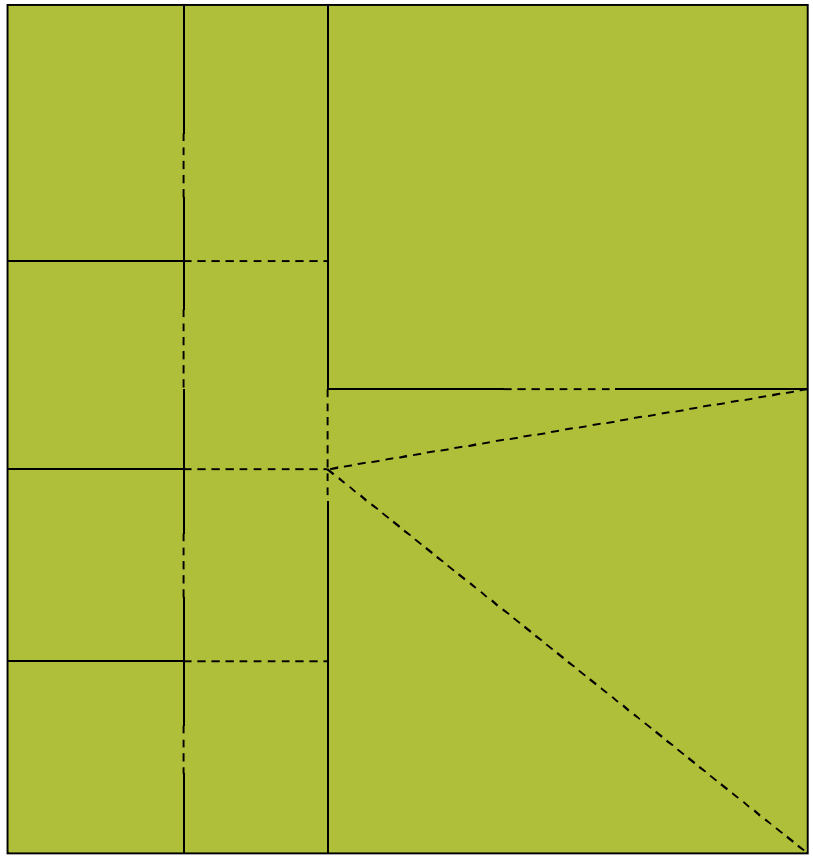
Grid Example



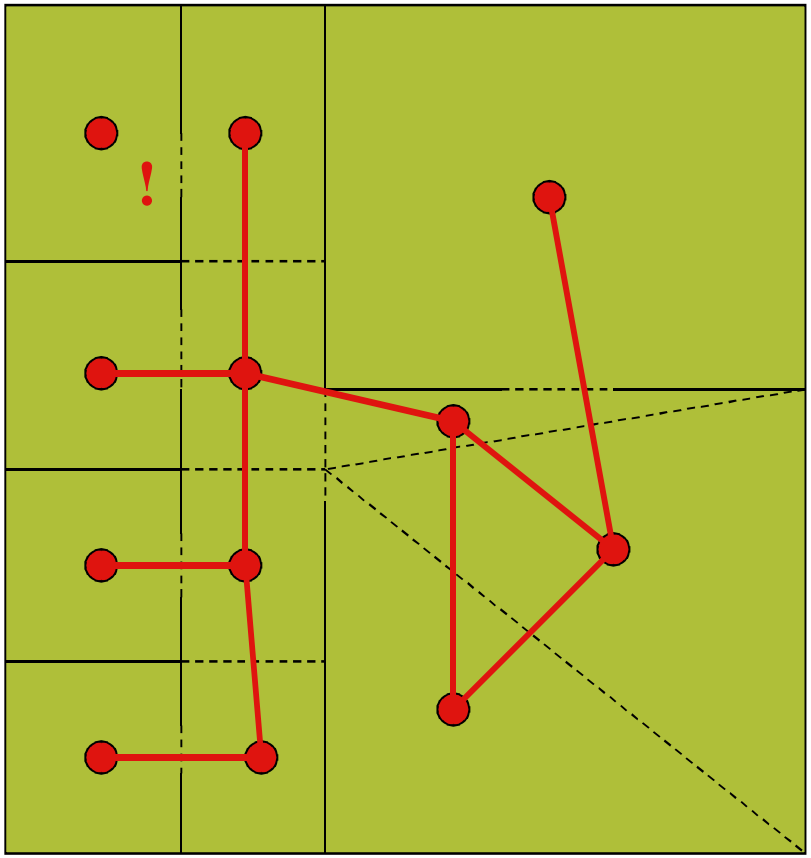
- Note that grid points pay no attention to the geometry
- Method can be improved:
  - Perturb grid to move closer to obstacles
  - Adjust grid resolution
  - Use different methods for inside and outside building
    - Join with waypoints in doorways

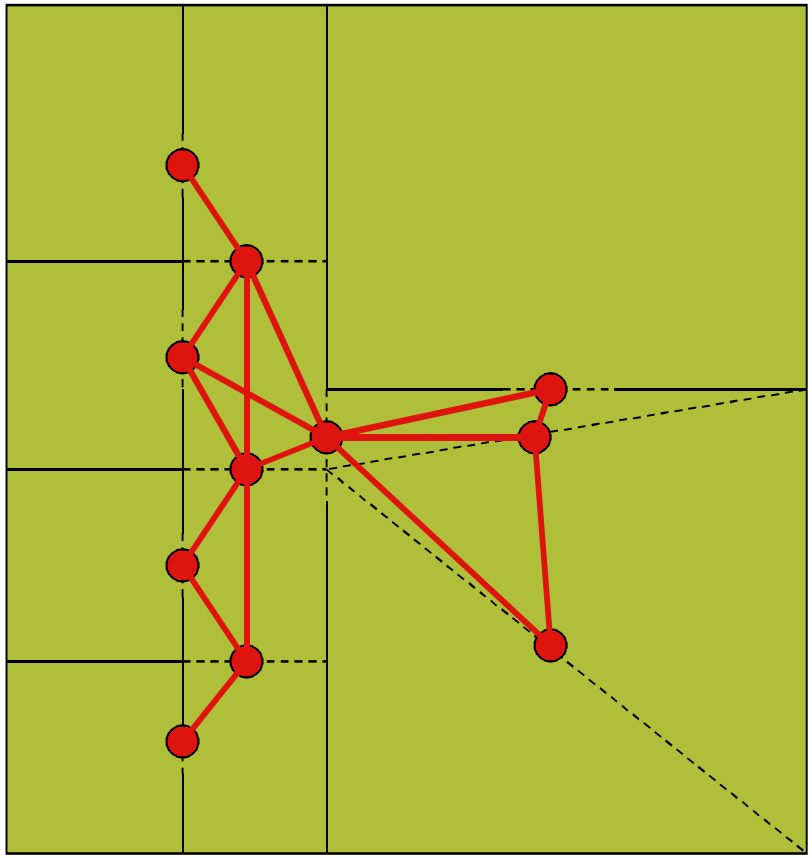
Waypoints From Polygons

- Choose waypoints based on the floor polygons in your world
- Or, explicitly design polygons to be used for generating waypoints
- How do we go from polygons to waypoints?
  - Hint: there are two obvious options



Waypoints From Polygons



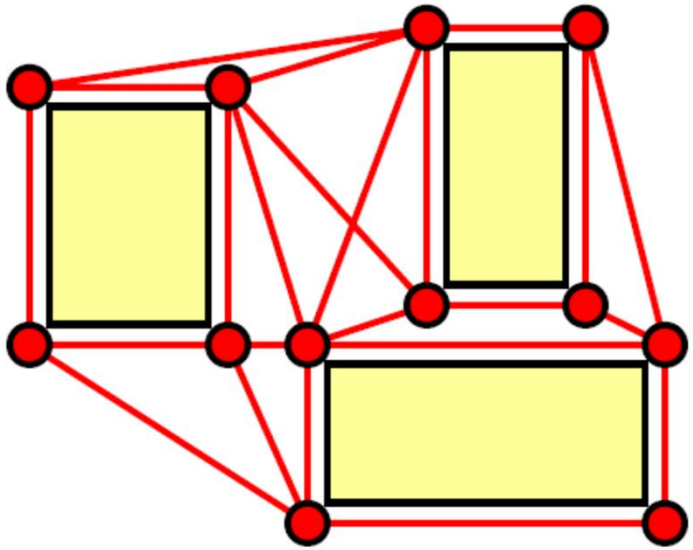


Could also add points on walls

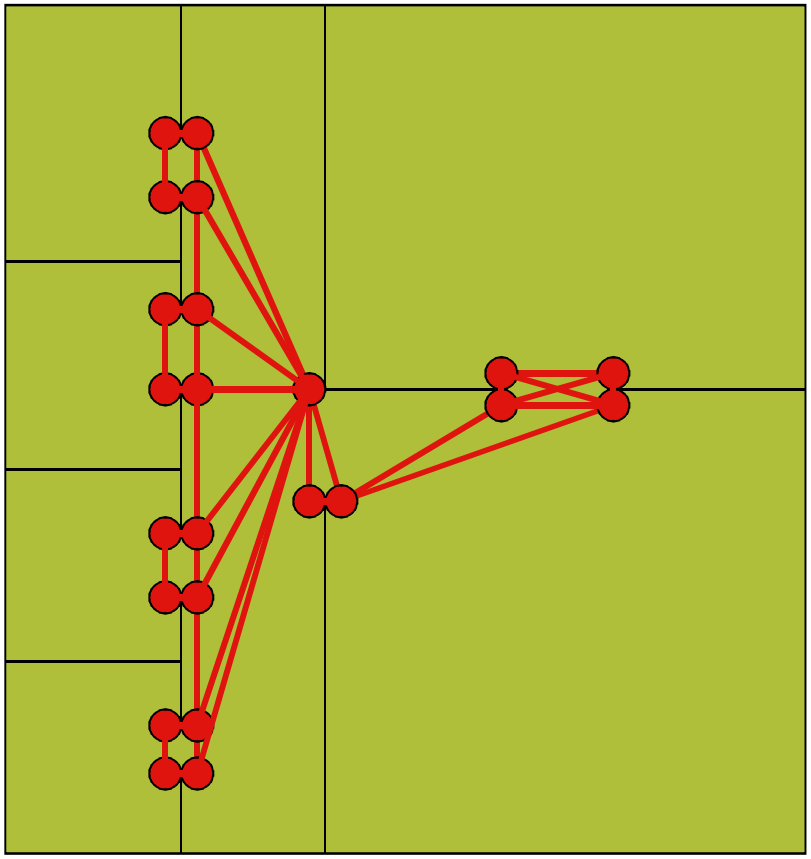
Waypoints From Corners

- Place waypoints at every convex corner of the obstacles
  - Actually, put a small distance from corners
- Connects all the corners that can see each other

- Paths will be the shortest
- However, some unnatural paths may result
  - Particularly along corridors – stick to walls



Waypoints From Corners

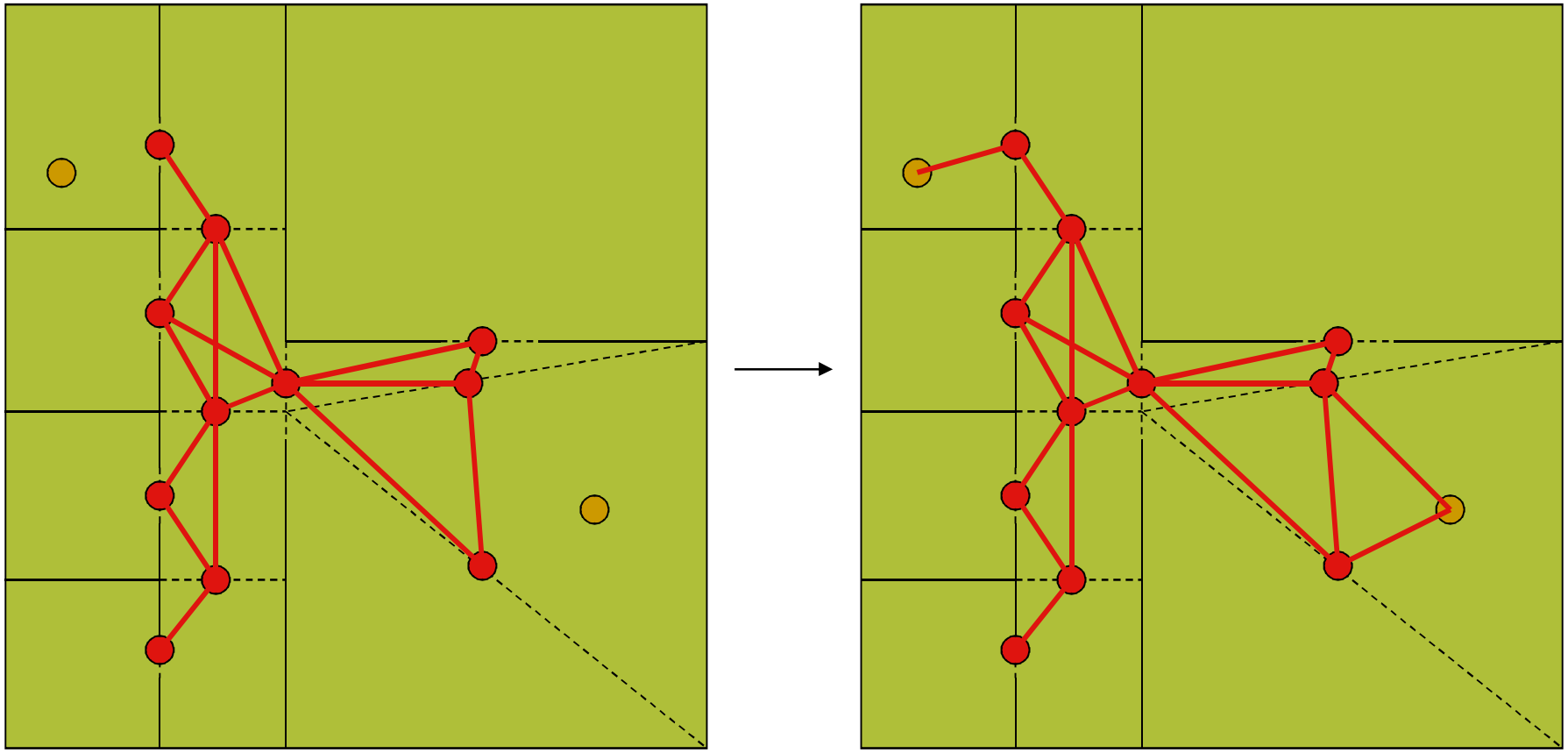


- NOTE: Not every edge is drawn
- Produces very dense graphs

Getting On and Off

- Typically, you do not wish to restrict the character to the waypoints or the graph edges
- When the character starts, find the closest waypoint and move to that first
  - Or, find the waypoint most in the direction you think you need to go
  - Or, try all of the potential starting waypoints and see which gives the shortest path
- When the character reaches the closest waypoint to its goal, jump off and go straight to the goal point
- Best option: Add a new, temporary waypoint at the precise start and goal point, and join it to nearby waypoints

Getting On and Off



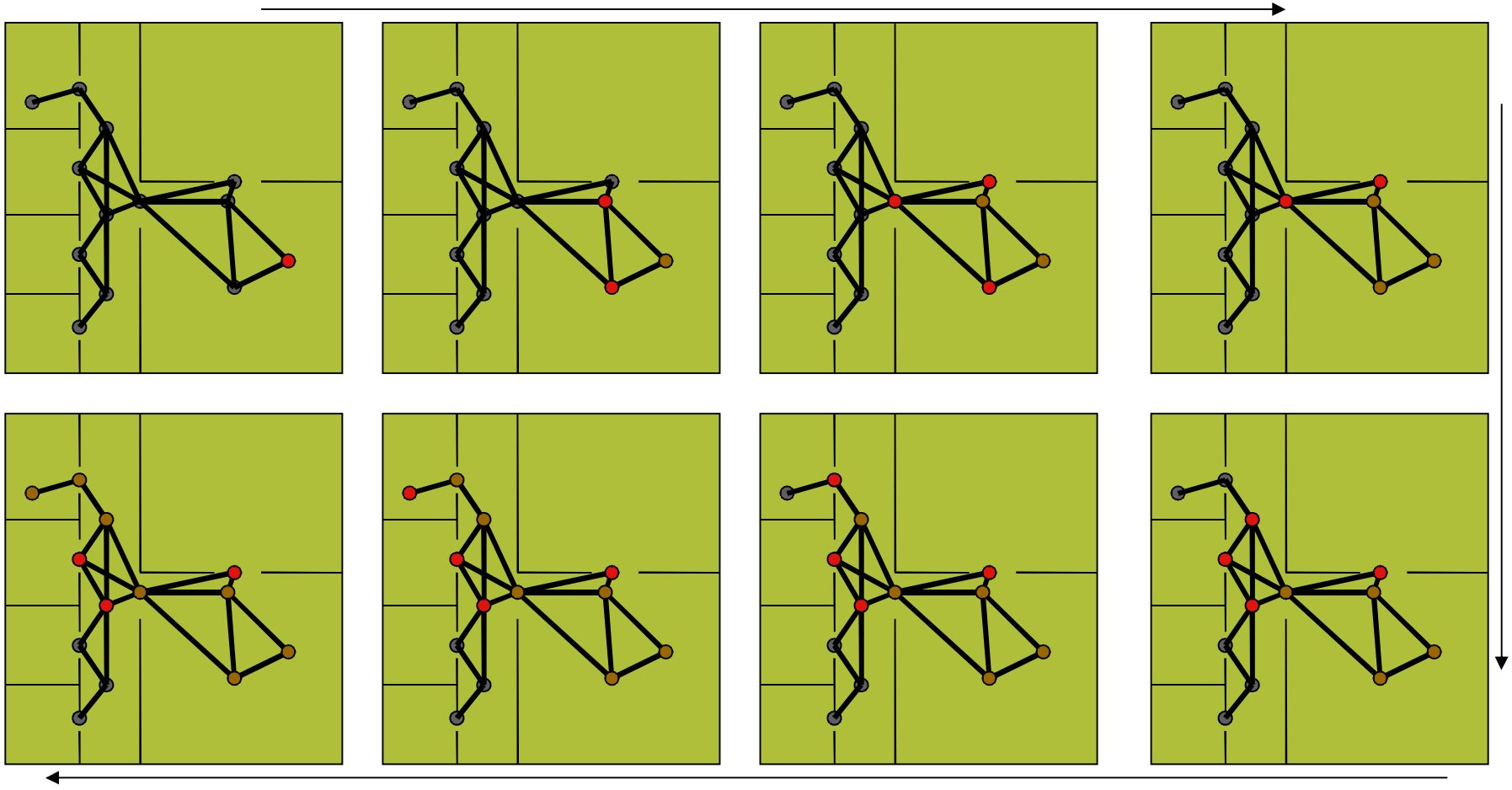
Best-First-Search

- Start at the start node and search outwards
- Maintain two sets of nodes:
  - Open nodes are those we have reached but don't know best path
  - Closed nodes that we know the best path to
- Keep the open nodes sorted by cost
- Repeat: *Expand* the "best" open node
  - If it's the goal, we're done
  - Move the "best" open node to the closed set
  - Add any nodes reachable from the "best" node to the open set
    - Unless already there or closed
  - Update the cost for any nodes reachable from the "best" node
    - New cost is min(old-cost, cost-through-best)

Best-First-Search

- Precise properties depend on how “best” is defined, but it will always find the goal if it can be reached
- To store the best path:
  - Keep a pointer in each node *n* to the previous node along the best path to *n*
  - Update these as nodes are added to the open set and as nodes are expanded (whenever the cost changes)
  - To find path to goal, trace pointers back from goal nodes

Example



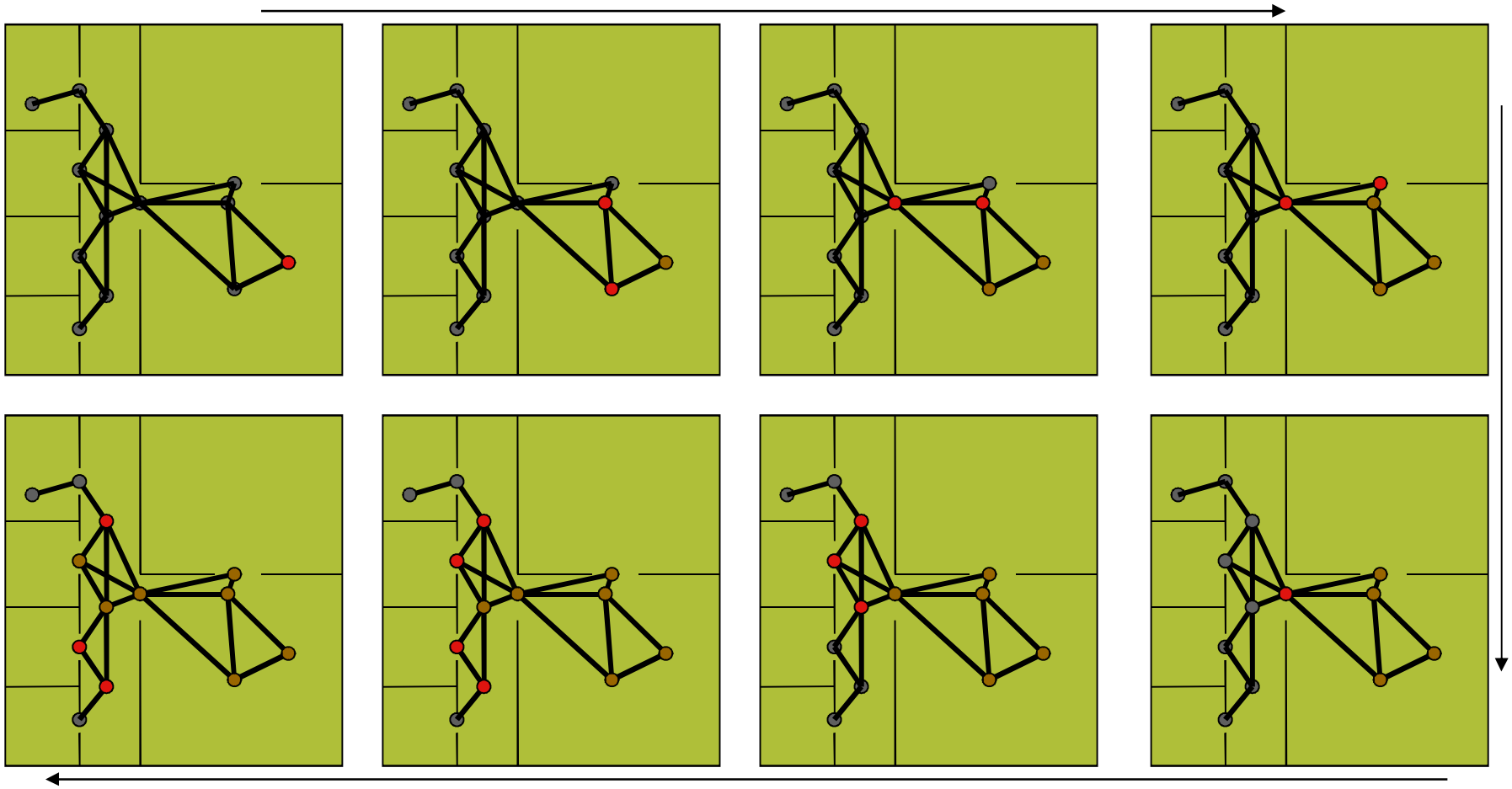
Definitions

- *g(n)*: The current known best cost for getting **to** a node from the start point
  - Can be computed based on the cost of traversing each edge along the current shortest path to *n*
- *h(n)*: The current estimate for how much more it will cost to get **from** a node to the goal
  - A *heuristic*: The exact value is unknown but this is your best guess
  - Some algorithms place conditions on this estimate
- *f(n)*: The current best estimate for the best path through a node: $f(n)=g(n)+h(n)$

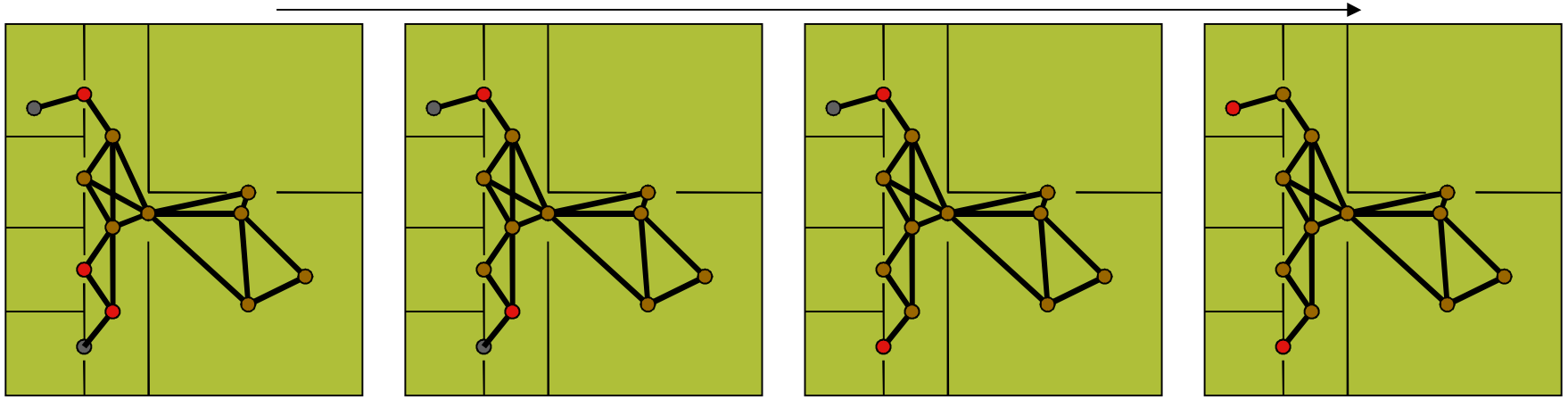
Using *g(n)* Only

- Define "best" according to *f(n)=g(n)*, the shortest known path from the start to the node
- Equivalent to breadth first search
- Is it optimal?
  - When the goal node is expanded, is it along the shortest path?
- Is it efficient?
  - How many nodes does it explore? Many, few, …?
- Behavior is the same as defining a constant heuristic function: *h(n)=const*

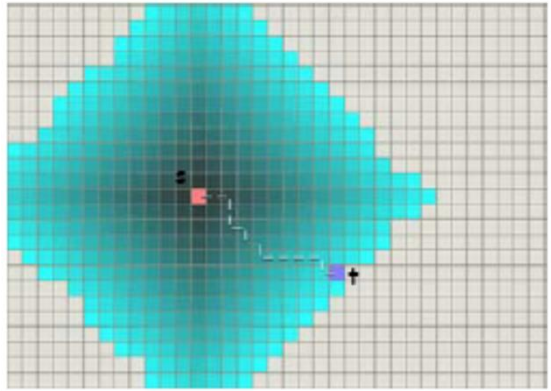
Breadth First Search

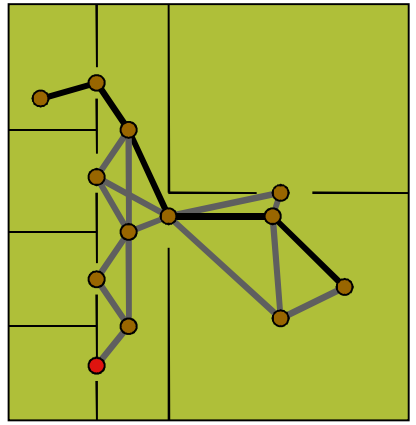


Breadth First Search



- On a grid with uniform cost per edge, the frontier expands in a circle out from the start point

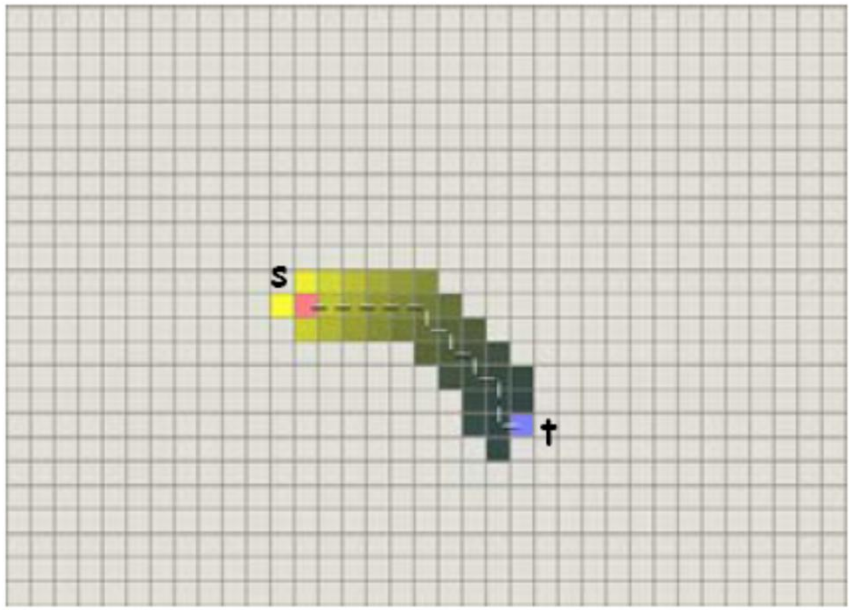


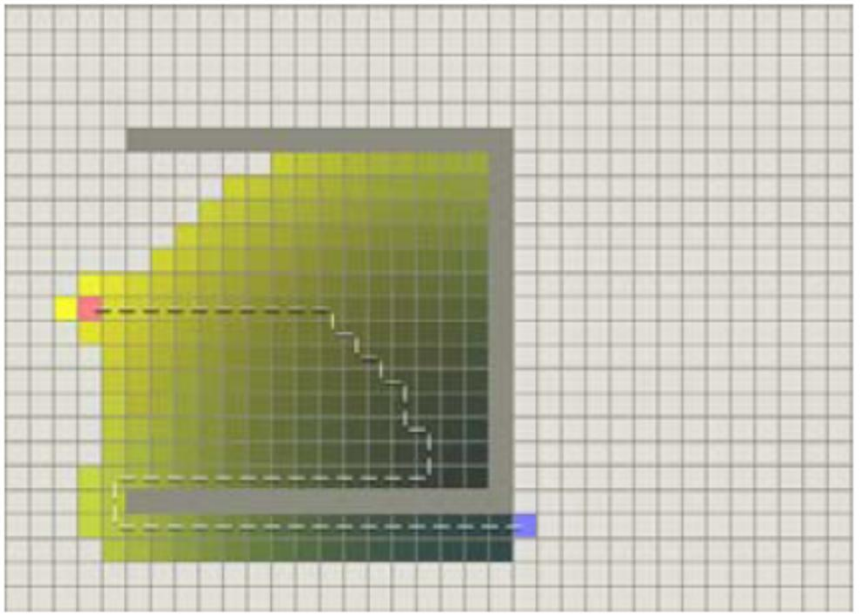


Using *h(n)* Only (Greedy Search)

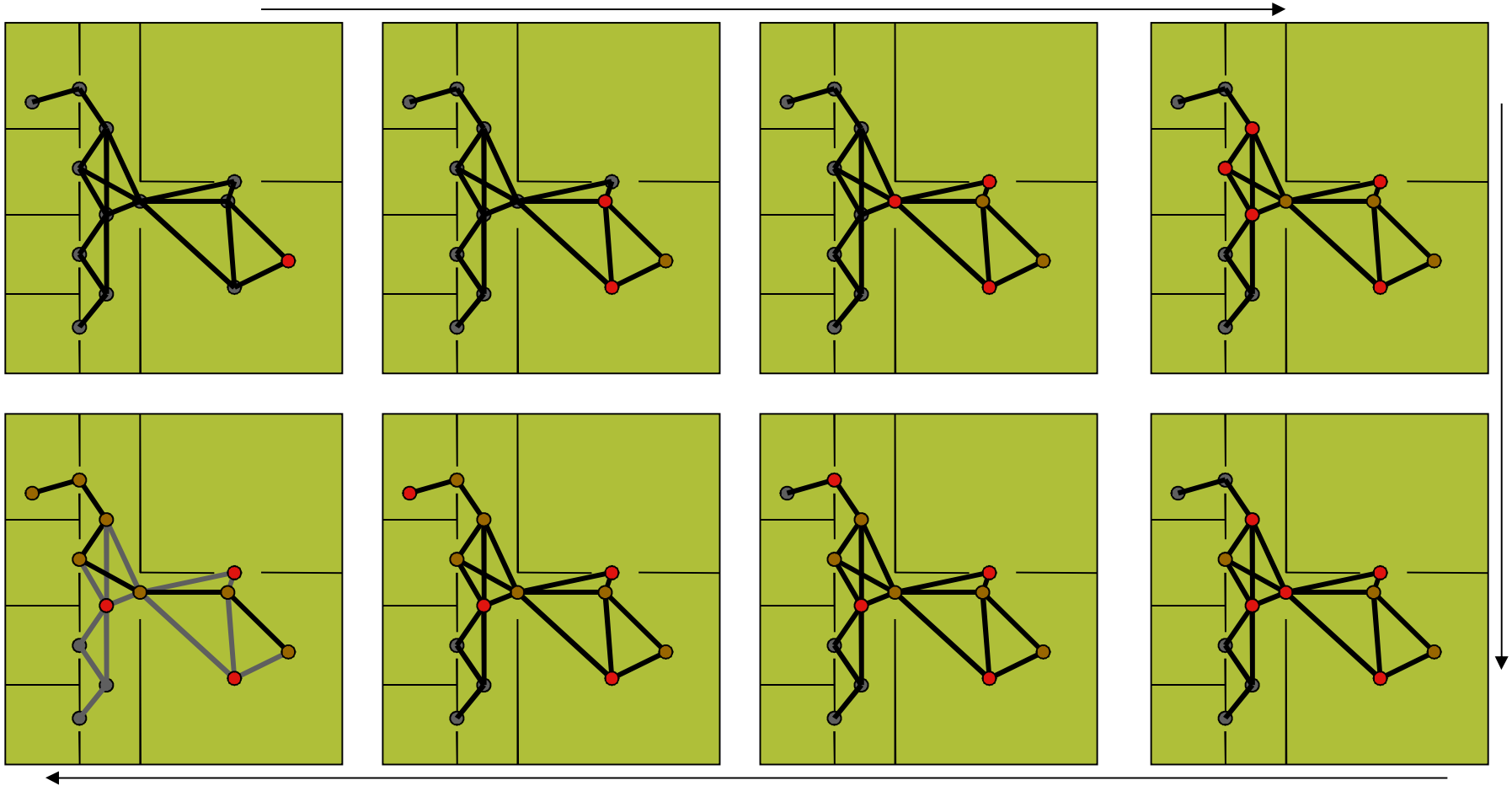
- Define "best" according to *f(n)=h(n)*, the best guess from the node to the goal state
  - Behavior depends on choice of heuristic
  - Straight line distance is a good one
- Is it optimal?
  - When the goal node is expanded, is it along the shortest path?
- Is it efficient?
  - How many nodes does it explore? Many, few, …?

Greedy Search





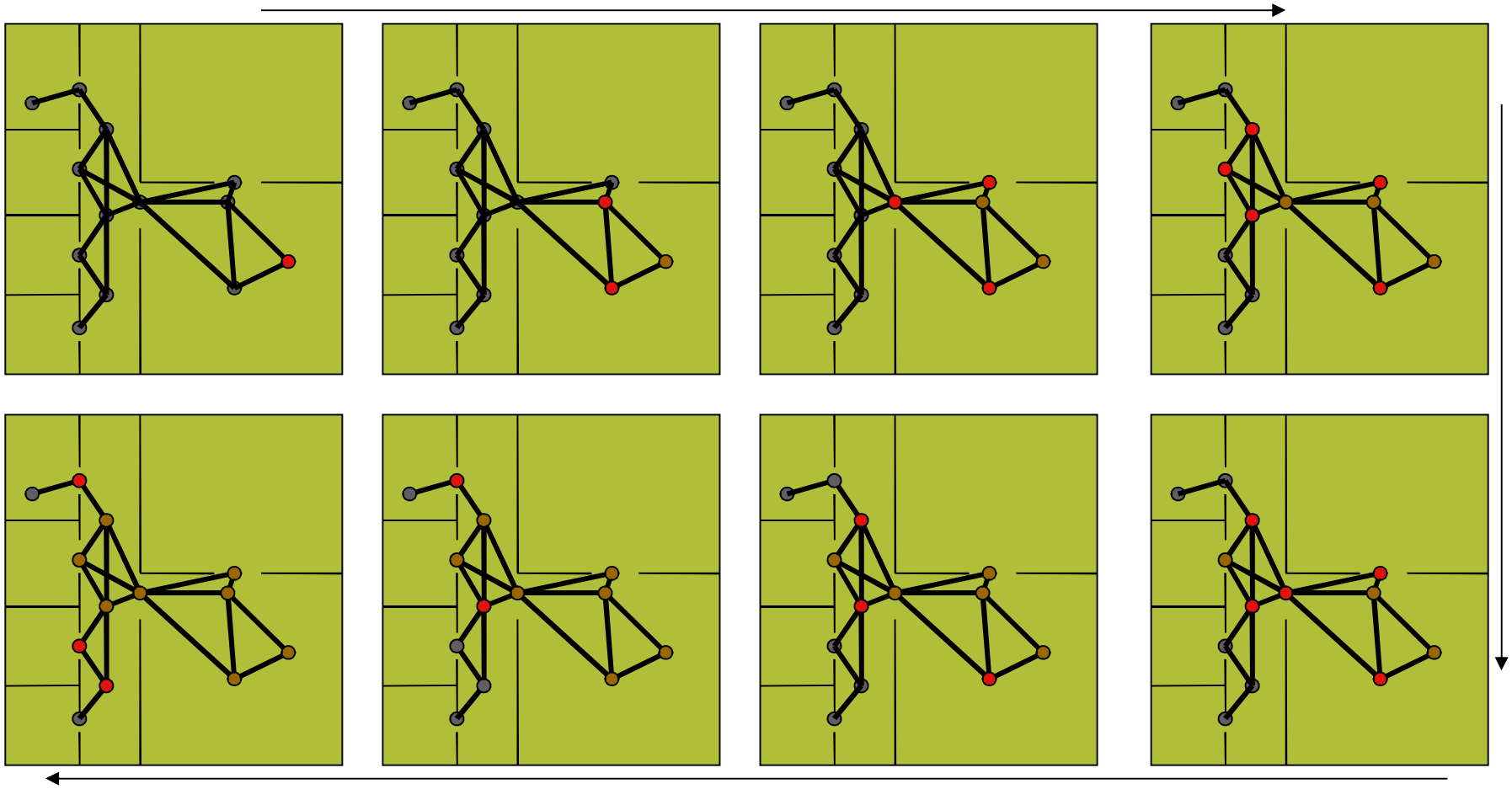
Greedy Search(Straight-Line-Distance Heuristic)



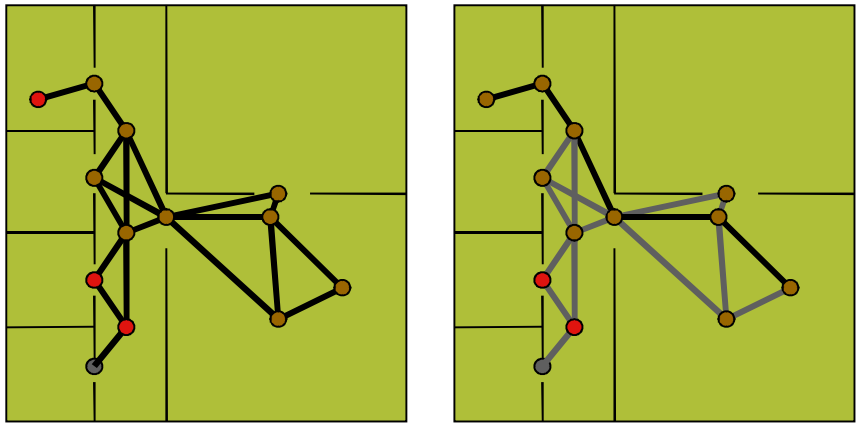
A* Search

- Set $f(n)=g(n)+h(n)$
  - Now we are expanding nodes according to best estimated total path cost
- Is it optimal?
  - It depends on $h(n)$
- Is it efficient?
  - It is the most efficient of any optimal algorithm that uses the same $h(n)$
- A* is the ubiquitous algorithm for path planning in games
  - Much effort goes into making it fast, and making it produce pretty looking paths
  - More articles on it than you can ever hope to read

A* Search (Straight-Line-Distance Heuristic)



A* Search (Straight-Line-Distance Heuristic)



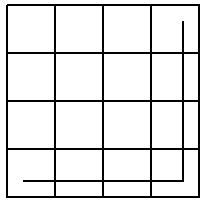
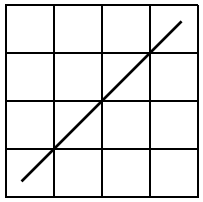
- Note that A* expands fewer nodes than breadth-first, but more than greedy
- It's the price you pay for optimality
- Keys are:
  - Data structure for a node
  - Priority queue for sorting open nodes
  - Underlying graph structure for finding neighbors

Heuristics

- For A* to be optimal, the heuristic must *underestimate* the true cost
  - Such a heuristic is *admissible*
- Also, the *f(n)* function must monotonically increase along any path out of the start node
  - True for almost any admissible heuristic, related to triangle inequality
  - If not true, can fix by making cost through a node max(*f(parent)* + edge, *f(n)*)
- Combining heuristics:
  - If you have more than one heuristic, all of which underestimate, but which give different estimates, can combine with: *h(n)*=max(*h1(n),h2(n),h3(n)*,...)

Inventing Heuristics

- Bigger estimates are always better than smaller ones
  - They are closer to the “true” value
  - So straight line distance is better than a small constant
- Important case: Motion on a grid
  - If diagonal steps are not allowed, use *Manhattan distance*


is a bigger estimate than


- General strategy: Relax the constraints on the problem
  - For example: Normal path planning says avoid obstacles
  - Relax by assuming you can go through obstacles
  - Result is straight line distance

Non-Optimal A*

- Can use heuristics that are not admissible – A* will still give an answer
  - But it won't be optimal: May not explore a node on the optimal path because its estimated cost is too high
  - Optimal A* will eventually explore any such node before it reaches the goal
- Non-admissible heuristics may be much faster
  - Trade-off computational efficiency for path-efficiency
- One way to make non-admissible: Multiply underestimate by a constant factor

Hierarchical Planning

- Many planning problems can be thought of hierarchically
  - To pass this class, I have to pass the exams and do the projects
  - To pass the exams, I need to go to class, review the material, and show up at the exam
  - To go to class, I need to go to classroom at 10:00am TuTh
- 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

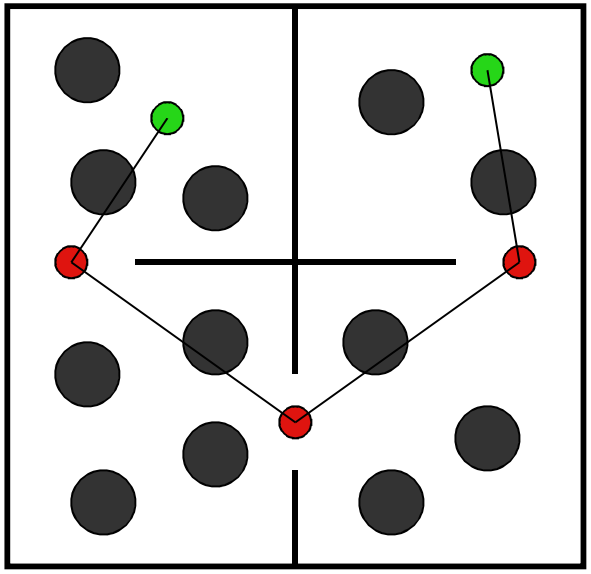
Doing Hierarchical Planning

- Define a waypoint graph for the top of the hierarchy
  - For instance, a graph with waypoints in doorways (the centers)
  - Nodes linked if there *exists* a clear path between them (not necessarily straight)

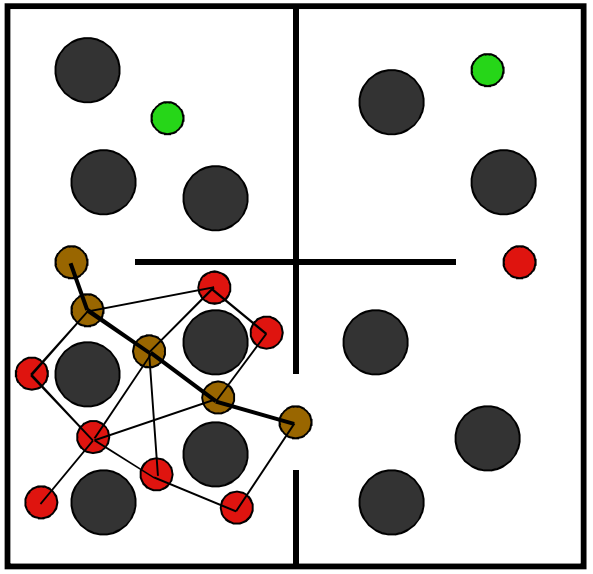
Doing Hierarchical Planning

- For each *edge* in that graph, define another waypoint graph
  - This will tell you how to get between each doorway in a single room
  - Nodes from top level should be in this graph
- First plan on the top level – result is a list of rooms to traverse
- Then, for each room on the list, plan a path across it
  - Can delay low level planning until required – smoothes out frame time

Hierarchical Planning Example



Plan this first



Then plan each room
(second room shown)

Hierarchical Planning Advantages

- The search is typically cheaper
  - The initial search restricts the number of nodes considered in the latter searches
- It is well suited to partial planning
  - Only plan each piece of path when it is actually required
  - Averages out cost of path over time, helping to avoid long lag when the movement command is issued
  - Makes the path more adaptable to dynamic changes in the environment

Hierarchical Planning Issues

- Result is 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
  - Have to restrict the number of nodes at the top level for efficiency
  - So cannot include all the options that would be available to a full planner

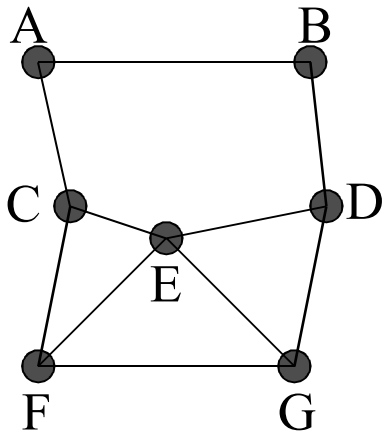
Pre-Planning

- If the set of waypoints is fixed, and the obstacles don't move, then 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

Storing All-Pairs Paths

- Trivial solution is to store the shortest path to every other node in every node: $O(n^3)$ memory
- A better way:
  - Say I have the shortest path from A to B: A-B
  - *Every* shortest path that goes through A on the way to B must use A-B
  - So, if I have reached A, and want to go to B, I *always* take the same next step
  - 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
  - But a path is just a sequence of steps – if I keep following the "next step" I will eventually get to B
  - 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

Dynamic Path Planning

- What happens when the environment changes *after* the plan has been made?
  - The player does something
  - Other agents get in the way (in this case, you know that the environment will change at the time you make the plan)
- The solution strategies are highly dependent on the nature of the game, the environment, the types of AI, and so on
- Three approaches:
  - Try to avoid the problem
  - Re-plan when something goes wrong
  - Reactive planning

Avoiding Plan Changes

- Partial planning: Only plan short segments of path at a time
  - Stop A* after a path of some length is found, even if the goal is not reached – use best estimated path found so far
    - Extreme case: Use greedy search and only plan one step at a time
    - Common case: Hierarchical planning and only plan low level when needed
  - Underlying idea is that a short path is less likely to change than a long path
  - But, optimality will be sacrificed
  - Another advantage is more even frame times
- Other strategies:
  - Wait for the blockage to pass – if you have reason to believe it will
  - Lock the path to other agents – but implies priorities

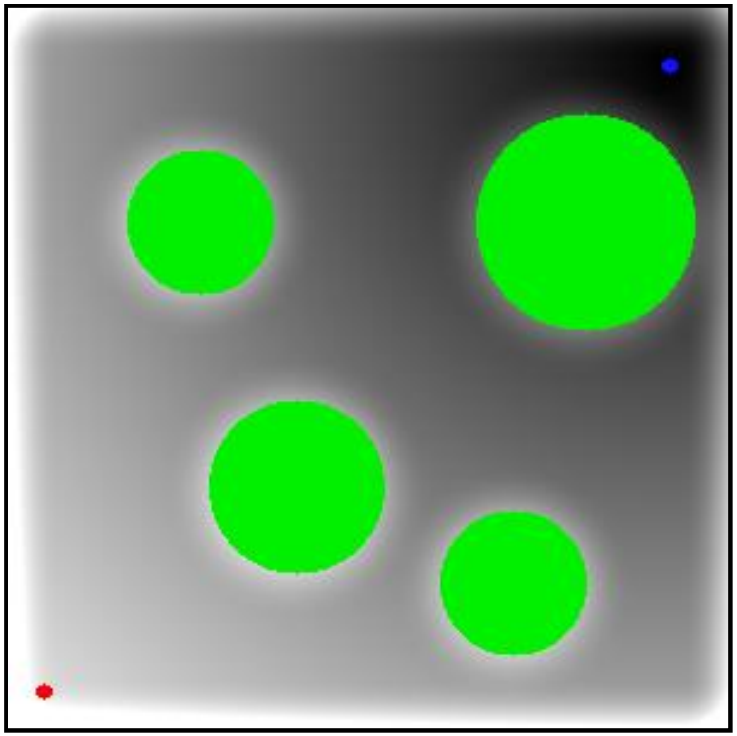
Re-Planning

- If you discover the plan has gone wrong, create a new one
- The new plan assumes that the dynamic changes are permanent
- Usually used in conjunction with one of the avoidance strategies
  - Re-planning is expensive, so try to avoid having to do it
  - No point in generating a plan that will be re-done – suggests partial planning in conjunction with re-planning

Reactive Planning

- A *reactive* agent plans only its next step, and only uses immediately available information
- Best example for path planning is *potential field* planning
  - Set up a force field around obstacles (and other agents)
  - Set up a gradient field toward the goal
  - The agent follows the gradient downhill to the goal, while the force field pushes it away from obstacles
  - Can also model velocity and momentum – field applies a *force*
- Potential field planning is reactive because the agent just looks at the local gradient at any instant
- Has been used in real robots for navigating things like hallways

Potential Field



- Red is start point, blue is goal
- This used a quadratic field strength around the obstacles
- Note that the boundaries of the world also contribute to the field

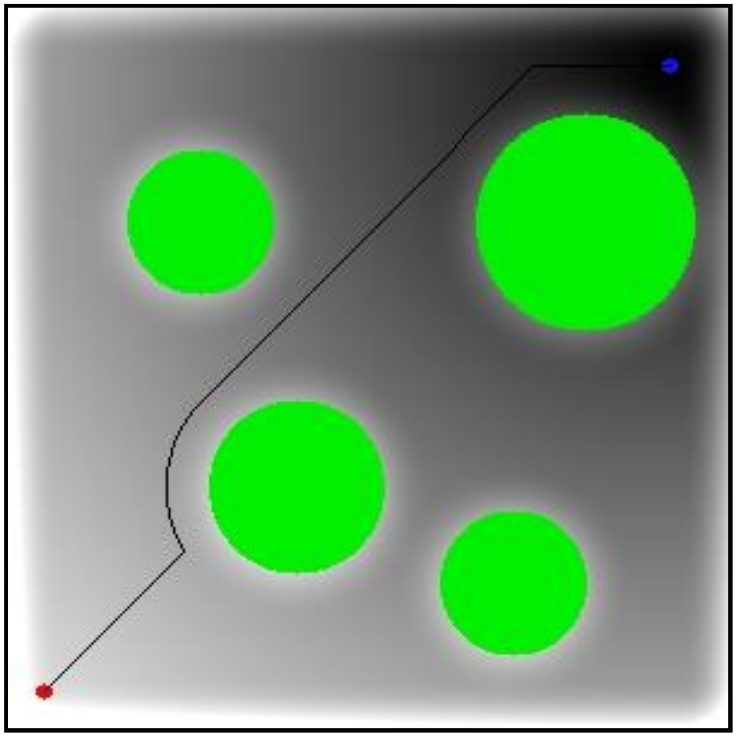
Creating the Field

- The constant gradient can be a simple linear gradient based on distance from the goal, d_{goal}: $f_{goal}=k\, d_{goal}$
- The obstacles contribute a field strength based on the distance from their boundary, $f_i(d_i)$
  - Linear, quadratic, exponential, something else
  - Normally truncate so that field at some distance is zero. Why?
  - Strength determines how likely the agent is to avoid it
- Add all the sub-fields together to get overall field

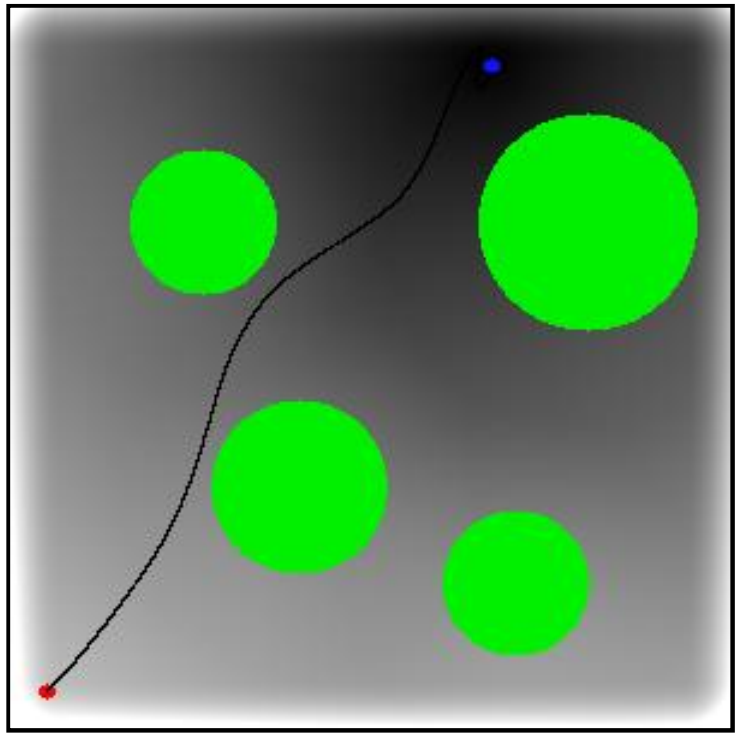
Following the Field

- At each step, the agent needs to know which direction is “downhill”
- Compute the gradient of the field
  - Compute the gradients of each component and add
  - Need partial derivatives in *x* and *y* (for 2D planning)
- Best approach is to consider the gradient as a force (acceleration)
  - Automatically avoids sharp turns and provides smooth motion
  - Higher mass can make large objects turn more slowly
  - Easy to make frame-rate independent
  - But, high velocities can cause collisions because field is not strong enough to turn the object in time
  - One solution is to limit velocity – want to do this anyway because the field is only a guide, not a true force

Following Examples



No momentum - choose to go to neighbor with lowest field strength



Momentum - but with linear obstacle field strength and moved goal

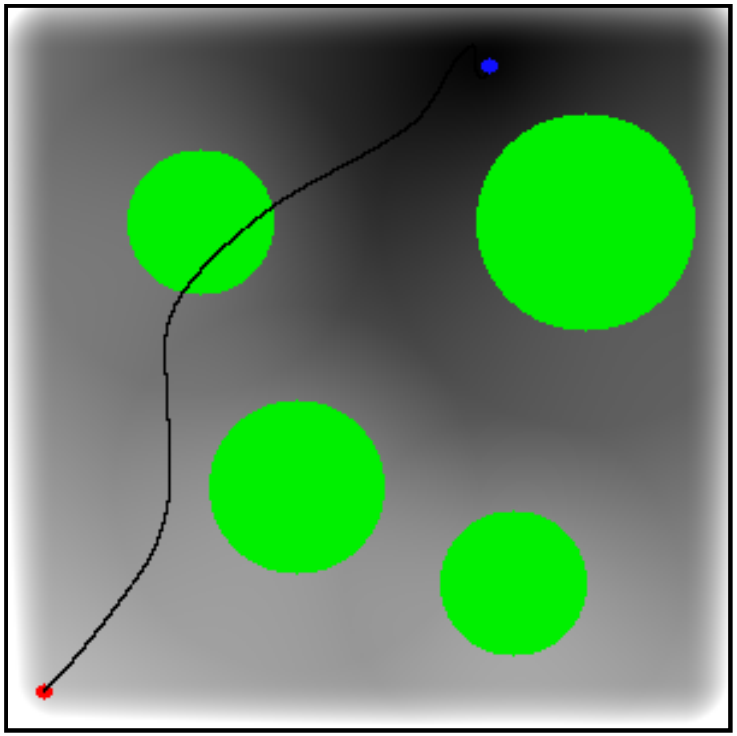
Discrete Approximation

- Compute the field on a grid
  - Allows pre-computation of fields that do not change, such as fixed obstacles
  - Moving obstacles handled as before
- Use discrete gradients
  - Look at neighboring cells
  - Go to neighboring cell with lowest field value
- Advantages: Faster
- Disadvantages: Space cost, approximate
- Left example on previous slide is a (very fine) discrete case

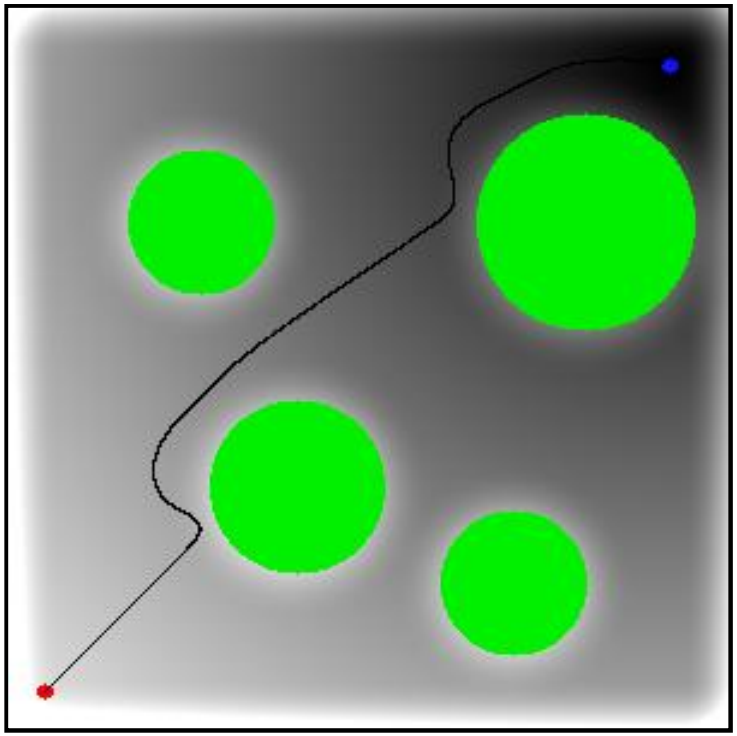
Potential Field Problems

- There are many parameters to tune
  - Strength of the field around each obstacle
  - Function for field strength around obstacle
  - Steepness of force toward the goal
  - Maximum velocity and mass
- Goals conflict
  - High field strength avoids collisions, but produces big forces and hence unnatural motion
  - Higher mass smoothes paths, but increases likelihood of collisions
- Local minima cause huge problems

Bloopers

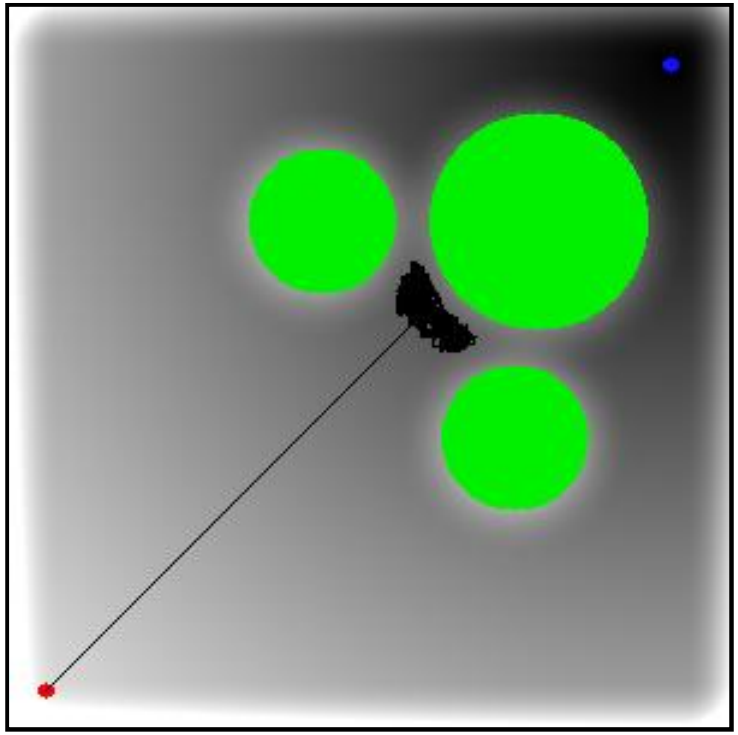


Field too weak



Field too strong

Local Minima Example



The Local Minima Problem

- Potential field planning is gradient descent optimization
- The biggest problem with gradient descent is that it gets stuck in local minima – potential field planning too
- Must have a way to work around this
  - Go back if a minima is found, and try another path
  - Virtual obstacles: create at the point where the search get stuck
  - Virtual sub-goals: backtrack along the path and attract it in a new direction