CS3027/CS5059: Planning and Navigation

N. Oren
n.oren@abdn.ac.uk

University of Aberdeen

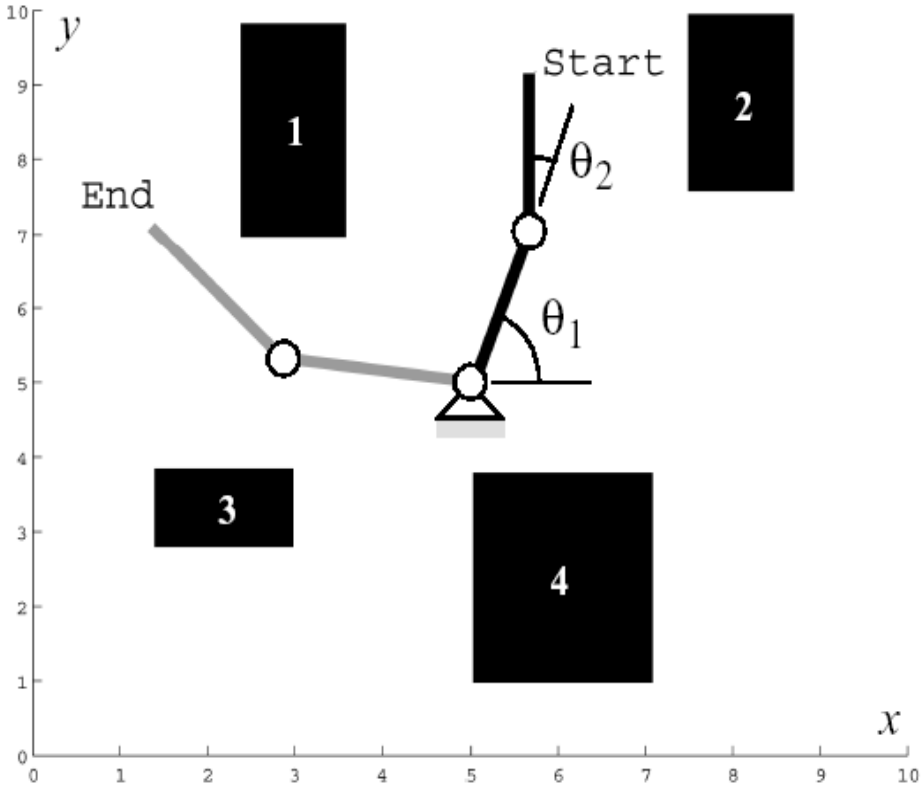
The Problem

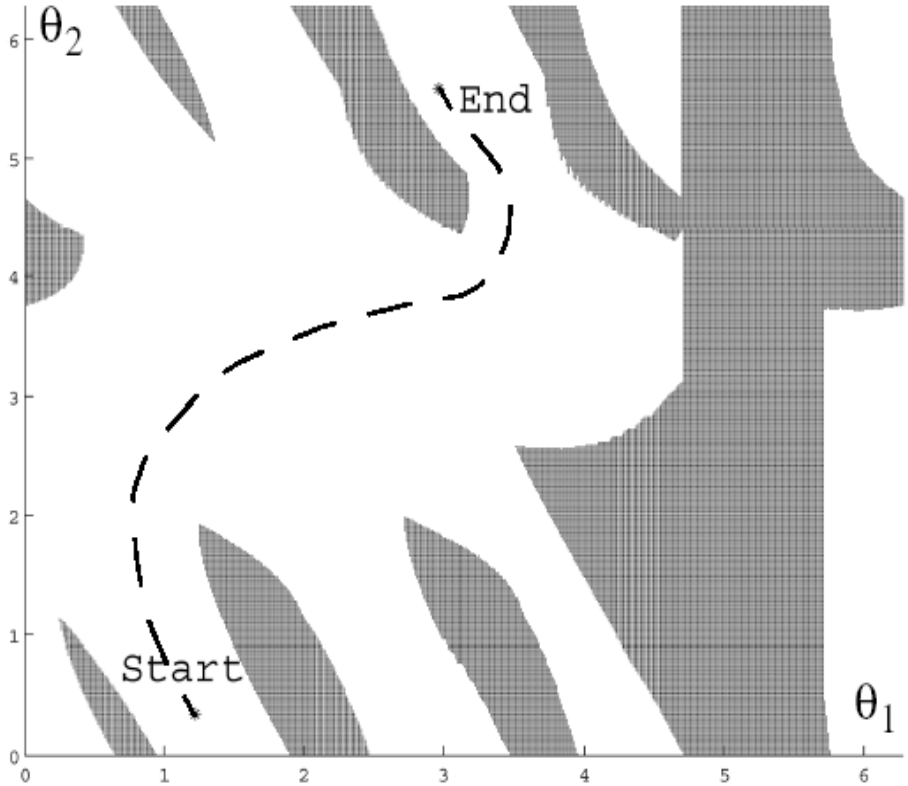
- How can one find a path in the work space from an initial position to a goal position avoiding collisions with obstacles?
- We assume access to a sufficiently accurate map for navigation.
- We distinguish between path planning (getting between points on the global map) and obstacle avoidance (which involves getting around local "small" inaccessible objects).
- Steps
  - Transform the map into a representation useful for planning.
  - Plan a path on the transformed map
  - Send motion commands to the controller

Configuration Space

- The robot operates in a workspace (the physical space) with known obstacles.
- Our aim is to find a path through this space from the initial position to the goal position avoiding collisions.
- Difficult to visualise in a high degree of freedom situation (e.g. a robot arm)
- Path planning is done within a configuration space — a space with dimensionality equal to the degrees of freedom of the robot.
- We map from the workspace to the configuration space, identifying a configuration space obstacle as the subspace of the configuration space wherein the robot would encounter an obstacle.
- The free space is the remaining part of the configuration space, wherein the robot can move freely.

Configuration Space





Configuration Space

- In mobile robots, we typically make the assumption that the robots are holonomic.
- We also assume that the robot is simply a point.
- So we can consider only x, y coordinates — identical to workspace.
- Except that obstacles must be inflated to compensate for robot size.

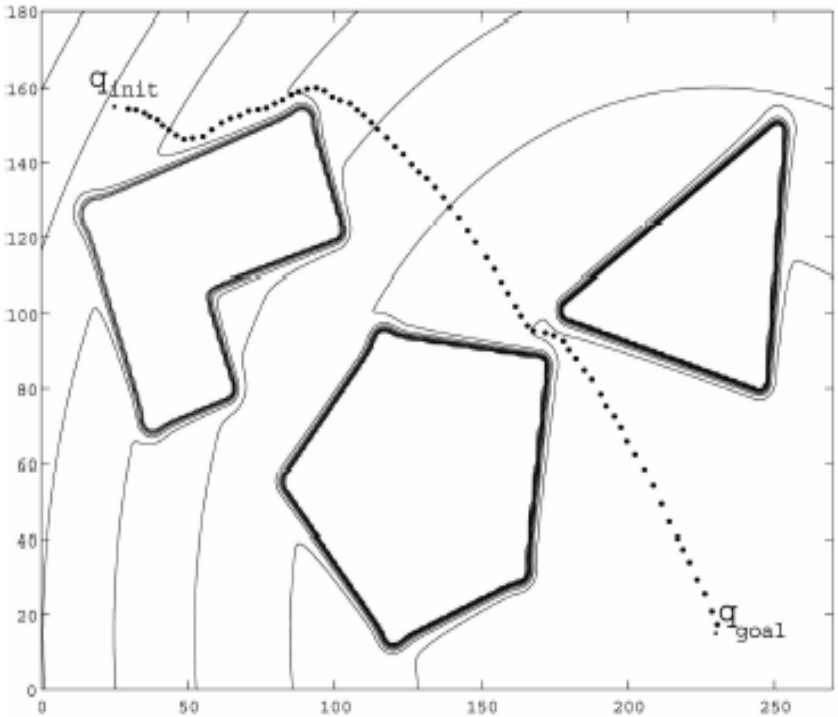
Path Planning

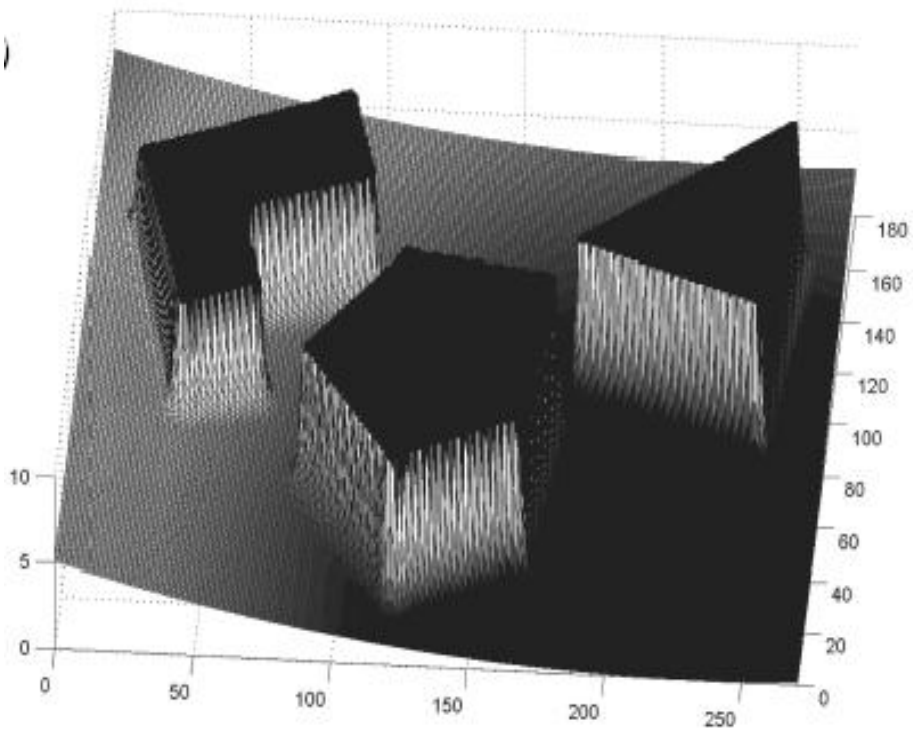
- We investigate two broad categories for path planning
  - Potential field planning (robot "rolls" from source to destination)
  - Graph search (over a connectivity graph)

Potential Field Path Planning

- Robot is treated as a point under the influence of an artificial potential field.
- Operates over continuous space.
- Robot movement is similar to that of a ball rolling on a downhill.
- Goal is attractive.
- Obstacles are repulsive.

Potential Field Path Planning





Potential Field Path Planning

- Our aim is to generate a potential field $U(q)$ where $q = (x, y)$.
- $U(q)$ is made up of attractive potentials (due to goal) and repulsive potentials (due to obstacles).

$$U(q) = U_{att}(q) + U_{rep}(q)$$

- Given a differentiable $U(q)$ we can determine a force $F(q) = -\nabla U(q)$ on the agent.

$$F_q = -\nabla U(q) = -\nabla U_{att}(q) - \nabla U_{rep}(q) = [\frac{\partial U}{\partial x}, \frac{\partial U}{\partial y}]^T$$

- Robot speed is set proportional to the force at the point.
- We therefore have a plan (saying how the robot the direction of movement at each point) and a control scheme (stating the velocity that should be used to move).

Attractive Potential Fields

- A parabolic function can be used to specify the attractive potential field, e.g.

$$U_{att}(q) = 0.5 k_{att} \rho^2_{goal}(q)$$

- $\rho_{goal}(q)$ is the distance to the goal $||q - q_{goal}||$

$$\begin{aligned} F_{att}(q) &= \nabla U_{att}(q) \\ &= -k_{att} \rho_{goal}(q) \nabla \rho_{goal}(q) \\ &= -k_{att}(q - q_{goal}) \end{aligned}$$

- $lim_{q \to q_{goal}} F = 0$

Repulsive Potential Fields

- The aim is to generate a barrier around the obstacle.
- Field should be strong near obstacles
- Weak (or non-existent) far from obstacles

$$U_{rep}(q) = \begin{cases} 0.5 k_{rep} \left(\frac{1}{\rho(q)} - \frac{1}{\rho_0}\right)^2 & \text{if } \rho_q \leq \rho_0 \\ 0 & \text{otherwise} \end{cases}$$

- $\rho(q)$ is the minimum distance of influence.
- $U_{rep} \geq 0$ and tends to infinity as q gets closer to the object.

Repulsive Potential Fields

$$U_{rep}(q) = \begin{cases} 0.5 k_{rep}(\frac{1}{\rho(q)} - \frac{1}{\rho_0})^2 & \text{if } \rho_q \leq \rho_0 \\ 0 & \text{otherwise} \end{cases}$$

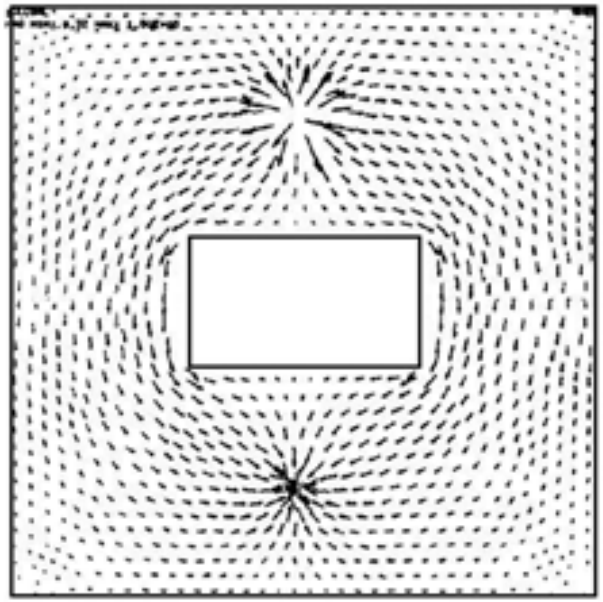
- If $\rho(q) \leq \rho_0$

$$F_{rep}(q) = -\nabla U_{rep}(q) = k_{rep}(\frac{1}{\rho(q)} - \frac{1}{\rho_0})\frac{1}{\rho^2(q)}\frac{q - q_{obst}}{\rho(q)}$$

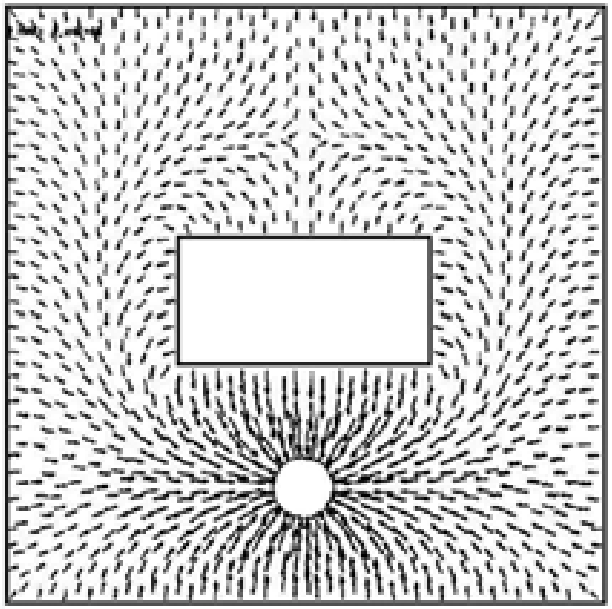
Potential Field Planning Problems

- Local minima can appear depending on obstacle shape and size.
- Concave objects create several minimal distances resulting in "bouncing" between objects.
- `https://www.youtube.com/watch?v=ka7Yb_XELAU`

More Potential Field Planning



Neumann



Dirichlet

- Analogies to fluid particles or electrostatic particles eliminates local minima.
- Equipotential lines orthogonal to object boundaries yield short paths close to objects.
- Equipotential lines parallel to object boundaries yield long (but safe) paths.

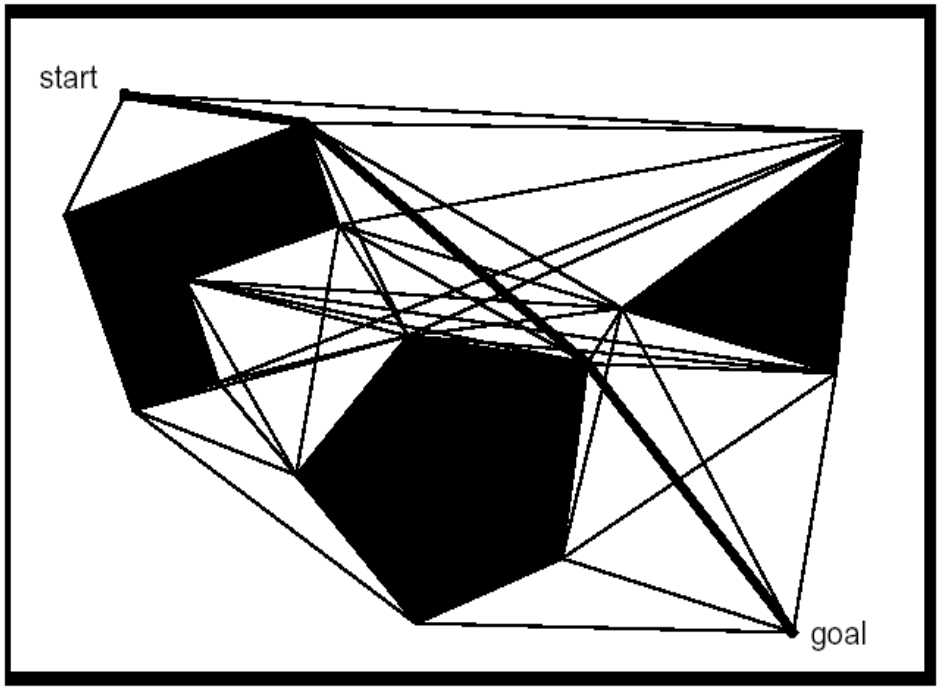
Graph Search

- By discretising the space we can seek a path between discrete states (with least cost).
- Discrete space is encoded in a graph.
- Such discretization can endanger completeness (i.e. a solution could exist which may not be found).
- Two main steps
  - Graph construction
  - Graph searching

Graph Construction

- The aim of graph construction is to construct nodes and edges that enable robot to reach any point in its free space.
- While limiting total size of the graph.
- Different construction approaches can yield vastly different graphs.

Visibility Graph

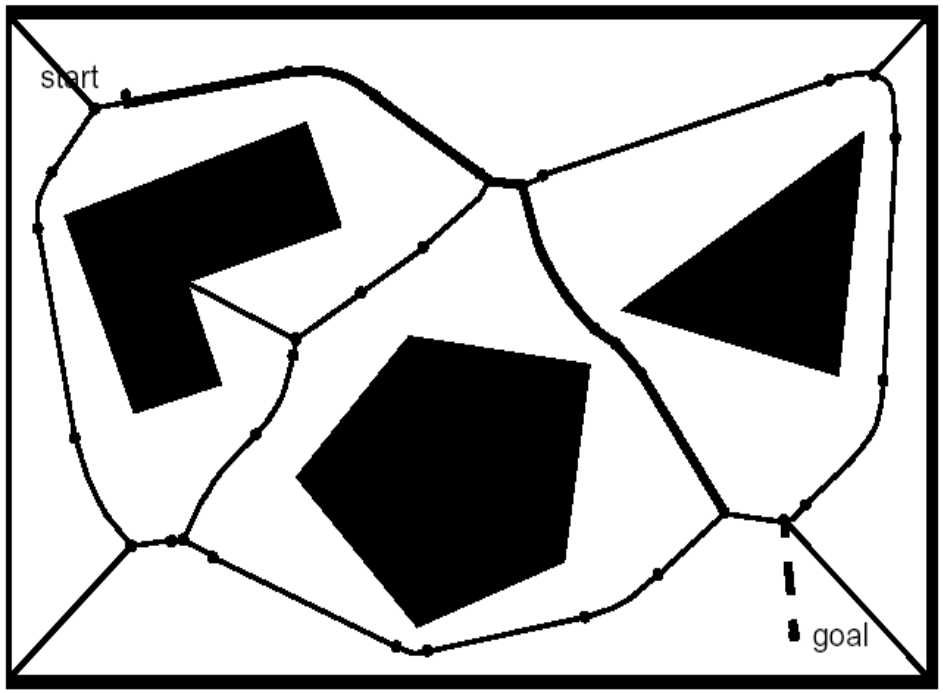


- Consists of edges joining all pairs of vertices that can see each other (including initial and goal positions).
- Edges (straight lines) are shortest paths joining the vertices.
- Path planner aims to find a shortest path from the initial to the goal position along roads defined by the visibility graph.

Visibility Graph

- Very simple implementation.
- Very suitable for polygons
- Number of edges and nodes increases with the number of obstacle polygons.
- Very efficient in sparse environments, poor in dense environments.
- Solutions take robot as close as possible to obstacles on the way to the goal — safety sacrificed for optimality.
- For safety, obstacles must typically be grown by a lot more, potentially sacrificing completeness (and optimality).

Voronoi Diagrams



- Maximises distance between the robot and obstacles.
- For each point in free space, identify distance to the nearest obstacle.
- Leads to ridges of maximal distance between obstacles, representing the edges.

Voronoi Diagrams

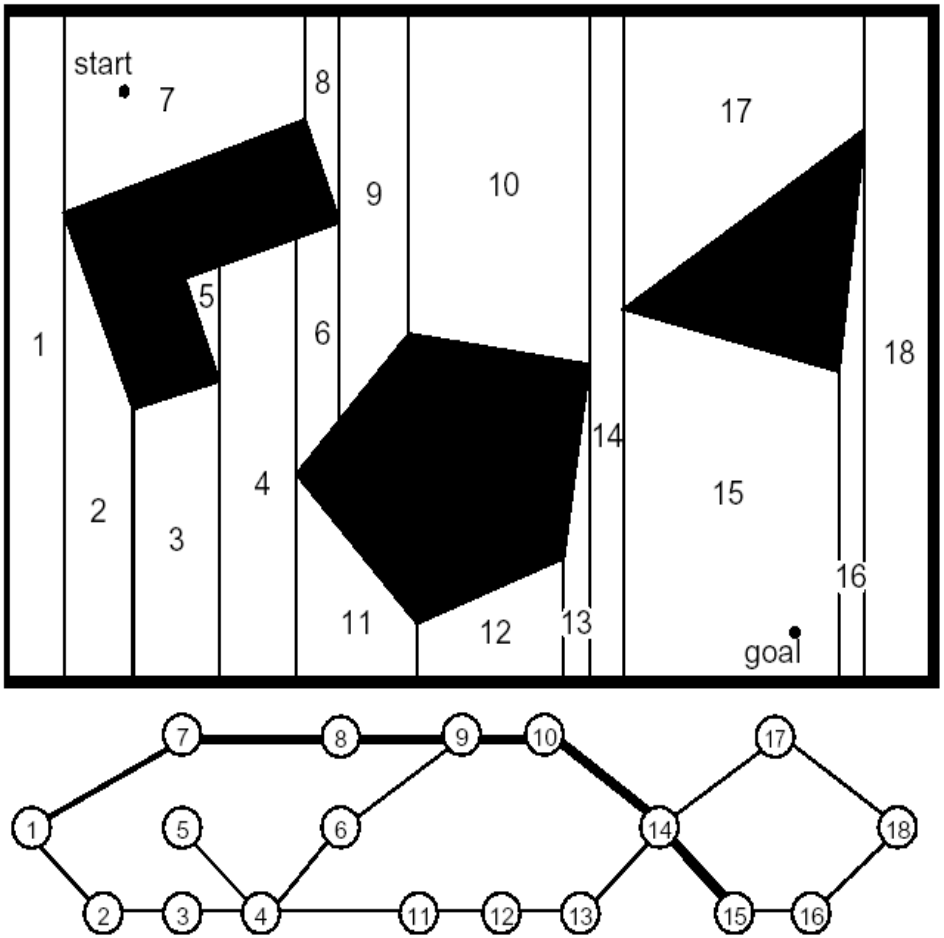
- Simple control rules can be used to keep robot along the Voronoi diagram - mitigation of encoder uncertainty.
- Automatic environment map: find and move along unknown Voronoi edges, construct a map of the environment.
- Path finding on a Voronoi diagram is complete — existence of a path in free space implies the existence of one on the Voronoi diagram.
- Resultant paths are long — robot stays as far away from obstacles as possible.
- Short range sensors may fail.

Cell Decomposition

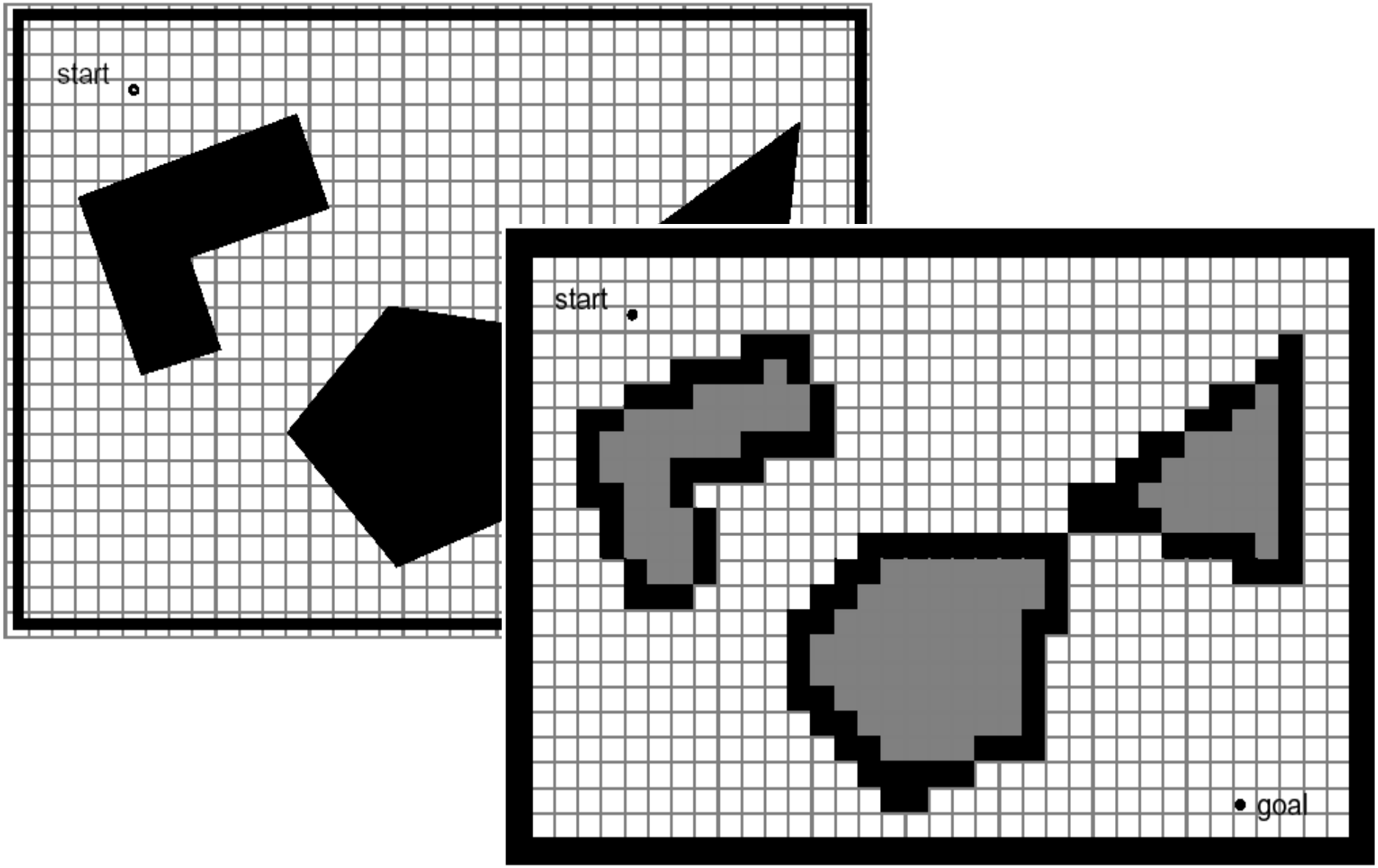
- Cell decomposition techniques split the space into simple connected regions (called cells).
- A connectivity graph between open cells is then generated.
- Motion occurs along this graph.
- Essentially a topological map of the environment.

Exact Cell Decomposition

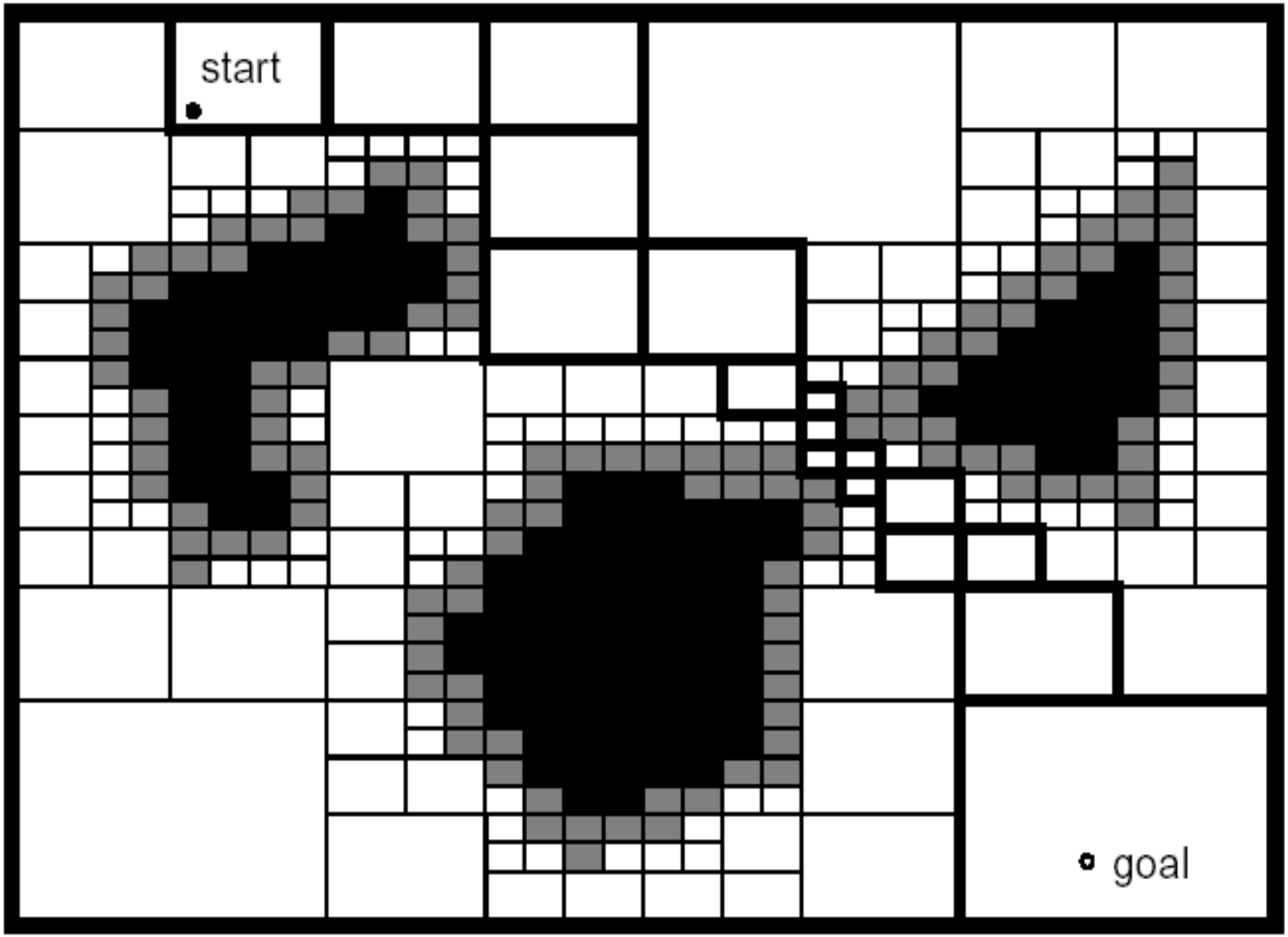
- Cells are either completely free or occupied.
- Ability to traverse between cells is important rather than exact location in cell.
- Number of cells depends on complexity o objects within the environment.
- Very efficient in sparse environments.
- Getting robots to actually move between cells is difficult — not commonly used in robotics.



Approximate Cell Decomposition

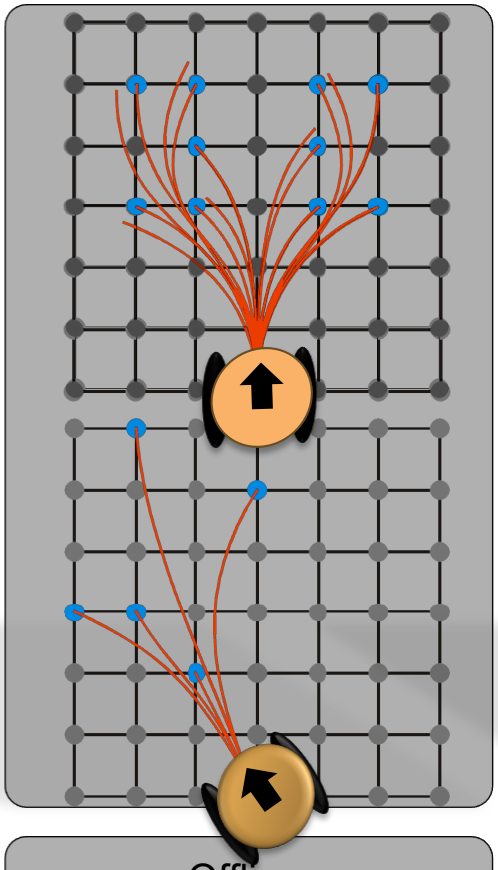


Adaptive Cell Decomposition

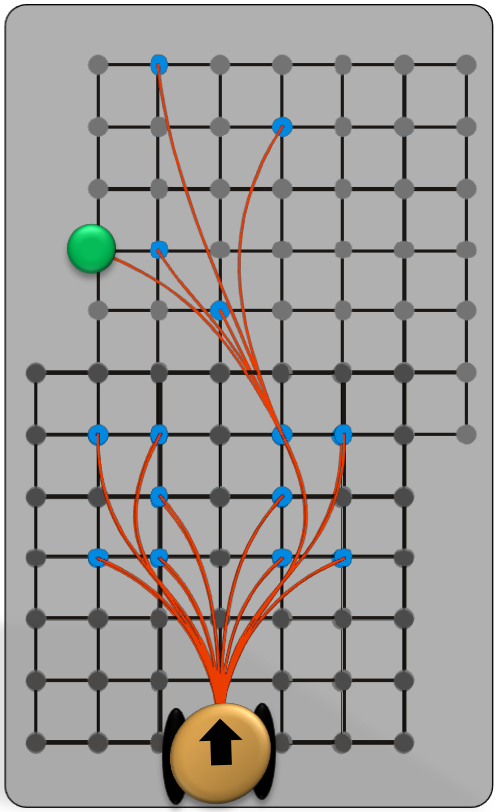


Lattice Cell Decomposition

Offline:
Motion Model



Offline:
Lattice Gen.



Online:
Incremental Graph Constr.

Graph Search

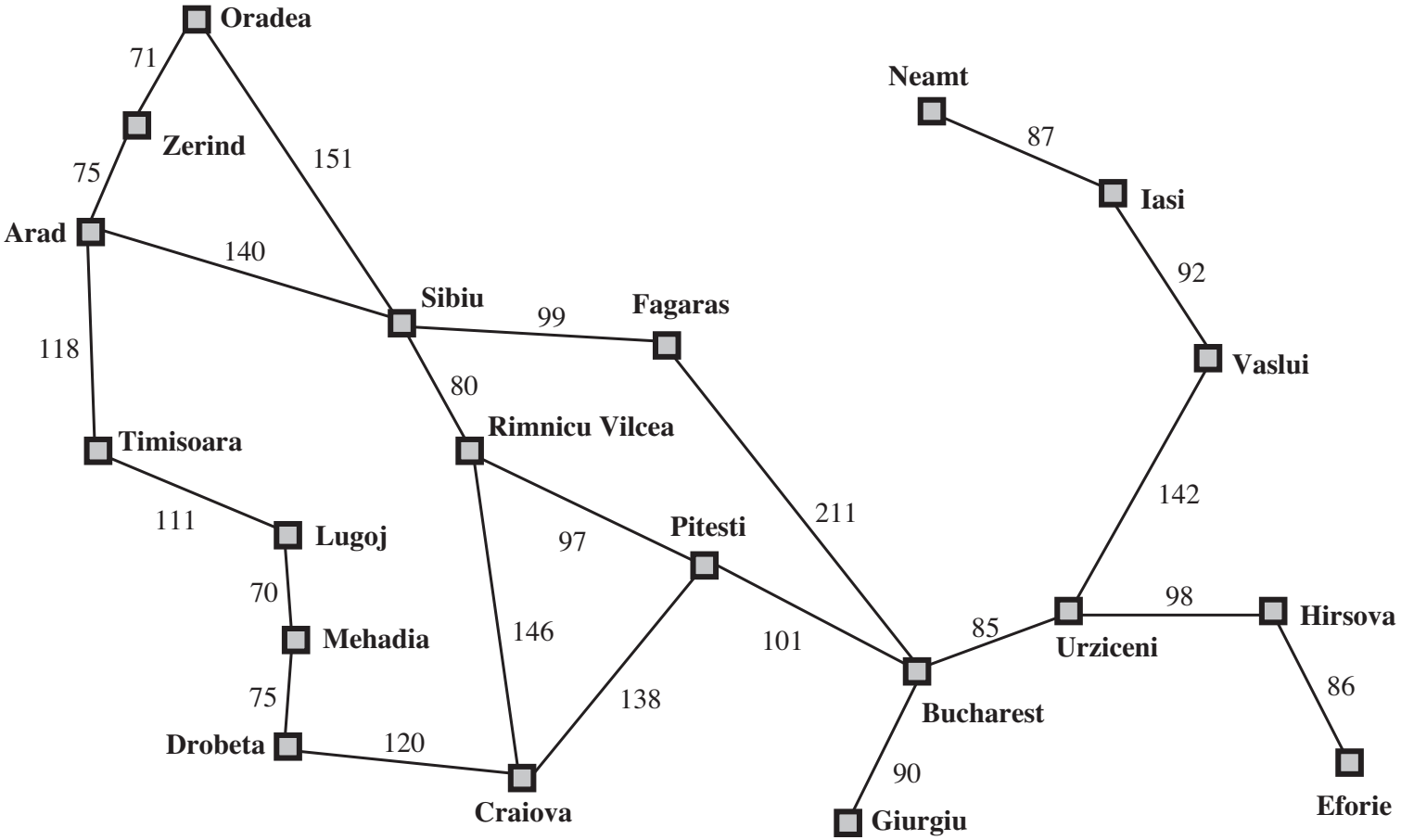
- Given a connectivity graph, we seek to find the best path between the start and the goal.
- Best is not necessarily shortest, but we must have some criterion to define it.
- Algorithms:
  - Breadth first
  - Depth first
  - Dijkstra
  - A*
  - D*
  - ...

Graph Search

```
1: function TreeSearch(problem, strategy)
2:     initialise search tree using initial state of problem
3:     while true do
4:         if there are no candidates for expansion then return failure
5:         choose a leaf node to expand using strategy
6:         if node contains a goal state then return the solution
7:         else expand the node and add resulting nodes to search tree
8:     end while
9: end function
```

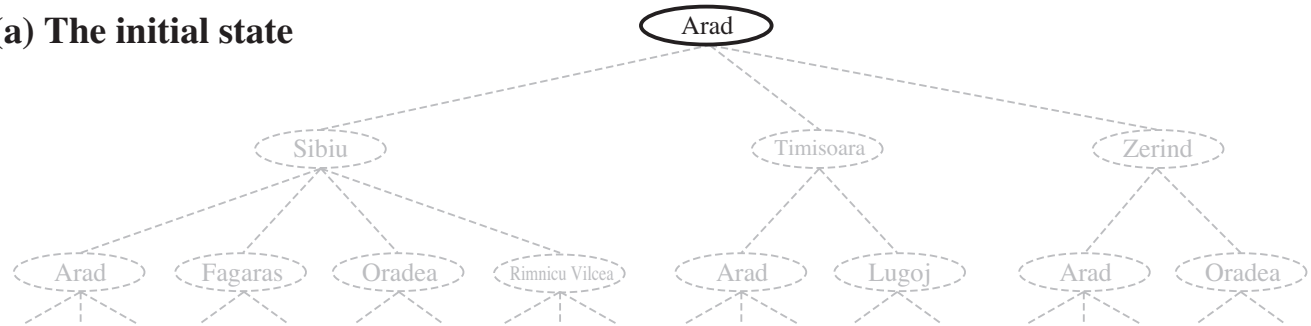
- Expansion takes place by applying successor functions to the selected node, providing new states.
- Nodes that have been generated but not expanded are referred to as the frontier
- Different search strategies behave very differently.

Search Tree

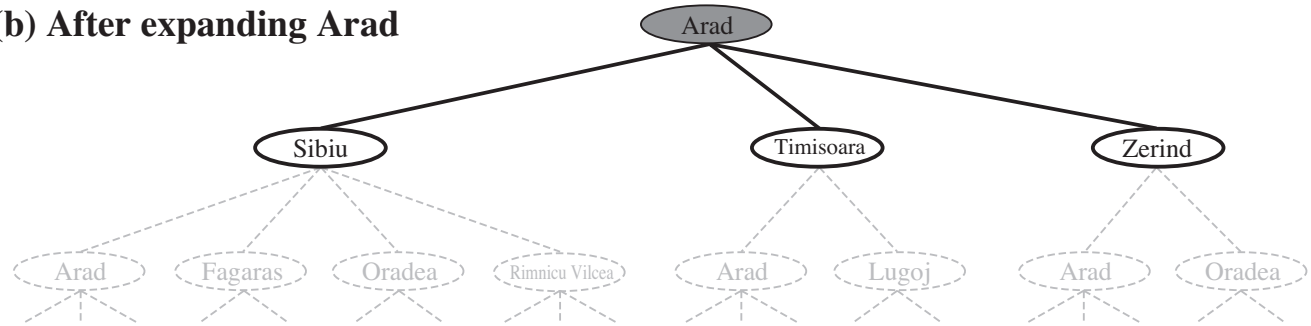

Oradea
71
Neamt
Zerind
87
151
75
Iasi
Arad
140
92
Sibiu
99
Fagaras
118
Vaslui
80
Rimnicu Vilcea
Timisoara
142
111
211
Lugoj
97
Pitesti
70
98
Hirsova
Mehadia
146
101
85
Urziceni
75
86
138
Bucharest
Drobeta
120
90
Craiova
Giurgiu
Eforie

Search Tree

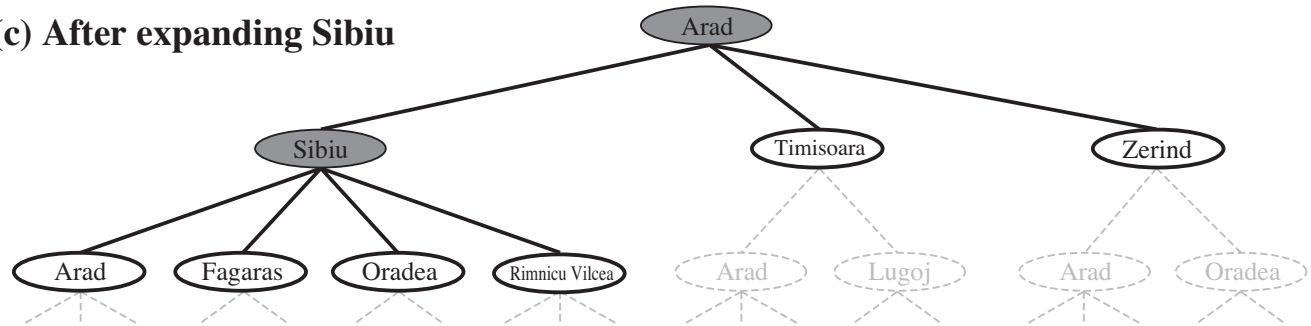
(a) The initial state



(b) After expanding Arad



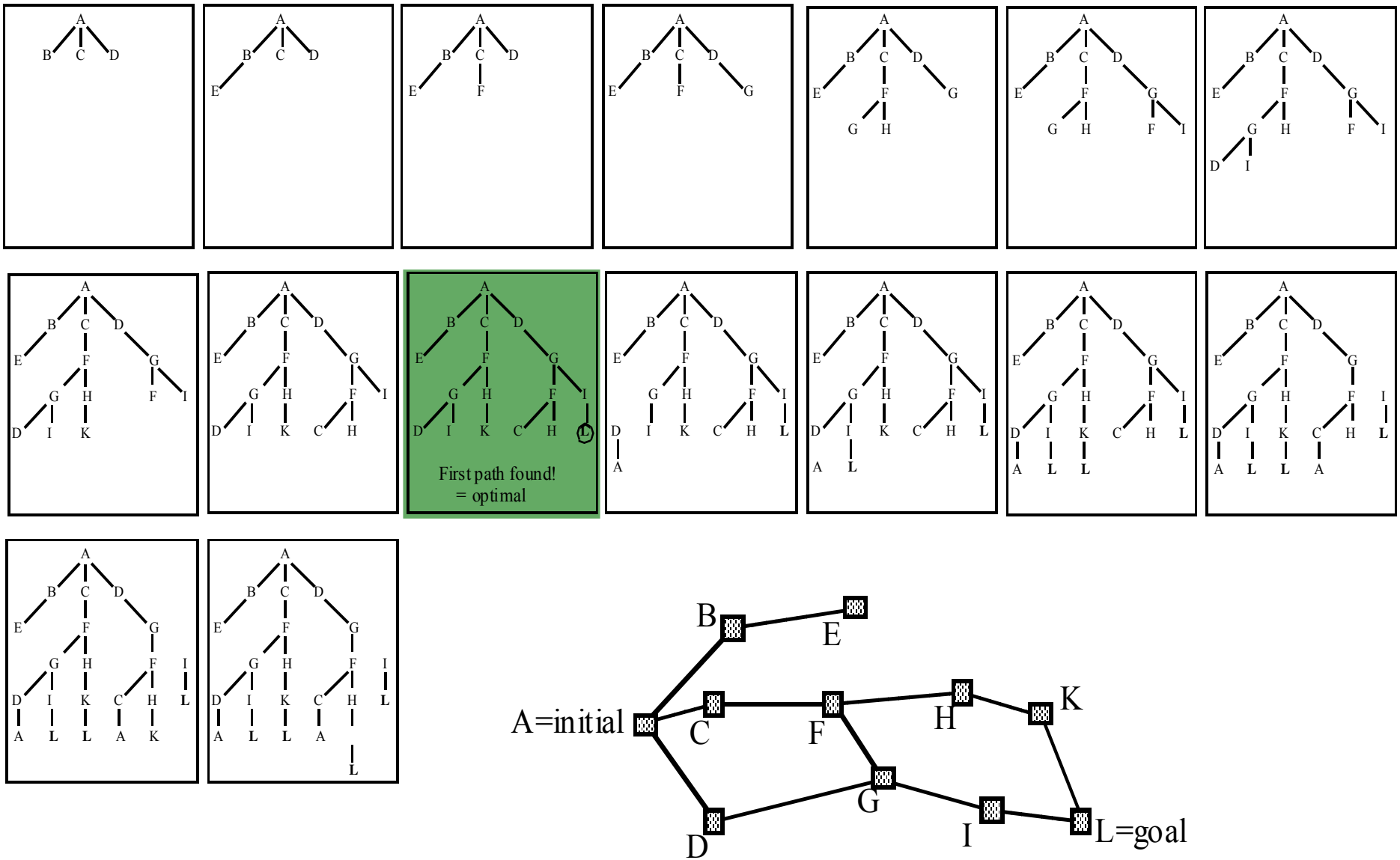
(c) After expanding Sibiu



Breadth First Search

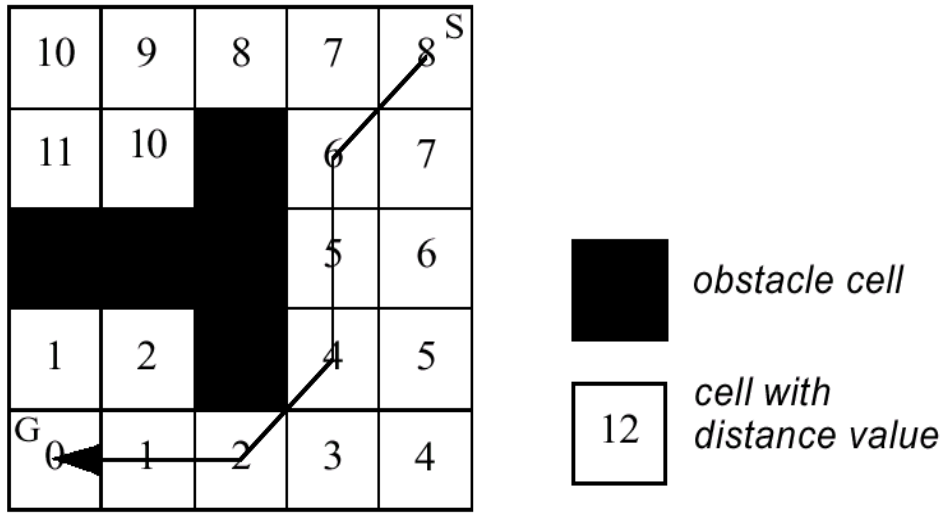
- Algorithm
  1. Begin at some start node.
  2. Add neighbours to tail of list
  3. mark parent node "visited"
  4. Repeat BFS on head of list as long as it is not visited.
- Corresponds to a wavefront expansion on a 2D grid
- Can use a FIFO queue — first found solution is optimal if all edges have equal cost.
- Each cell is visited once — linear in the number of cells.

Breadth First Search



Breadth First Search in Robotics

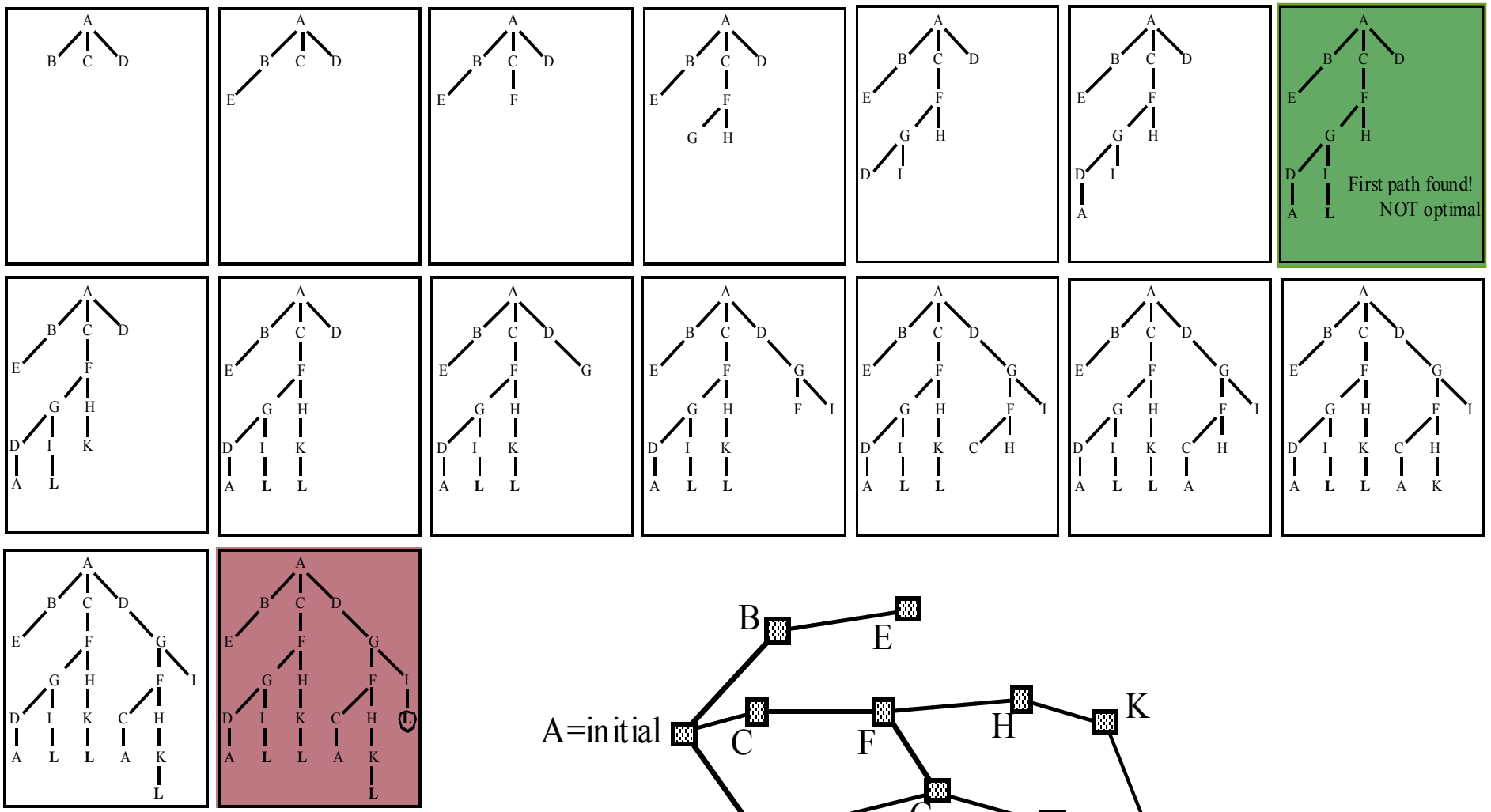
- Wavefront expansion/NF1/grassfire technique for finding routes in fixed size cell arrays.
- Each cell is marked by its Manhattan distance from the goal cell.
- Process continues until the robot's initial position is reached.
- An estimate of the robot's distance to the goal as well as a solution trajectory can be obtained.
- Search is linear in the number of cells.



Depth First Search

- Expand a subtree as far as possible before exploring neighbours.
- Memory efficient as fully explored subtrees can be deleted.
- Can be implemented with a LIFO queue.

Depth First Search

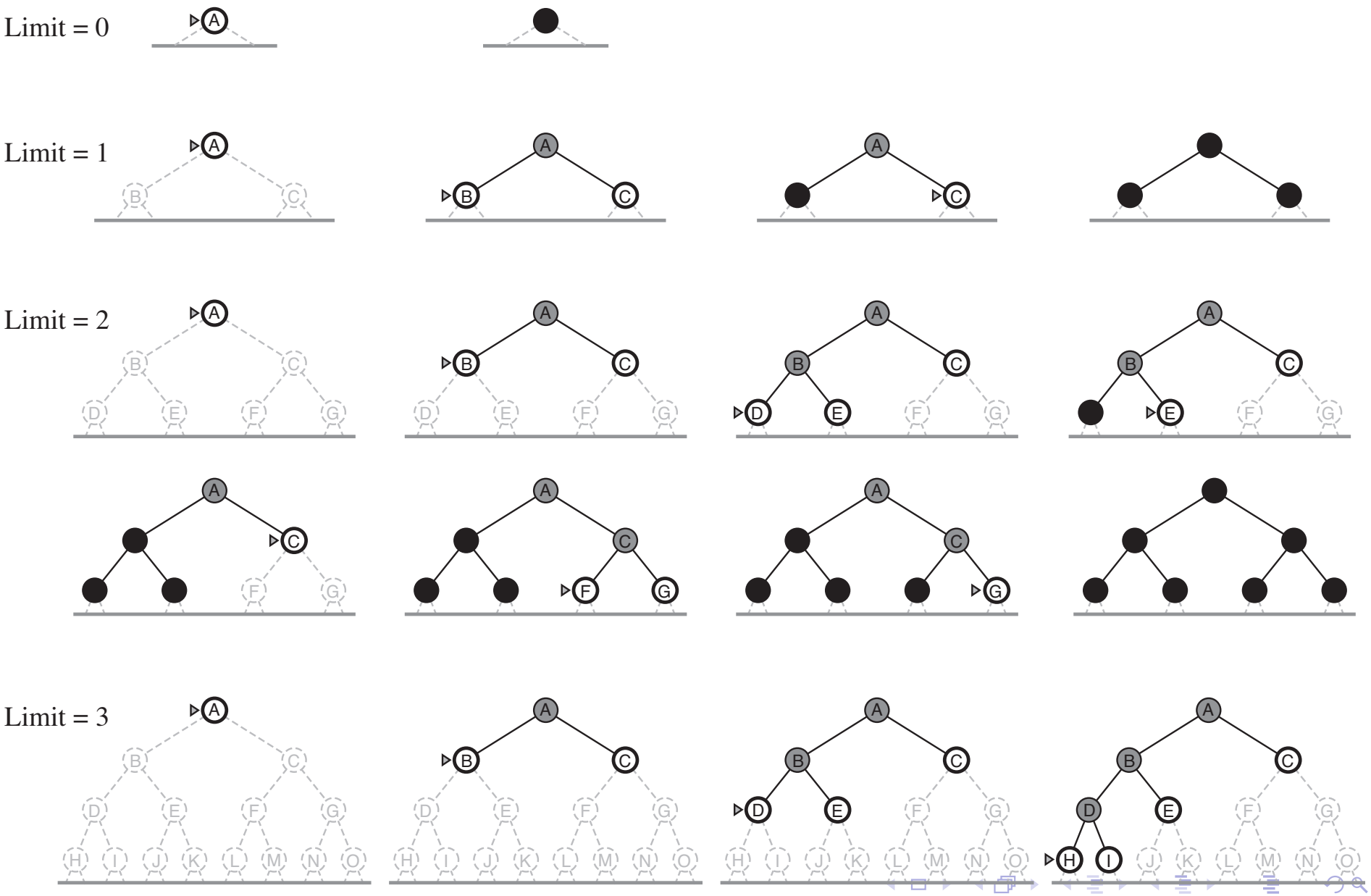


- Use of a LIFO queue

Enhancements to DFS

- Depth-limited search does not search below a certain depth.
  Introduces incompleteness.
- Iterative Deepening DFS.
  - Acts as a DFS with a gradually increasing limit until a goal is reached.
  - Combines benefits of DFS and BFS
  - States are generated multiple times, but not (very) costly.

Iterative Deepening DFS



Uniform Cost Search

- Breadth first search is optimal when all step costs are equal — it always expands the shallowest unexpanded nodes.
- What if we have different costs for expanding different nodes?
- UCS expands the node with the lowest path cost.
- There must always be a positive cost (otherwise, infinite loops can arise).

Where are we?

- BFS,DFS,UCS all expand new states according to some strategy and test them against a goal.
- This approach is highly inefficient
  - Apart from taking connectivity and (potentially) cost into account, they ignore any structure in the problem.
  - They represent uninformed search.
- By using problem specific knowledge (i.e. an informed search), we can search more efficiently.

Informed Search

- Basic idea: select a node for expansion based on the value of an evaluation function $f(n)$.
- This function measures distance from the goal, so we expand the node with lowest evaluation.
- The search is a best-first one; we expand the node that (we believe) is the best one first.
- Different informed searches use different evaluation functions.
- A key component is the heuristic function $h(n)$ which estimates the cost of the cheapest path from the current node to a goal node.
- E.g. straight line distance from current point to the goal.
- For any $h(n)$, if n is a goal node, $h(n) = 0$

A* search

- The A* search is probably the most used type of best first search.
- It combines the cost to reach a node ($g(n)$) with the cost to get from the node to the goal ($h(n)$)

$$f(n) = g(n) + h(n)$$

- $f(n)$ is the estimated cost of the cheapest solution through n.
- Since we're trying to find the cheapest solution, it makes sense to try the node with the lowest $f(n)$ first.
- Under certain conditions for $h(n)$, A* search is complete and optimal.

A* Search

- In the case of tree search, A* is optimal if $h(n)$ is an admissible heuristic.
- A heuristic is admissible if it never overestimates the cost to reach the goal.
- These are "optimistic", they think the cost of solving a problem is less than it is.
- Using such a heuristic means that $f(n)$ never overestimates the cost of reaching the goal.
- The straight line distance is admissible. Why?

Admissible Heuristics

- Tree search with A* is optimal if the heuristic used is admissible.
  - Assume that there is suboptimal goal node G in the frontier, and let the cost of the optimal solution be C. Now,

$$f(G) = g(G) + h(G)$$

  - Since $h(G) = 0$ (as h is admissible), $g(G) > C$
  - Now consider a different node on an optimal solution path. If h does not overestimate, then

$$f(n) = g(n) + h(n) < C$$

  - So $f(n) \leq C < f(G)$ so G will not be expanded, and therefore A* must return an optimal solution.
- This doesn't work for graph search as the optimal state can be discarded.

A* in a Graph

- To ensure optimality of admissible heuristics in a graph search requires either
  - Always discarding the more expensive of any two paths to the same node; or
  - Ensuring that the optimal path to any repeated state is always the first one followed.
- The second property holds if h is consistent (a.k.a monotonic): if for every node n and successor n' generated by a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n'.

$$h(n) \leq c(n, a, n') + h(n')$$

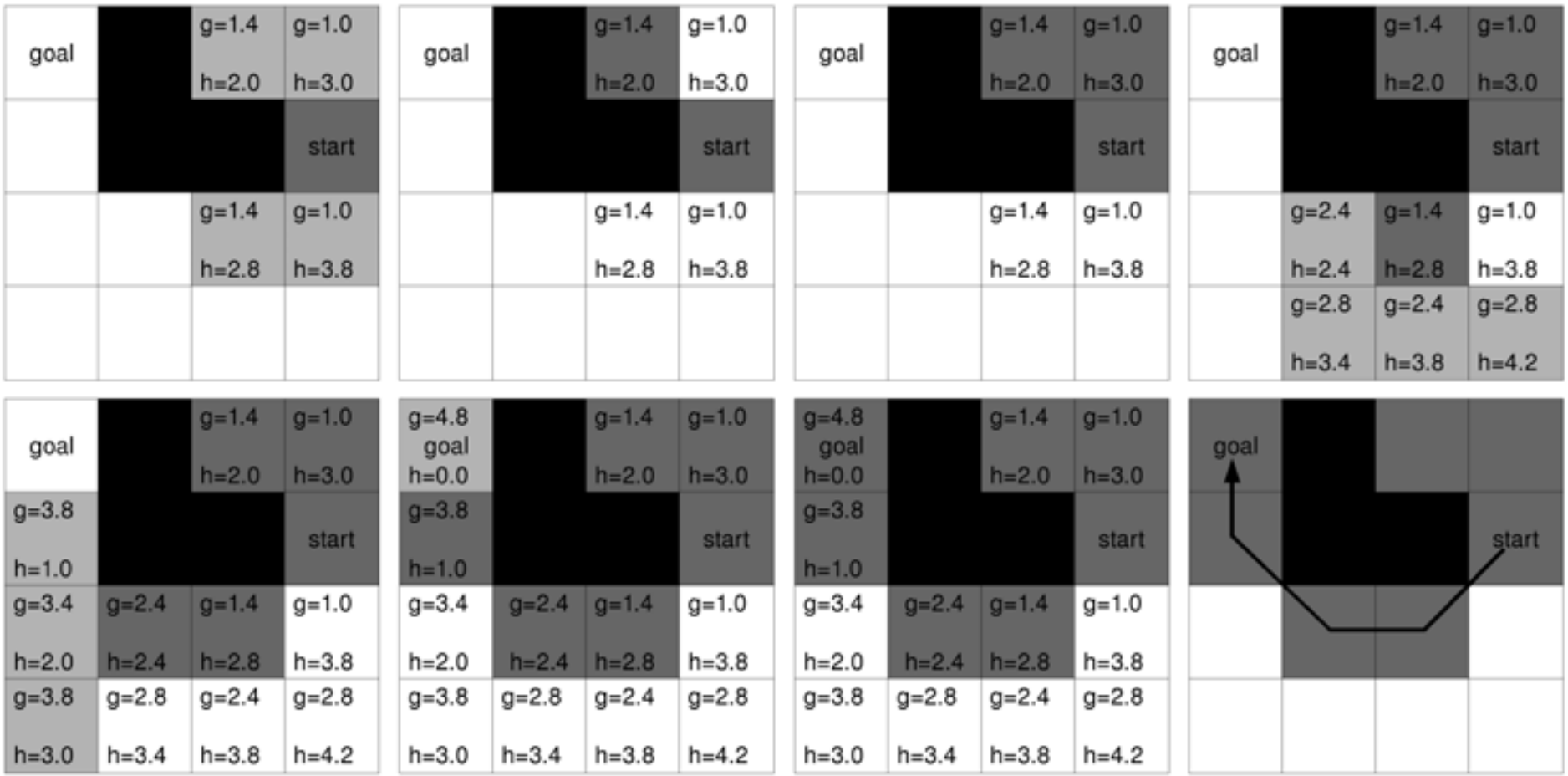
- It's hard to find admissible heuristics which are not consistent.
- If h is consistent, then the values of f along any path are nondecreasing.

The Algorithm

Require: a start state s
Require: a graph G
Require: a goal state *goal*

1: create an empty priority queue *Queue*
2: set *closed* to be the empty set
3: add s to *Queue*
4: **while** P is not empty **do**
5: $\quad$ *node* $\leftarrow$ dequeue an element of *Queue*
6: $\quad$ If *node* $=$ *goal* **return** solution
7: $\quad$ **if** *node* $\notin$ *closed* **then**
8: $\quad\quad$ add *node* to *closed*
9: $\quad\quad$ add all successors of *node* to *Queue*
10: $\quad$ **end if**
11: **end while**
12: **return** *fail*

A* Search


goal
g=1.4 h=2.0
g=1.0 h=3.0
start
g=1.4 h=2.8
g=1.0 h=3.8
goal
g=1.4 h=2.0
g=1.0 h=3.0
start
g=1.4 h=2.8
g=1.0 h=3.8
goal
g=1.4 h=2.0
g=1.0 h=3.0
start
g=1.4 h=2.8
g=1.0 h=3.8
goal
g=1.4 h=2.0
g=1.0 h=3.0
start
g=2.4 h=2.4
g=1.4 h=2.8
g=1.0 h=3.8
g=2.8 h=3.4
g=2.4 h=3.8
g=2.8 h=4.2
goal
g=1.4 h=2.0
g=1.0 h=3.0
g=3.8 h=1.0
start
g=3.4 h=2.0
g=2.4 h=2.4
g=1.4 h=2.8
g=1.0 h=3.8
g=3.8 h=3.0
g=2.8 h=3.4
g=2.4 h=3.8
g=2.8 h=4.2
g=4.8 goal h=0.0
g=1.4 h=2.0
g=1.0 h=3.0
g=3.8 h=1.0
start
g=3.4 h=2.0
g=2.4 h=2.4
g=1.4 h=2.8
g=1.0 h=3.8
g=3.8 h=3.0
g=2.8 h=3.4
g=2.4 h=3.8
g=2.8 h=4.2
g=4.8 goal h=0.0
g=1.4 h=2.0
g=1.0 h=3.0
g=3.8 h=1.0
start
g=3.4 h=2.0
g=2.4 h=2.4
g=1.4 h=2.8
g=1.0 h=3.8
g=3.8 h=3.0
g=2.8 h=3.4
g=2.4 h=3.8
g=2.8 h=4.2
goal
start

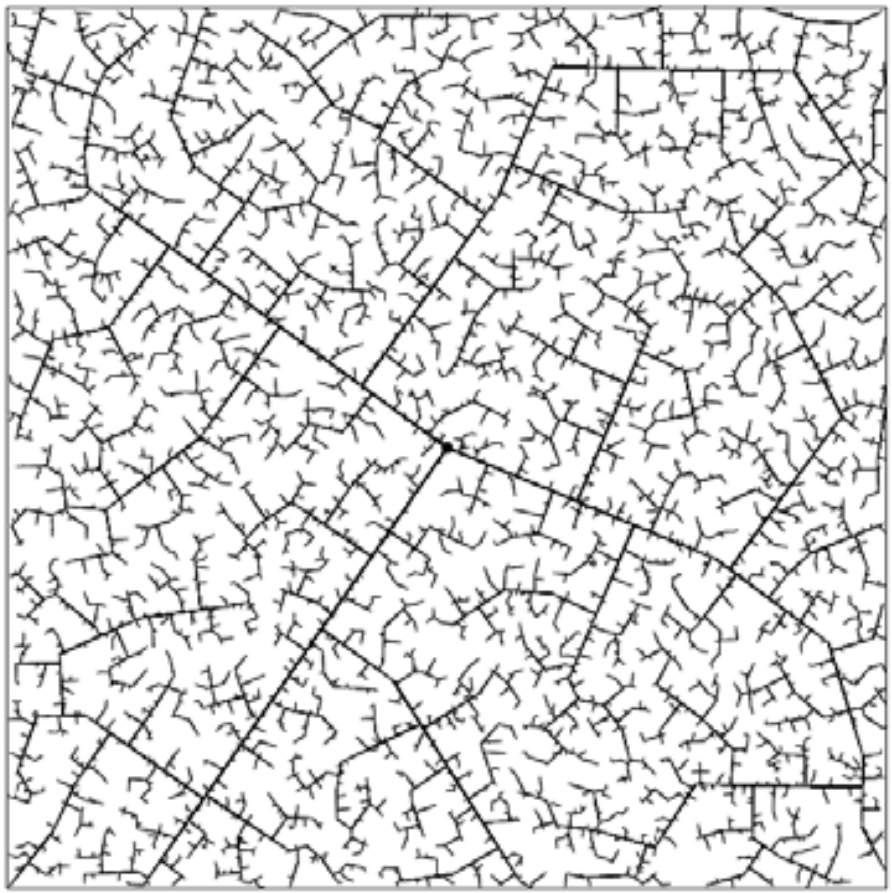
Randomised Search

1. Start with an initial tree T
2. While not finished
3. Pick a random point r in free space
4. Find nearest node in tree n
5. Create an edge from r towards n, with some fixed length
6. If the goal is reached, finished=true

- Variations allow picking of the goal rather than a random node.
- Probabilistically complete
- Works (surprisingly) well in sparse environments.
- Highly parallel.

Randomised Tree Search

45 iterations

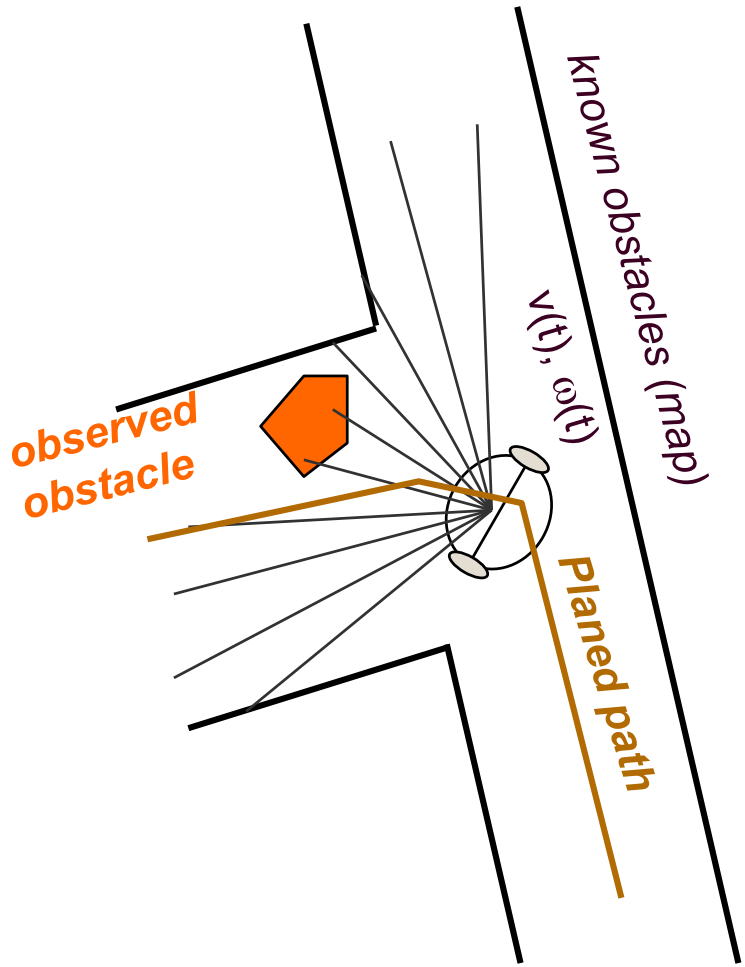


2345 iterations

C. S. LaValle

Where are we?

- Given a global map, we can get from a start position to a goal position.
- Such techniques make no use of the robot's sensors to detect and avoid obstacles.
- Obstacle avoidance techniques do so, typically delying on only a very local map.

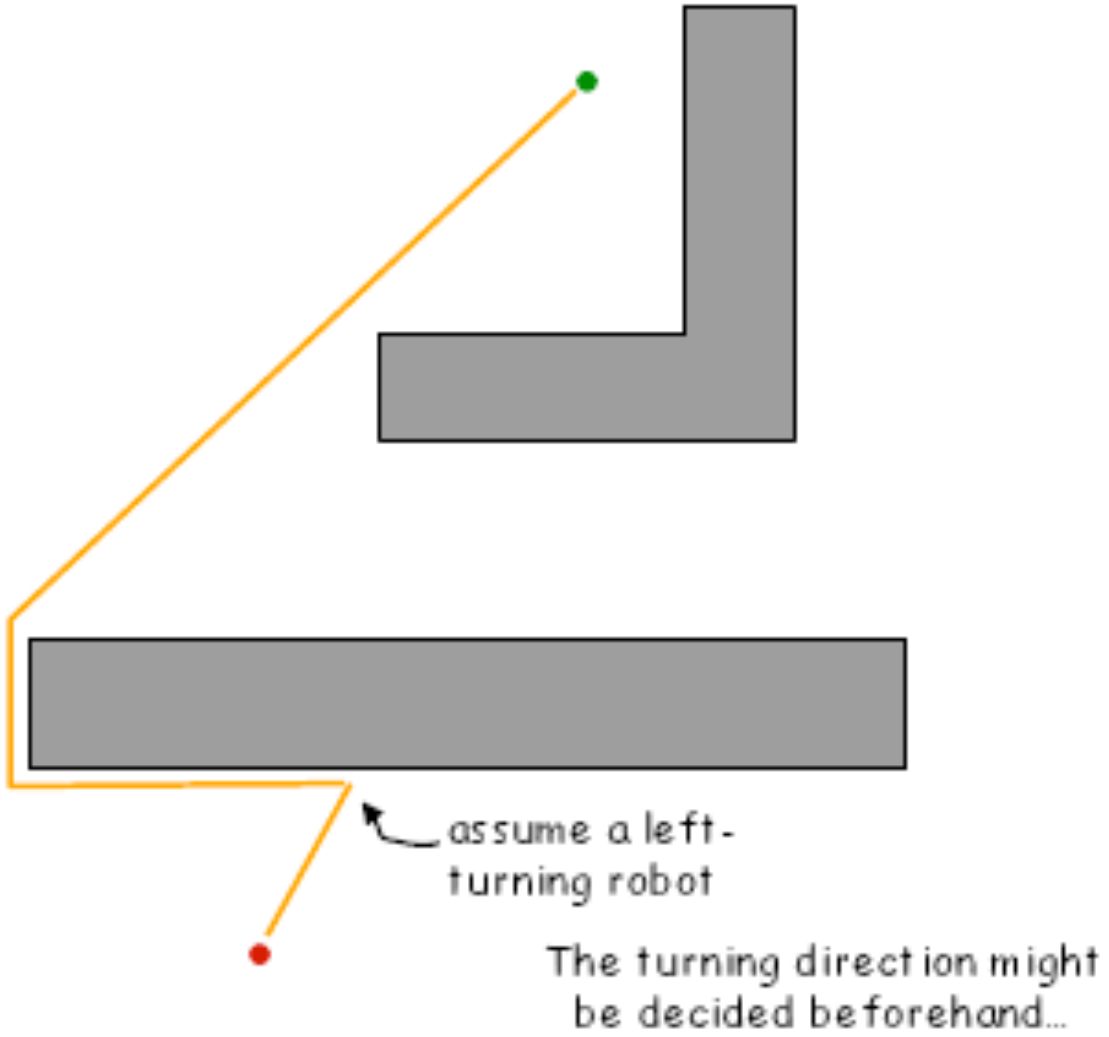


Bug Algorithms

- The Bug family of algorithms assume only very local knowledge of the environment, together with a global goal.
- Tactile, or very short range sensing is assumed.

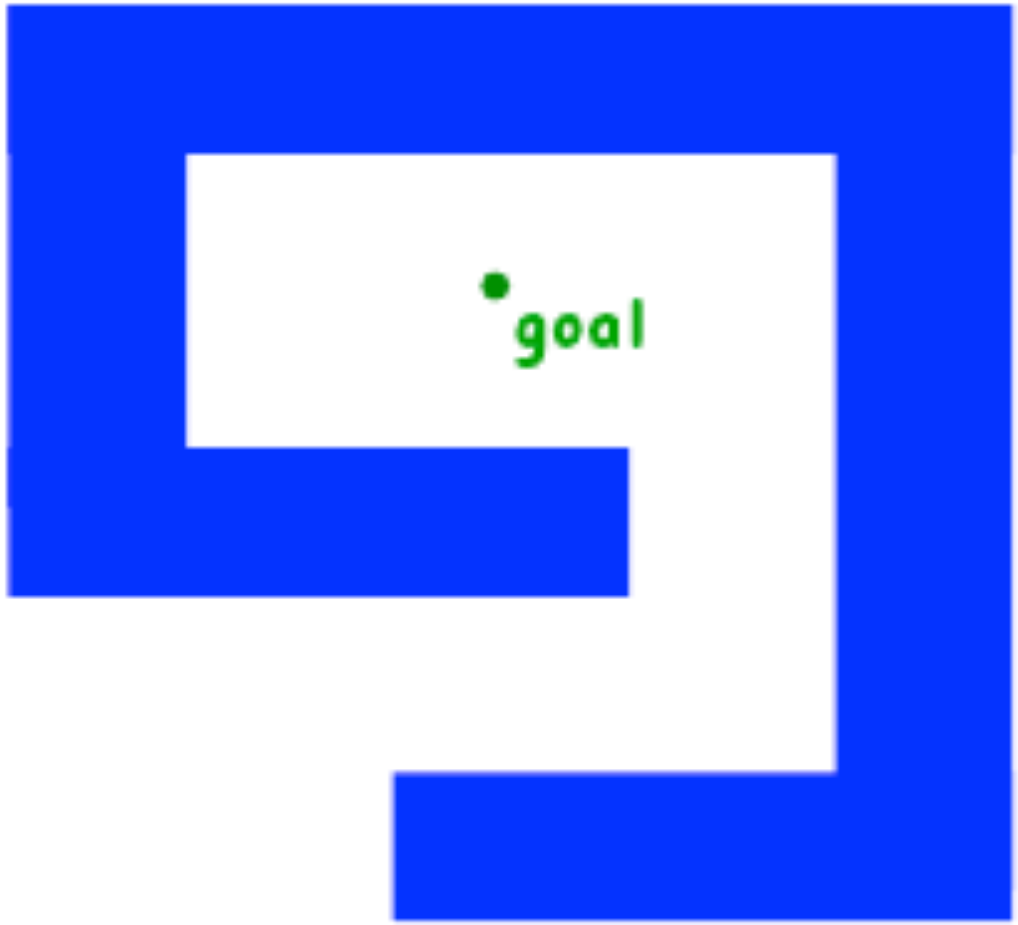
Bug 0

1. Head towards goal
2. If an obstacle is encountered, follow it until you can head towards your goal again
3. rinse, repeat



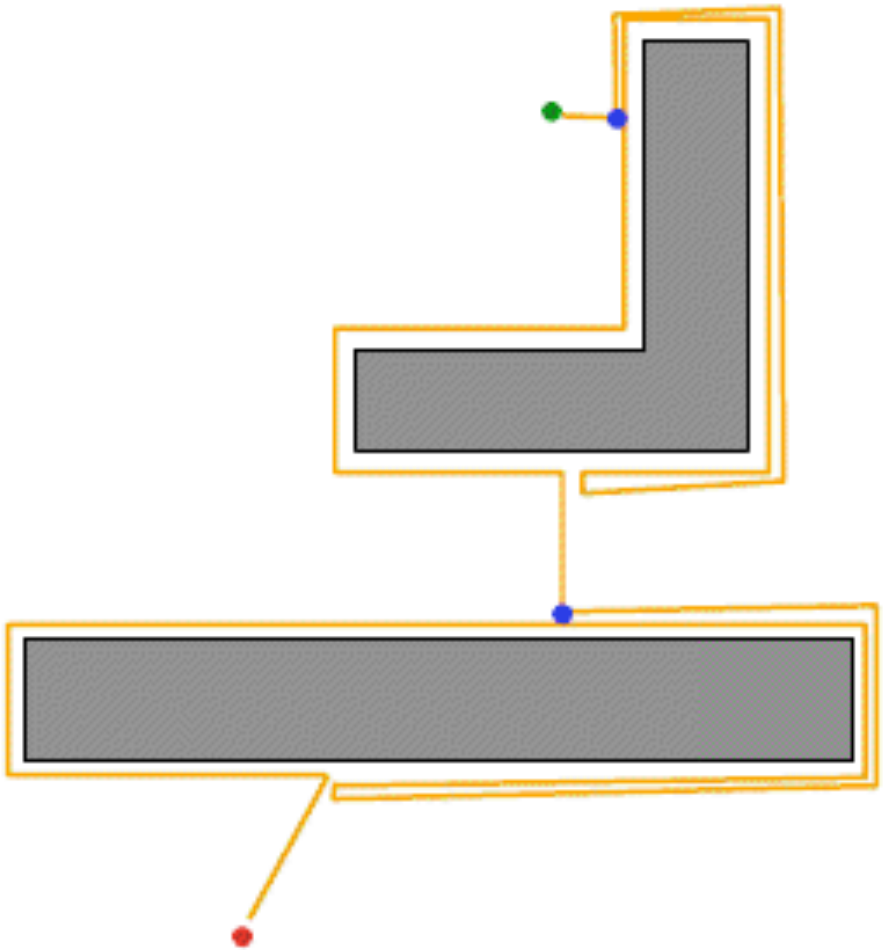
Bug 0

1. Head towards goal
2. If an obstacle is encountered, follow it until you can head towards your goal again
3. rinse, repeat



Bug 1

1. Head towards goal
2. If an obstacle is encountered, circumnavigate it, identifying closest point to goal.
3. Once finished circumnavigating, return to closest point (by wall following) and continue towards goal.
4. rinse, repeat



Bug 1

- Given a distance D from start to end, and n obstacles with perimeters $P_1 \dots P_n$
- Lowest distance Bug 1 will travel is D
- Upper distance is $D + \sum_{i=1}^{n} 1.5 P_i$
- When will this upper distance be reached?
- `https://www.youtube.com/watch?v=FMERewWFHLI`

Bug 1 Completeness

- Can we show that Bug1 will always work correctly?
- Assume Bug 1 is incomplete.
  - Then there is a path from the start to the goal, which Bug1 does not find.
  - Either due to incorrect termination, or taking an infinite amount of time.

Bug 1 Completeness

- Suppose Bug 1 takes an infinite amount of time.
- But we know that each time Bug1 leaves an obstacle, it is closer to the goal than when it encountered it.
- This means the each hit point is closer than the previous leave point.
- Since there are only a finite number of obstacles, there are a finite number of hit/leave pairs. After going through them all, the robot will head towards the obstacle.

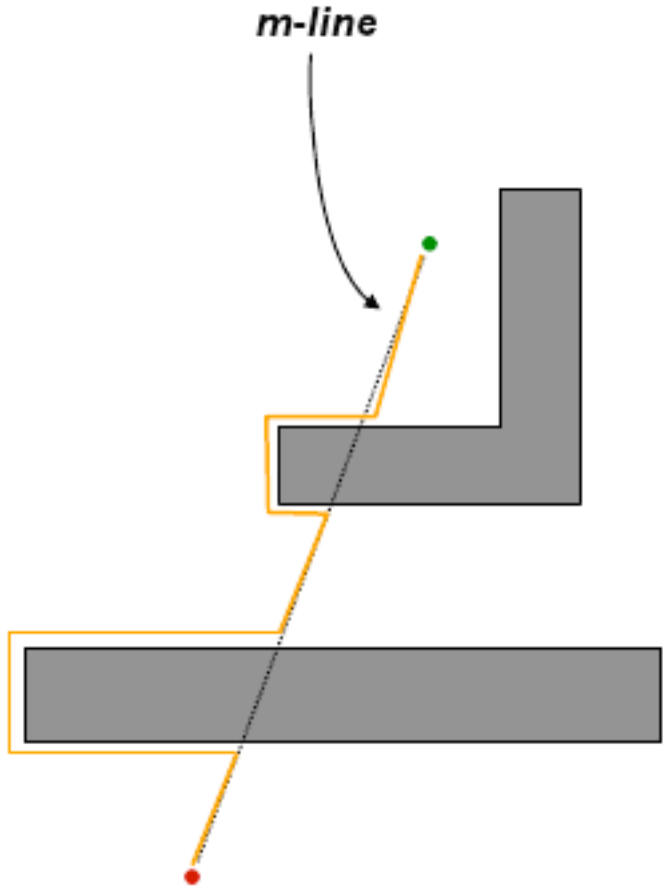
Bug 1 Completeness

- Suppose Bug 1 terminates incorrectly
- Then the leave point and direction must drive it into the obstacle
- The line from the robot to the goal must intersect the obstacle an even number of times.
- If there is a path, the closer intersection point to the goal must have been crossed, and we must leave from this towards the object.
- This leave point is in contradiction with the earlier one.

Bug 2

1. Define an m-line as the line from the start to the goal.
2. Head towards the goal on the m-line
3. If an obstacle is detected, follow it until the m-line is encountered again .
4. Leave the obstacle and continue towards the goal

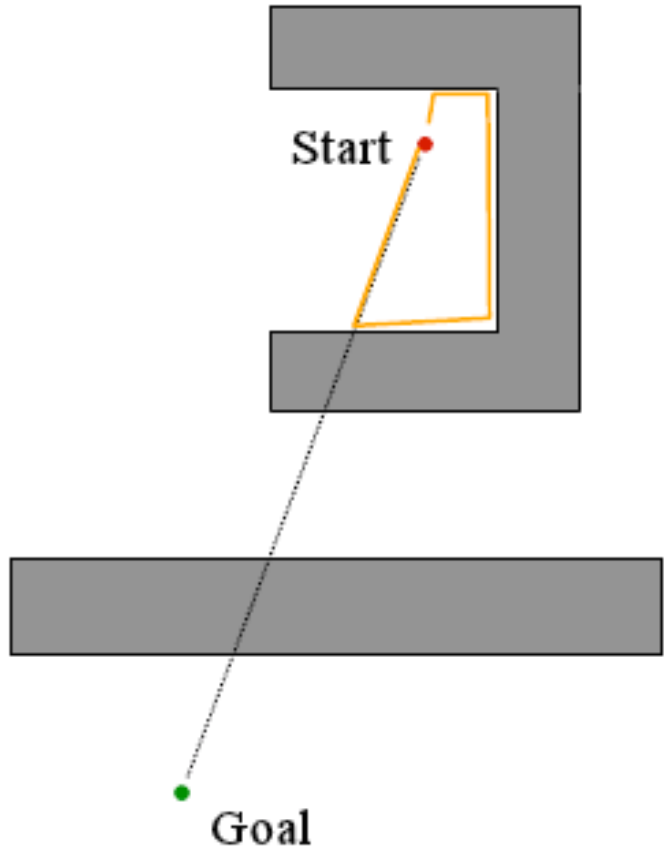
- `https://www.youtube.com/watch?v=-iOu0bawbcA`



Bug 2

1. Define an m-line as the line from the start to the goal.
2. Head towards the goal on the m-line
3. If an obstacle is detected, follow it until the m-line is encountered again .
4. Leave the obstacle and continue towards the goal

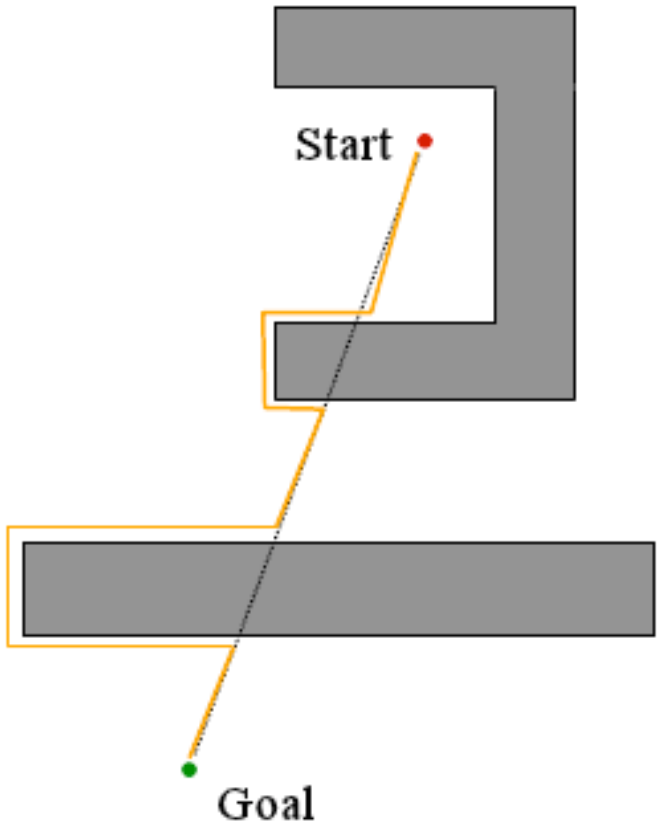
- `https://www.youtube.com/watch?v=-iOu0bawbcA`



Bug 2

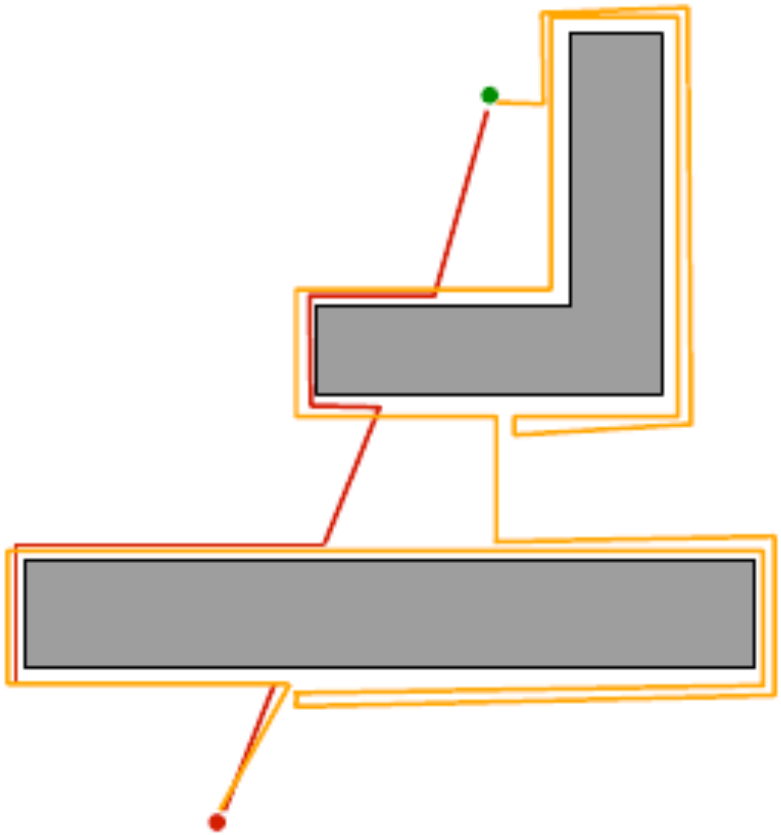
1. Define an m-line as the line from the start to the goal.
2. Head towards the goal on the m-line
3. If an obstacle is detected, follow it until the m-line is encountered again closer to the goal.
4. Leave the obstacle and continue towards the goal

- `https://www.youtube.com/watch?v=-iOu0bawbcA`

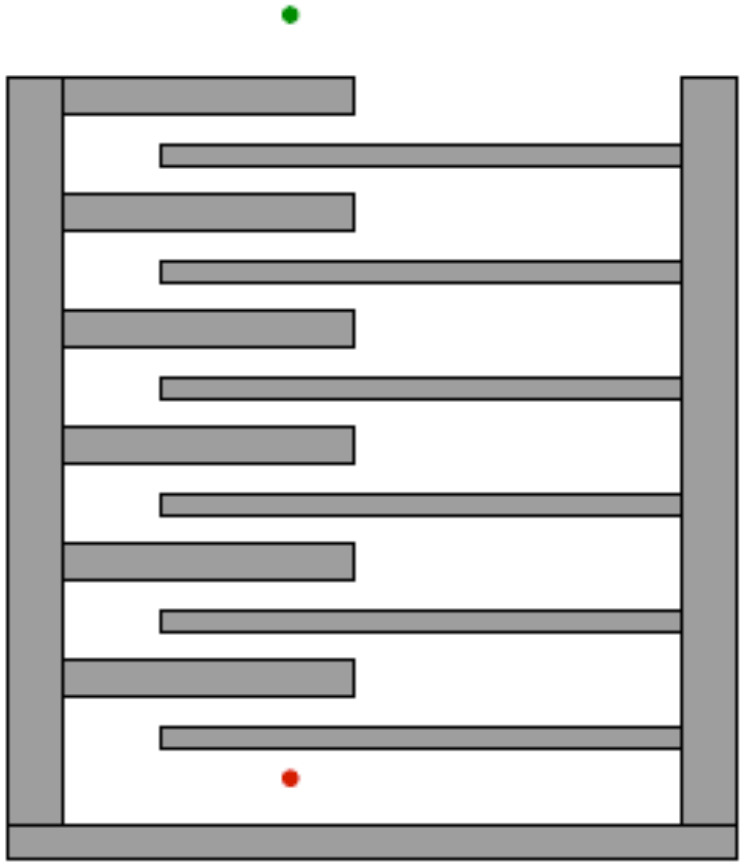


Battle of the bugs

Bug 2 beats Bug 1



Bug 1 beats Bug 2



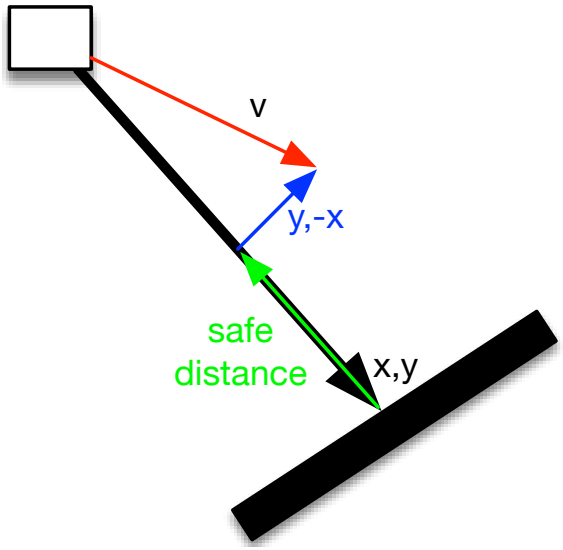
Battle of the bugs

- Bug 1 is an exhaustive search algorithm — examines all options before committing.
- Bug 2 is a greedy search algorithm — takes first possible "good" choice.
- In many situations Bug 2 beats Bug 1
- But performance of Bug 1 is more predictable.
- Bug 2 lower distance bound is D — distance from goal to destination
- Bug 2 upper distance bound is $D + \sum_i n_i P_i/2$ where n_i is the number of m-line intersections object i has.
- Additional enhancements to these algorithms exist.

Bug Practicalities

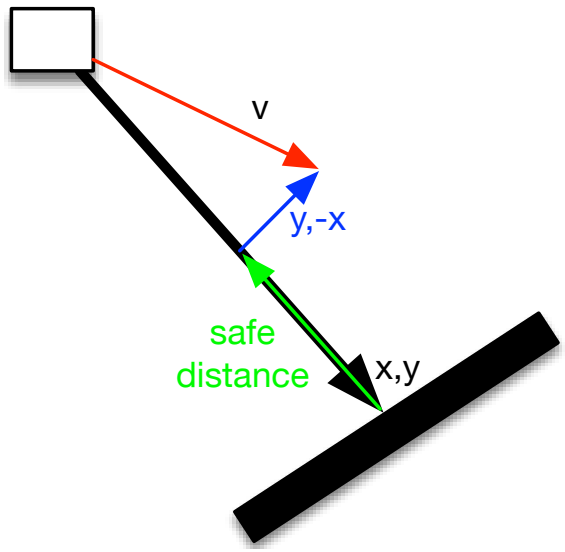
- Bug algorithms ask us to "follow a wall". But how do we do this?
- One simple (but not foolproof) approach involves trying to stay the same distance from the nearest obstacle, while maintaining forward movement.

Bug Practicalities



- We can turn by $arctan(v.y/v.x)$ (open loop control)
- Problems?

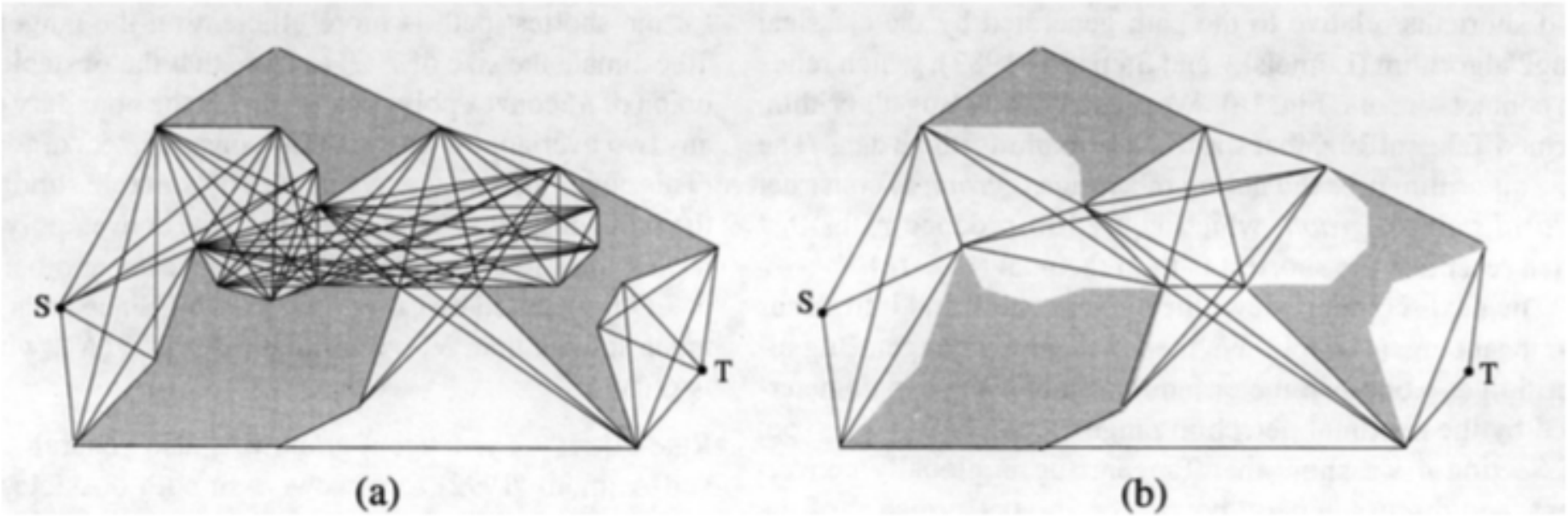
Bug Practicalities



- if $v.x \leq 0$ *arctan* will give the incorrect result
- Two tangents exist, we could end up going in the wrong direction
- We don't remember which wall we're following, just the closest one. This can be problematic for convex shapes, or where there are many obstacles.

Tangent Bug[1]

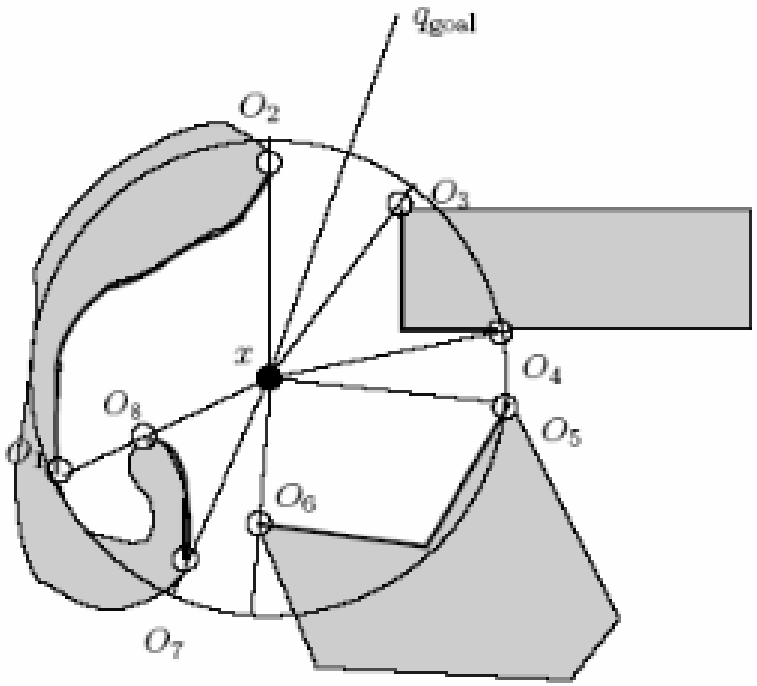
- In reality we have some range sensors, these let us look ahead.
- Ideally we could move along tangent lines to goal.



[1] Image from Kamon et al., 1998

Tangent Bug[2]

- Tangents are approximated based on what we can see — Tangent Bug relies on finding endpoints of continuous segments by using angles and a threshold heuristic.



[2]images from Choset, Hager and Dodds

Tangent Bug[3]

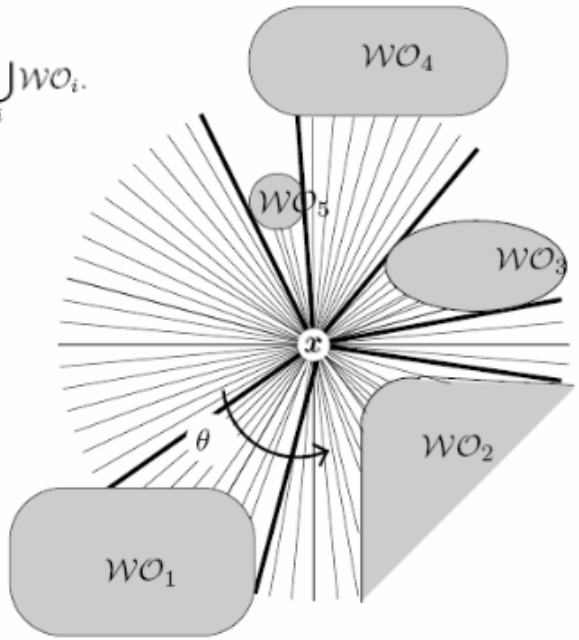
- Tangents are approximated based on what we can see — Tangent Bug relies on finding endpoints of continuous segments by using angles and a threshold heuristic.

$$\rho(x,\theta) = \min_{\lambda\in[0,\infty]} d(x, x+\lambda[\cos\theta,\sin\theta]^T),$$
$$\text{such that} \quad x+\lambda[\cos\theta,\sin\theta]^T \in \bigcup_i \mathcal{WO}_i.$$

$$\rho\colon \mathbb{R}^2 \times S^1 \to \mathbb{R}$$

Saturated raw distance function

$$\rho_R(x,\theta) = \begin{cases} \rho(x,\theta), & \text{if } \rho(x,\theta) < R \\ \infty, & \text{otherwise.} \end{cases}$$

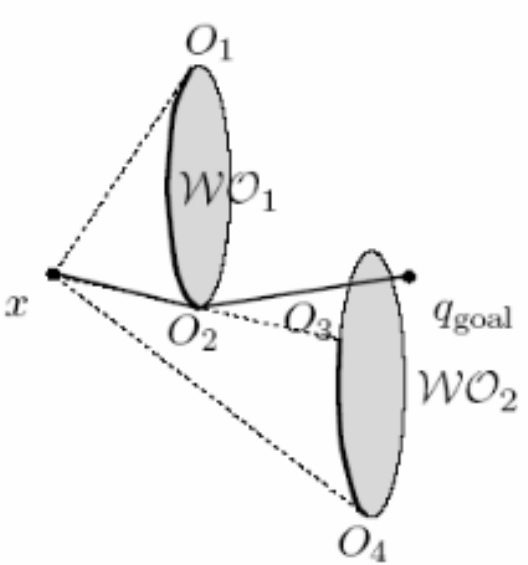


[3] images from Choset, Hager and Dodds

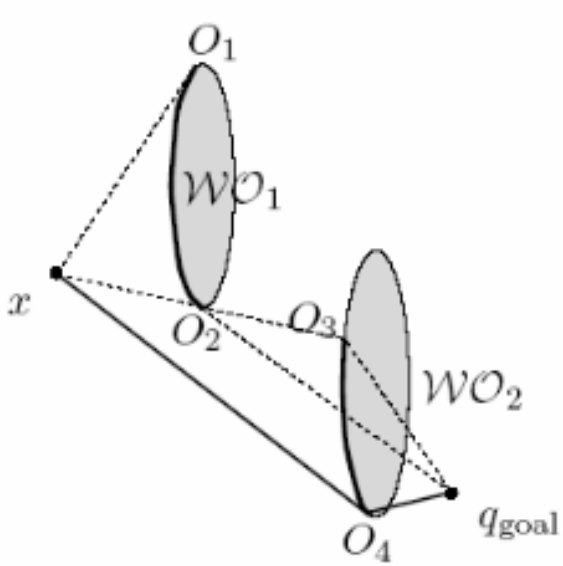
Tangent Bug[4]

- Initially the robot uses a motion-to-goal appraoch, trying to move along the tangent it is on towards the goal. If an obstacle is encountered, it must select an appropriate tangent to move along.

At x, robot knows only what it sees and where the goal is,

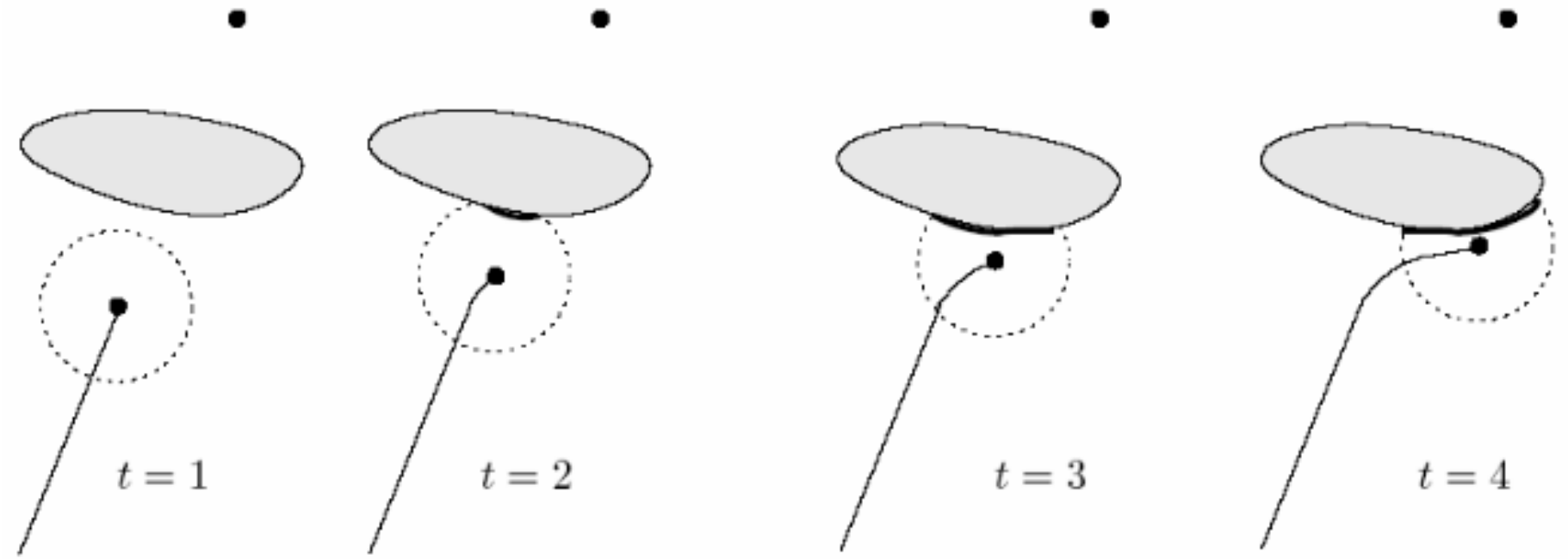


so moves toward O_2. Note the line connecting O_2 and goal pass through obstacle



so moves toward O_4. Note some "thinking" was involved and the line connecting O_4 and goal pass through obstacle

Tangent Bug[5]



Choose the pt O_i that minimizes $d(x,O_i) + d(O_i,q_{goal})$

[5] images from Choset, Hager and Dodds

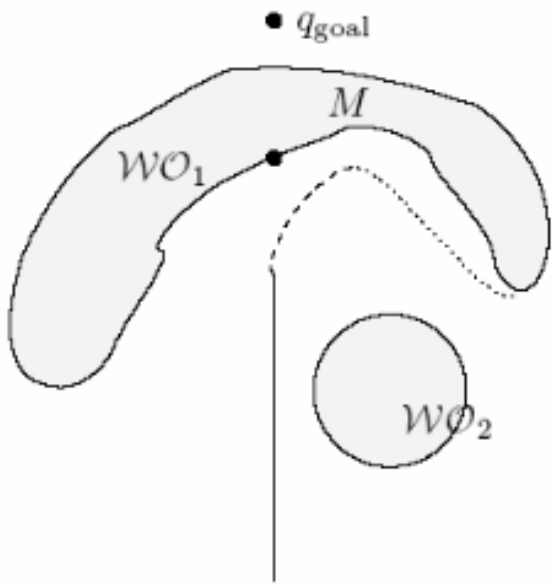
Tangent Bug

- But what if the heuristic distance starts to go up (i.e., it's stuck in a corner)?
- Answer: start acting like a bug and follow the boundary.

Tangent Bug[6]

Move toward the O_i on the followed obstacle in the "chosen" direction

Maintain $d_{followed}$ and d_{reach}



M is the point on the "sensed" obstacle which has the shorted distance to the goal

Followed obstacle: the obstacle that we are currently sensing

Blocking obstacle: the obstacle that intersects the segment

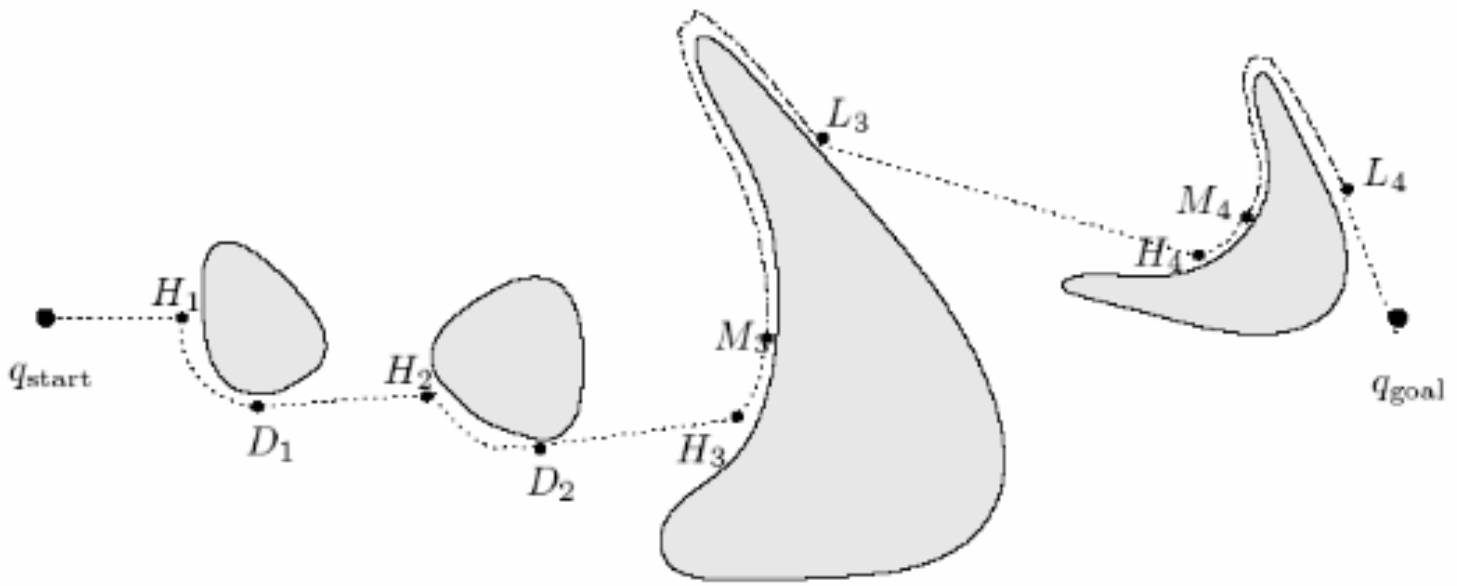
They start as the same

[6] images from Choset, Hager and Dodds

Tangent Bug

- $d_{followed}$ is the shortest distance seen between the sensed boundary and the goal.
- d_{reach} is the shortest distance currently observed between the obstacle and goal.
- When $d_{reach} < d_{followed}$, the robot should move to $d_r each$ and then start motion towards goal again.

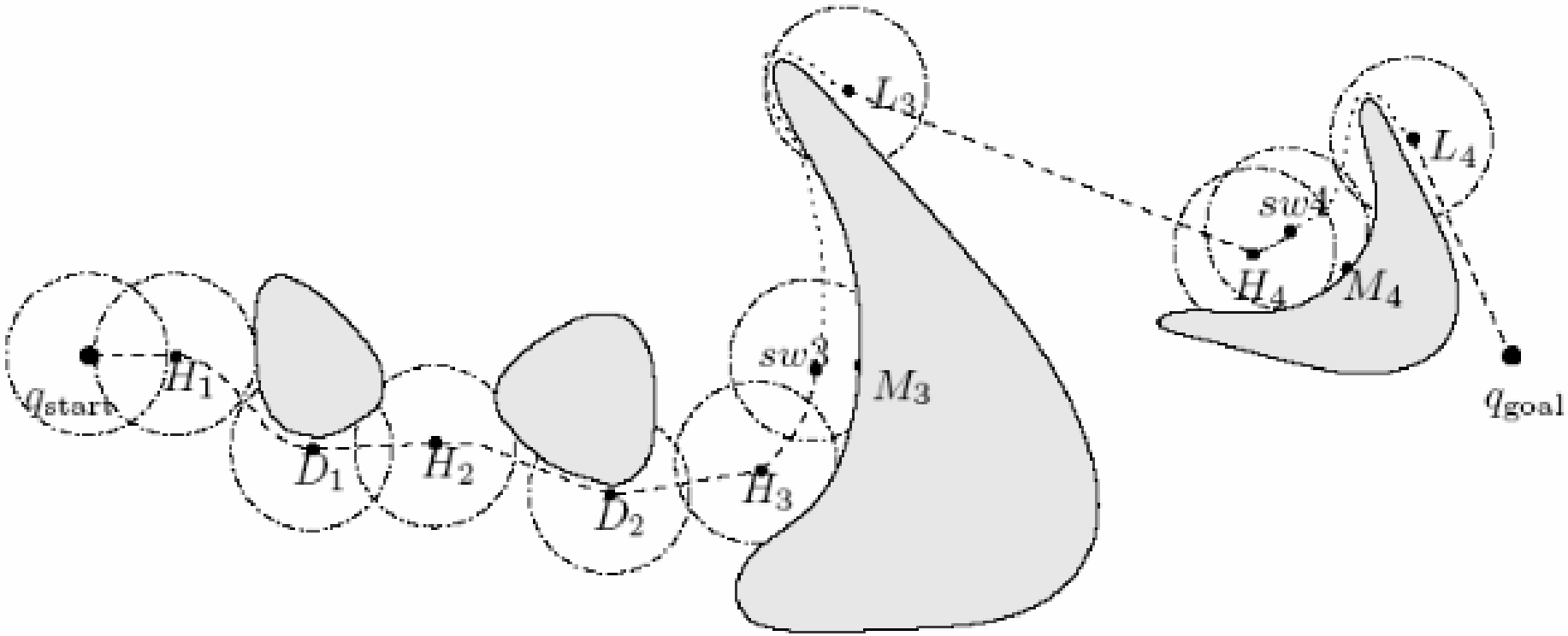
Tangent Bug - Zero range sensor[7]



1. Robot moves toward goal until it hits obstacle 1 at H1
2. Pretend there is an infinitely small sensor range and the Oi which minimizes the heuristic is to the right
3. Keep following obstacle until robot can go toward obstacle again
4. Same situation with second obstacle
5. At third obstacle, the robot turned left until it could not increase heuristic
6. $D_{followed}$ is distance between M_3 and goal, d_{reach} is distance between robot and goal because sensing distance is zero

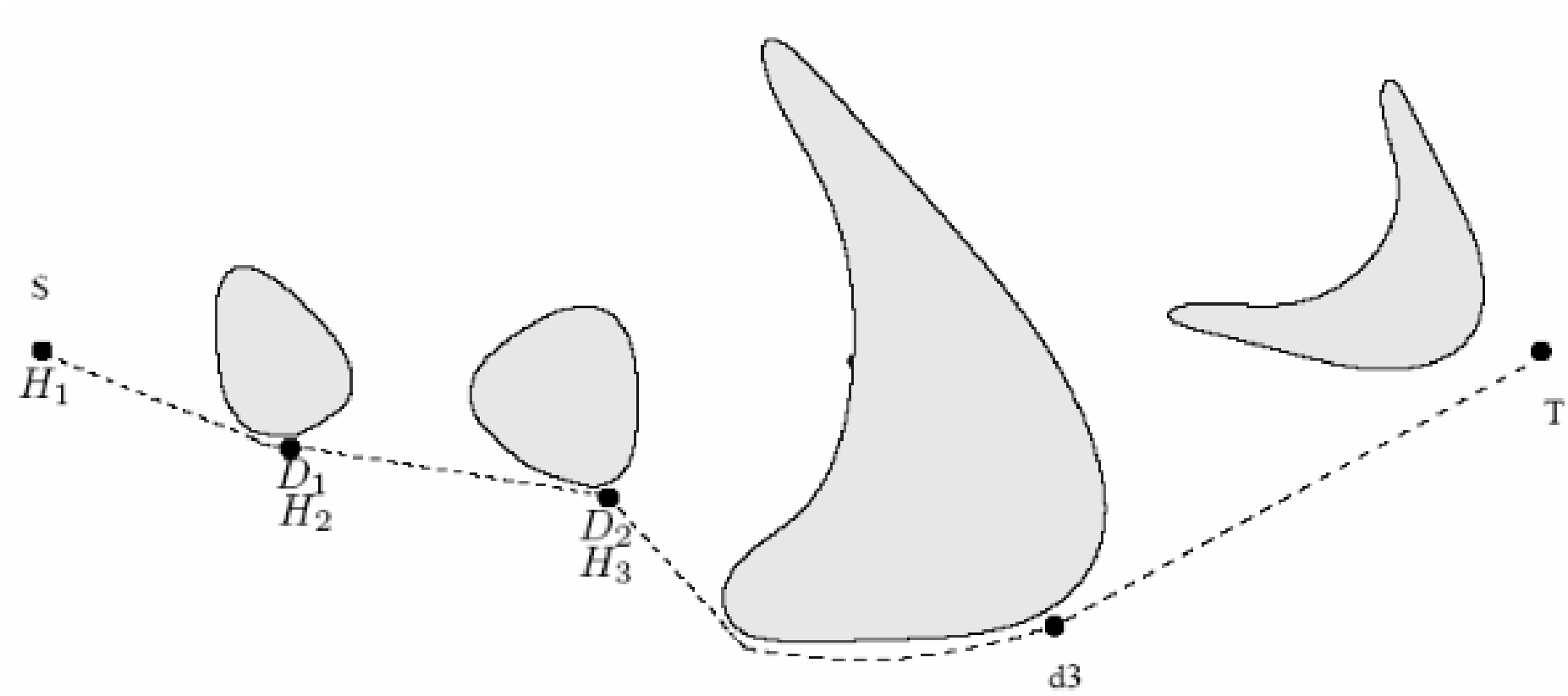
[7] images from Choset, Hager and Dodds

Tangent Bug - Finite range sensor[8]



[8] images from Choset, Hager and Dodds

Tangent Bug - Infinite range sensor[9]



[9] images from Choset, Hager and Dodds

The algorithm

Require: q_{goal} the goal position

1: state:=move towards goal
2: **repeat**
3: Compute $O = \{O_1, \ldots, O_n\}$, range segments in view; x, robot position
4: **if** state=move towards goal **then**
5: move towards $n \in \{q_{goal}, O_i\}$ minimising $h(x, n) = d(x, n) + d(n, q_{goal})$
6: **if** h(x,n) is increasing **then**
7: initialize $d_{followed}$
8: state:=boundary following
9: **if** state=boundary following **then**
10: Keep following boundary in same direction, update d_{reach}
11: **if** $d_{reach} < d_{followed}$ **then**
12: state:=move towards goal (after d_{reach} is reached)
13: **until** at goal or at previous visited position

Implementation Issues

- The tricky bit is "keep following boundary in same direction"...
- Head towards appropriate tangent point.
- Always in front of robot
- of same "type" as before (e.g., s.t. boundary start is further from robot then boundary end)
- of same obstacle as was followed.

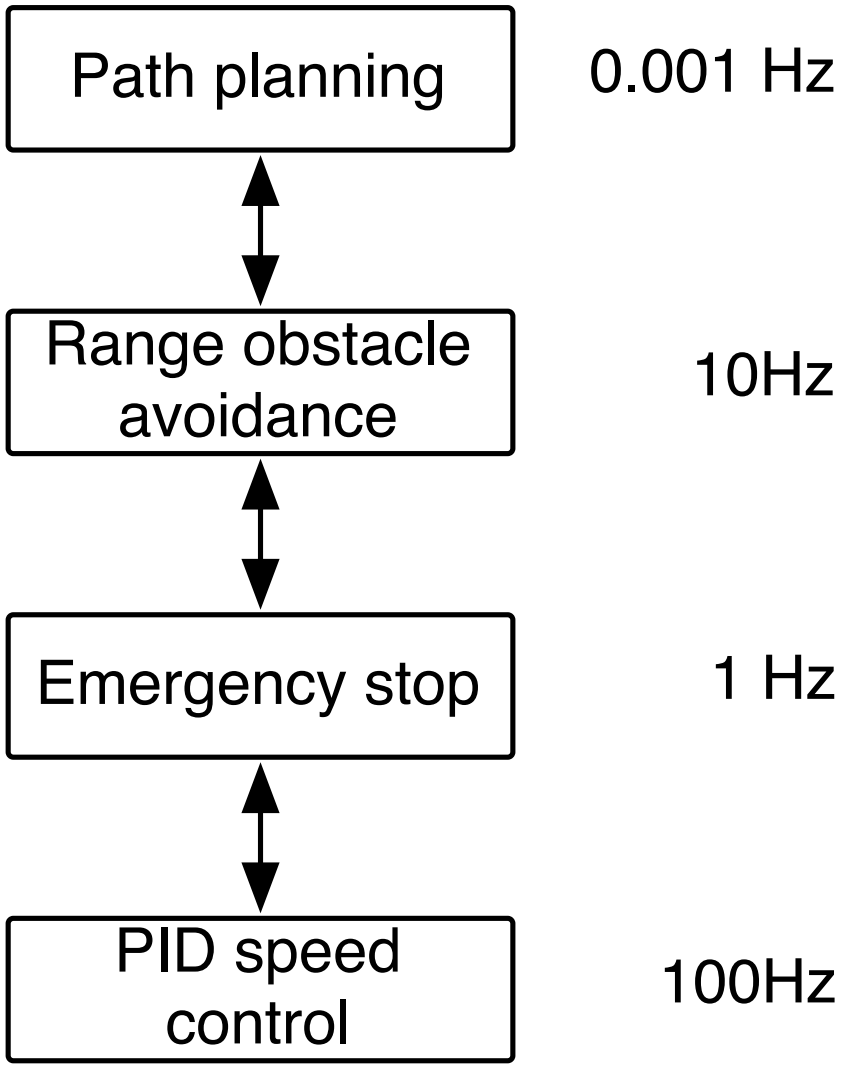
Tying it all together

- How do we combine path planning, obstacle avoidance, localisation and perception interpretation to obtain a complete system?
- We could design monolithic software to do so, works for simple cases.
- For larger more permanent (robust) systems, we need a more well designed navigation architecture.
- Basic ideas:
  - modularity — allows for code reuse, changes in hardware (e.g. change ultrasonic sensor to laser rangefinder) or new algorithms.
  - control localisation — each module is responsible for one function; allows for individual testing and verification. For example, by having all obstacle avoidance in a single place, testing can be done via simulation.

Decompositions

- Temporal Decomposition distinguishes modules based on the time demands of the modules.
- Control decomposition identifies how control outputs combine to yield physical actions.

Temporal Decomposition



Temporal Decomposition

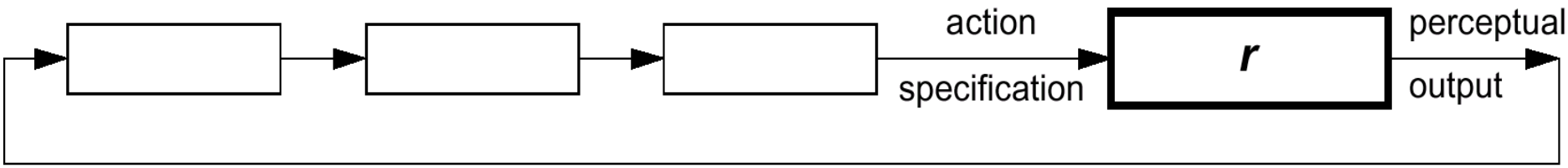
- Sensor response time — amount of time between a sensor event and the module output. Higher up in the stack, sensor response time increases.
- Temporal depth — the size of the temporal window that affects the module's output. Temporal horizon is the lookahead, temporal memory is the history. Low level modules usually have little temporal depth.
- Spatial locality — area module considers. Real time modules are very local, high level modules consider global situations.
- Context specificity — decision making occurs not only due to inputs, but other variables. Low level modules use little context.

Control Decomposition

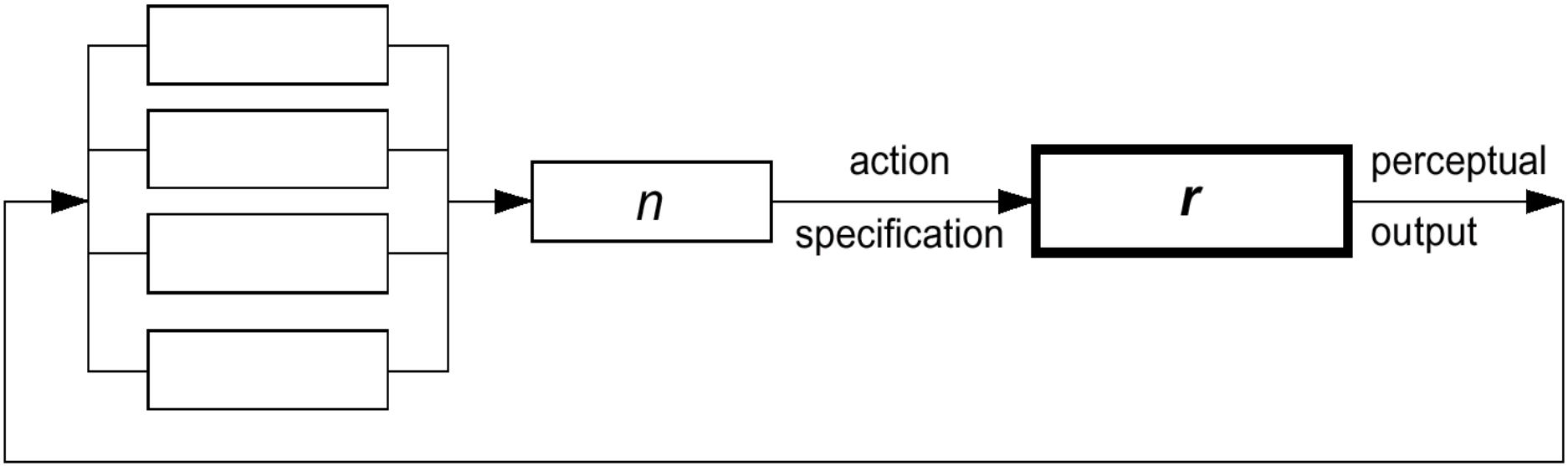
- Rather than discriminating based on the time behaviour of software modules, we can decompose according to the module's contribution to the control outputs.
- A robot is treated as a set of M modules, with each module connected to others via inputs and outputs.
- Such a system is assumed to be closed — any input is the output of one or more other modules.
- Each module has one output and one or more inputs.
- There is a special module r representing the physical object on which the algorithm has impact, and which provides perceptual inputs.
- Inputs to r are the action specifications, outputs from r all perceptual information.

Serial Decomposition

- Percepts enter the system and are transformed from module to module until an action specification is generated.
- Very predictable and verifiable — module state and outputs depend only on inputs from the previous module, so forward simulation methods can be used to evaluate the system.



Parallel Decomposition



Parallel Decomposition

- There is now a combination step which merges all inputs.
- Different ways of implementing this merger
  - Switched parallel systems ensure that the output can always be attributed to one specific module.
  - Mixed parallel modes share control by multiple modules (e.g. vector addition of output directions).
- Parallel decomposition makes it easy to combine behaviour, e.g. path finding and obstacle detection depending on distance from the obstacle.

Switched Parallel

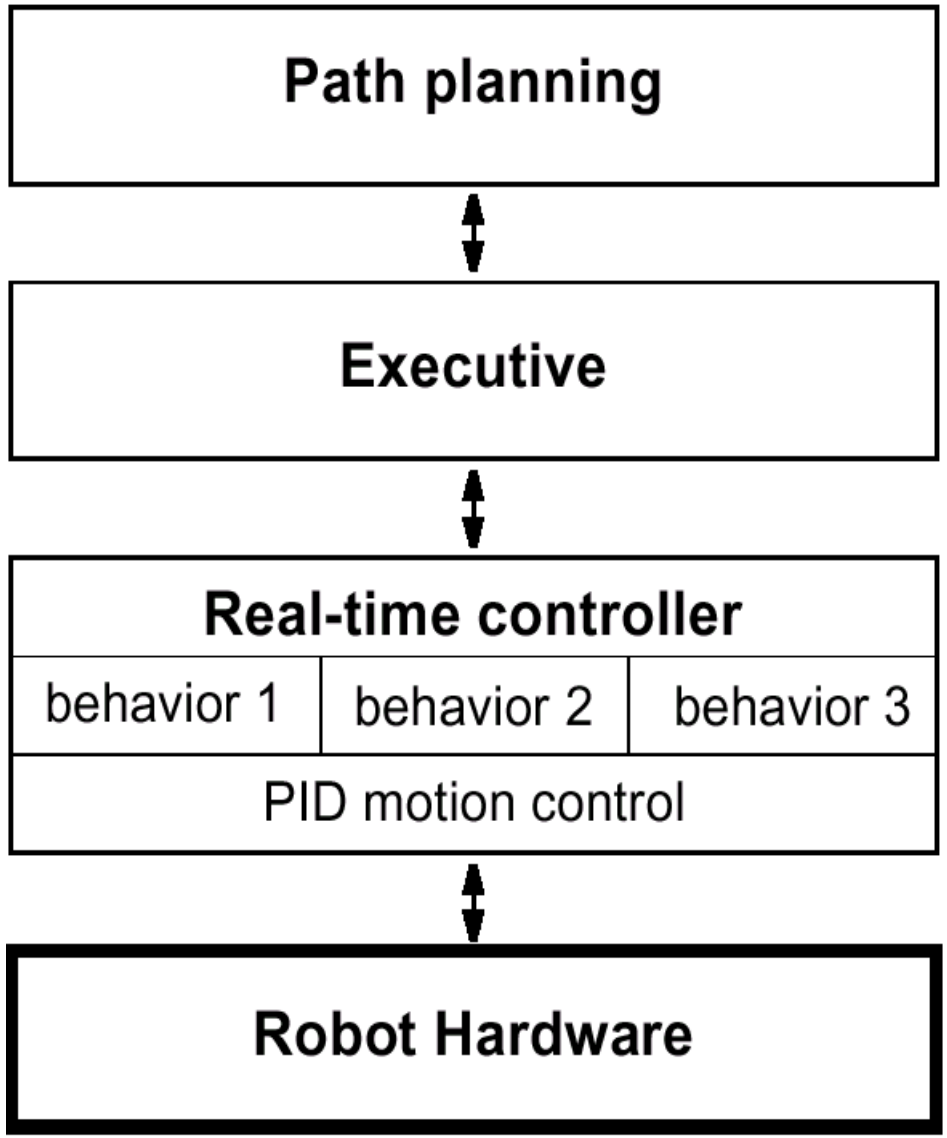
- Switched control can lead to "bouncing" between modes — instability
- Sometimes it is desirable to mix outputs of different modules (e.g. path following bias in the presence of obstacles).
- Switched parallel control leads to well understood behaviours.
- Independent testing of modules is possible.

Mixed Parallel

- More general than switched parallel control.
- But harder to debug/obtain good behaviour.
- Typically, testing of both types of parallel approaches is done using real robots as simulations may not capture all inter-module interactions appropriately.

Generalised Architecture

- Temporal decomposition view.
- Executive layer
  - decides which behaviours to activate
  - Recognises failure
  - Reinitialises the planner
- Offline planning enables one to remove the path planner component.
- Executive must then have a priori schemes for travelling to destinations
- Useful for static route applications and when reliability is critical (e.g. space shuttle)



Episodic Planner

- Offline planning cannot make use of runtime perceptual inputs (e.g. a blocked route).
- Online planning is computationally expensive.
- An episodic planner architecture allows the executive to decide when enough information has been encountered to trigger a change in strategic direction and invoke the planner.
- Deferred planning is an instantiation of this (Cye robot):
  - Given a set of goal locations, a BFS planning algorithm plots a path to the closest location and executes it.
  - Motion to this location might change the map.
  - Once at the first location, a new path is planned to the second location, etc.
- Episodic planners allow fast response and versatility and are thus very popular.

Integrated planning and execution

- It is also possible to combine the planner and executive.
- Constant replanning occurs, always taking new information into account.
- Modern graph search algorithms and lattice graphs are very efficient, and fast enough to enable this integrated executive.
- To overcome problems as the environment size increases, multi resolution approaches can be used — high fidelity lattices close to the robot, low fidelity further away (or road network representations).