

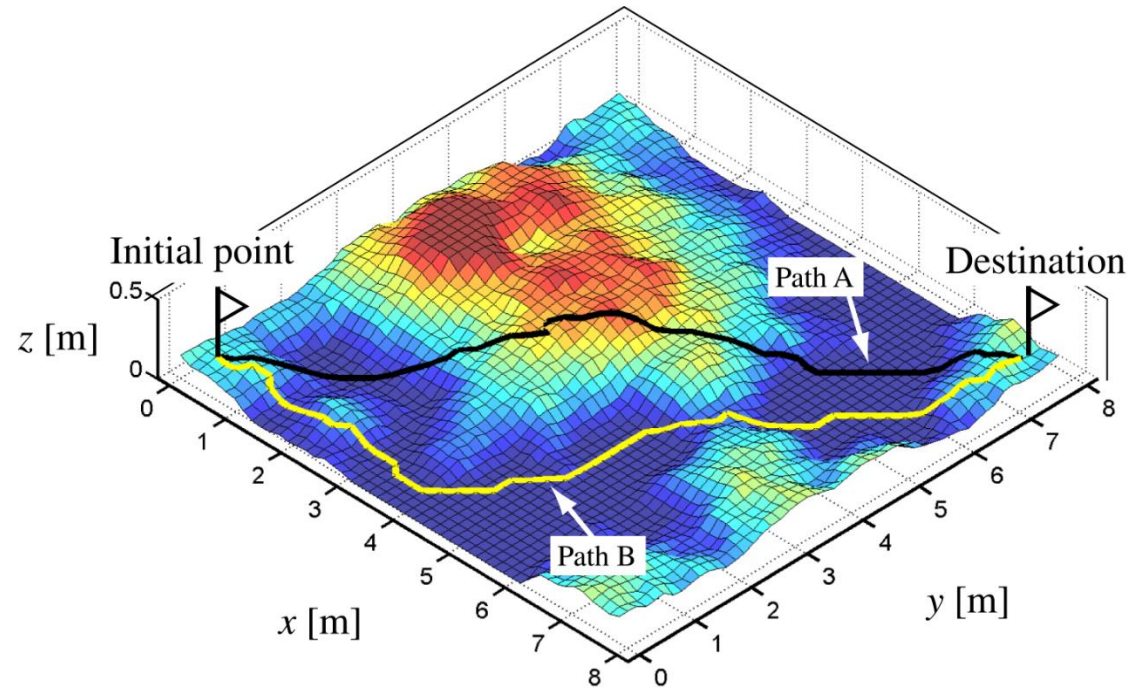
6 – Path planning

Advanced Methods for Mapping and Self-localization in Robotics
MPC-MAP

Tomas Lazna
Brno University of Technology
2025

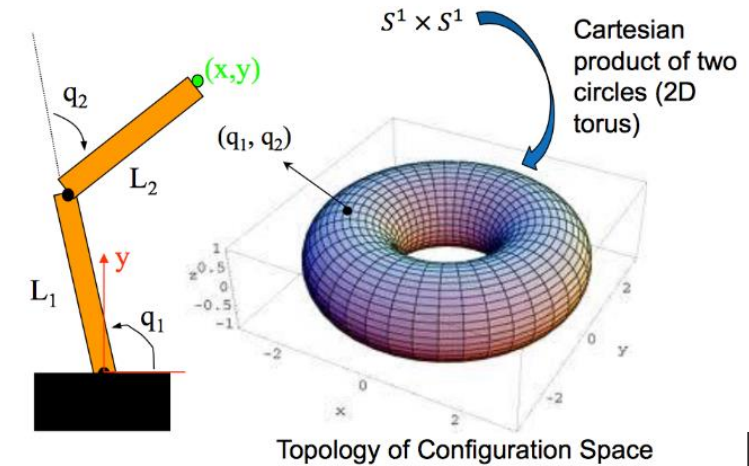


- What is path planning for?
 - Map, obstacles
 - Self-localization
 - Where to go?
 - How to get there?
 - What is the best way?
 - What robot do I have?
- Path planning \approx motion planning

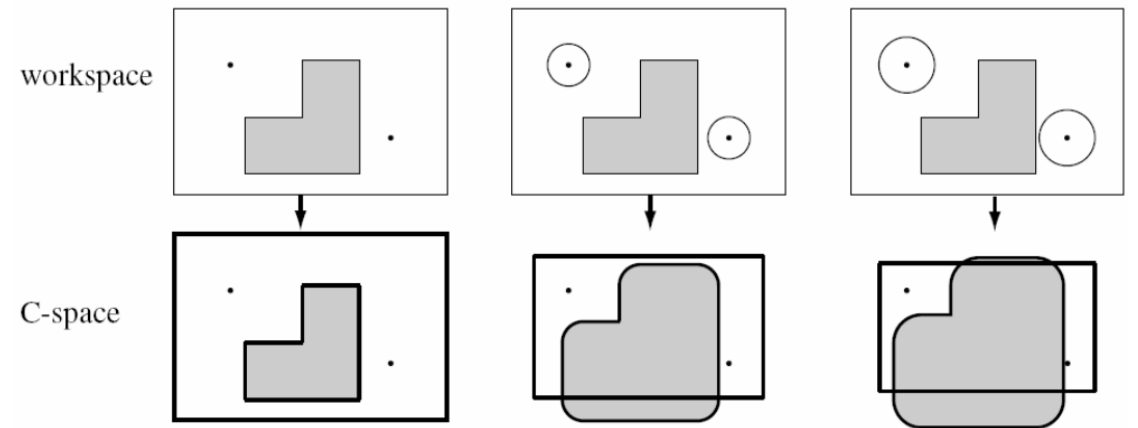


[1]

- Configuration
 - A complete specification of the position of every point in the system
- Configuration space = C-space
 - Space of all possible configurations
 - A manifold in higher-dimension Cartesian space
- C-obstacle
 - Set of configurations where the robot collides with a workspace obstacle or with itself
- Free configuration
 - Null intersection of the configuration with workspace obstacles
 - Semi-free = config. touches an obstacle



[1]



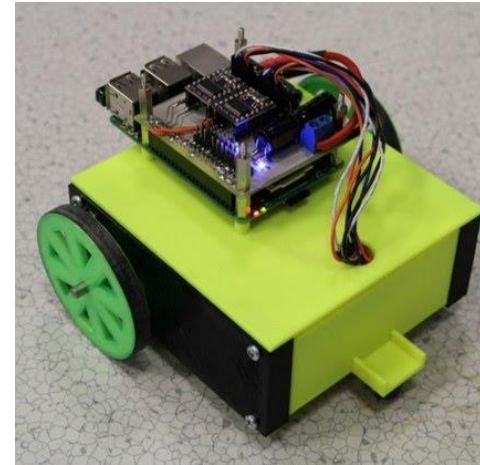
[2]



- Inputs
 - Initial and goal configurations
 - Workspace representation and obstacles
 - Allowed actions with associated costs
 - (Geometric) Description of robot
- Outputs
 - Collision-free sequence of configurations
(optimality: the shortest / the fastest / the least computationally demanding / ...)
 - Cost of the sequence



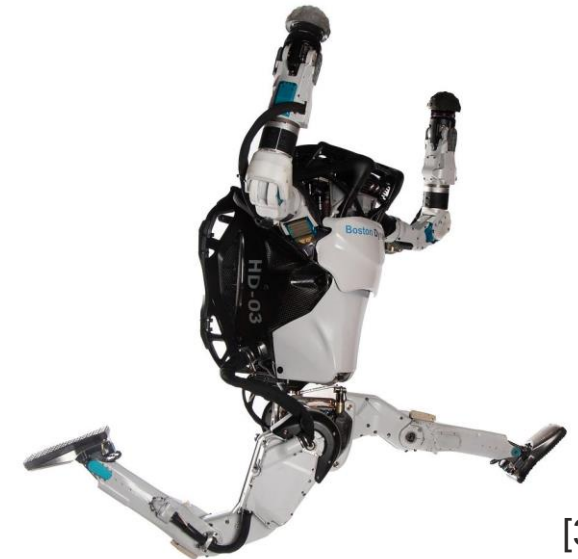
- Rigid vs. articulated robots
 - Rigid: all parts of a robot have constant mutual positions
 - Mobile robots with differential drive
 - Unmanned aircraft systems (drones)
 - Articulated: A robot consist of parts connected by rotary joints
 - Industrial manipulators
 - Legged robots



[1]



[2]



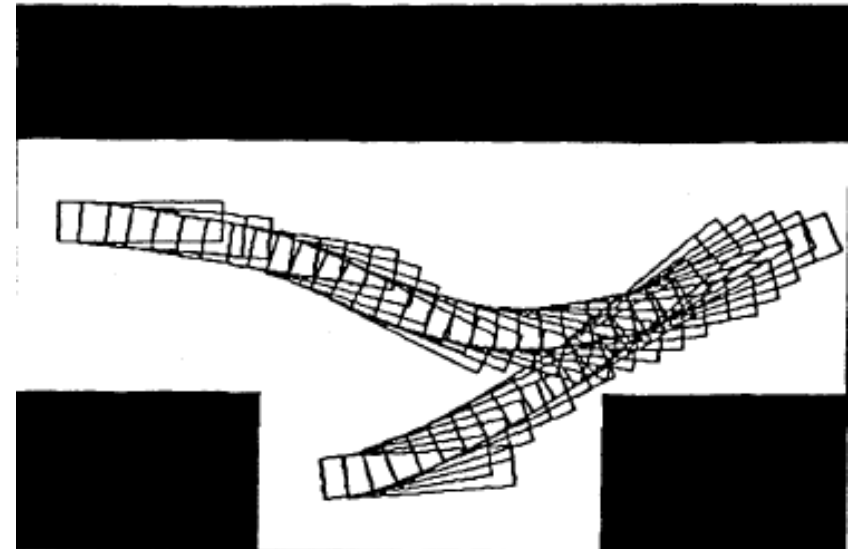
[3]



- Holonomic vs. nonholonomic robots
 - Holonomic: Has the same or higher number of controllable DOFs than their total number in configuration space
 - Robots with omnidirectional drive
 - Nonholonomic: lower number of controllable DOFs
 - Cars (Ackermann drive)



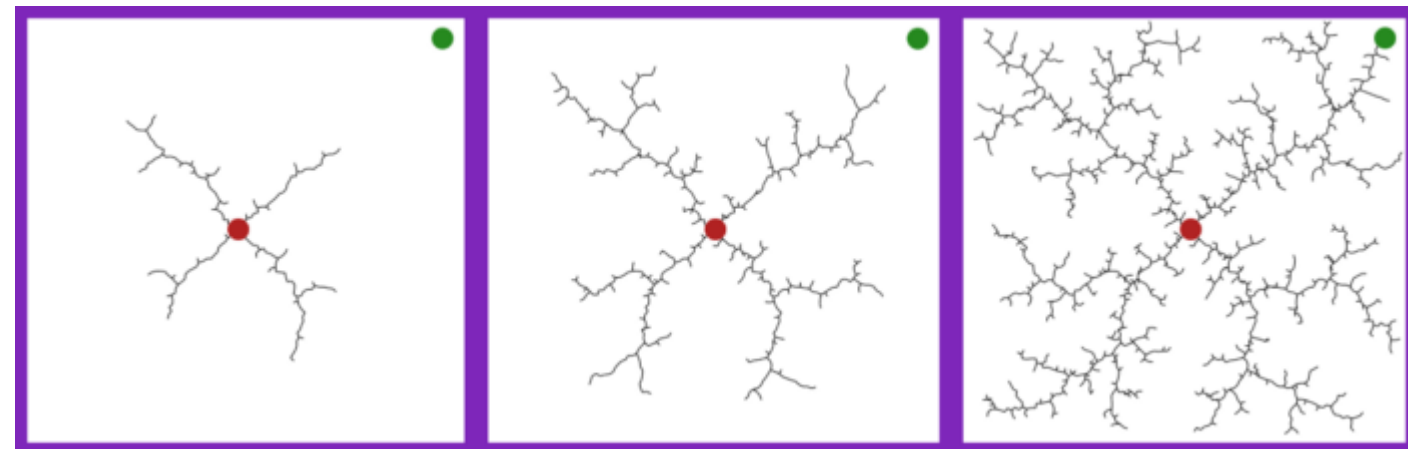
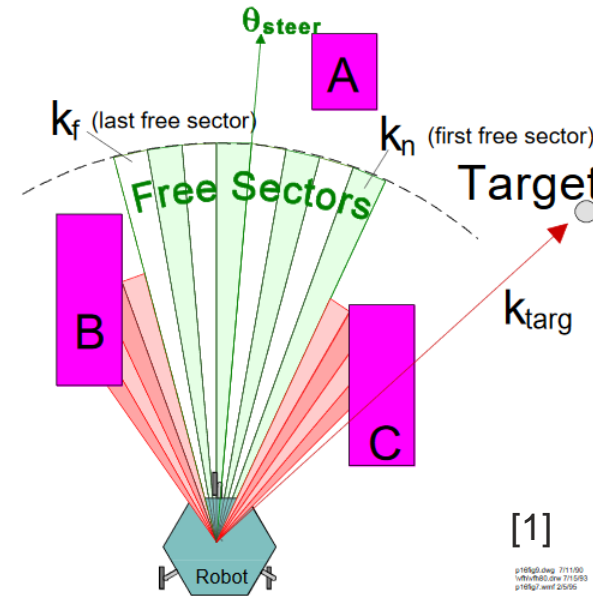
[1]



[2]



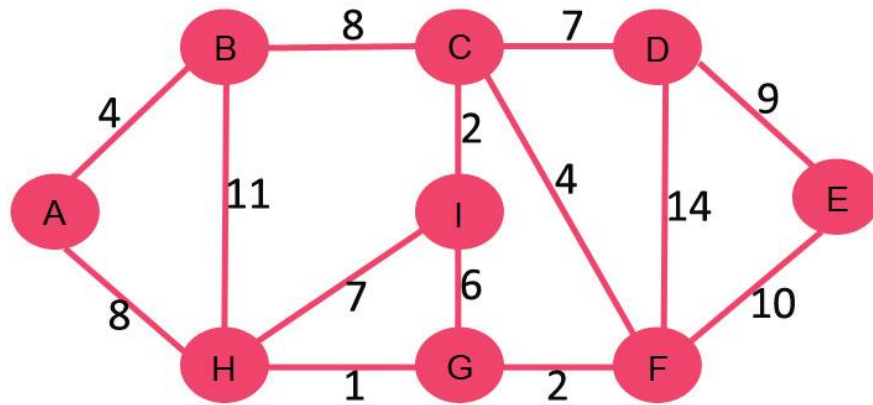
- Local planners
 - Are limited by range of sensors
 - Provide a *direction* to move in
 - Do not guarantee finding the goal
- Global planners
 - Work with the whole configuration space
 - Provide complete path to the goal
 - Find an *optimal* solution
- Stochastic planners
 - Monte Carlo methods
 - Fast (especially in higher dimensions)
 - Not suitable for all scenarios
 - Does not find an optimal path



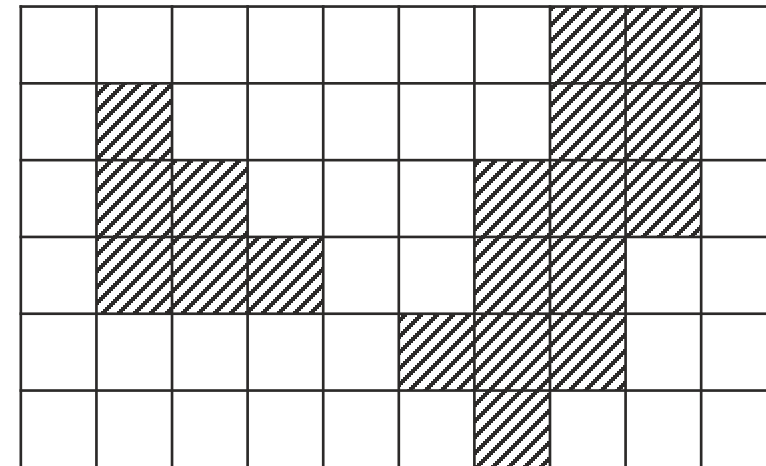
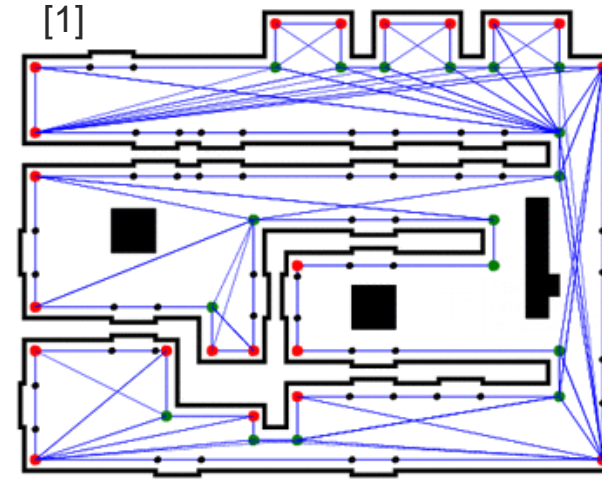


- Types of maps

- Continuous vs. discrete
- Metric vs. topological
- Planning algorithms are designed to search in a graph (topological map)



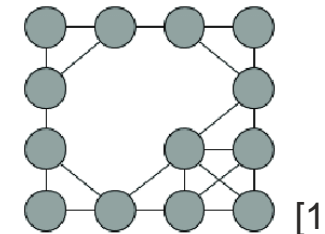
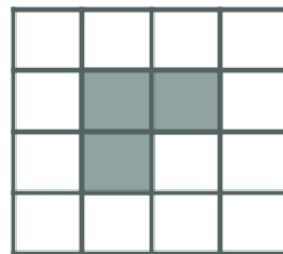
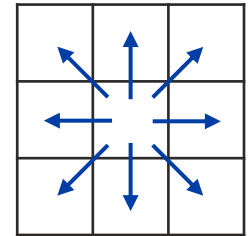
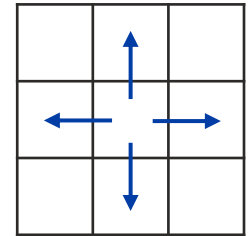
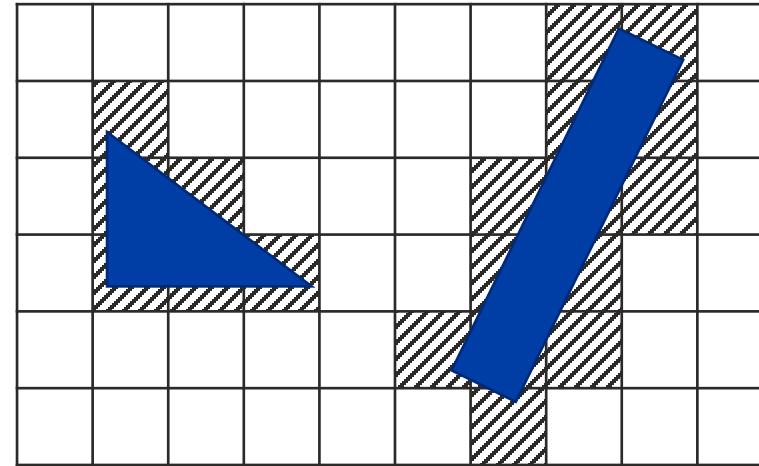
[2]



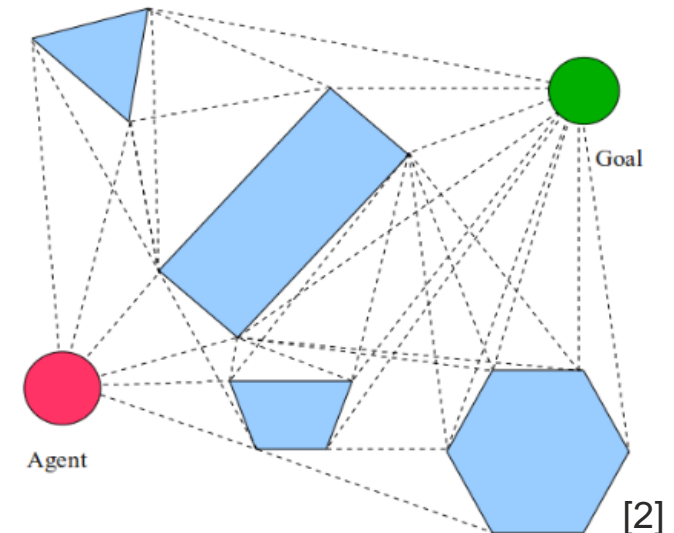


Converting maps to graphs

- Continuous → discrete
 - Choose a cell size
 - Cells with not-null intersection with an obstacle are occupied and vice versa
 - Occupancy grid is created
- Discrete → graph
 - Cell adjacency graph
 - 4-connected or 8-connected
- Continuous → graph
 - Visibility graph
 - Voronoi diagrams



[1]



[2]

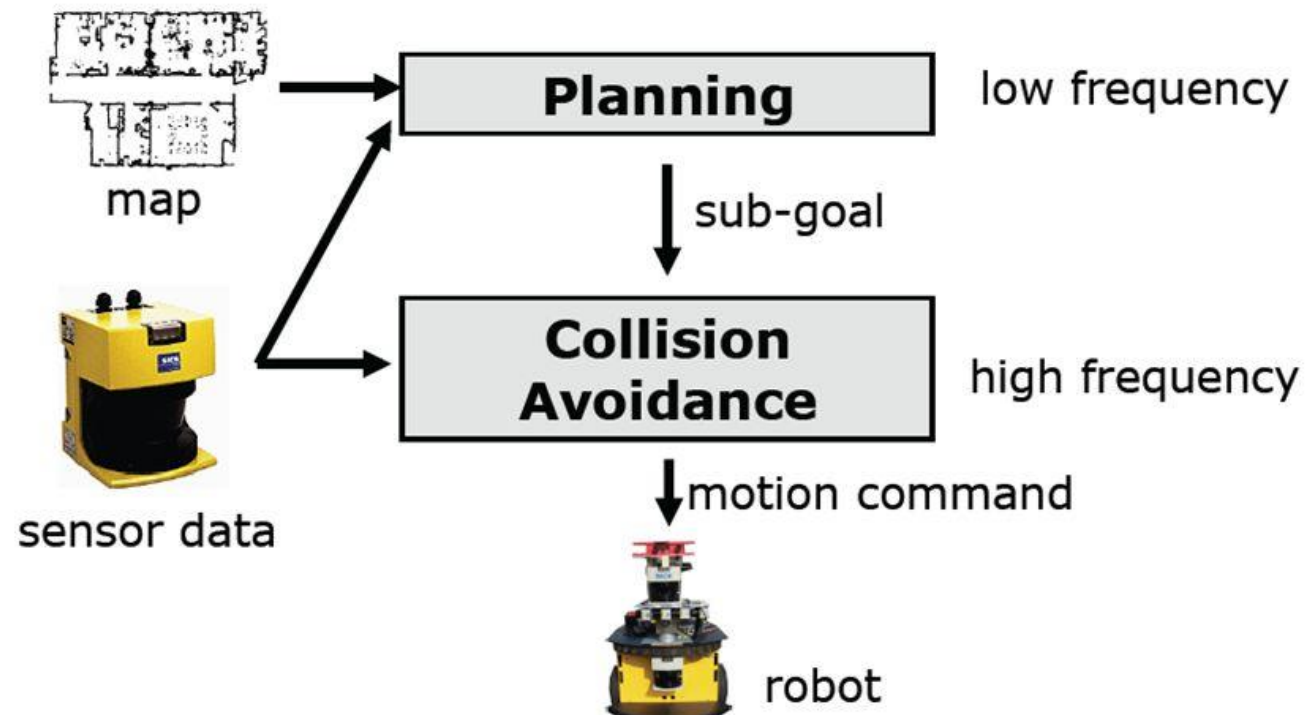


- Measuring the performance of search algorithms
 - Completeness: If there is a solution, it is found
 - Optimality: The provided solution is the best in terms of some criterion
 - Time complexity: Time required to find the solution
 - Space complexity: Memory and/or other assets required to find the solution

- What other attributes do we require?
 - Take uncertainties into account
 - React to dynamic obstacles quickly
 - Guarantee safety



Classic Two-Layered Architecture for Mobile Robots



[1]



1	$Q.Insert(x_0)$
2	$x_0.visited = \text{true}$
3	while $\text{length}(Q) > 0$ do
4	$x = Q.GetFirst()$
5	if $x \in X_G$
6	return SUCCESS
7	foreach $u \in U(x)$
8	$x' = f(x, u)$
9	if $\neg x'.visited$
10	$x'.visited = \text{true}$
11	$Q.Insert(x')$
12	else
13	Resolve duplicity of x'
14	return FAILURE

Q	Priority queue
x_0	Initial state
X_G	Set of goal states
x, x'	States
u	State transition
$U(x)$	Set of all transitions in state x

Various planners differ especially in a type of the employed priority queue Q and a mechanism for resolving state conflicts (line 13).



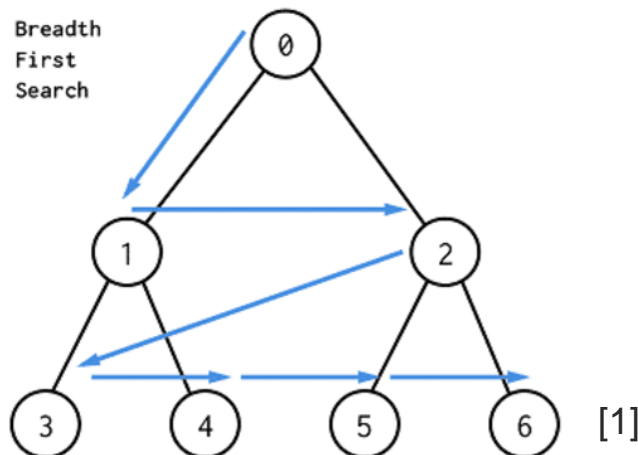
- Unvisited
 - This state has not been encountered yet
- Dead (Closed)
 - Has been visited, and every possible next state has also been visited
 - Can no longer contribute to the search
 - State picked from the priority queue
- Alive (Open)
 - Has been visited, but possibly has unvisited next states
 - State *is* in the priority queue



- No information on goal is provided
- Configuration space is systematically explored until a goal is encountered

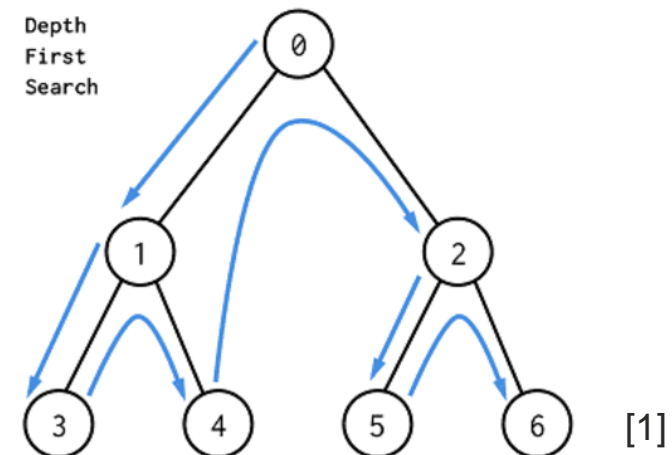
- Breadth-first search

- A FIFO type priority queue \rightarrow all states in depth k are explored before those in depth $k + 1$
- Complete
- Optimal in case of equal transition costs



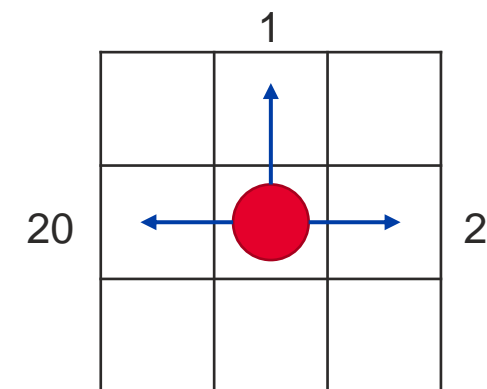
- Depth-first search

- A LIFO type priority queue \rightarrow whole branch is explored to the end of graph before a new branch is opened
- Not complete in infinite C-spaces
- Not optimal



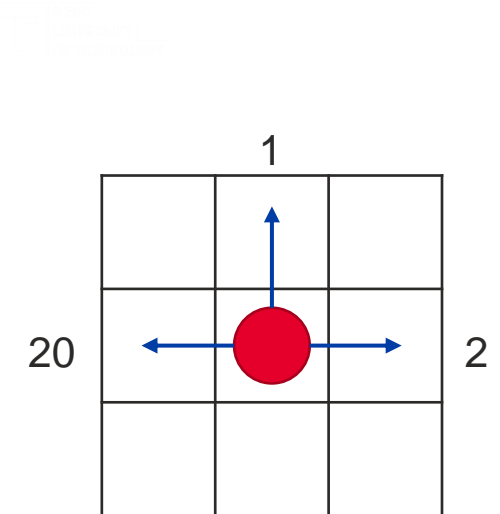
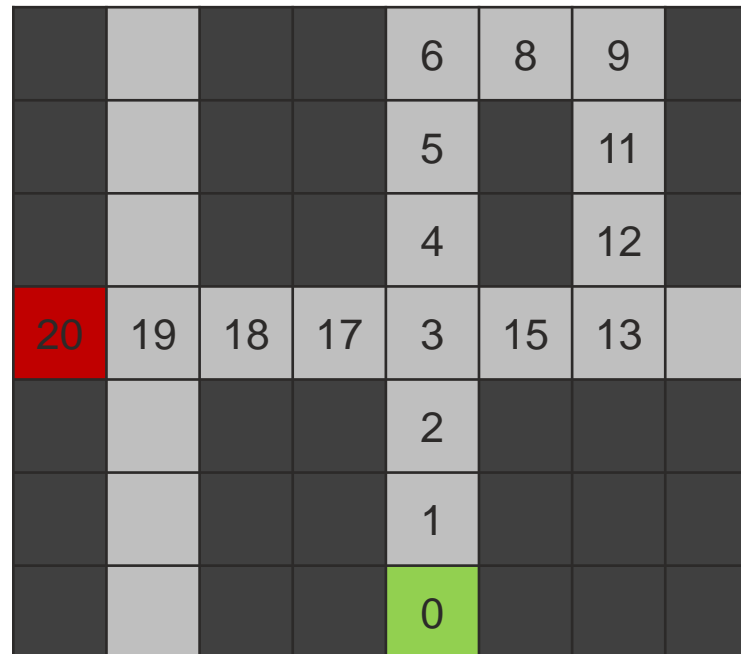
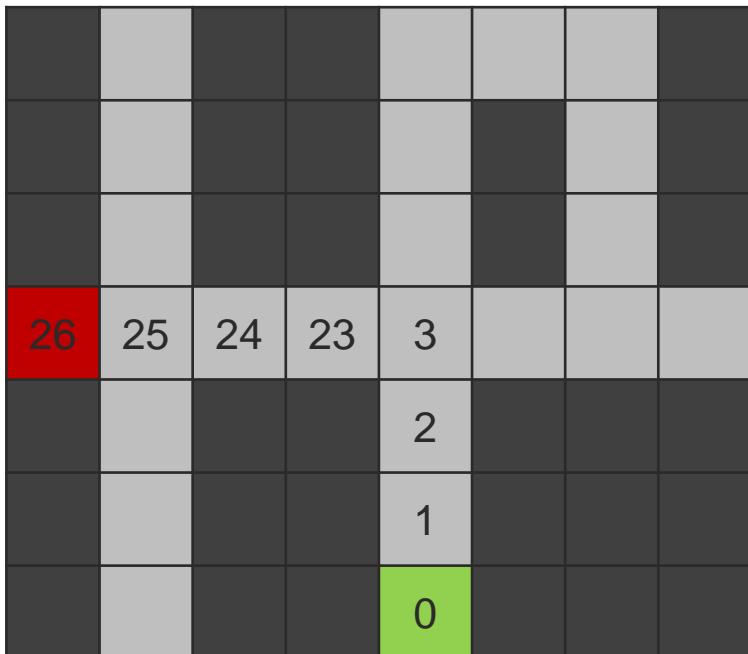


- Some actions can be more „expansive“ than others in terms of complexity
- Examples:
 - It takes more energy to go up a steep slope
 - It is possible to go faster on a road than on an agricultural field
 - Turning right takes less time than turning left in city traffic
- To represent such circumstances, we can assign various costs to:
 - Graph edges (what is the cost of going to adjacent cell)
 - Actions (what are the costs of different operations performed by a robot)





- Example: expansive left turns

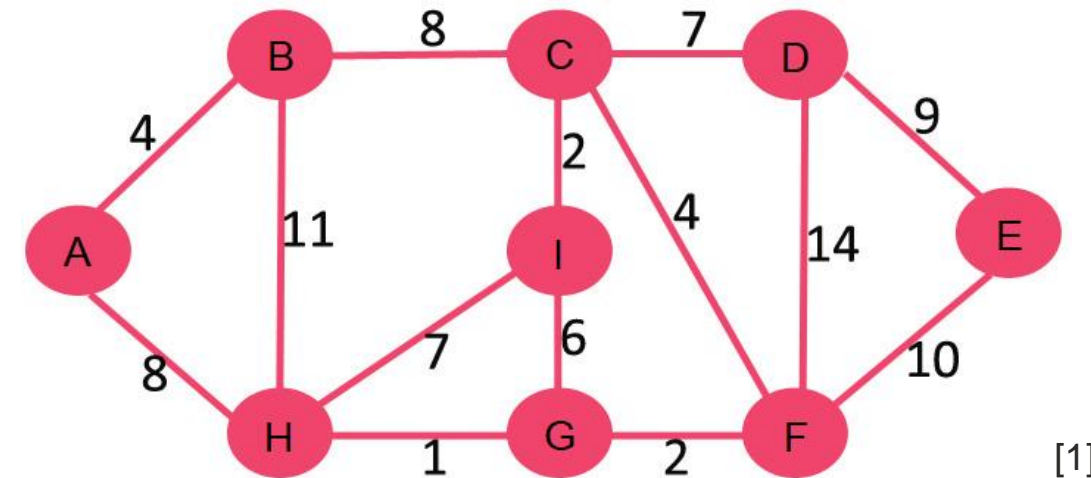




- The priority queue is ordered by the cost function $g(x)$ (aka cost-to-come)
- Cost function of a state is equal to sum of costs of all transitions and actions necessary to reach the state from the initial configuration

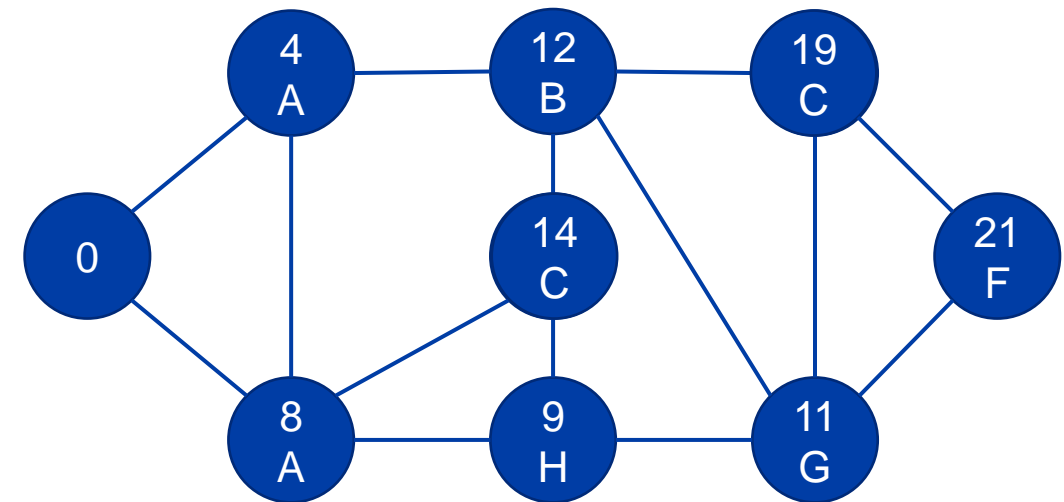
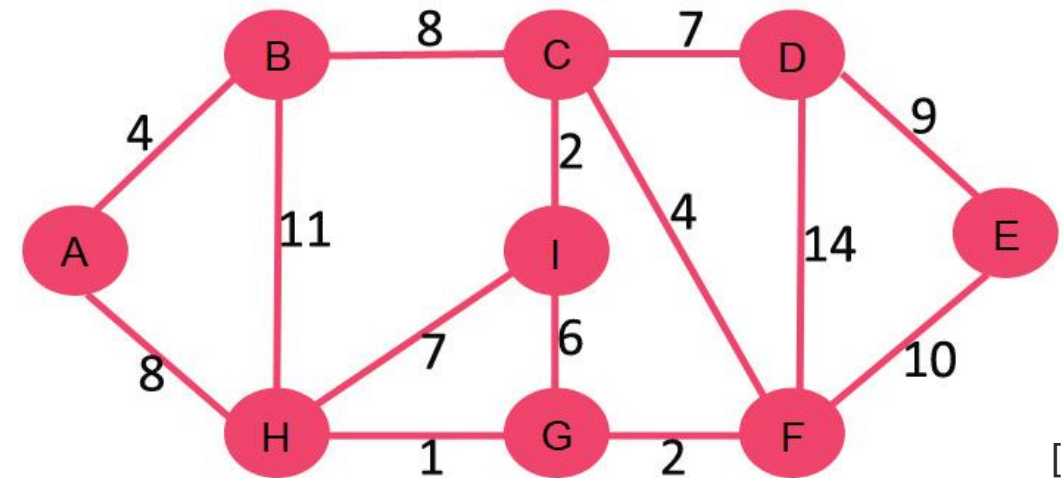
$$g(x_n) = \sum_{i=0}^{n-1} C(x_i \rightarrow x_{i+1})$$

- Each explored state is assigned value of cost function and respective action the state has been reached
- If an **alive** state is revisited more efficiently, its corresponding cost function and action are updated (dead states are **ignored**)



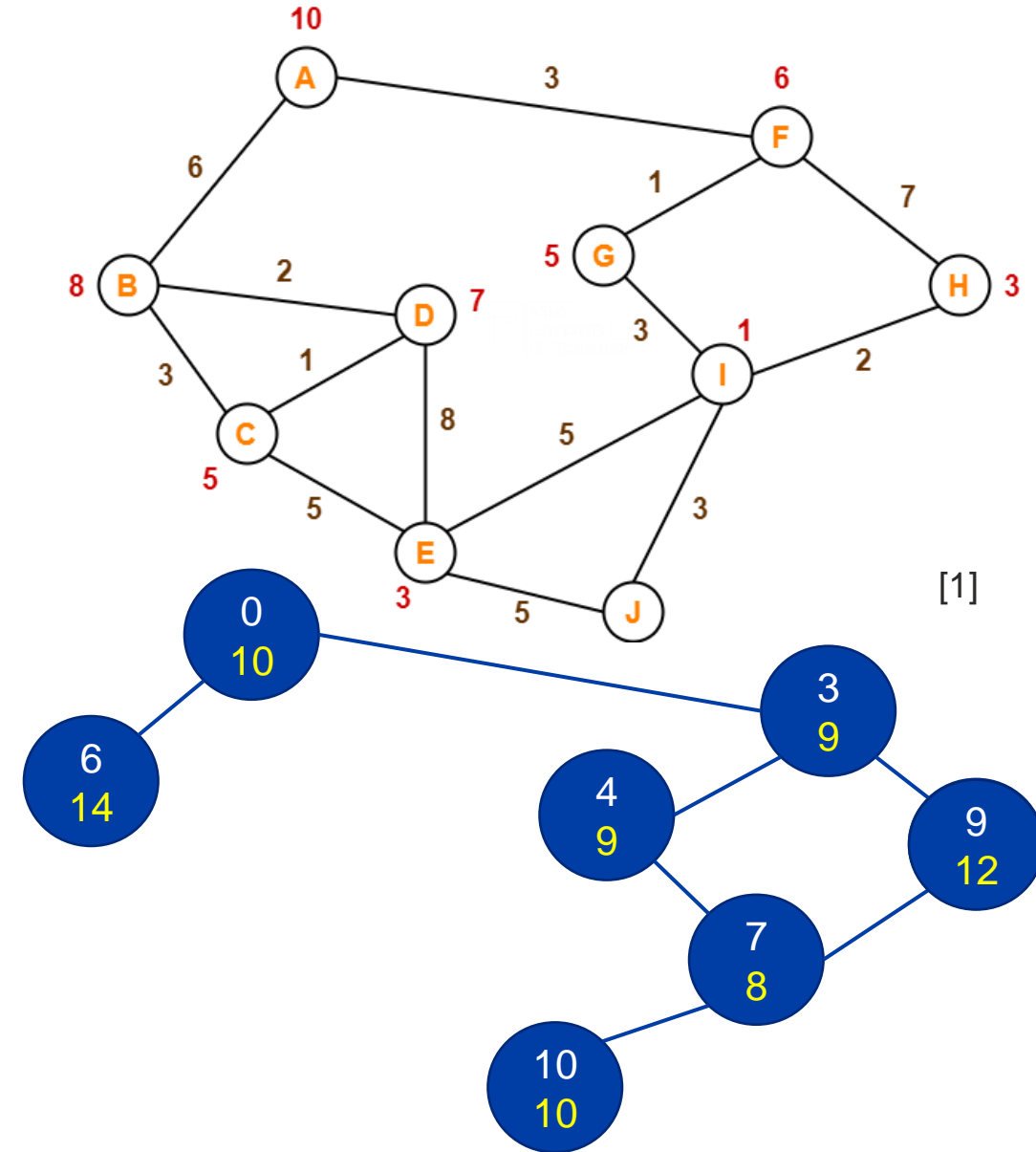


- The priority queue is ordered by the cost function $g(x)$ (aka cost-to-come)
- Dijkstra's algorithm = full graph is expanded \rightarrow optimal path from one state to each other



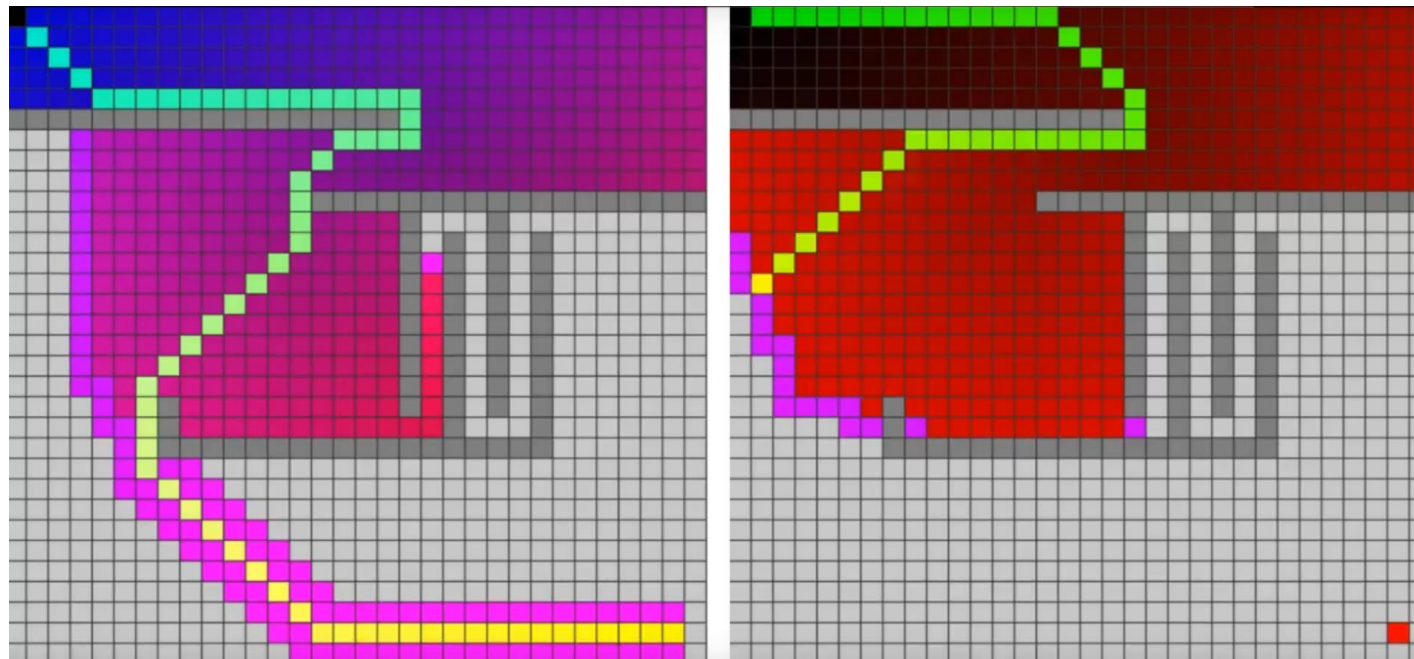


- A heuristic function $h(x)$ is introduced (aka cost-to-go) – an **underestimated** guess of cost necessary to reach the goal, e.g., Euclidean distance $h(x_n) = \|x_G - x_n\|$
- The priority queue is ordered by function
$$f(x) = g(x) + h(x)$$
- Revisited states: Same as Dijkstra
- States closer to the goal are preferred in the exploration → smaller part of the graph is explored compared to the greedy search
- Critical condition to ensure optimality: $\forall x: h(x) \leq h^*(x)$ where $h^*(x)$ is the actual cost from x to the goal
- Works only for non-negative edges





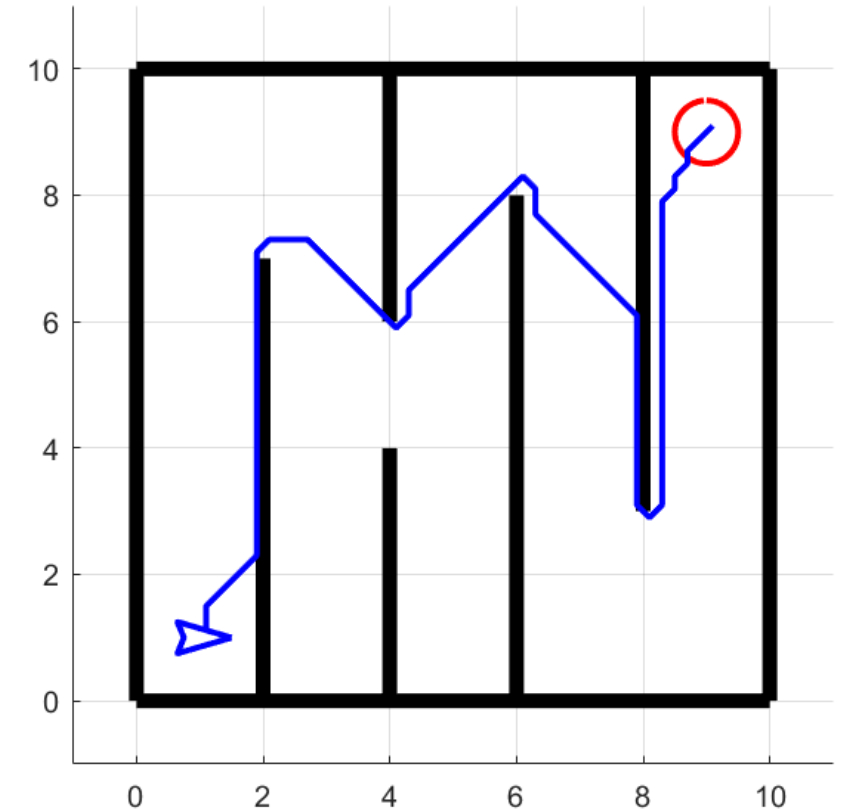
- A*
 - Informed search – use of heuristics
 - Finds optimal sequence between two configurations
 - What is the best heuristics?
- Dijkstra
 - Uniformed search
 - Find optimal sequence between one and all other configurations
 - The most accurate estimate for the A*



[1]

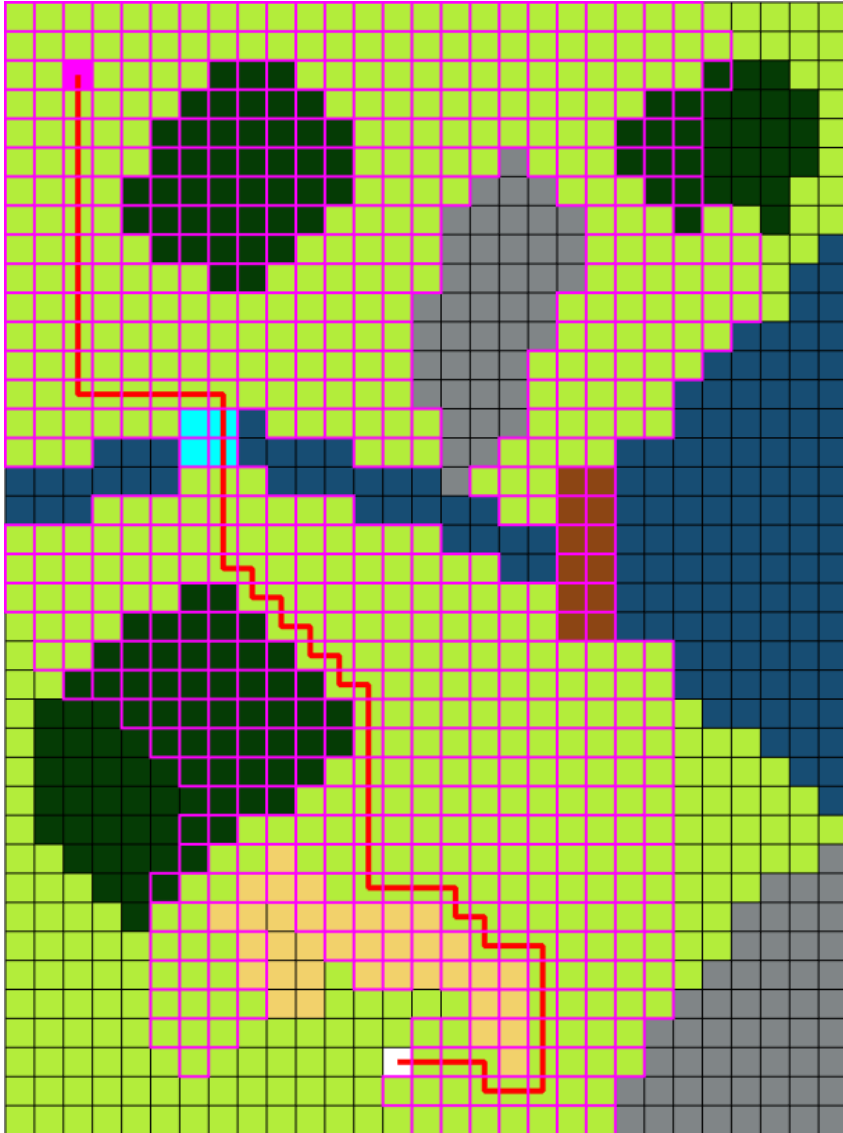


- Possible problems
 - Robot is not localized correctly
 - The shortest path is usually found in the vicinity of obstacles
 - Trajectory is aligned with the grid structure
- Best-first (greedy) search
 - Uses only the heuristics (cost-to-go) to order the priority queue
 - Does not guarantee optimal solution
 - Can be faster than A* but provides generally worse results – one has to pay for being too greedy







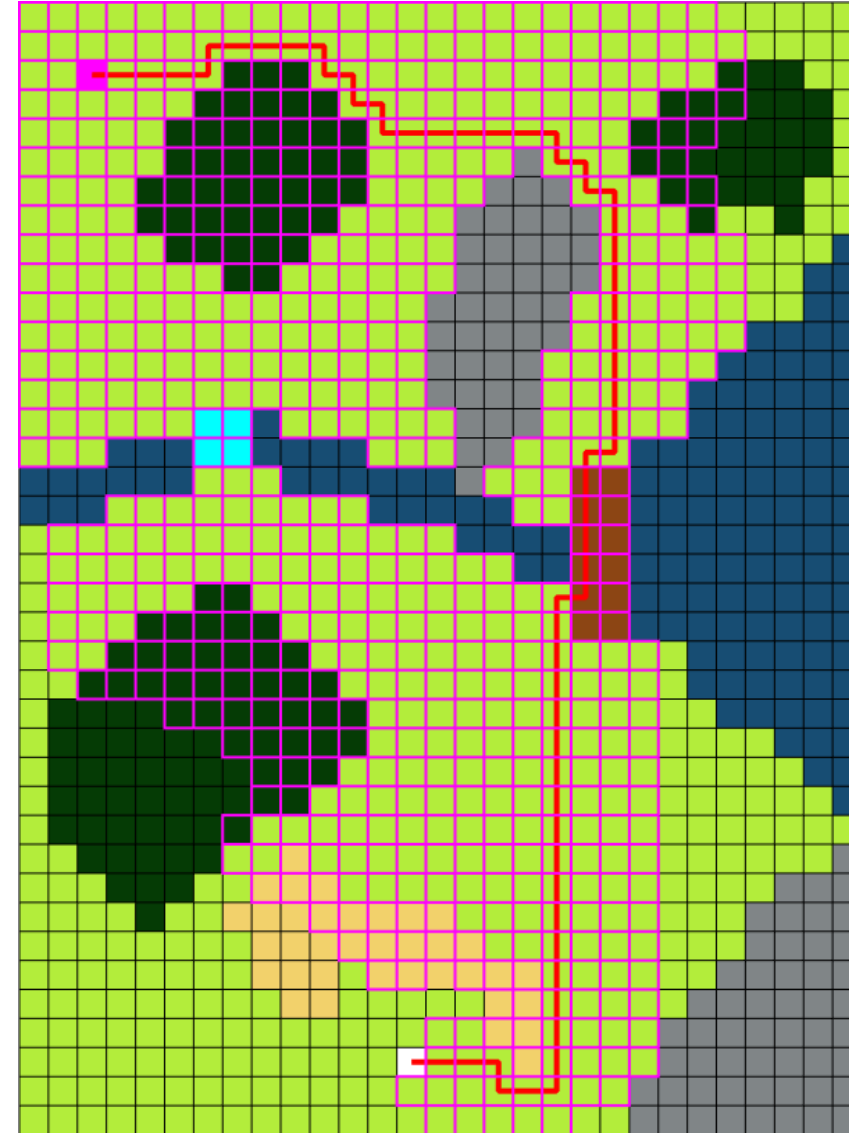


A* – examples







Grass = 1
Forest = 3
Sand = 7
Rocks = Inf
Water = Inf
Crossing = 6
Bridge = 2

Start 
Goal 
Visited 
Path 

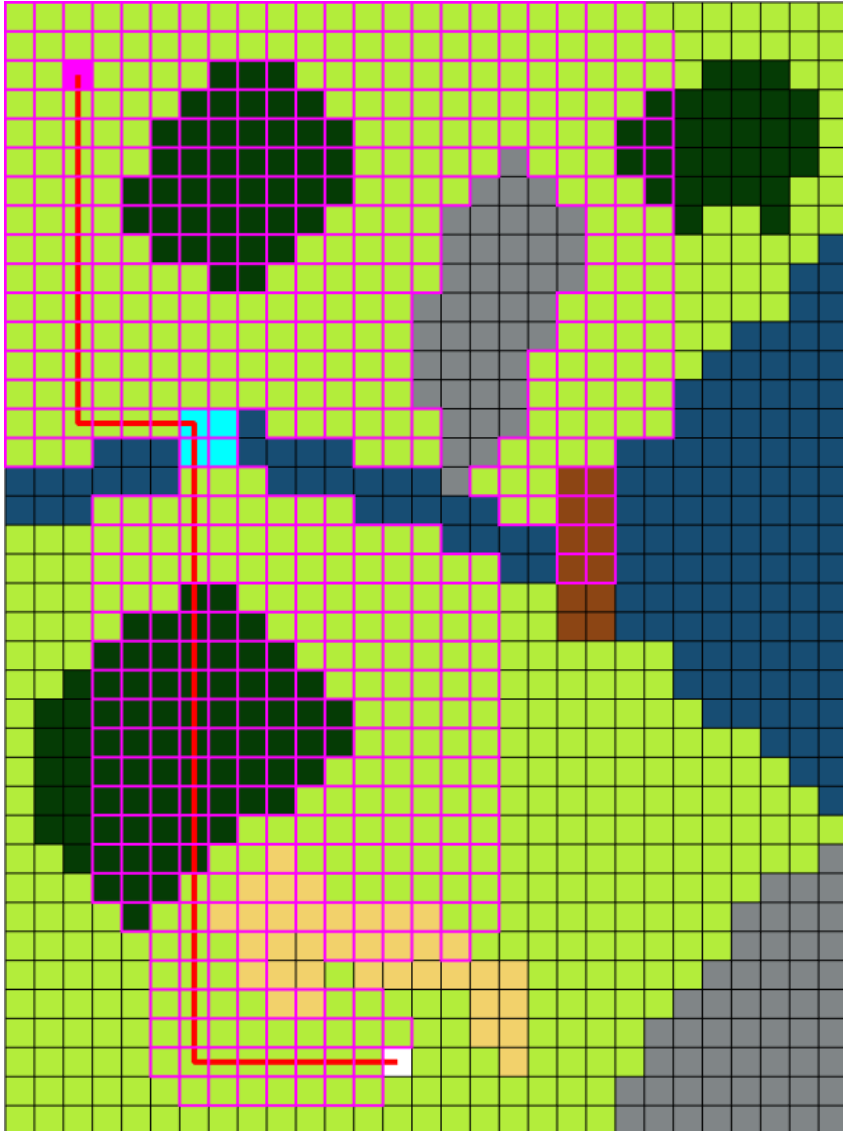


Grass = 1
Forest = 3
Sand = 5
Rocks = Inf
Water = Inf
Crossing = 8
Bridge = 2





Start 
Goal 
Visited 
Path 

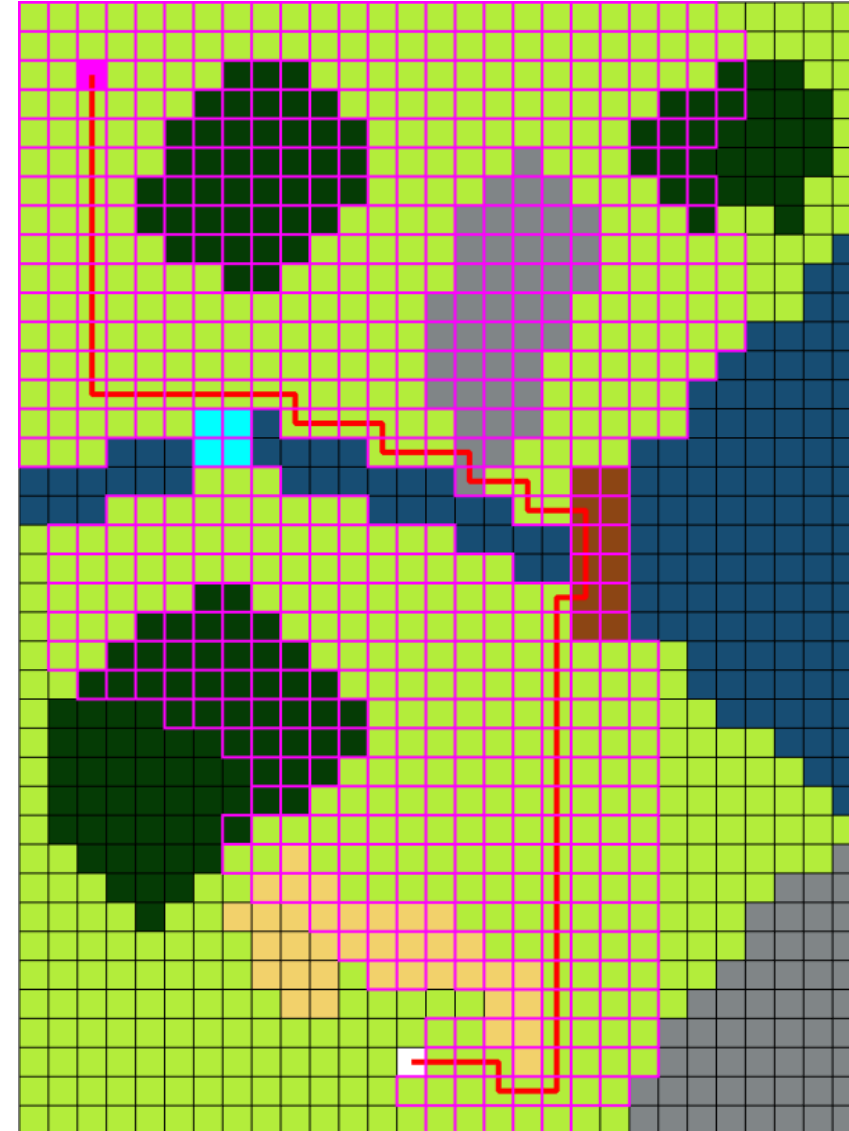


A* – examples







Grass = 1
Forest = 1.5
Sand = 5
Rocks = Inf
Water = Inf
Crossing = 8
Bridge = 2

Start 
Goal 
Visited 
Path 

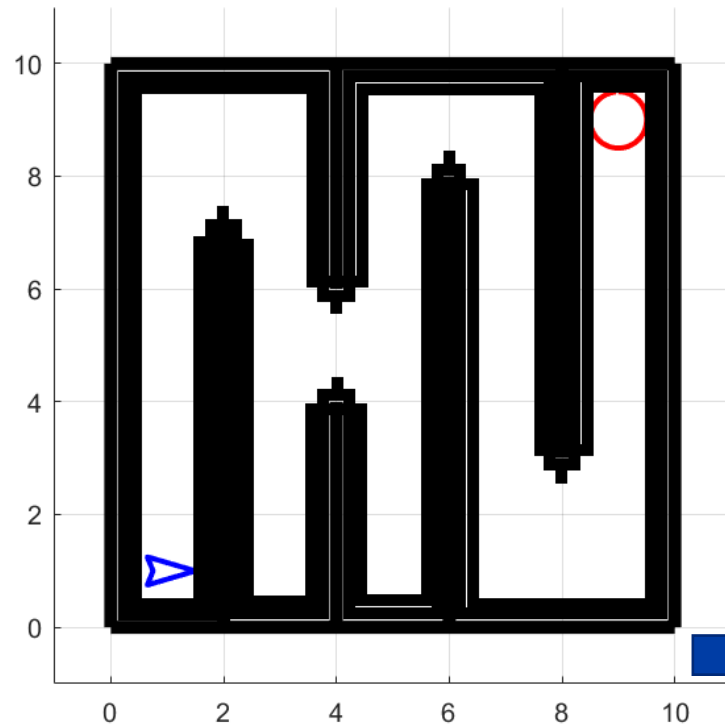


Grass = 1
Forest = 3
Sand = 5
Rocks = 3
Water = Inf
Crossing = 8
Bridge = 2

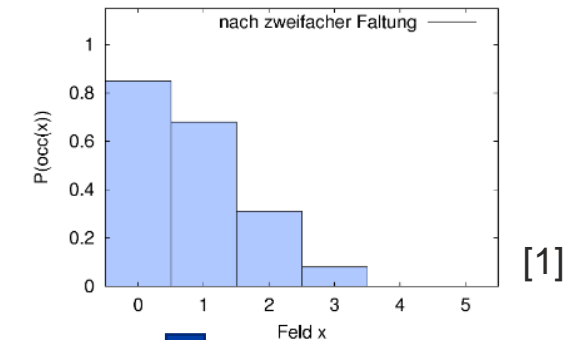
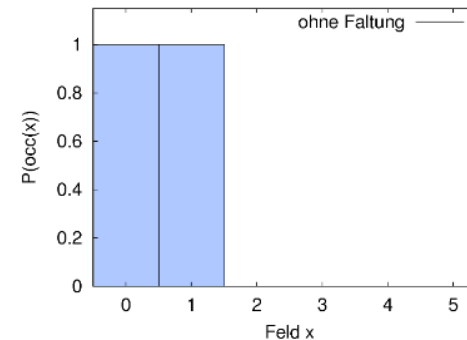
Start 
Goal 
Visited 
Path 



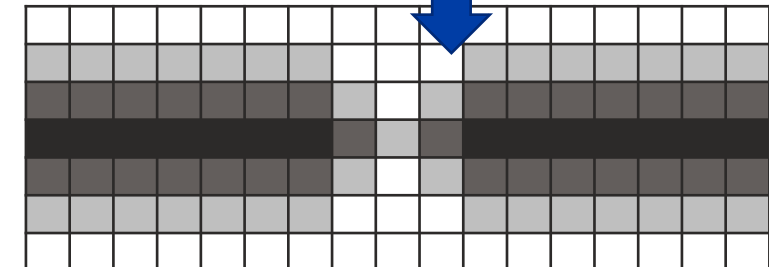
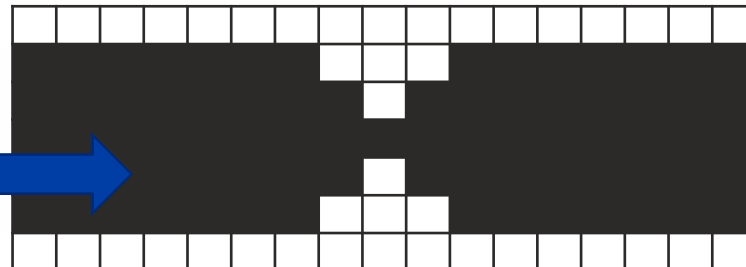
- Dilating obstacles
 - Works well in open spaces
 - Inconvenient for narrow passages



- Applying convolution
 - Keeps passages open
 - Cells closer to obstacles have higher cost
 - Disk-shaped or diamond-shaped kernel



[1]





- A way to deal with stochastic environments
- For each cell, a *value* is computed and optimal policy is set
- Iterative process

	16	17	18	19	20	21	22		20	
	15								19	
	14		12	13	14	15	16	17	18	
	13		11							
	12	11	10	9	8	7	6	7	8	
	13						5			
	14	15	16	17	18		4	3	2	
	15				19		5		1	
	16		22	21	20		6		0	

Input: map, value (> 0), policy (= null), actions, goal, cost_step

1	$value[goal[1]][goal[2]] = 0$
2	$change = true$
3	while $change$ do
4	$change = false$
5	foreach (x, y) st. $map[x][y]$ is empty
6	foreach a in $actions$
7	$x2 = x + a[1]$
8	$y2 = y + a[2]$
9	if $(x2, y2)$ is in range && $map[x2][y2]$ is empty
10	$v2 = value[x2][y2] + cost_step$
11	if $v2 < value[x][y]$
12	$change = true$
13	$value[x][y] = v2$
14	$policy[x][y] = a$



- A way to deal with stochastic environments
- For each cell, a *value* is computed and optimal policy is set
- Iterative process

	16	17	18	19	20	21	22		20	
	15								19	
	14		12	13	14	15	16	17	18	
	13		11							
	12	11	10	9	8	7	6	7	8	
	13						5			
	14	15	16	17	18		4	3	2	
	15				19		5		1	
	16		22	21	20		6		0	

Input: map, value (> 0), policy (= null), actions, goal, cost_step

```
1  value[goal[1]][goal[2]] = 0
2  change = true
3  while change do
4      change = false
5      foreach (x, y) st. map[x][y] is empty
```

	v	<	<	<	<	<	<		v	
	v								v	
	v		v	<	<	<	<	<	<	
	v		v							
	>	>	>	>	>	>	v	<	<	
	^						v			
	^	<	<	<	<		>	>	v	
	^				^		^		v	
	^		>	>	^		^		*	

ns

in range && map[x2][y2] is empty

lue[x2][y2] + cost_step

value[x][y]

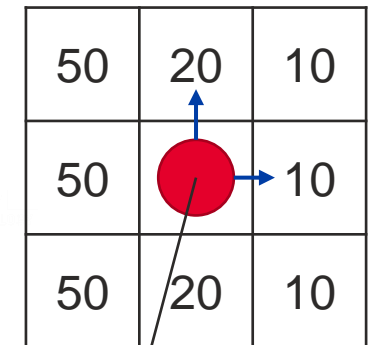
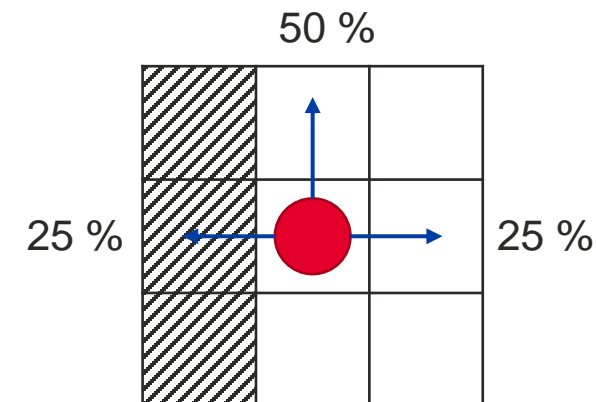
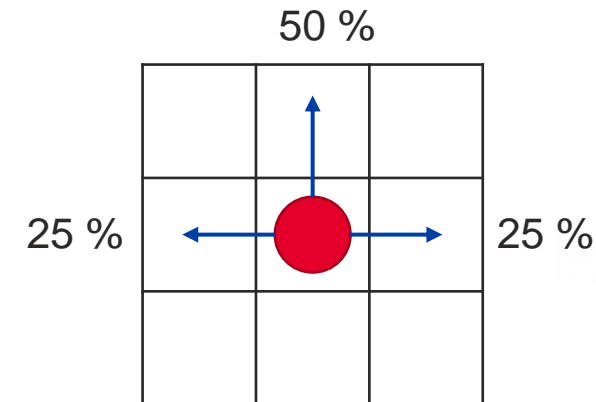
ange = true

lue[x][y] = v2

lity[x][y] = a

- Why not use Dijkstra to compute the values?
- Iterative approach can be extended to take the uncertainty of motion into account (stochastic actions)
- DP with stochastic actions keeps obstacle clearance

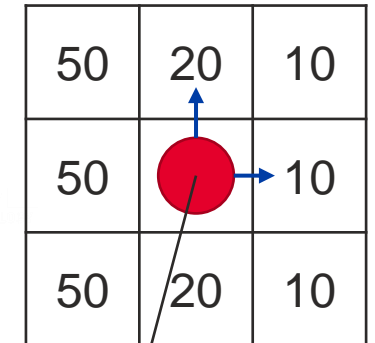
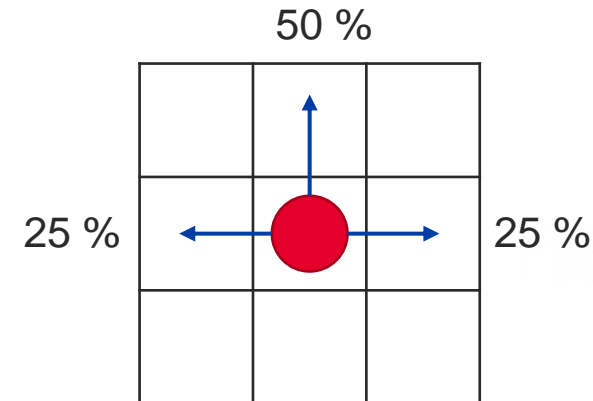
99	99	99	99	99	99	99	99	99	99	99	99	99
99	91	89	91	99	67	57	61	99	56	48	60	99
99	87	85	87	99	54	47	43	39	37	37	43	99
99	83	80	82	99	49	44	41	37	33	32	36	99
99	79	76	78	99	49	45	42	37	30	28	30	99
99	75	73	73	99	51	49	49	99	26	24	26	99
99	72	69	66	62	55	53	54	99	22	20	22	99
99	71	68	64	61	58	56	59	99	19	16	19	99
99	73	69	66	63	61	61	65	99	18	10	18	99
99	80	73	68	66	64	66	75	99	30	0	30	99
99	99	99	99	99	99	99	99	99	99	99	99	99



$$V_{UP} = 1 + 50 \% \cdot 20 + 25 \% \cdot 50 + 25 \% \cdot 10 = 26$$

$$V_{RIGHT} = 1 + 50 \% \cdot 10 + 25 \% \cdot 20 + 25 \% \cdot 20 = 14$$

- Why not use Dijkstra to compute the values?
- Iterative approach can be extended to take the uncertainty of motion into account (stochastic actions)
- DP with stochastic actions keeps obstacle clearance



99	99	99	99	99	99	99	99	99	99	99	99	99
99	91	89	91	99	67	57	61	99	56	48	60	99
99	87	85	87	99	54	47	43	39	37	37	43	99
99	83	80	82	99	49	44	41	37	33	32	36	99
99	79	76	78	99	49	45	42	37	30	28	30	99
99	75	73	73	99	51	49	49	99	26	24	26	99
99	72	69	66	62	55	53	54	99	22	20	22	99
99	71	68	64	61	58	56	59	99	19	16	19	99
99	73	69	66	63	61	61	65	99	18	10	18	99
99	80	73	68	66	64	66	75	99	30	0	30	99
99	99	99	99	99	99	99	99	99	99	99	99	99

	v	v	v		v	v	v		v	v	v	
	>	v	<		>	v	v	v	v	v	<	
	>	v	<		>	>	>	>	v	v	<	
	>	v	<		>	>	^	^	>	v	<	
	>	v	<		>	^	<		>	v	<	
	>	>	v	v	>	^	<		>	v	<	
	>	>	>	>	^	^	<		>	v	<	
	>	^	^	^	^	^	^		>	*	<	

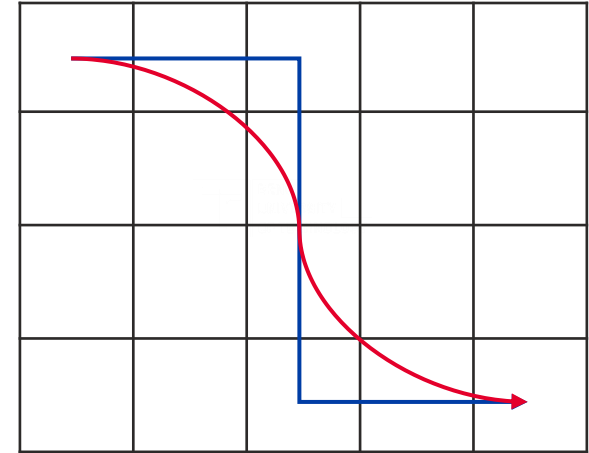
$$V_{UP} = 1 + 50\% \cdot 20 + 25\% \cdot 50 + 25\% \cdot 10 = 26$$

$$V_{RIGHT} = 1 + 50\% \cdot 10 + 25\% \cdot 20 + 25\% \cdot 20 = 14$$



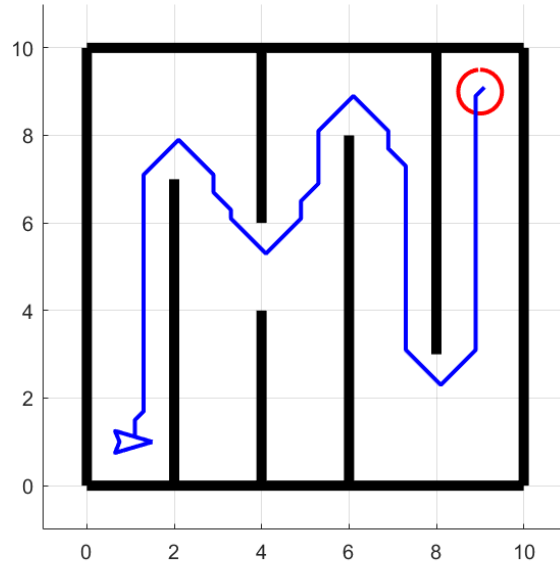
- Grid-based planners yield paths that are aligned with the grid and do not consider kinematic constraints of the robot
- In actual scenario, we would like the path to be *smooth*, i.e., it does not contain sharp turns
- Smoothing algorithm:
 1. Let us have path $X = \{x_i\}_{i=1}^N$
 2. Create copy of the original path $Y = X$
 3. Optimize:
 - $(x_i - y_i)^2 \rightarrow \min$ $y_i = y_i + \alpha(x_i - y_i)$
 - $(y_i - y_{i+1})^2 \rightarrow \min$ $y_i = y_i + \beta(y_i - y_{i+1})$
- Iterative method:

$$y_i^{(k+1)} = y_i + \alpha(x_i - y_i) + \beta(y_{i-1} + y_{i+1} - 2y_i), \quad i = 2 \dots N - 1$$

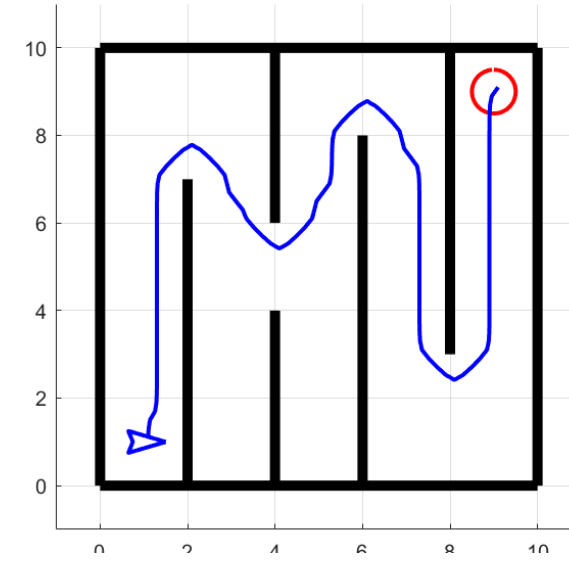




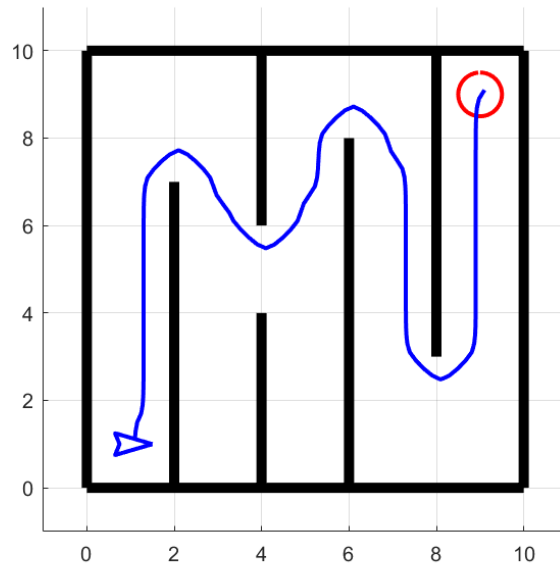
$$\alpha = 0, \beta = 0$$



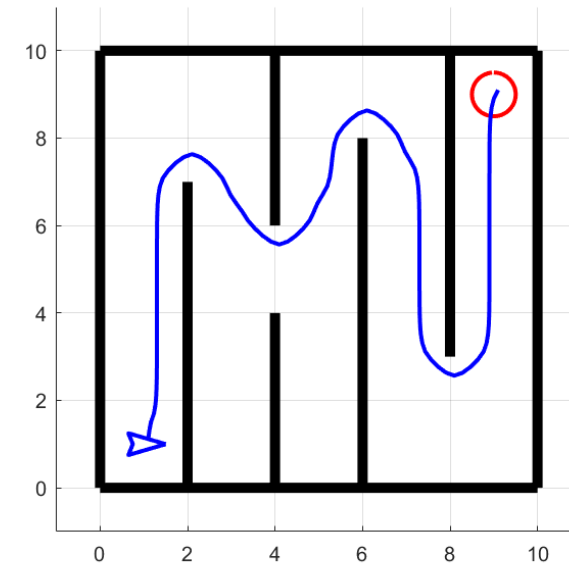
$$\alpha = 0.5, \beta = 0.25$$



$$\alpha = 0.5, \beta = 0.5$$



$$\alpha = 0.25, \beta = 0.5$$





- Path (motion) planning
 - Given map of the workspace and description of the robot, find collision-free sequence of steps (configurations) from the initial to the goal point.
 - Completeness and optimality
 - Assign costs to configuration transitions
 - Usually reduced to a search in a graph
- Planning based on A*
 - Convert map to occupancy grid and connect adjacent cells by graph edges
 - Sort open nodes in queue by sum of the cost-to-come and the cost-to-go (heuristic) function
 - Faster than uninformed search
 - Finds optimal path between two points
- Path smoothing – iterative gradient algorithm



[1]



Tomas Lazna

tomas.lazna@ceitec.vutbr.cz

Brno University of Technology
Faculty of Electrical Engineering and Communication
Department of Control and Instrumentation



Robotics and AI Research Group