Discrete Motion Planning

Lecture 8 – Thursday November 23, 2016

Objectives

When you have finished this lecture you should be able to:

- Recognize what a **motion plan** is, what it is supposed to achieve, requirements of a **path planner**, how to **represent** a plan, how to measure the **quality** of a plan and different **planning algorithms** available.
- Understand different **discrete planning algorithms**.

Outline

- Introductory Concepts
- Discrete Planning
- Breadth-first Search (BFS)
- Depth-first Search (DFS)
- Dijkstra's Algorithm
- Best-first
- A* Algorithm

Outline

- **Introductory Concepts**
- Discrete Planning
- Breadth-first Search (BFS)
- Depth-first Search (DFS)
- Dijkstra's Algorithm
- Best-first
- A* Algorithm

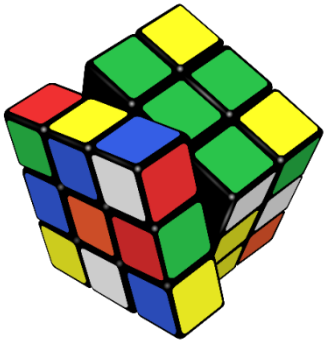
Introductory Concepts

- **Planning**

  Planning is a complex activity that determines a **future course of actions/activities** for an executing entity to drives it from an **initial state** to a specified **goal state**. Planning often involves reasoning with incomplete information.

Introductory Concepts

- **Planning is everywhere**

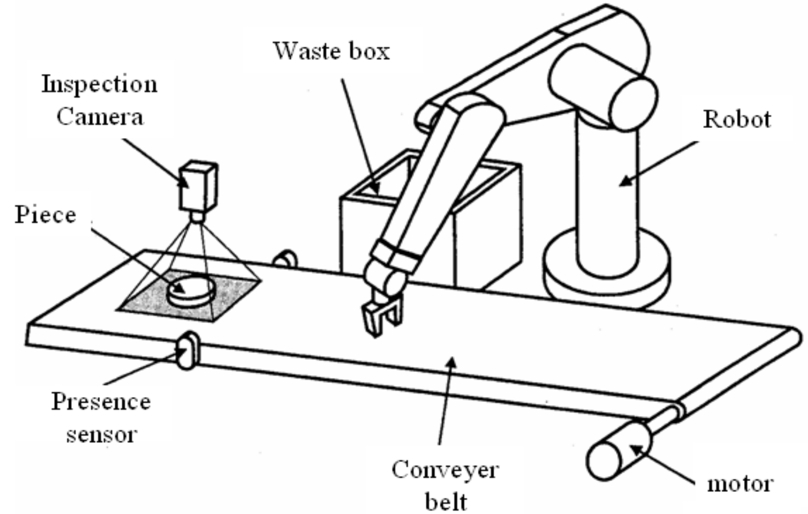


Rubik's cube

12	1	2	15
11	6	5	8
7	10	9	4
	13	14	3

Sliding Puzzle

AI Discrete Planning [1]



Robot Manipulator Planning

Piano Mover Problem [1]

Unmanned Ground Vehicles (UGV)

Unmanned Aerial Vehicles (UAV) & Micro Aerial Vehicles (MAV)

Unmanned Surface Vehicles (USV)

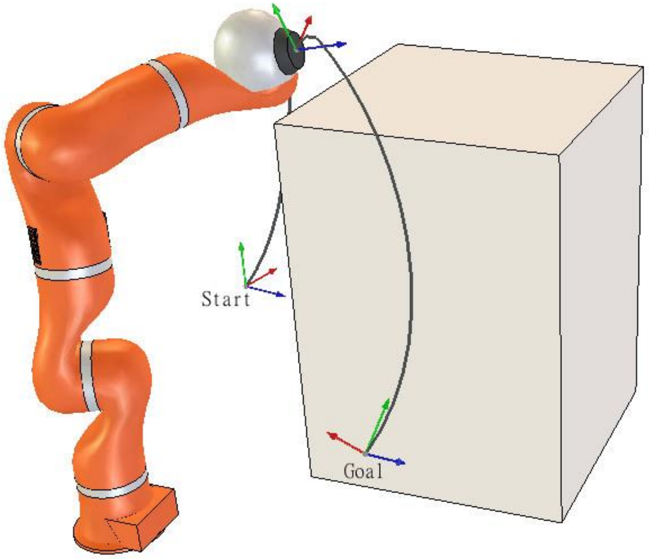
Unmanned Underwater Vehicles (UUV)

Unmanned Vehicles (UXVs) Planning

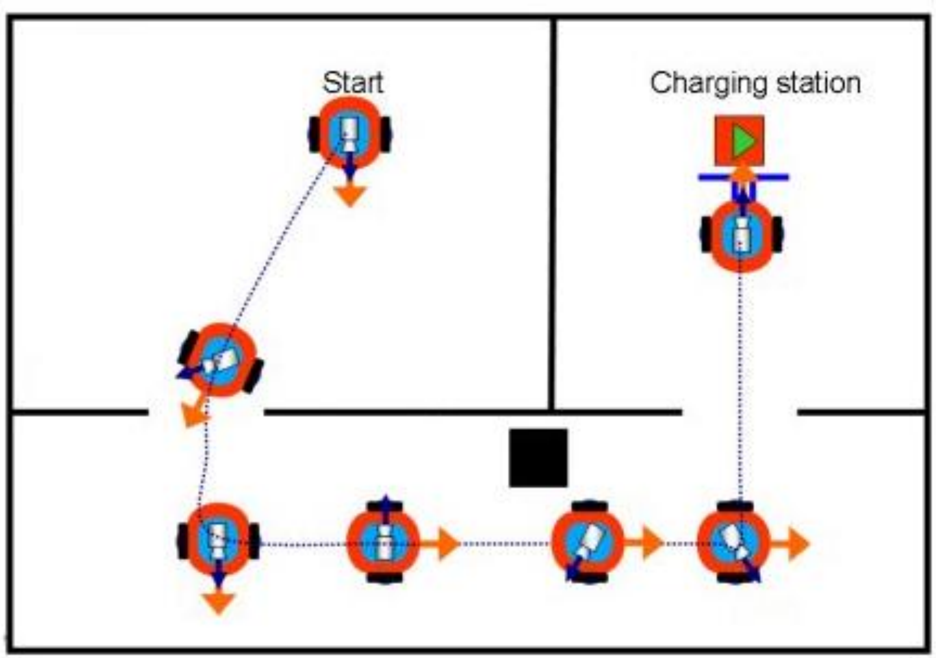
Introductory Concepts

- **Motion Planning**

A robot has to compute a **collision-free path** from a start position (s) to a given goal position (G), amidst a collection of obstacles.



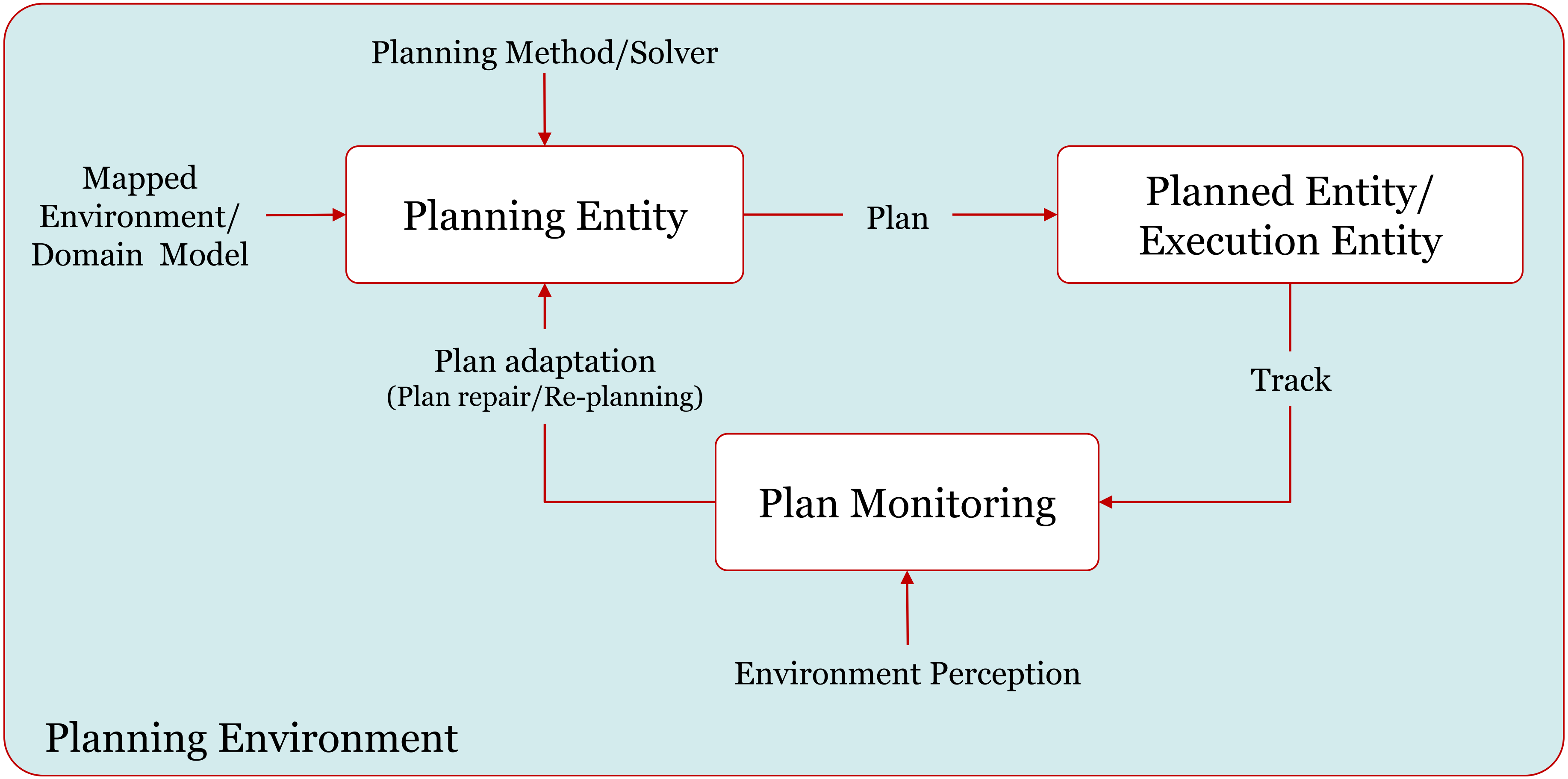
Articulated Robot



Rigid Robot

Introductory Concepts

- **Planning Overview**



Introductory Concepts

- **Planning Overview: Planning Environment**

Different planning approaches for different environments.

Modeling Dimensions:

- **Structure:** structured versus unstructured.
- **Observerability:** fully versus partially observable.
- **Determinism:** deterministic versus stochastic.
- **Continuity:** discrete versus continuous.
- **Adversary:** benign versus adversarial.
- **Number of agents:** single agent versus multiagent.
- …

Introductory Concepts

- **Planning Overview: Planning Environment**

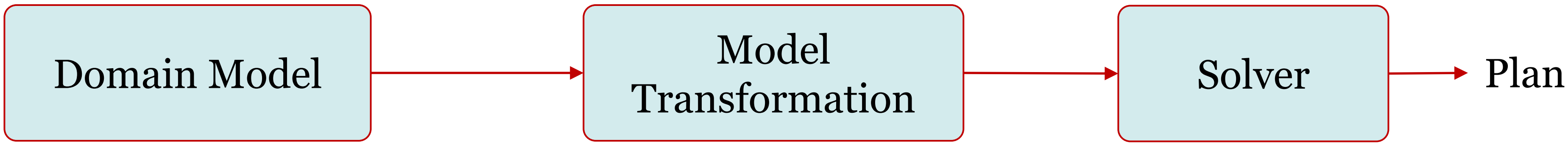
Kinds of events that trigger planning [2]:

- **Time-based:** for example, a plan for automated guided vehicle (AGV) needs to be made each season.
- **Event-based:** a plan for AGV must be made after an event, for example, a rush order in a factory.
- **Disturbance-based:** a plan must be adjusted because a disturbance occurs that renders the plan invalid- for example, unexpected moving obstacle in the robot way.

For reading: Can You Program Ethics Into a Self-Driving Car?

Introductory Concepts

- **Planning Process**



- Initial state
- Goal state
- Possible states of the robot(s)
- Primitive actions or activities
- Hard and soft constraints
- Uncertainty (sensor and actuators)
- A priori knowledge

- C-Space configuration
- Graph Decomposition
- Cell decomposition
- Road maps
- Potential fields
- …

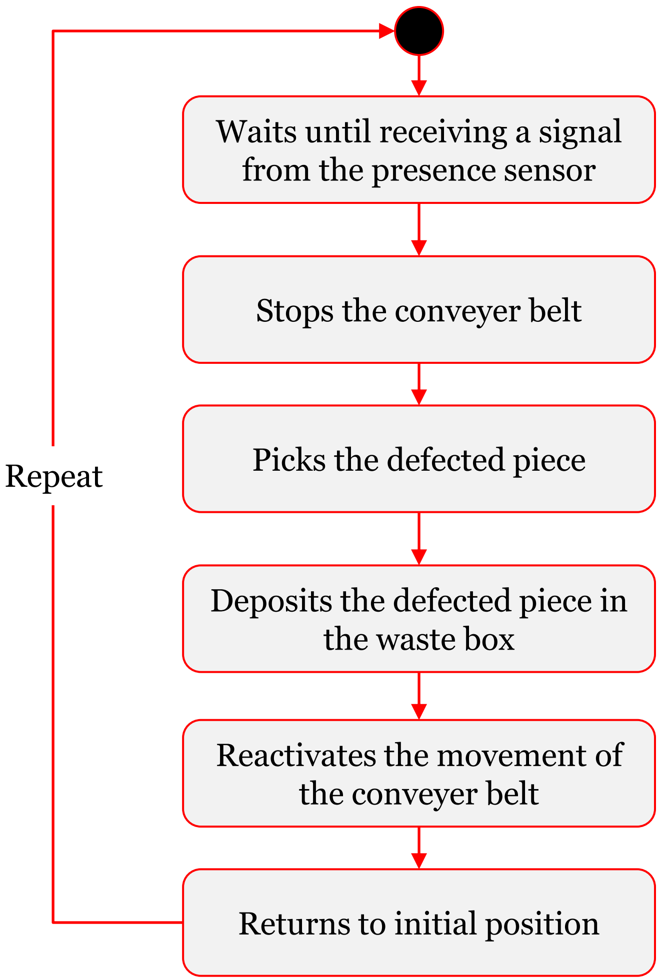
- Discrete Planning
- Combinatorial Planning
- Sampling-based Planning
- Potential Field Method
- Metaheuristics
- Planning under Uncertainty
- Hybrid approaches
- …

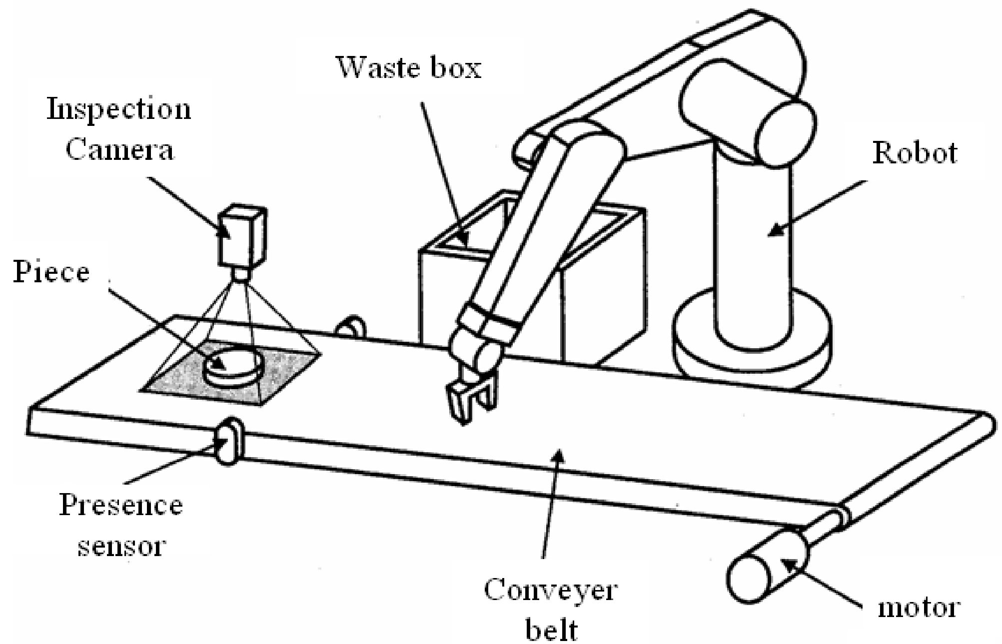
Introductory Concepts

- **Natural Questions:**
  - ◇ What is a plan?
  - ◇ What is plan supposed to achieve?
  - ◇ What are the requirements of a path planner?
  - ◇ How is an environment transformed?
  - ◇ How will plan quality be evaluated?
  - ◇ Who or what is going to use the plan?
  - ◇ What are the different planning algorithms?

Introductory Concepts

- **What is a plan?**



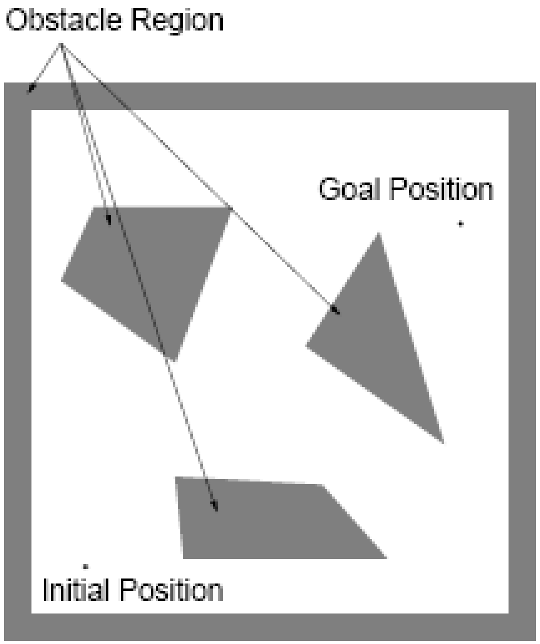


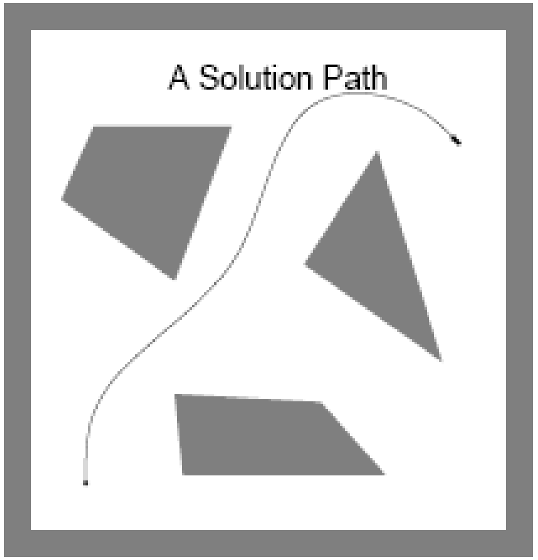

```
PROC main()
  go_wait_position;                          !Move to initial position
 WHILE Dinput(finish)=0 Do                   !wait end program signal
  IF Dinput(defected_piece)=1 THEN           !Wait defected piece signal
  SetDO activate_belt,0;                     !Stop belt
  pick_piece                                 !Pick the defected piece
  SetDO activate_belt,1;                     !Activate the belt
  place_piece                                !Place the defected piece
  go_wait_position;                          !Move to the initial position
  ENDIF
 ENDWHILE
ENDPROC
```

Plan: Activity Diagram → ABB RAPID Program

Introductory Concepts

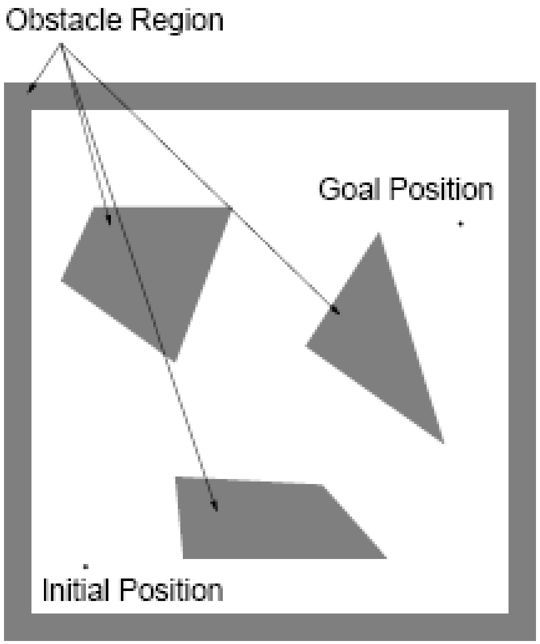
- **What is a plan?**
  - ◇ Suppose that we have a **tiny mobile robot** that can move along the floor in a building.
  - ◇ The task is to **determine a path** that it should follow from a **starting location** to a **goal location**, while **avoiding collisions**.
  - ◇ A reasonable model can be formulated by assuming that the robot is a **moving point** in a **two-dimensional environment** that contains obstacles.

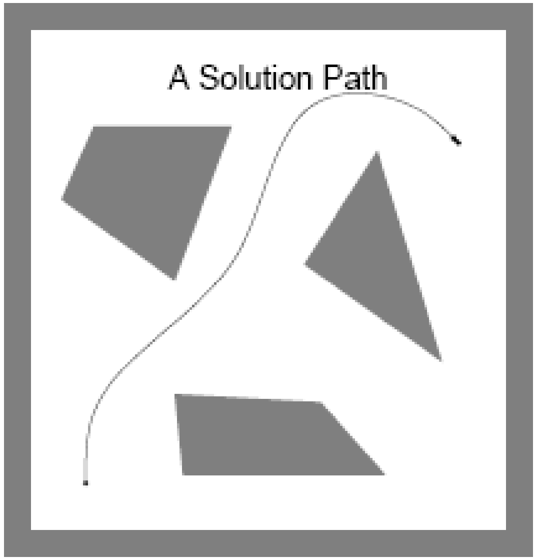




Introductory Concepts

- **What is a plan?**
  - ◊ The task is to **design an algorithm** that **accepts** an **obstacle region** defined by a set of polygons, an **initial position**, and a **goal position**.
  - ◊ The algorithm must **return a path** that will bring the robot from the **initial position** to the **goal position**, while only **traversing the free space**.





Introductory Concepts

- **What is plan supposed to achieve?**
  - ◇ A plan might simply **specify a sequence of actions** to be taken; however, it may be more complicated.
  - ◇ If it is impossible to **predict future states**, the plan may provide actions as a **function of state**.
  - ◇ In this case, **regardless of future states**, the appropriate action is determined.
  - ◇ It might even be the case that the **state cannot be measured**.
  - ◇ In this case, the action must be chosen based on **whatever information** is available up to the current time.

Introductory Concepts

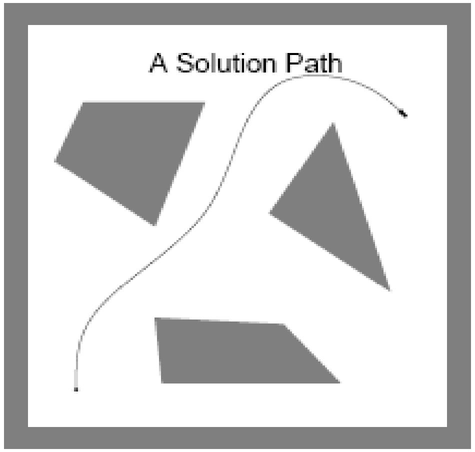
- **What are the requirements of a path planner?**

1. to **find a path** through a **mapped environment** so that the robot can travel along it without colliding with anything,
2. to **handle uncertainty** in the sensed world model and **errors** in **path execution**,
3. to **minimize** the impact of objects on the field of view of the robot's sensors by keeping the robot **away from** those objects,
4. to find the **optimum path**, if that path is to be **negotiated** regularly.

Introductory Concepts

- **What are the requirements of a path planner?**

5. Determine the following:

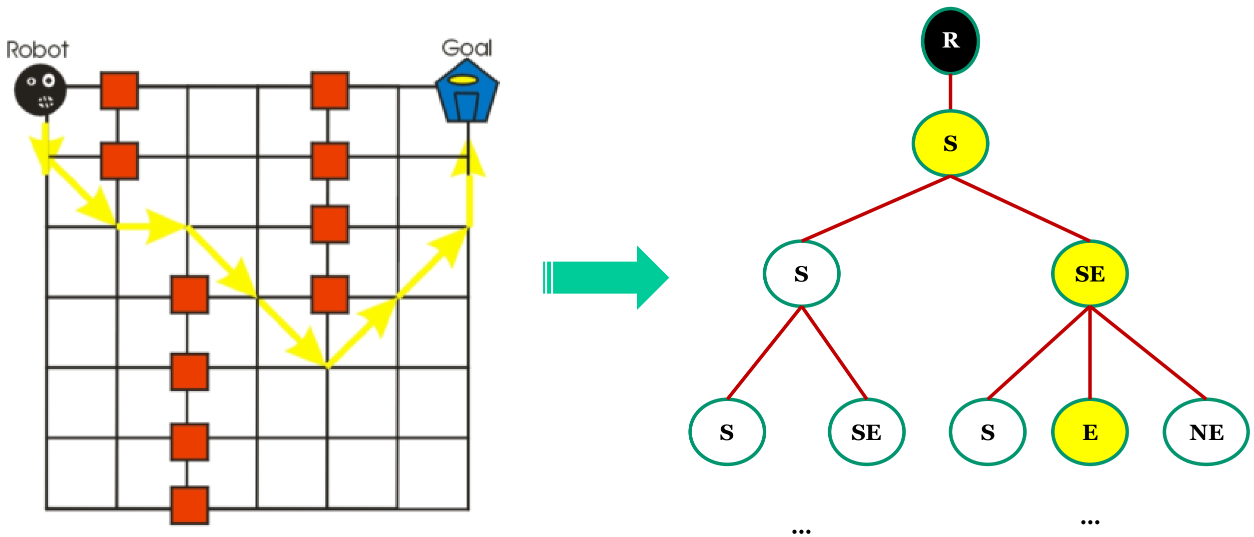


- ◇ **Horizon:** What time/space span does the plan cover? Receding Horizon Control/planning (RHC)
- ◇ **Frequency:** How often is the plan created or adapted?
- ◇ **Level of detail:** Does the plan need more detail in order to be executed? Does the executing entity have to fill in the details, or the plan used as a template for another planner (multiresolutional planning)?
- ◇ **(Re)presentation:** How is the plan represented and depicted? Does it specify the end state, or does it provide a process description that leads to the end state?

[2]

Introductory Concepts

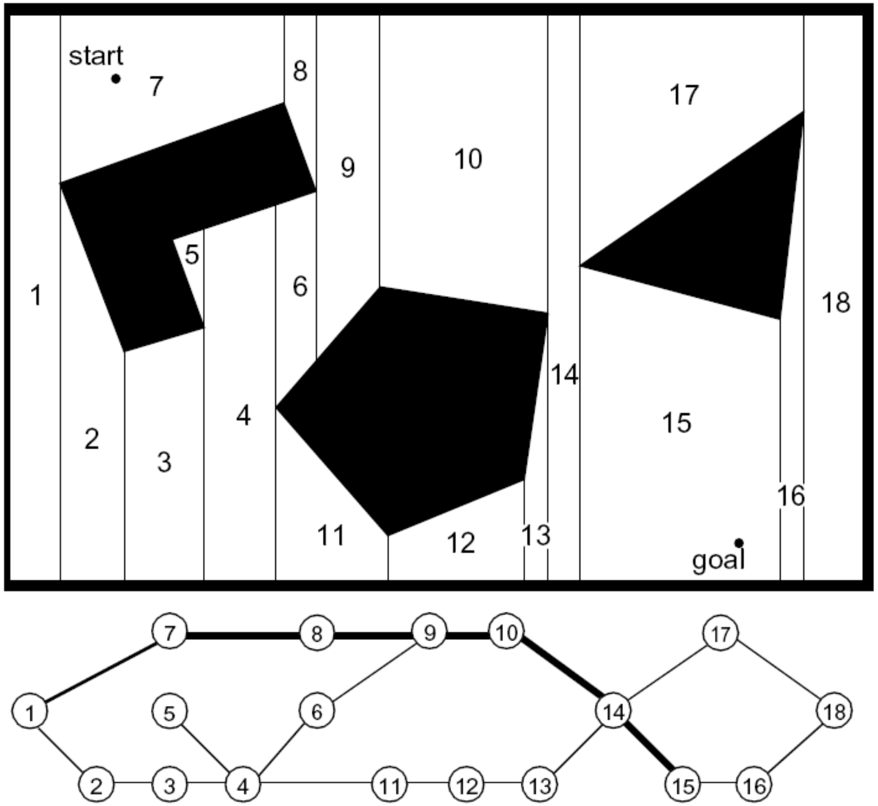
- **How is an environment transformed?**
  - ◇ The first step of any path-planning system is to **transform** the **continuous environmental model** into a **discrete map** suitable for the chosen path-planning algorithm.
  - ◇ Path planners differ as to how they effect this **discrete decomposition**.



Graph Decomposition: Environment is represented as an ordered directed graph.

Introductory Concepts

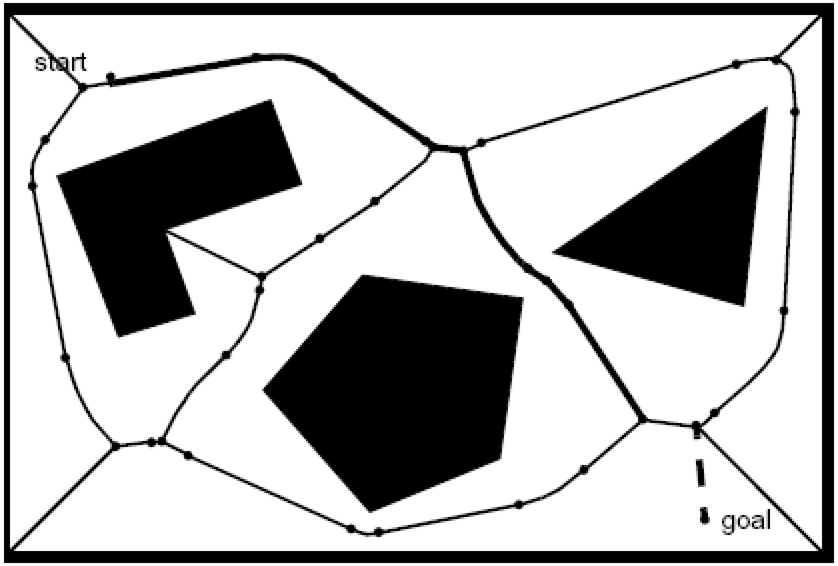
- **How is an environment transformed?**



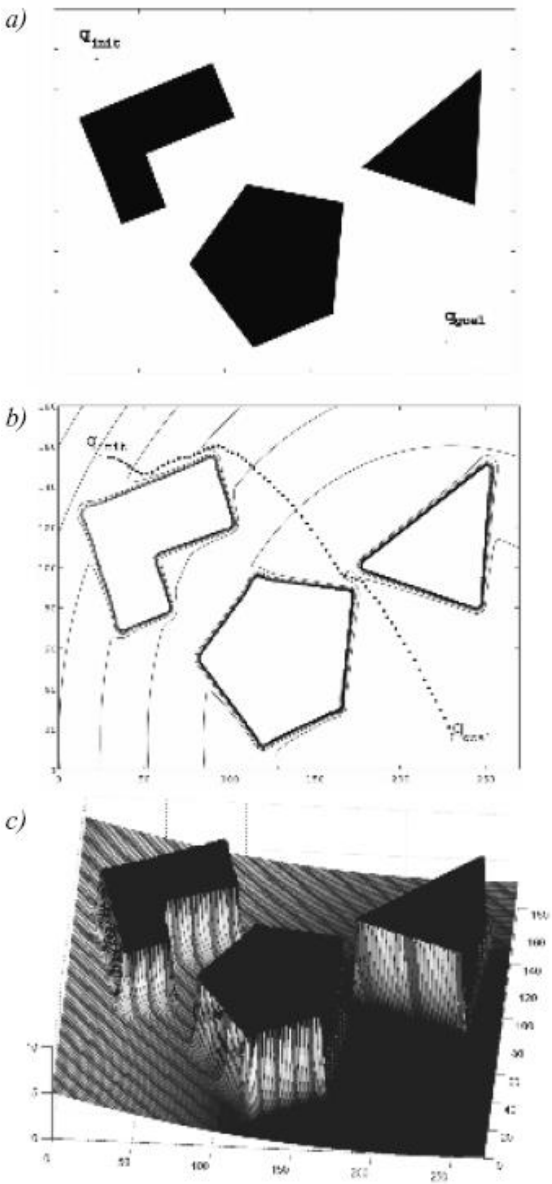
Cell decomposition: discriminate between free and occupied cells.

Introductory Concepts

- **How is an environment transformed?**



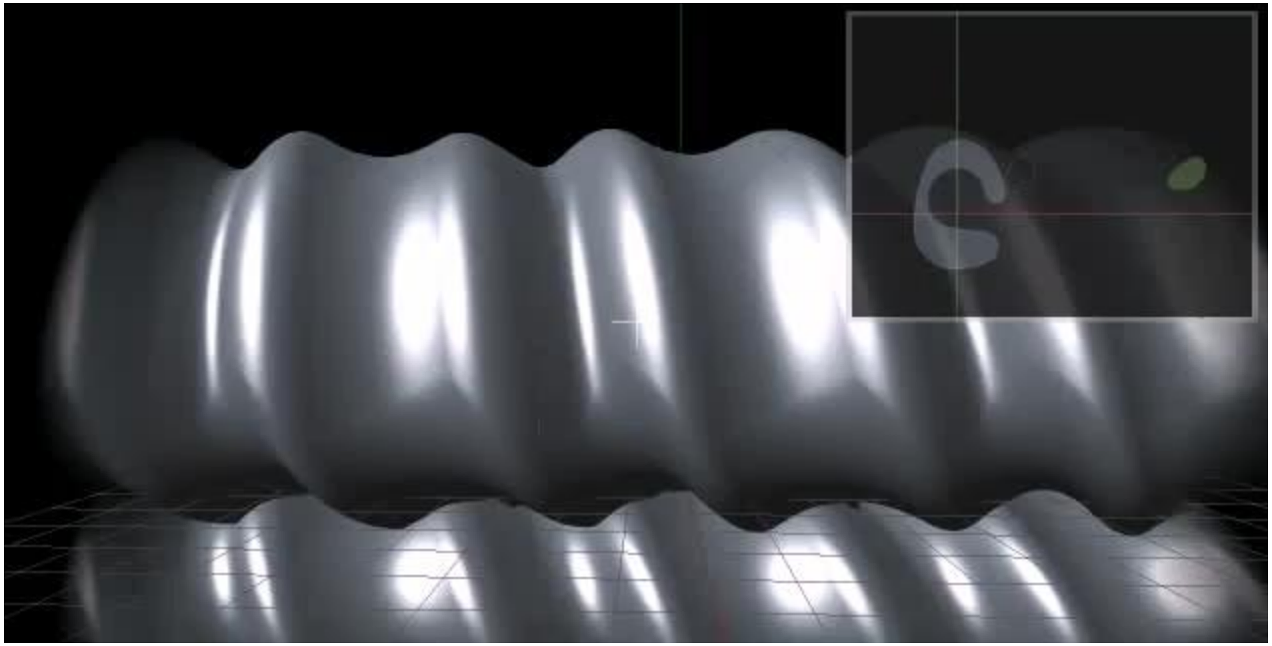
Road map: identify a set of routes within the free space.



Potential field: impose a mathematical function over the space.

Introductory Concepts

- **Configuration space**
  - ◇ Configuration space (**C-Space**) is the set of **legal configurations** of the robot.
  - ◇ C-Space also defines the topology of continuous motions. **Transform**.

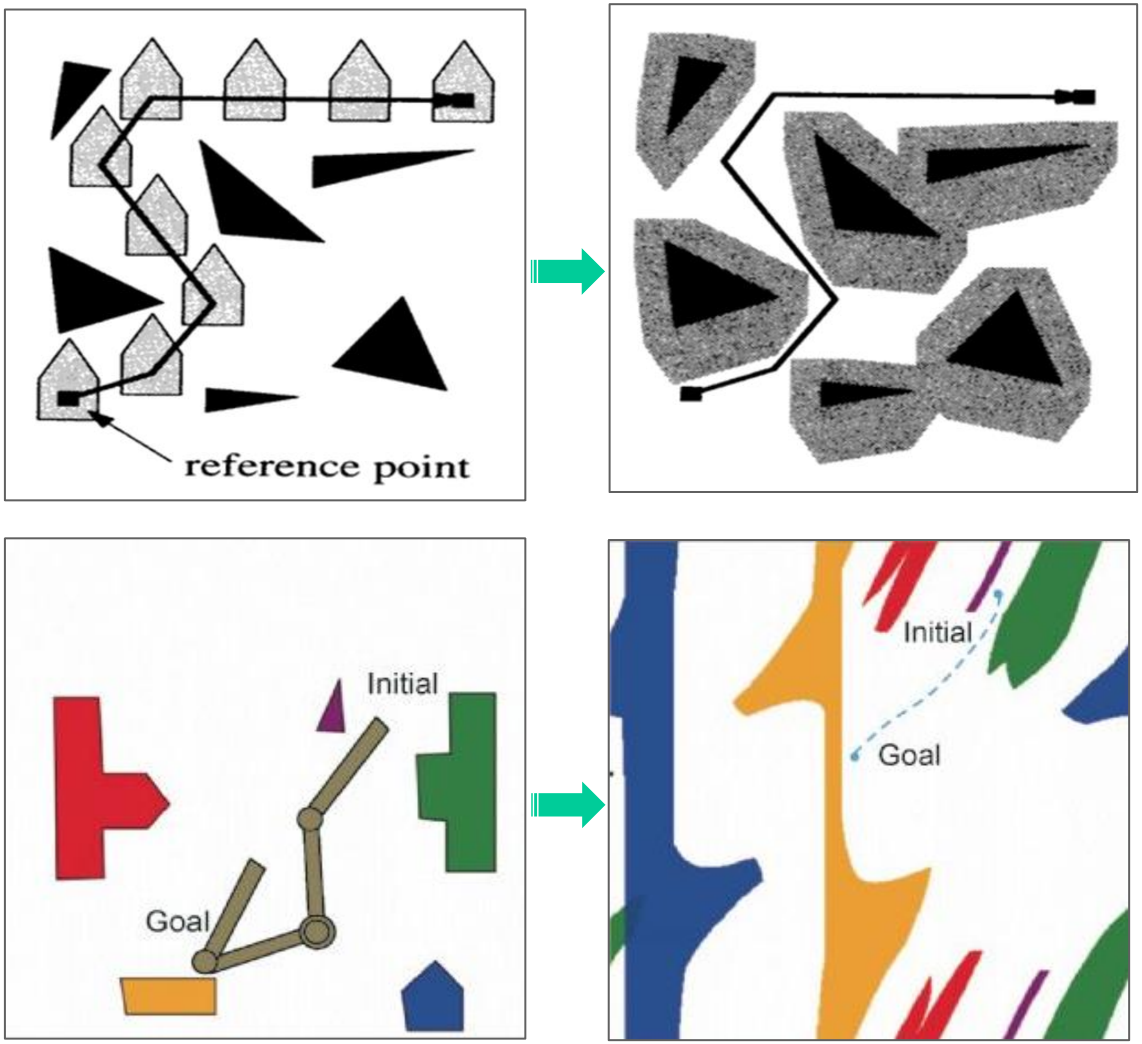


Source: https://www.youtube.com/watch?v=zLEIWt6XZDY

Introductory Concepts

- **Configuration space**

  For both articulated robots and rigid-object robots (no joints) there exists a transformation to the robot and obstacles that **turns the robot into a single point.** The **C-Space Transform**.



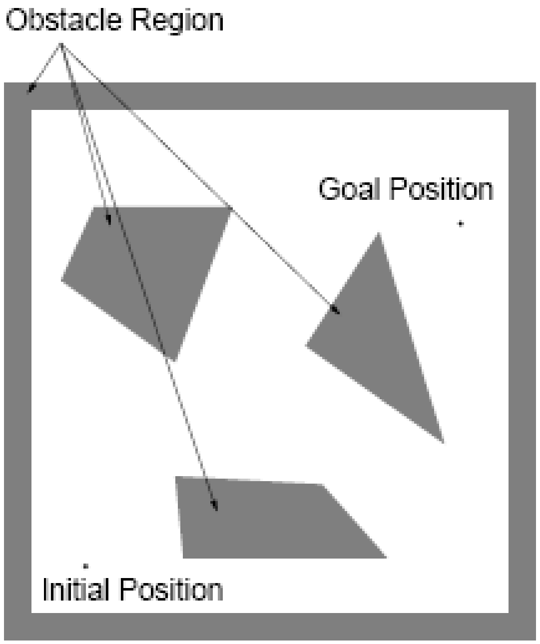
Workspace | Configuration Space

Introductory Concepts

- **Configuration space**

  Assume the following:

  $\mathcal{A}$: a rigid/articulated **robot**;
  $\mathcal{W}$: the **workspace** (i.e., the Cartesian space in which the robot moves);
  $\mathcal{A}(q)$: the subset of the workspace that is occupied by the robot at configuration q
  $\mathcal{O}_i$: the **obstacles** in the workspace;



  $\mathcal{O} = \cup \mathcal{O}_i$, **obstructive region** and
  $\mathcal{C}$: **configuration space**, which is the set of all possible configurations. A complete specification of the location of every point on the robot is referred to as a configuration.

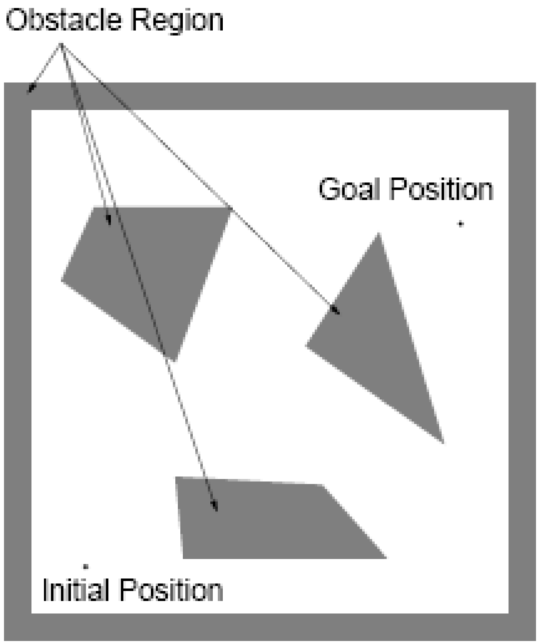
[3]

Introductory Concepts

- **Configuration space**

  $\mathcal{C}_{\text{obs}}$: **configuration space obstacle**, which is the set of configurations for which the robot collides with an obstacle, $\mathcal{C}_{\text{obs}} \subseteq \mathcal{C}$

$$\mathcal{C}_{\text{obs}} = \{q \in \mathcal{C} | \mathcal{A}(q) \cap \mathcal{O} \neq \phi\}$$



The set of collision-free configurations, referred to as the **free configuration space**, is then simply

$$\mathcal{C}_{\text{free}} = \mathcal{C} \backslash \mathcal{C}_{\text{obs}}$$

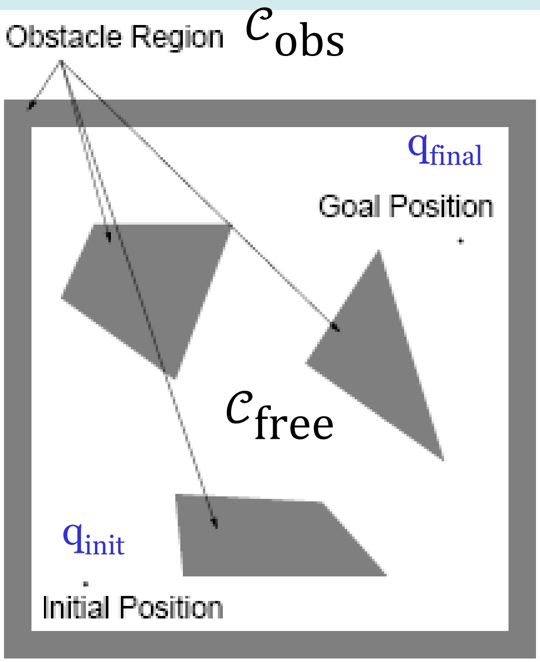
$$\mathcal{C} = \mathcal{C}_{\text{free}} \cup \mathcal{C}_{\text{obs}}$$

[3]

Introductory Concepts

- **Configuration space**

The **path planning problem** is to find a path from an initial configuration q_{init} to a final configuration q_{final}, such that the robot does not collide with any obstacle as it traverses the path.



A collision-free path from q_{init} to q_{final} is a continuous map,

$$\mathcal{T}: [0,1] \rightarrow \mathcal{C}_{free}$$

with

$\mathcal{T}(0)= q_{init}$ and

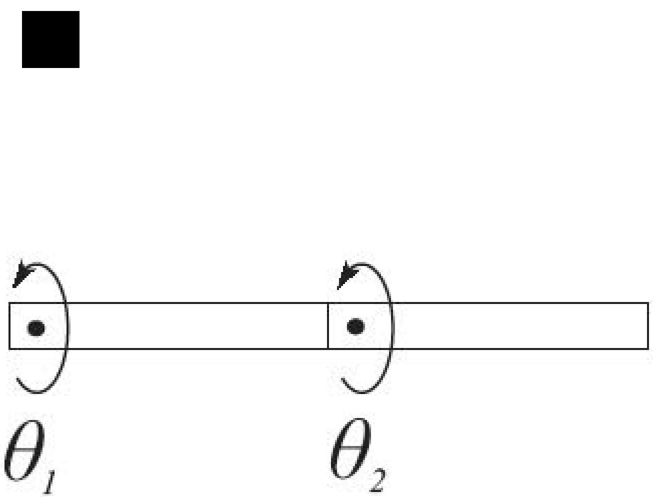
$\mathcal{T}(1)= q_{final}$

[3]

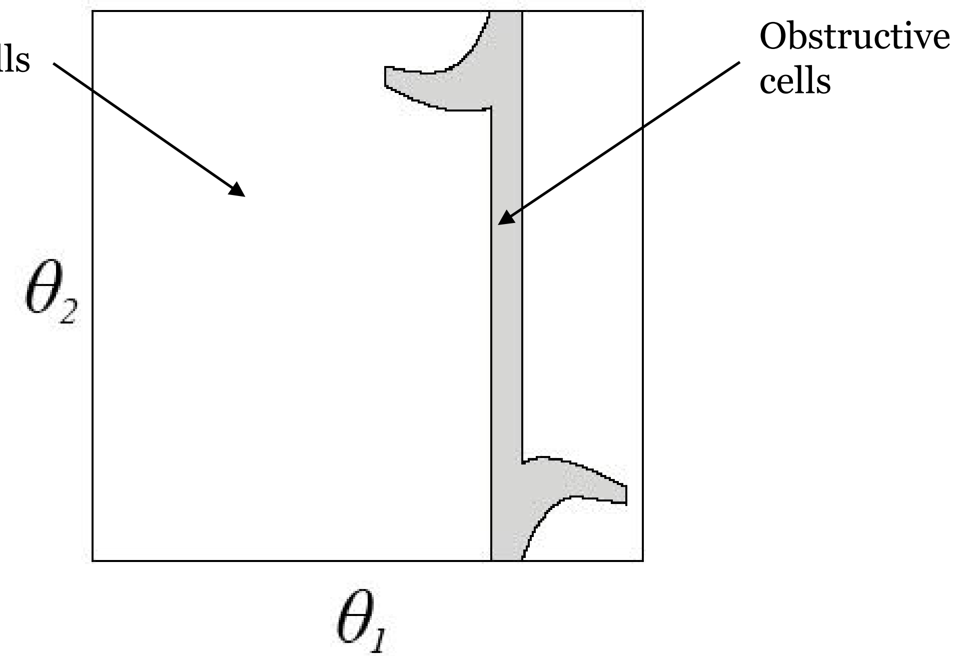
Introductory Concepts

- **Configuration space: Articulated Robots**

Consider a two-link planar arm in a workspace containing a single obstacle



Two-link Planar Arm



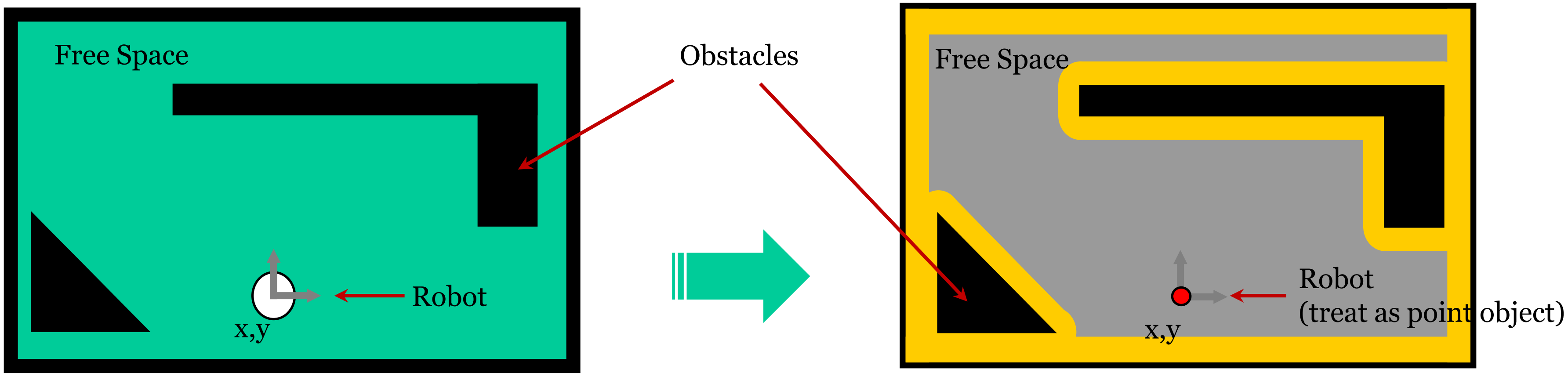
Configuration Space

The region $\mathcal{C}_{\text{obs}}$ was computed using a **discrete grid** on the configuration space

[3]

Introductory Concepts

- **Configuration space: Rigid Robots**



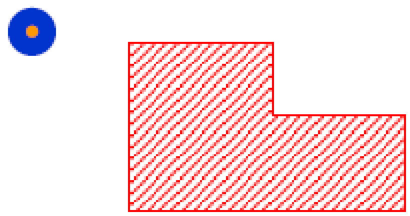
Workspace

Configuration Space

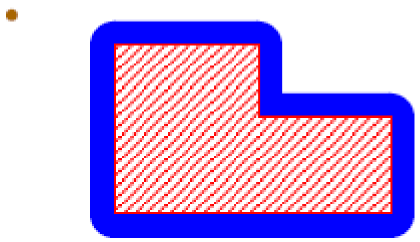
Introductory Concepts

• C-Space Transform Examples

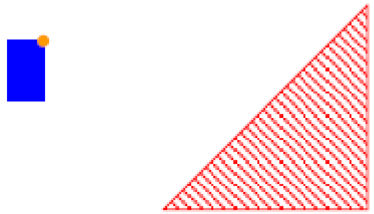
Where can I move this robot in the vicinity of this obstacle



Where can I move this point in the vicinity of this expanded obstacle

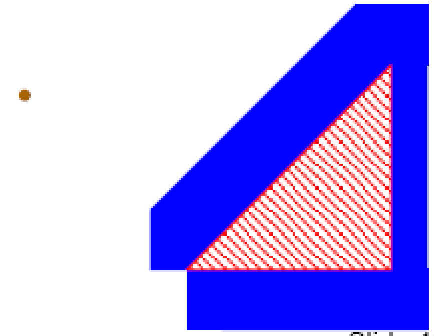


Where can I move this robot in the vicinity of this obstacle



Assuming you're not allowed to rotate

Where can I move this point in the vicinity of this expanded obstacle



Slide 10

Introductory Concepts

- **Object-growth Algorithm**

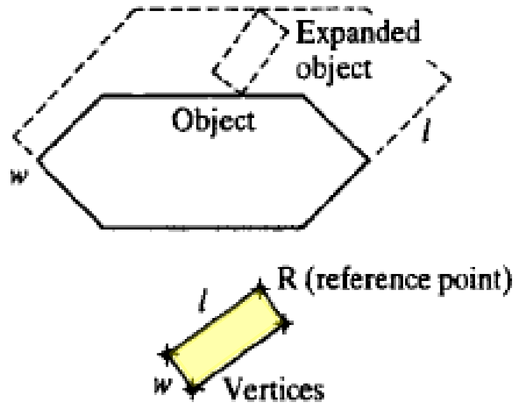
1. Attach a **coordinate frame** to the reference point,
2. **Flip the robot** about the two axes of the coordinate frame, one axis at a time.
3. Place the reference point of the flipped robot at **each of the vertices of the object** and calculate the positions of the vertices of the robot.
4. Find the **convex hull of the vertices**. The convex hull is the convex polygon formed by stretching a rubber band around the vertices.

Reading Material

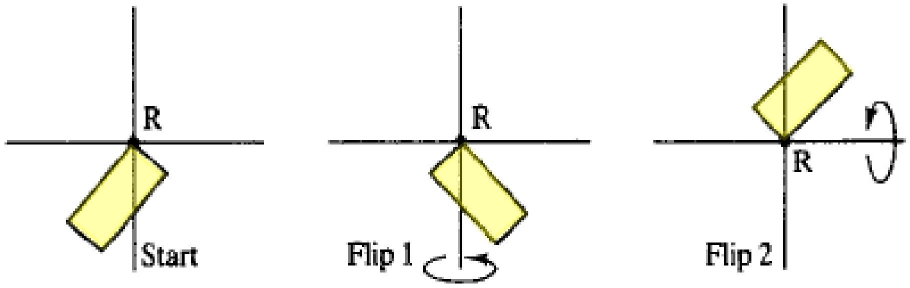
Introductory Concepts

- **Object-growth Algorithm**

1. Attach a coordinate frame to the reference point,



2. Flip the robot about the two axes of the coordinate frame, one axis at a time.

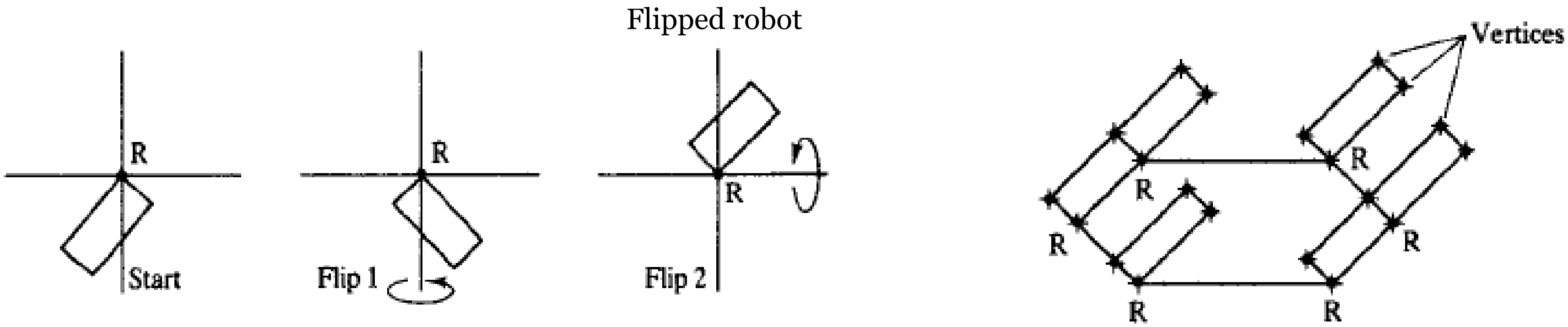


Reading Material

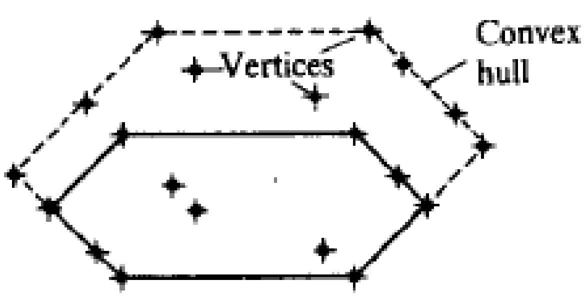
Introductory Concepts

- **Object-growth Algorithm**

3. Place the reference point of the flipped robot at each of the vertices of the object and calculate the positions of the vertices of the robot.



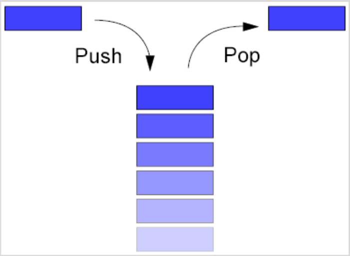
4. Find the convex hull of the vertices. The convex hull is the convex polygon formed by stretching a rubber band around the vertices.



Reading Material

Introductory Concepts

- **Convex hull Algorithm: Sklansky's Algorithm**



Stack (LIFO)

1. Place vertices into a counter-clockwise ordered list
2. Allocate a stack
3. Push vertex 0 onto stack
4. Push vertex 1 onto stack
5. **For** *i*=2 to n **do**

    {compare vertex *i* to ray formed by the 2 vertices at the top of the stack}

    if vertex *i* is on or to the right of the ray (stacktop-1, stacktop) then

        pop {discard top element of stack}

    End

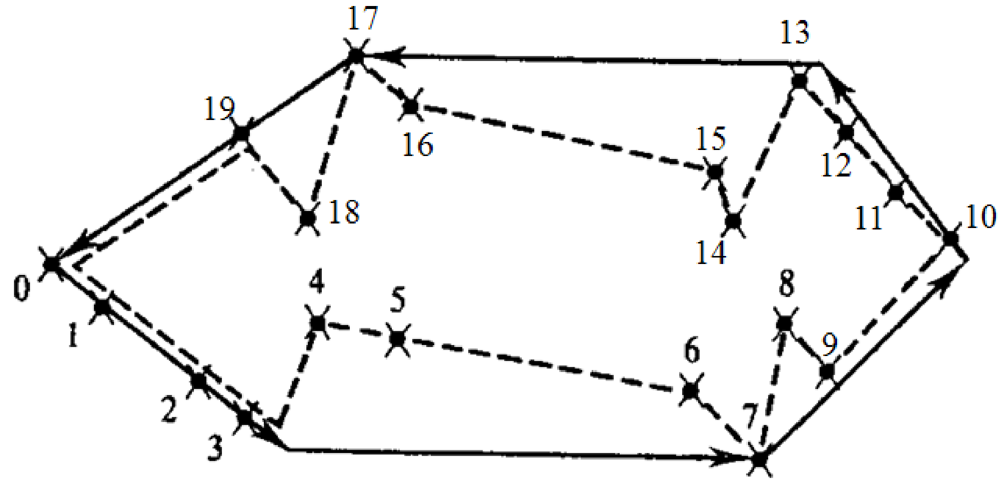
    push vertex *i* {put vertex *i* onto stack}

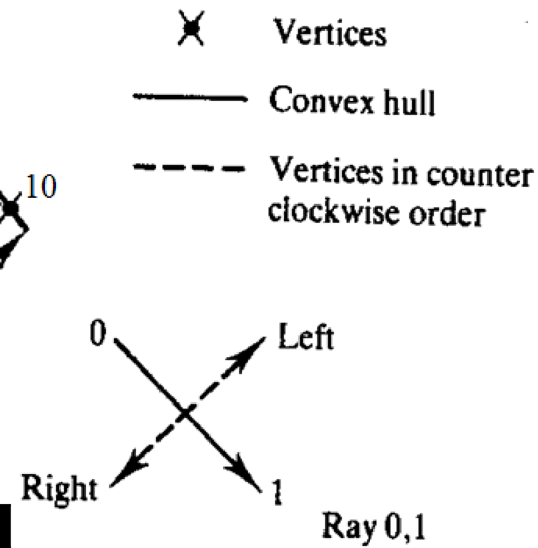
    End

Reading Material

Introductory Concepts

- **Convex hull Algorithm: Sklansky's Algorithm**





Point	Pushed	Popped	Stack
0	0	-	0
1	1	-	1,0
2	2	1	2,0
3	3	2	3,0
4	4	-	4,3,0
5	5	4	5,3,0
6	6	5	6,3,0
7	7	6	7,3,0
8	8	-	8,7,3,0
9	9	8	9,7,3,0
…	…	…	…

Reading Material

↓ Repeat…

Introductory Concepts

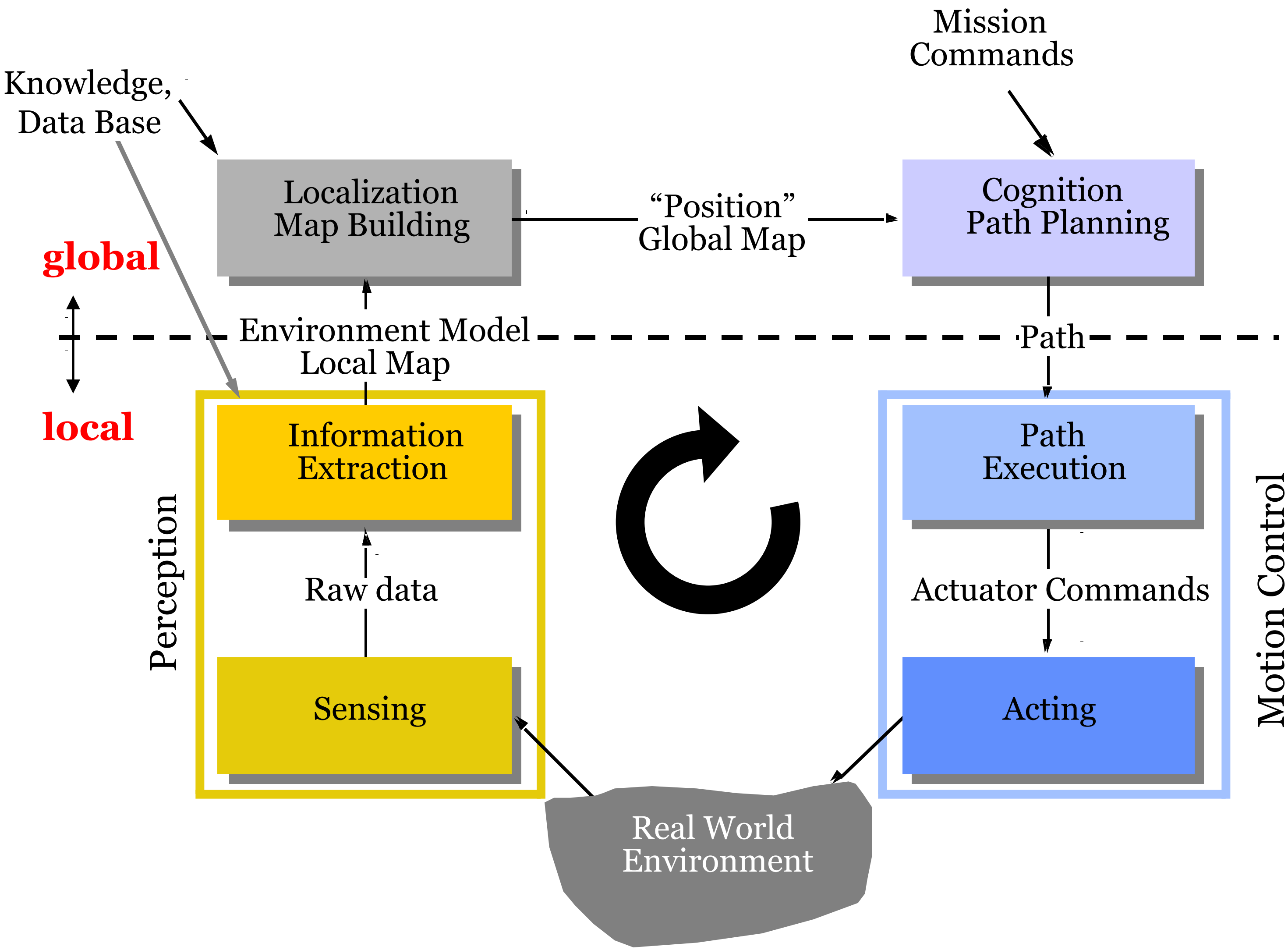
- **How will plan quality be evaluated?**

Evaluation Metrics

- Time
- Distance travelled
- Expense
- Distance from obstacles
- Smoothness of the path
- Optimality of the path
- etc…

Introductory Concepts

- **Who or what is going to use the plan?**



[4]

Introductory Concepts

- **What are the different planning algorithms?**

Motion Planning Methods

- Discrete Planning
  - Forward Search
  - Backward Search
    - ◇ Breadth first
    - ◇ Depth first
    - ◇ Dijkstra's algorithm
    - ◇ A*
    - ◇ …
- Combinatorial Planning
  - Cell Decomposition
    - ◇ Exact
    - ◇ Approximate (Fixed/ Adaptive)
  - Roadmaps
    - ◇ Voronoi Diagrams
    - ◇ Visibility Graph
    - ◇ Free Way Net
    - ◇ Silhouette
    - ◇ MAKLINK
    - ◇ …
- Other Approaches
  - ◇ Sampling-Based Motion Planning (Rapidly-exploring Random Trees-RRTs, Probabilistic Roadmaps-PRMs)
  - ◇ Potential Field Methods
  - ◇ Metaheuristics
  - ◇ Decision-theoretic Planning
  - ◇ Hybrid approaches

Outline

- Introductory Concepts
- **<u>Discrete Planning</u>**
- Breadth-first Search (BFS)
- Depth-first Search (DFS)
- Dijkstra’s Algorithm
- Best-first
- A* Algorithm

Discrete Planning

- The typical approach towards the problem of path planning assumes a **graph representation** of the terrain. The problem of minimum-path planning on a graph is well known, and a variety of algorithms are applicable under different input specification.
- Therefore path planning problem is handled as a **search problem** to find the shortest path between start and goal points on a **graph**.

Discrete Planning

Forward Graph Search Methods/ Enumerative Algorithms

Uninformed/Blind Search

- Breadth-first (BFS)
- Depth-first (DFS)
- British Museum Search or Brute-force Search
- Dijkstra
- …

Informed Search

- Best-first
- A*
- …

Outline

- Introductory Concepts
- Discrete Planning
- **Breadth-first Search (BFS)**
- Depth-first Search (DFS)
- Dijkstra’s Algorithm
- Best-first
- A* Algorithm

Breadth-first Search (BFS)

Step 1. Form a queue Q and set it to the initial state (for example, the Root).

Step 2. Until the Q is empty or the goal state is found

do:

Step 2.1 Determine if the first element in the Q is the goal.

Step 2.2 If it is not

Step 2.2.1 remove the first element in Q.

Step 2.2.2 Apply the rule to generate new state(s) (successor states).

Step 2.2.3 If the new state is the goal state quit and return this state

Step 2.2.4 Otherwise add the new state to the end of the queue.

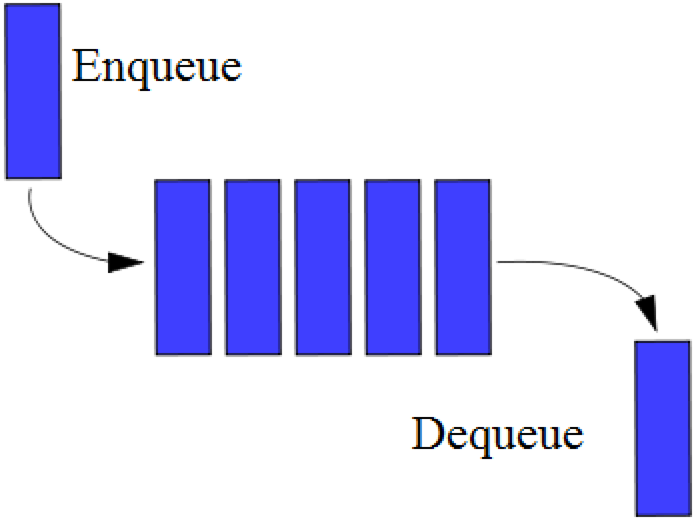
Step 3. If the goal is reached, success; else failure.

Reading Material

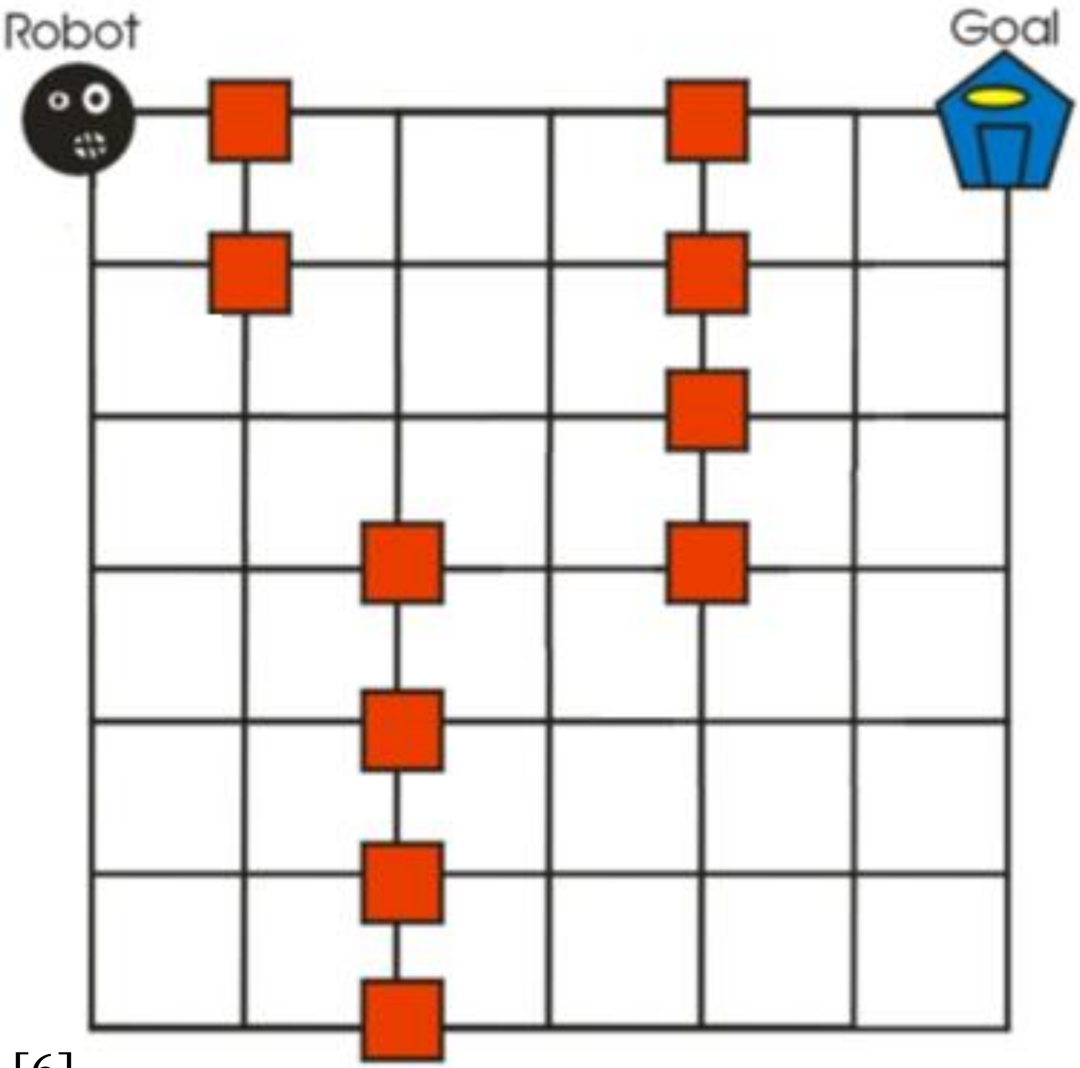
[5]

Breadth-first Search (BFS)

- BFS uses the **queue** as data structure.
- Queue is a **First-In-First-Out (FIFO)** data structure.
- A FIFO means that the node that has been sitting on the queue for the **longest amount of time** is the **next node to be expanded**.



Breadth-first Search (BFS)



[6]

Start

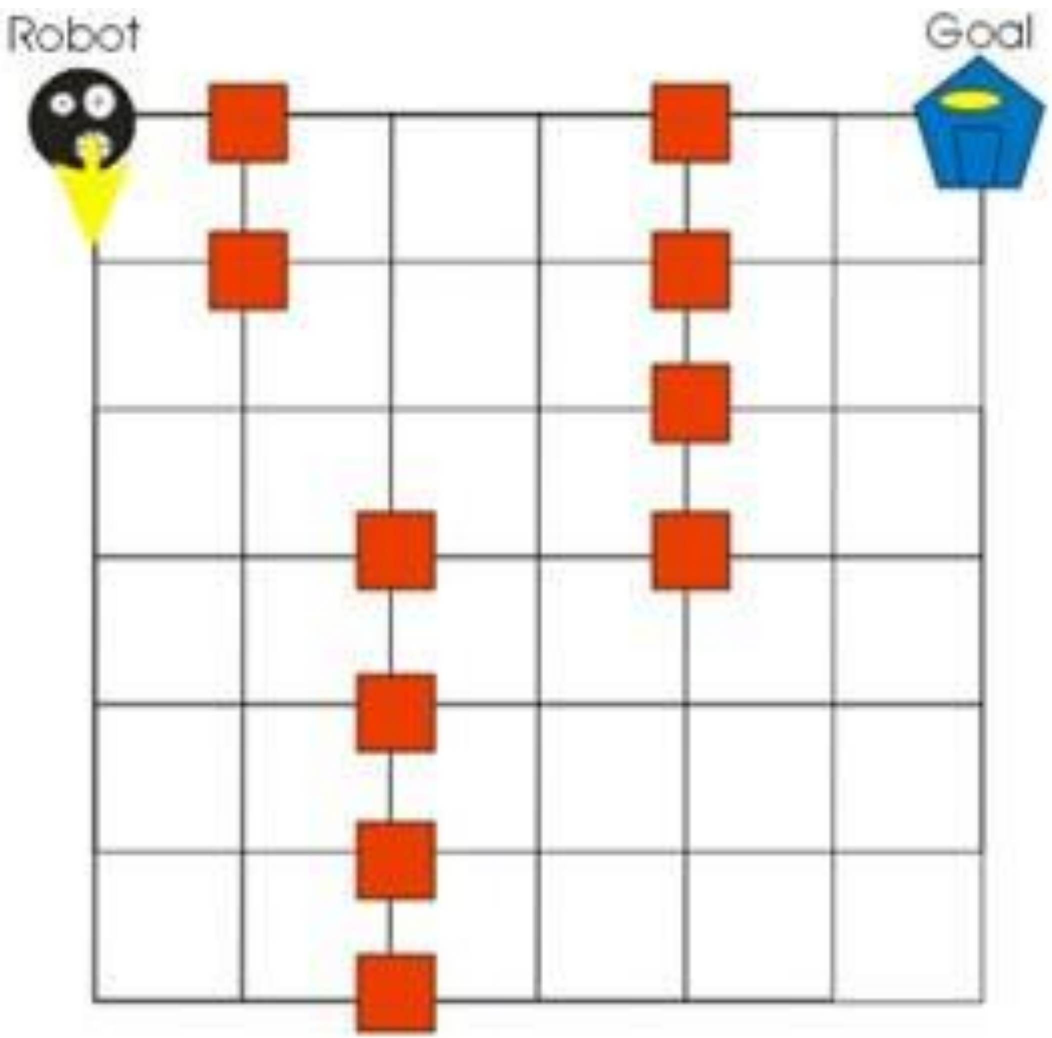
Queue

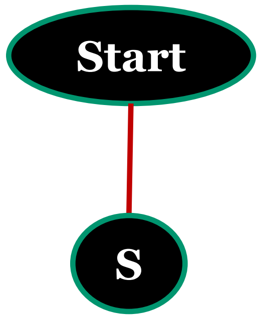
Start

IN

OUT

Breadth-first Search (BFS)





Queue

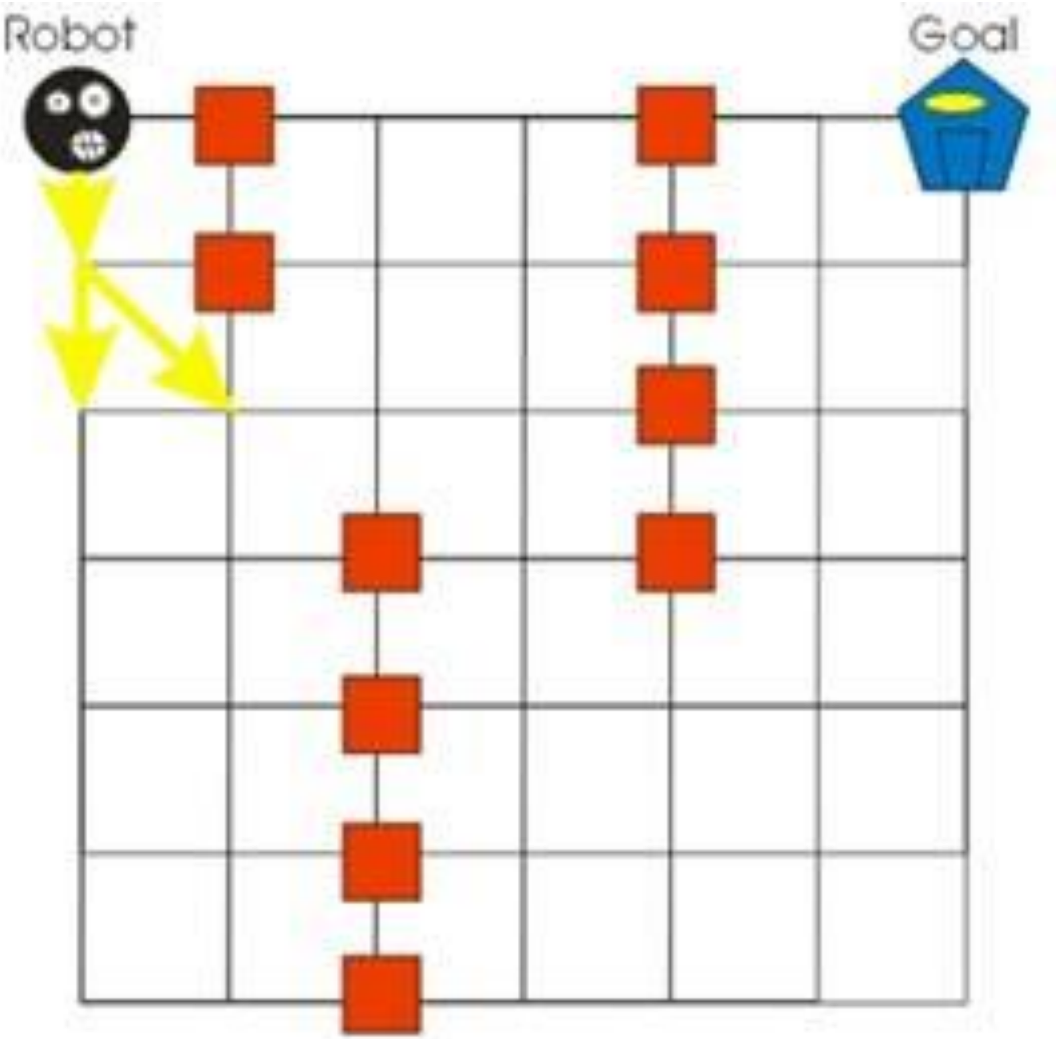
S

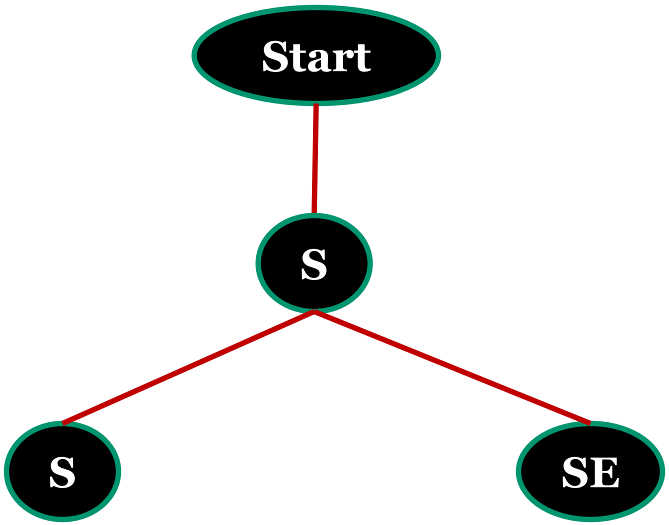
IN

Start

OUT

Breadth-first Search (BFS)





Queue

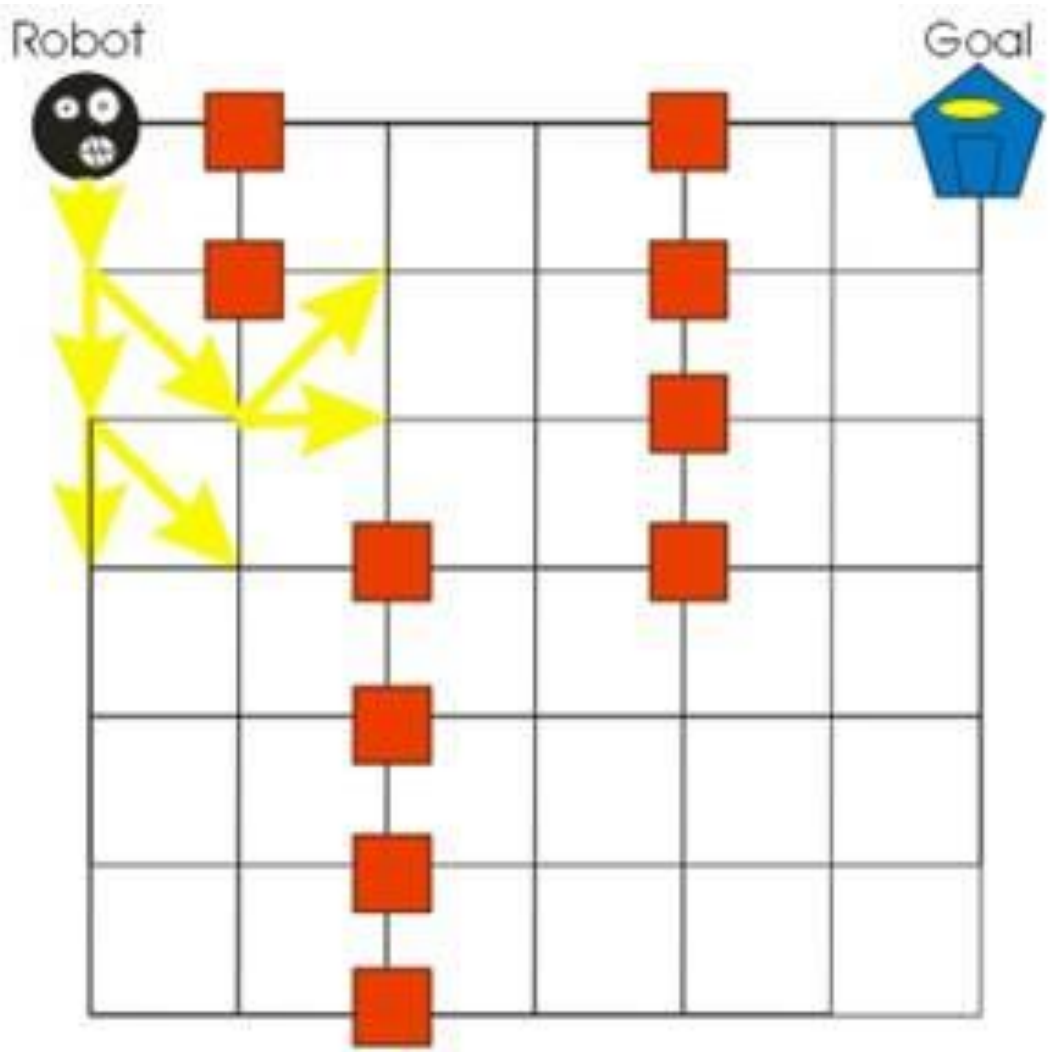
SE S

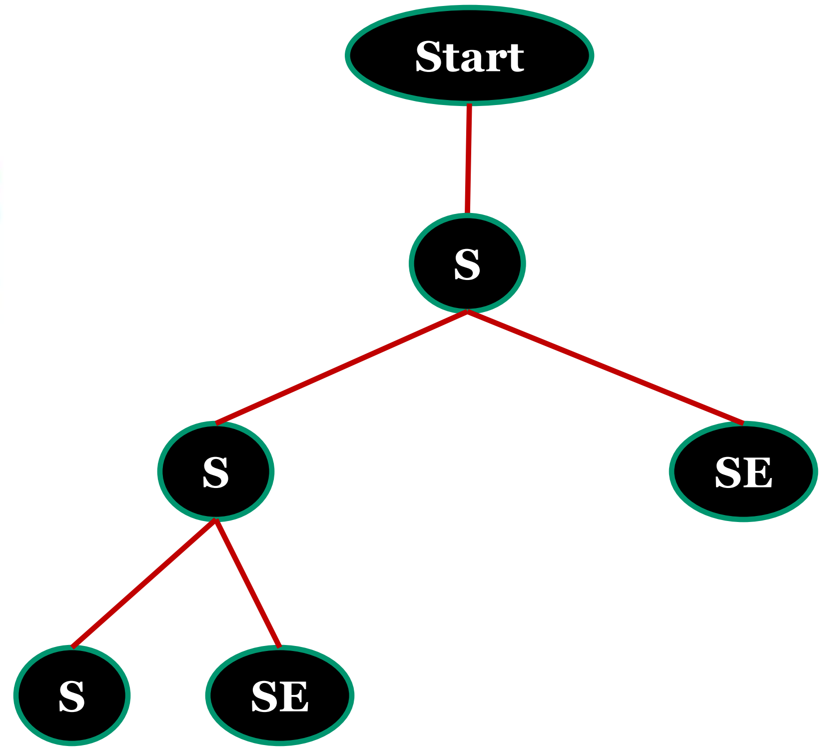
IN

Start S

OUT

Breadth-first Search (BFS)





Queue

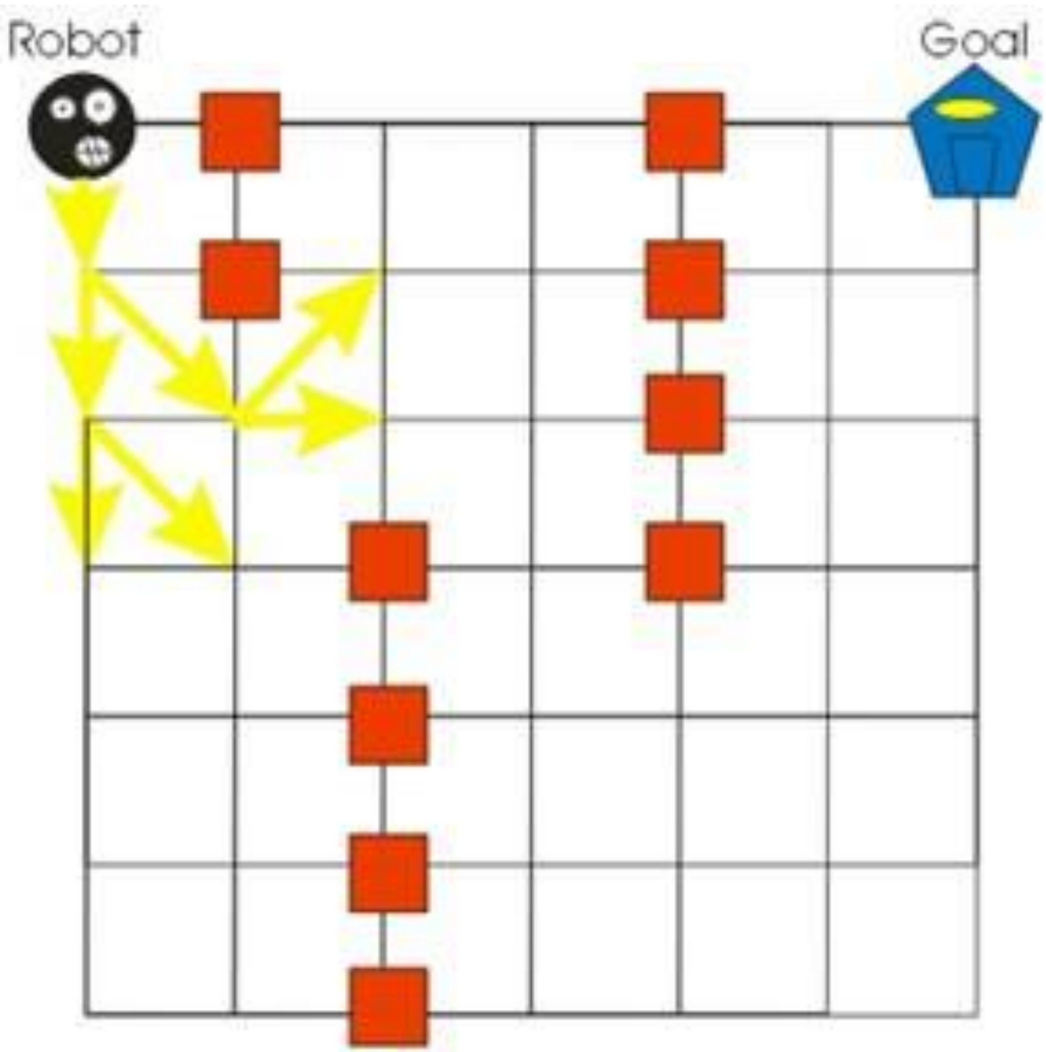
SE S SE

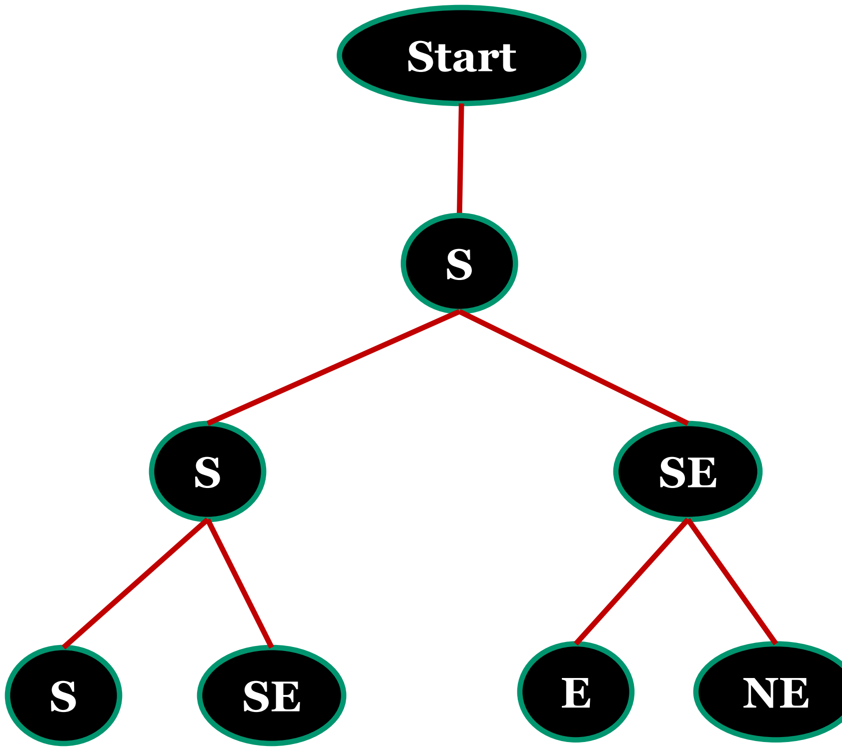
IN

Start S S

OUT

Breadth-first Search (BFS)





Queue

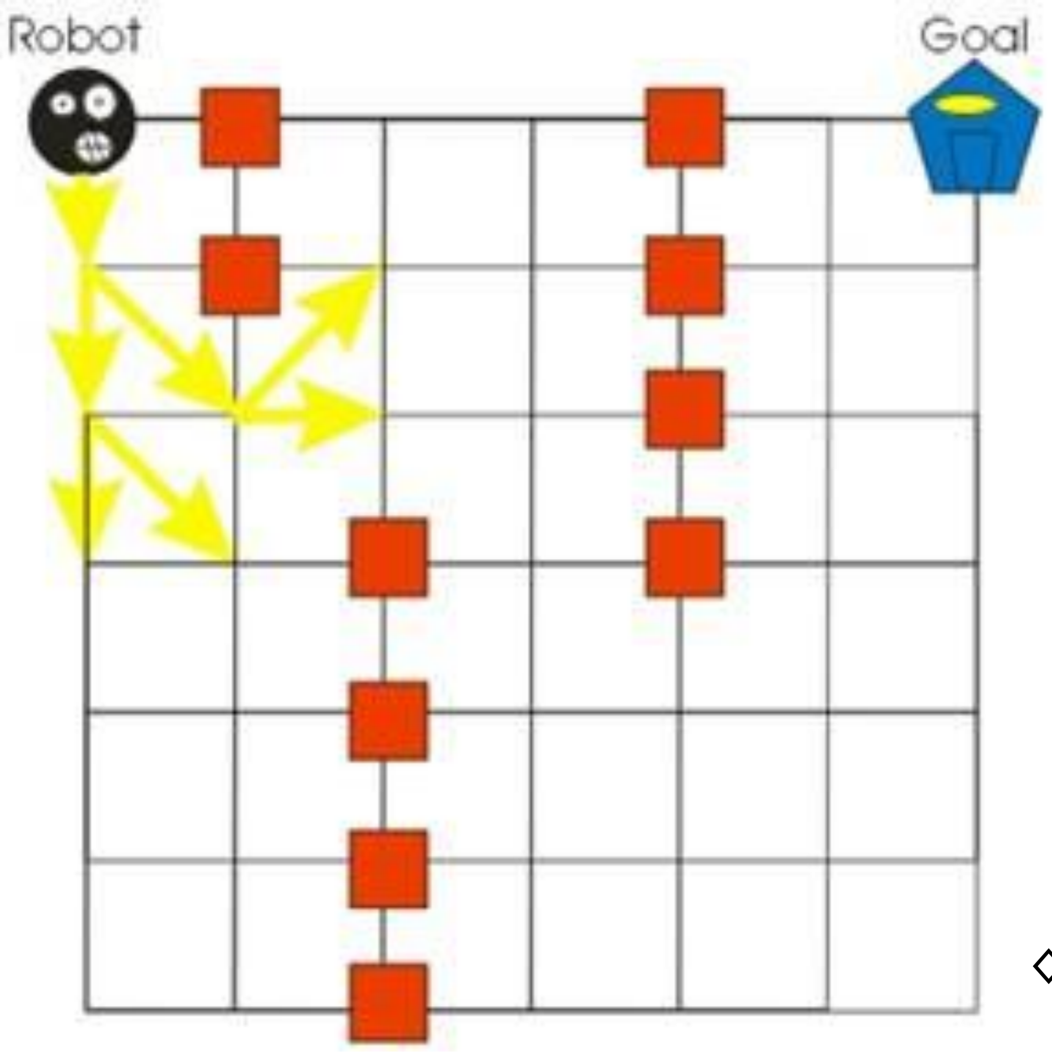
NE E SE S

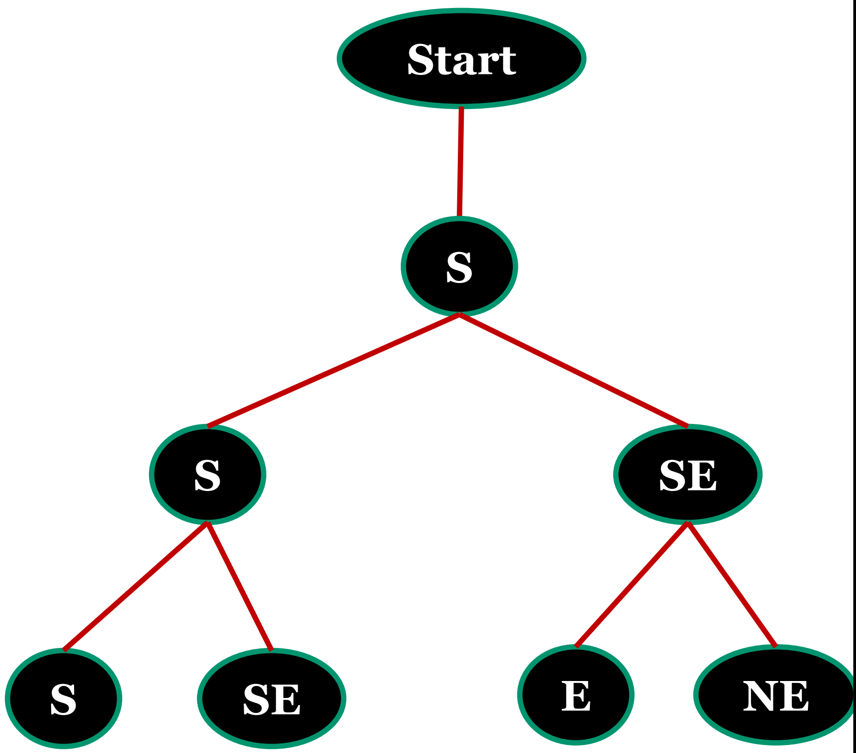
IN

Start S S SE

OUT

Breadth-first Search (BFS)

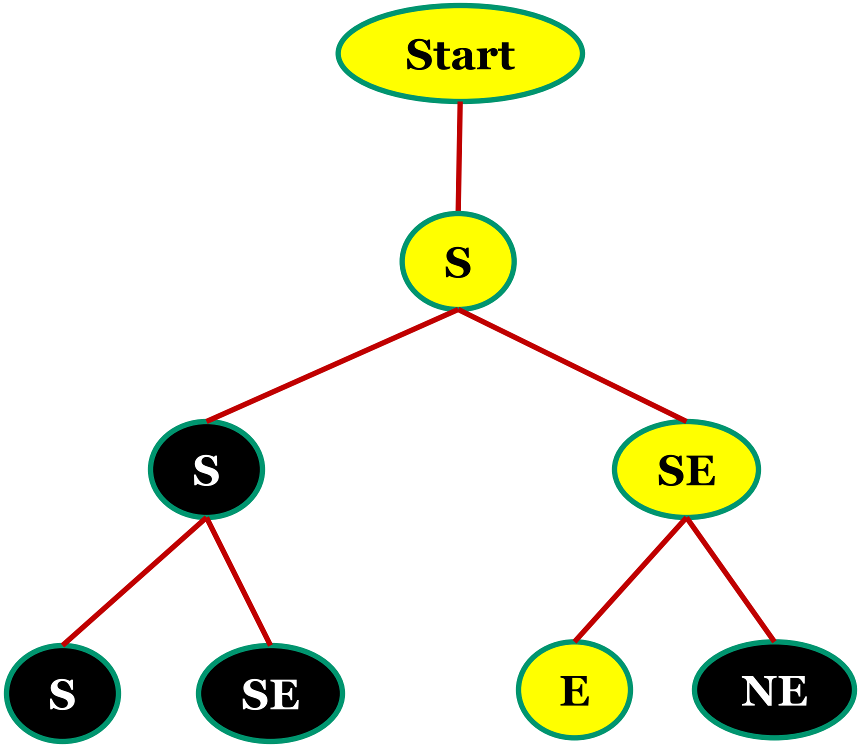




- The FIFO queue continues until the goal node is found

Breadth-first Search (BFS)

- The path leading to the goal node (E) is **traced back up the tree** which maps out the directions that the robot must follow to reach the goal.
- Traveling **back up the tree**, we can see that the robot from the start would have to go south, then south east, then east to reach the goal.



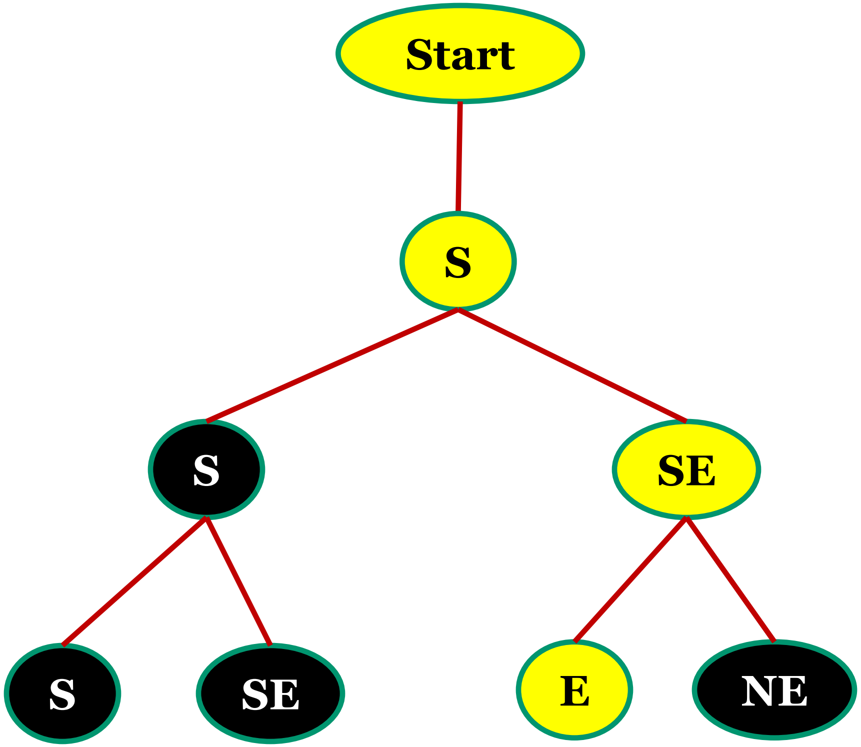
Assume that **E** is the goal,
Path is: **Start→ S→SE→E**

Breadth-first Search (BFS)

- In BFS, every node generated must remain in memory.
- The **number of nodes generated** is at most:

  $$O(b^d)$$

  where **b** represents the **maximum branching factor** for each node and **d** is the **depth one must expand to reach the goal**.



b=2 and d=3

Total # of nodes=2^3=8

Breadth-first Search (BFS)

- **Space complexity: O($\mathbf{b^d}$)** (keep every node in memory)

- We can see from this that a for **very large workspace** where the **goal is deep within the workspace**, the number of nodes could **expand exponentially** and demand a **very large memory requirement**.

- **Time Complexity:**

$$1+b+b^2+....+b^d=(b^{d+1}-1)/(b-1)=O(b^d)$$

i.e., **exponential** in d.

Breadth-first Search (BFS)

Depth	Nodes	Time	Memory
2	1100	.11 seconds	1 megabyte
4	111,100	11 seconds	106 megabytes
6	10^7	19 minutes	10 gigabytes
8	10^9	31 hours	1 terabytes
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3,523 years	1 exabyte

Figure 3.11 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 10,000 nodes/second; 1000 bytes/node.

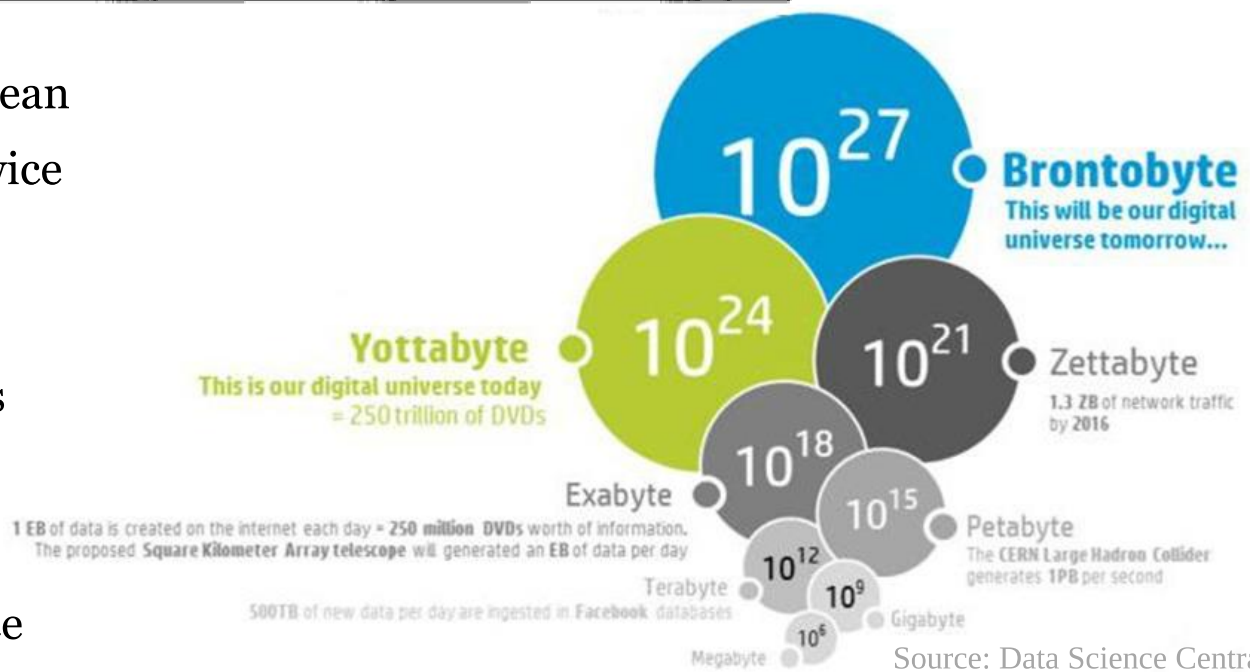
Space Complexity

$O(b^d)$

Time Complexity

$O(b^d)$

- Zettabyte: fills the Pacific ocean
- Exabyte: covers Germany twice
- Petabyte: covers Manhattan
- Terabyte: 2 container ships
- Gigabyte: 3 container lorries
- Megabyte: 8 bags of rice
- Kilobyte: cup of rice
- Byte of data: one grain of rice



Source: Data Science Central

Breadth-first Search (BFS)

◇ High memory requirement.

◇ **Exhaustive** search as it will process every node.

◇ Doesn’t get stuck.

◇ Finds the shortest path (minimum number of steps).

Outline

- Introductory Concepts
- Discrete Planning
- Breadth-first Search (BFS)
- **Depth-first Search (DFS)**
- Dijkstra's Algorithm
- Best-first
- A* Algorithm

Depth-first Search (DFS)

Step 1. Form a stack S and set it to the initial state (for example, the Root).

Step 2. Until the S is empty or the goal state is found

do:

Step 2.1 Determine if the first element in the S is the goal.

Step 2.2 If it is not

Step 2.2.1 Remove the first element in S.

Step 2.2.2 Apply the rule to generate new state(s) (successor states).

Step 2.2.3 If the new state is the goal state quit and return this state

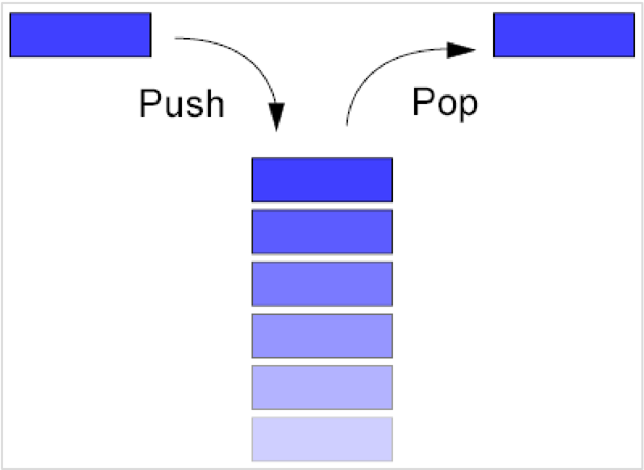
Step 2.2.4 Otherwise add the new state to the beginning of the stack

Step 3. If the goal is reached, success; else failure.

Reading Material

Depth-first Search (DFS)

- DFS uses the **stack** as data structure.
- Stack is a **Last-In-First-Out (LIFO)** data structure.
- The stack contains the list of discovered nodes. The **most recent discovered node** is put (pushed) on top of the LIFO stack.
- The **next node** to be expanded is then taken (popped) **from the top** of the stack and all of its successors are added to the stack.

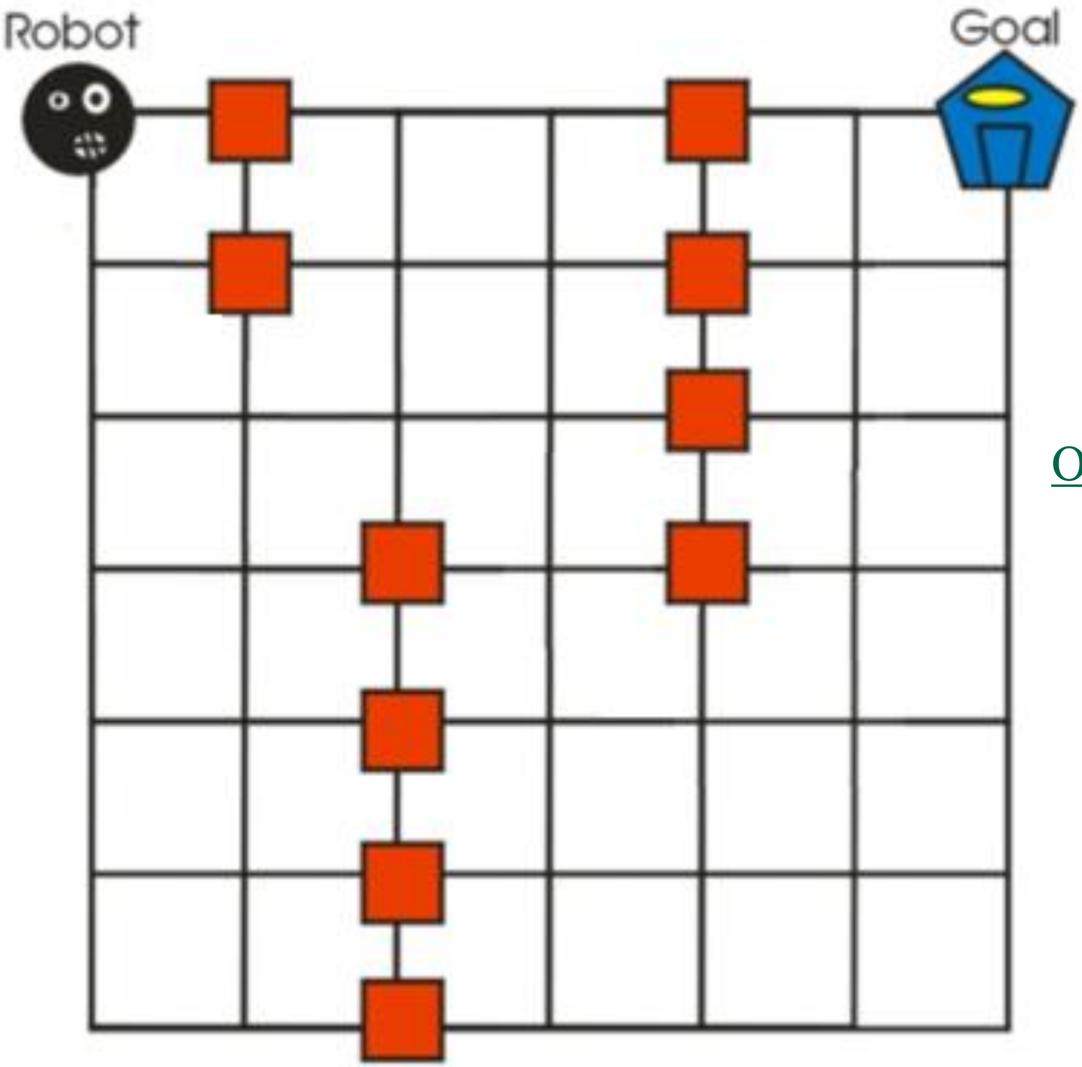


Stack (LIFO)

http://wiki.ugcnet4cs.in/t/Stack

Depth-first Search (DFS)

Start

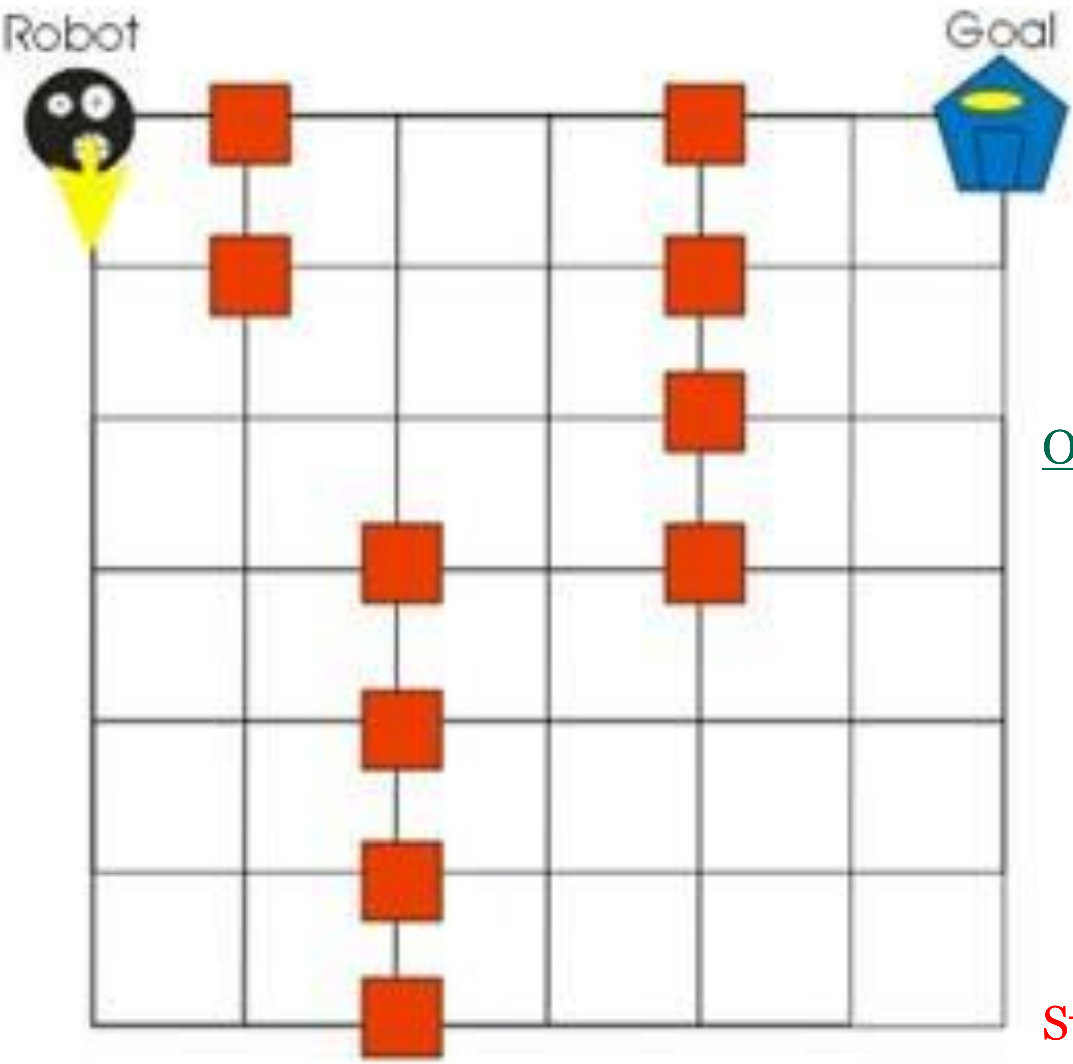


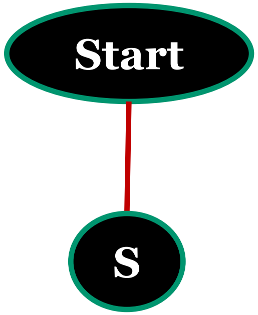
OUT IN

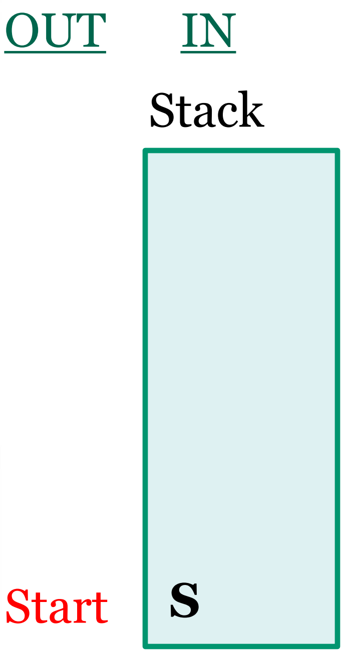
Stack

Start

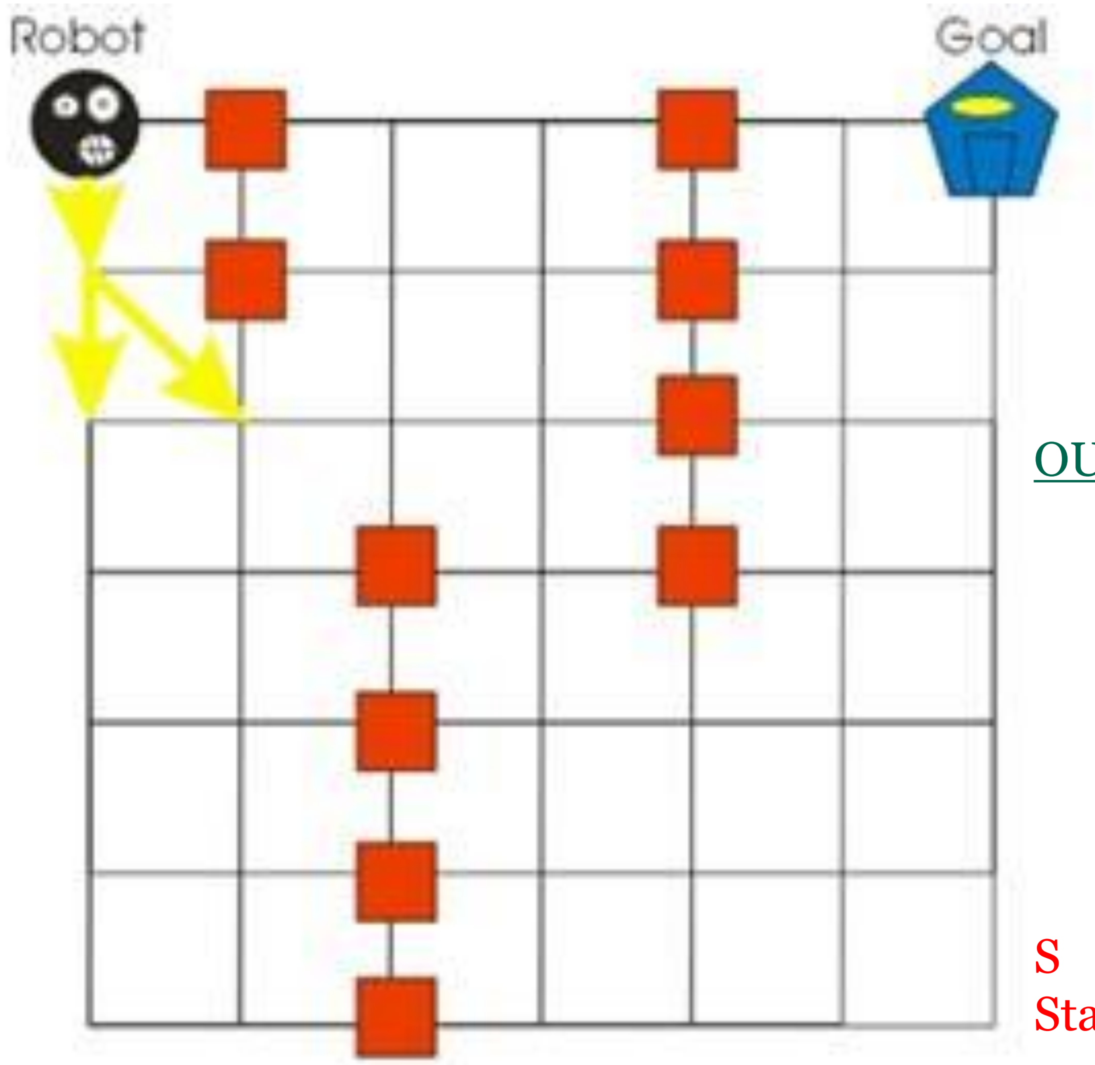
Depth-first Search (DFS)

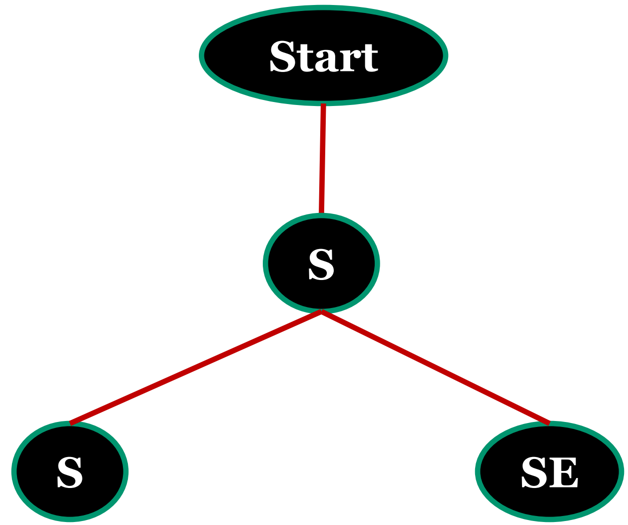






Depth-first Search (DFS)



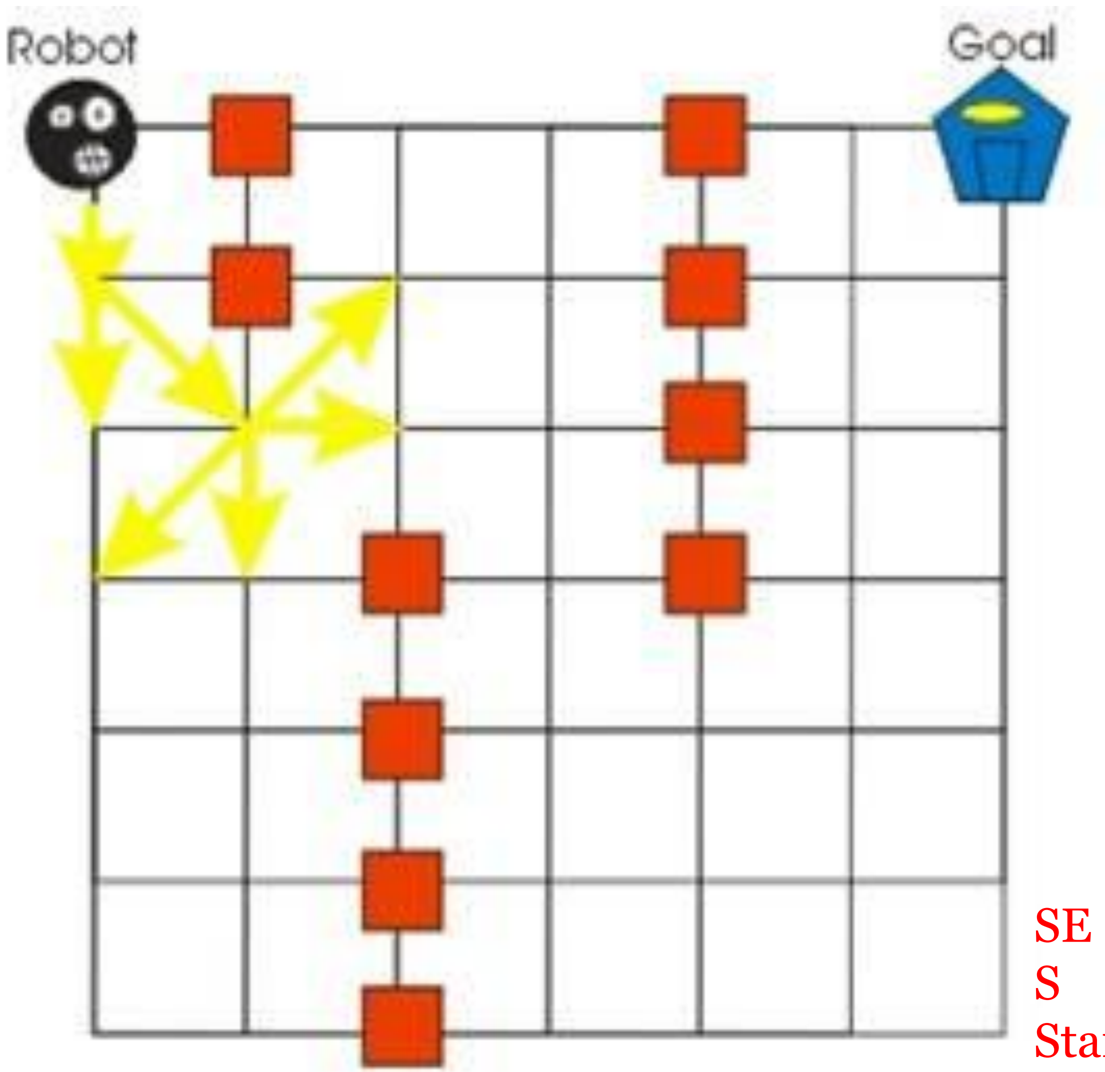


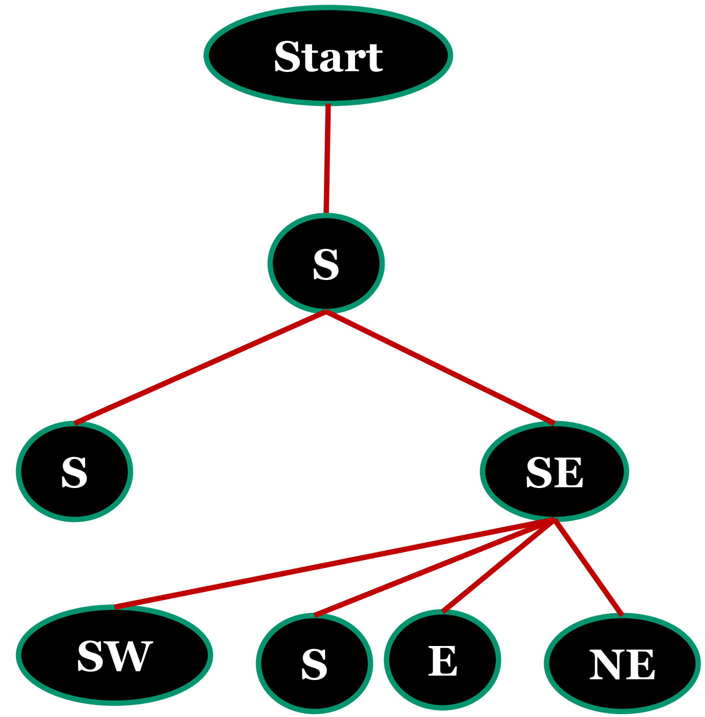
OUT IN

Stack

OUT	Stack
S	SE
Start	S

Depth-first Search (DFS)

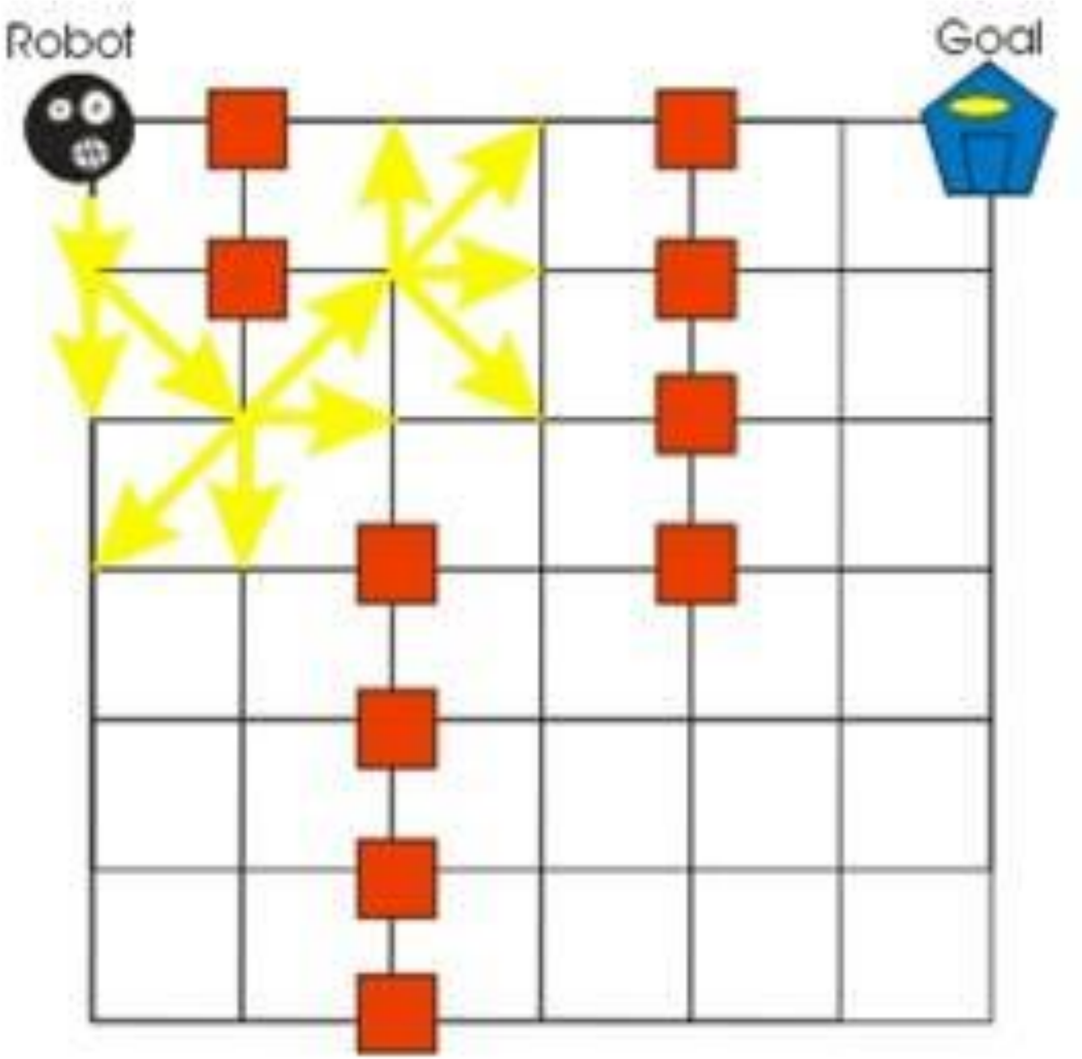




SE
S
Start

NE
E
S
SW
S

Depth-first Search (DFS)



- The next node to be expanded would be **NE** and its successors would be added to the stack and this loop continues until the goal is found.
- Once the goal is found, you can then **trace back** through the tree to obtain the path for the robot to follow.

Depth-first Search (DFS)

- Depth first search usually requires a **considerably less amount of memory** that BFS.
- This is mainly because DFS does not always expand out every single node at each depth.
- However the DFS could **continue down an unbounded branch forever even if the goal is not located** on that branch.
- **Time Complexity:** $O(b^d) \rightarrow$ terrible if d is much larger than b but if solutions are dense, may be much faster than BFS.
- **Space Complexity:** O(bd), i.e., linear space!

Depth-first Search (DFS)

- Low memory requirement.
- Full state space search in that every node is processed, i.e, it’s **exhaustive** within its set limits.
- Could get stuck exploring infinite paths.
- Used if there are many solutions and you need only one.

Depth-first Search (DFS)

- **BFS vs. DFS**

	BFS	DFS
Space Complexity	More expensive	Less expensive. Requires only O(d) space irrespective of number of children per node.
Time Complexity	More time efficient. A vertex at lower level (closer to the root) is visited first before visiting a vertex that is at higher level (far away from the root)	Less time efficient.

Depth-first Search (DFS)

- **BFS vs. DFS**

Space Complexity: $O(b^d)$
Time Complexity: $O(b^d)$

BFS is preferred if

◇ The branching factor is not too large (hence memory costs);

◇ A solution appears at a relatively shallow level;

◇ No path is excessively deep.

Space Complexity: O(bd)
Time Complexity: $O(b^d)$

DFS is preferred if

◇ The tree is deep;

◇ Solutions occur deeply in the tree;

◇ The branching factor is not excessive.

Outline

- Introductory Concepts
- Discrete Planning
- Breadth-first Search (BFS)
- Depth-first Search (DFS)
- **Dijkstra’s Algorithm**
- Best-first
- A* Algorithm

Dijkstra's Algorithm

- **British Museum Search**
  - ◊ Blind search finds only one **arbitrary solution** instead of the optimal solution.
  - ◊ To find the optimal solution with DFS or BFS, you must not stop searching when the first solution is discovered. Instead, the search needs to continue until it reaches all the solutions, so you can compare them to pick the best.
  - ◊ The strategy for finding the optimal solution is called **British Museum search or brute-force search**.

The inventors called this procedure the British Museum algorithm "*... since it seemed to them as sensible as placing monkeys in front of typewriters in order to reproduce all the books in the British Museum [5].*"

Dijkstra’s Algorithm

Dijkstra’s algorithm (**Dynamic Programming**) is a graph search algorithm that solves **the single-source shortest path problem** for a fully connected graph with nonnegative edge path costs, producing a shortest path tree.

Dynamic programming (a fancy name for **divide-and-conquer** with a table) approaches are based on the recursive division of a problem into simpler sub-problems.

Dijkstra’s algorithm is **uninformed/blind**, meaning it does not need to know the target node before hand and doesn’t use heuristic information.

Dijkstra's Algorithm

1. Init

Set start distance to 0, dist[s]=0,

others to infinite: dist[i]=∞ (for i $\neq$s),

Set Ready = { } .

2. Loop until all nodes are in Ready

Select node n with shortest known distance that is not in Ready
set Ready = Ready + {n} .

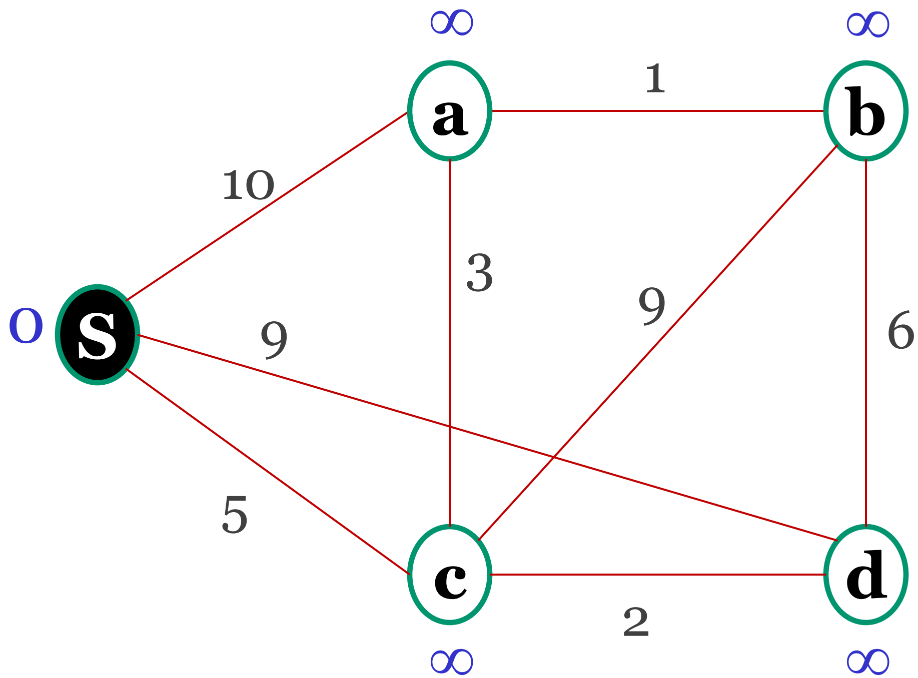
FOR each neighbor node m of n

IF dist[n]+edge(n,m) < dist[m] /* shorter path found */

THEN { dist[m] = dist[n]+edge(n,m); pre[m] = n;}

Reading Material

Dijkstra's Algorithm



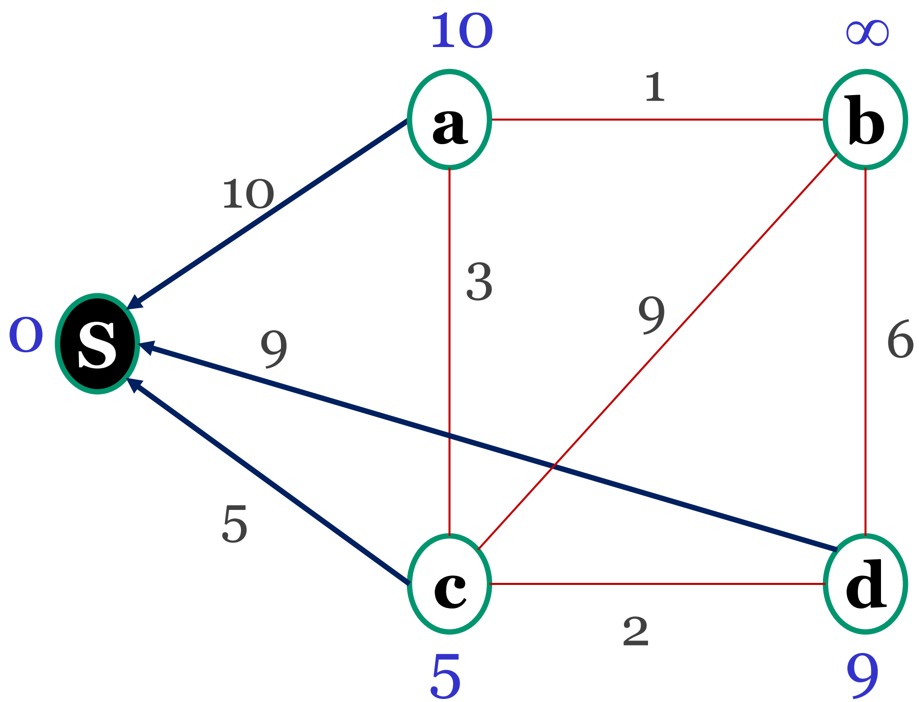
From s to:	S	a	b	c	d
Distance	0	∞	∞	∞	∞
Predecessor	-	-	-	-	-

Step 0: Init list, no predecessors

Ready = {}

[6]

Dijkstra's Algorithm

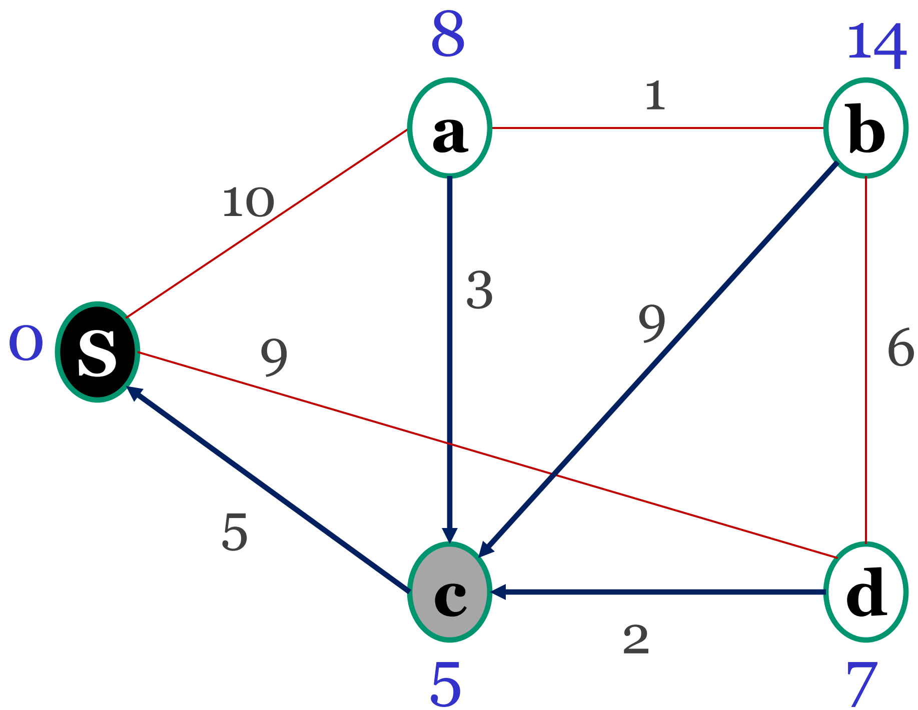


From s to:	S	a	b	c	d
Distance	0	10	∞	5	9
Predecessor	-	s	-	s	s

Step 1: Closest node is s, add to Ready

Update distances and pred. to all neighbors of *s*, **Ready = {S}**

Dijkstra's Algorithm

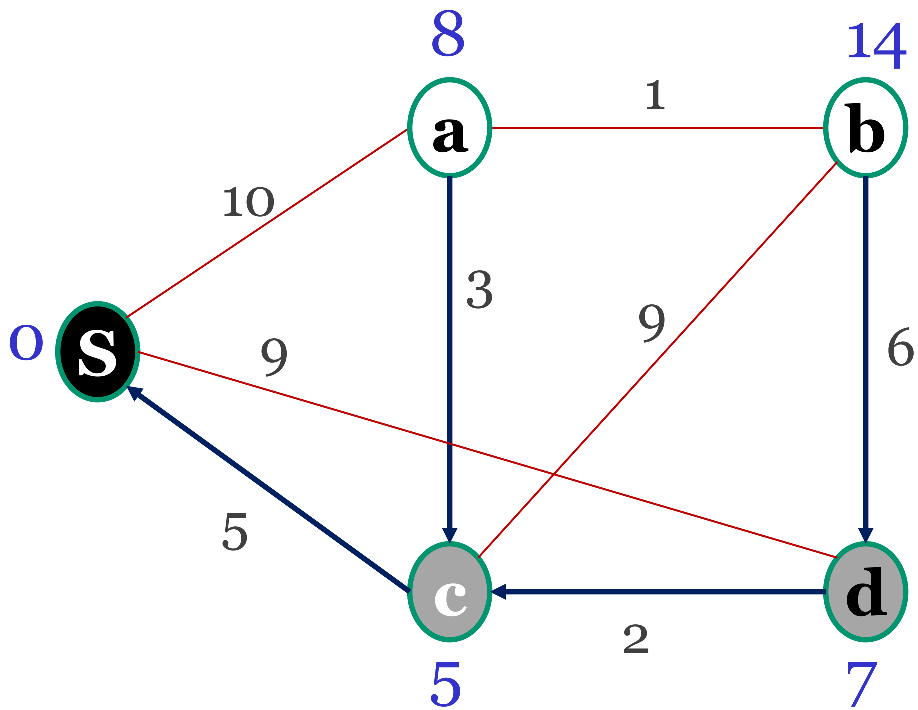


From s to:	S	a		b	c	d	
Distance	0	~~10~~	8	14	5	~~9~~	7
Predecessor	-	~~s~~	c	c	s	~~s~~	c

Step 2: Next closest node is c, add to Ready

Update distances and pred. for a and d, **Ready = {S, c}**

Dijkstra's Algorithm

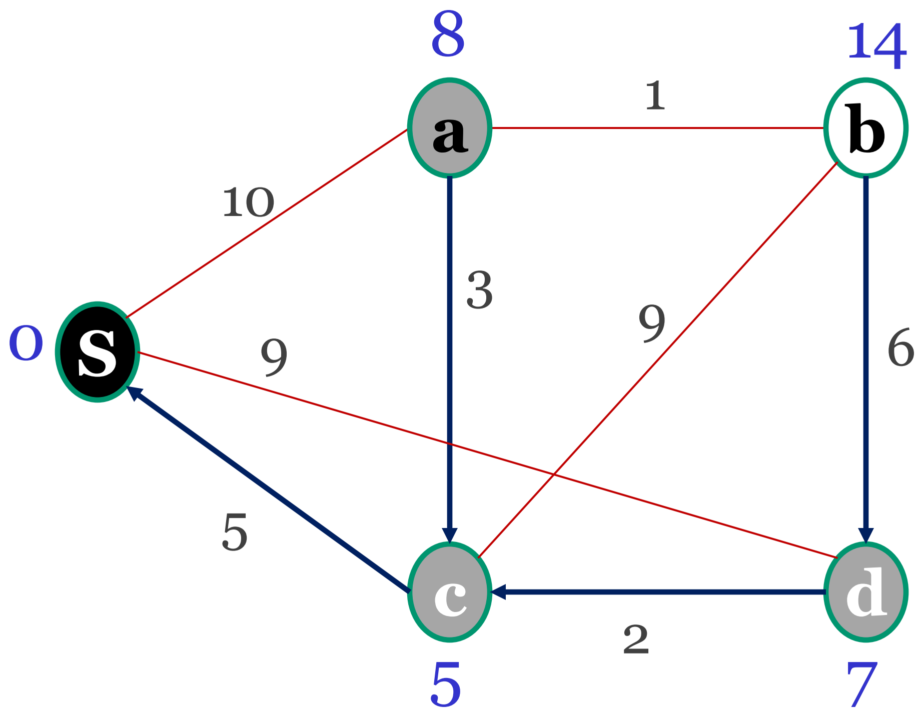


From s to:	S	a	b		c	d
Distance	0	8	~~14~~	13	5	7
Predecessor	-	c	~~c~~	d	s	c

Step 3: Next closest node is d, add to Ready

Update distance and pred. for b. **Ready = {s, c, d}**

Dijkstra's Algorithm

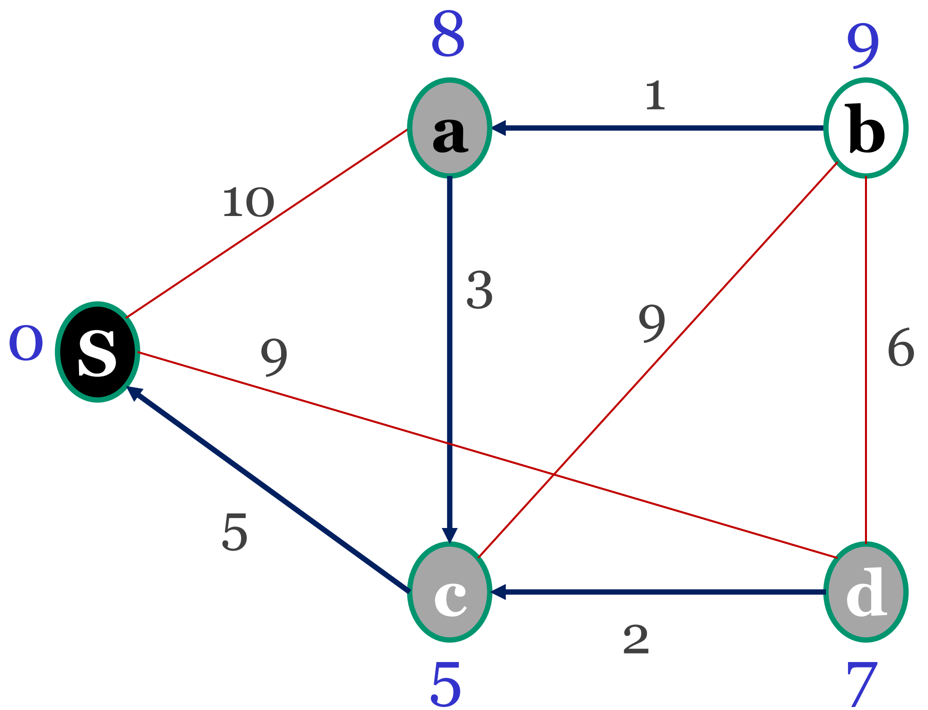


From s to:	S	a	b		c	d
Distance	0	8	~~13~~	9	5	7
Predecessor	-	c	~~d~~	a	s	c

Step 4: Next closest node is a, add to Ready

Update distance and pred. for b. **Ready = {S, a, c, d}**

Dijkstra's Algorithm

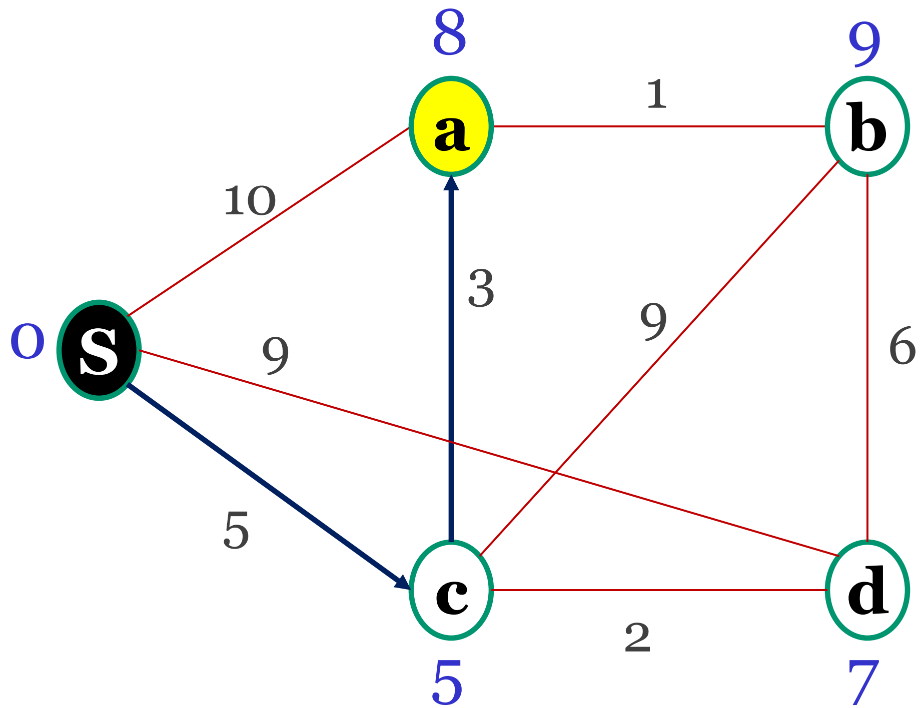


From s to:	S	a	b	c	d
Distance	0	8	9	5	7
Predecessor	-	c	a	s	c

Step 5: Closest node is b, add to Ready

check all neighbors of s. Ready = {S, a, b, c, d} **complete!**

Dijkstra's Algorithm

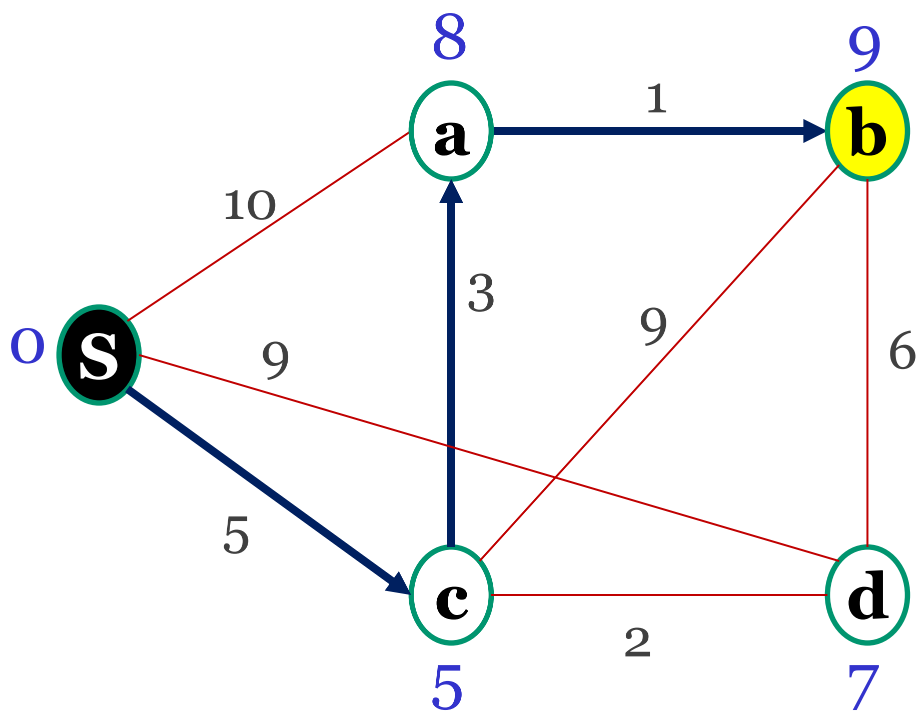


- dist[a] = 8
- pre[a] = c
- pre[c] = S
- Shortest path: S→c →a,
- Length is 8

From s to:	S	a	b	c	d
Distance	0	8	9	5	7
Predecessor	-	c	a	s	c

Shortest path between s and a is {S, c, a} ⇒ length=8

Dijkstra's Algorithm

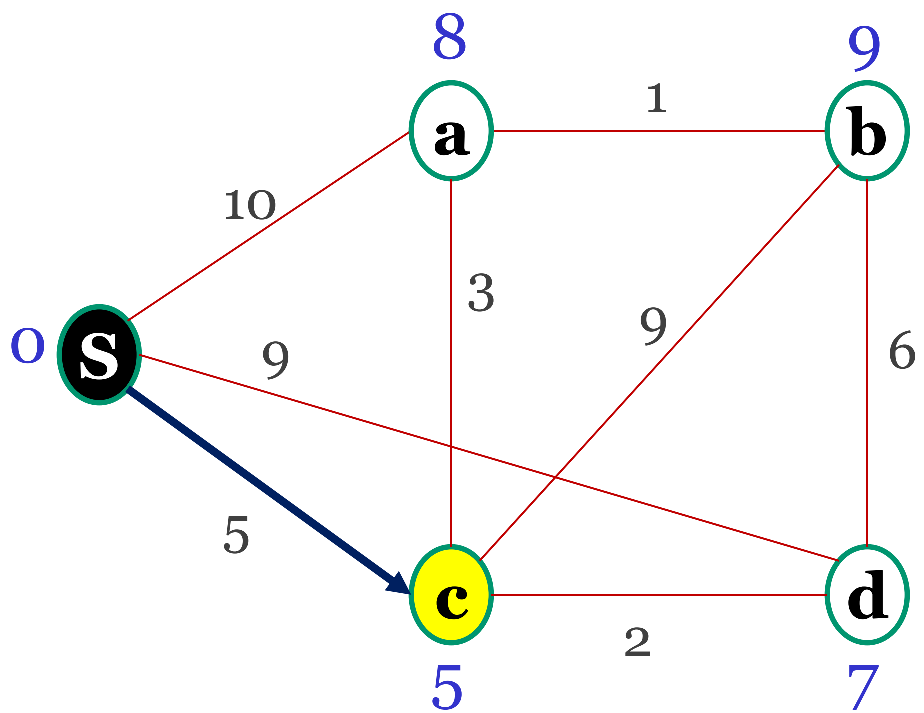


- dist[b] = 9
- pre[b] = a
- pre[a] = c
- pre[c] = S
- Shortest path: S→c →a →b

From s to:	S	a	b	c	d
Distance	0	8	9	5	7
Predecessor	-	c	a	s	c

Shortest path between s and b is {S, c, a,b} ⇒ length=9

Dijkstra's Algorithm

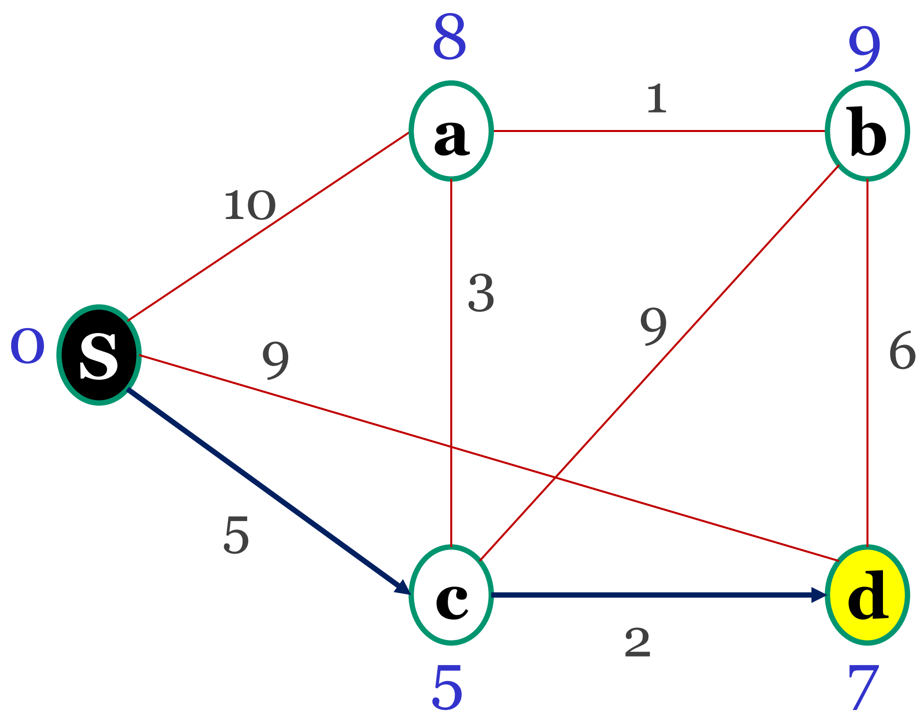


- dist[c] = 5
- pre[c] = S
- Shortest path: S→c

From s to:	S	a	b	c	d
Distance	0	8	9	5	7
Predecessor	-	c	a	s	c

Shortest path between s and d is {S, c} ⇒ length=5

Dijkstra's Algorithm



- dist[d] = 7
- pre[d] = c
- pre[c] = S
- Shortest path: S→c →d

From s to:	S	a	b	c	d
Distance	0	8	9	5	7
Predecessor	-	c	a	s	c

Shortest path between s and d is {S, c, d} ⇒ length=7

Dijkstra’s Algorithm

As can be seen from previous example, instead of solving subproblems recursively, solve them sequentially and store their solutions in a table. The trick is to solve them in the right order so that whenever the solution to a subproblem is needed, it is already available in the table [I. Parberry. *Problems on algorithms*. Prentice-Hall, 1995].

Outline

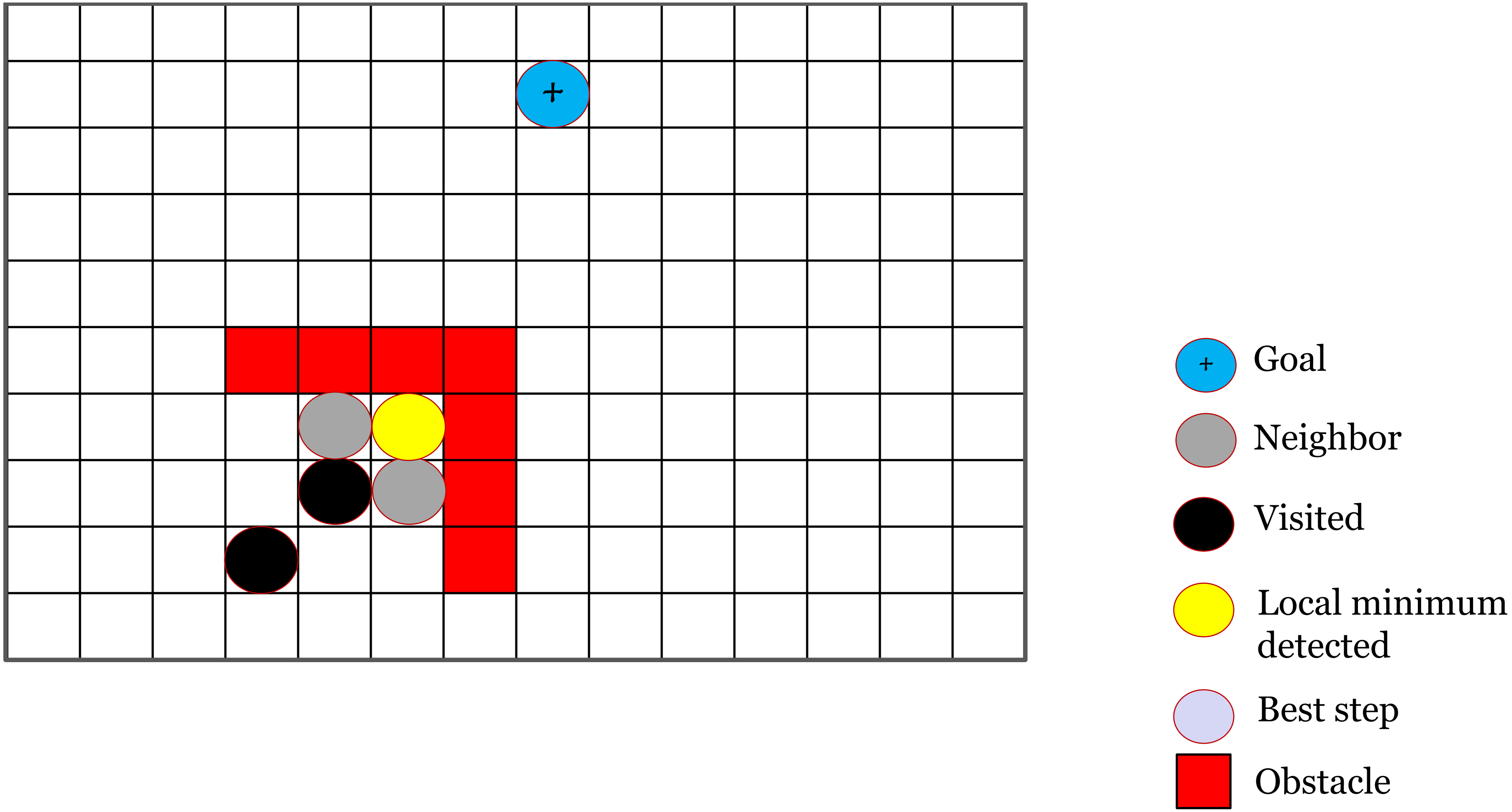
- Introductory Concepts
- Discrete Planning
- Breadth-first Search (BFS)
- Depth-first Search (DFS)
- Dijkstra’s Algorithm
- **Best-first**
- A* Algorithm

Best-first

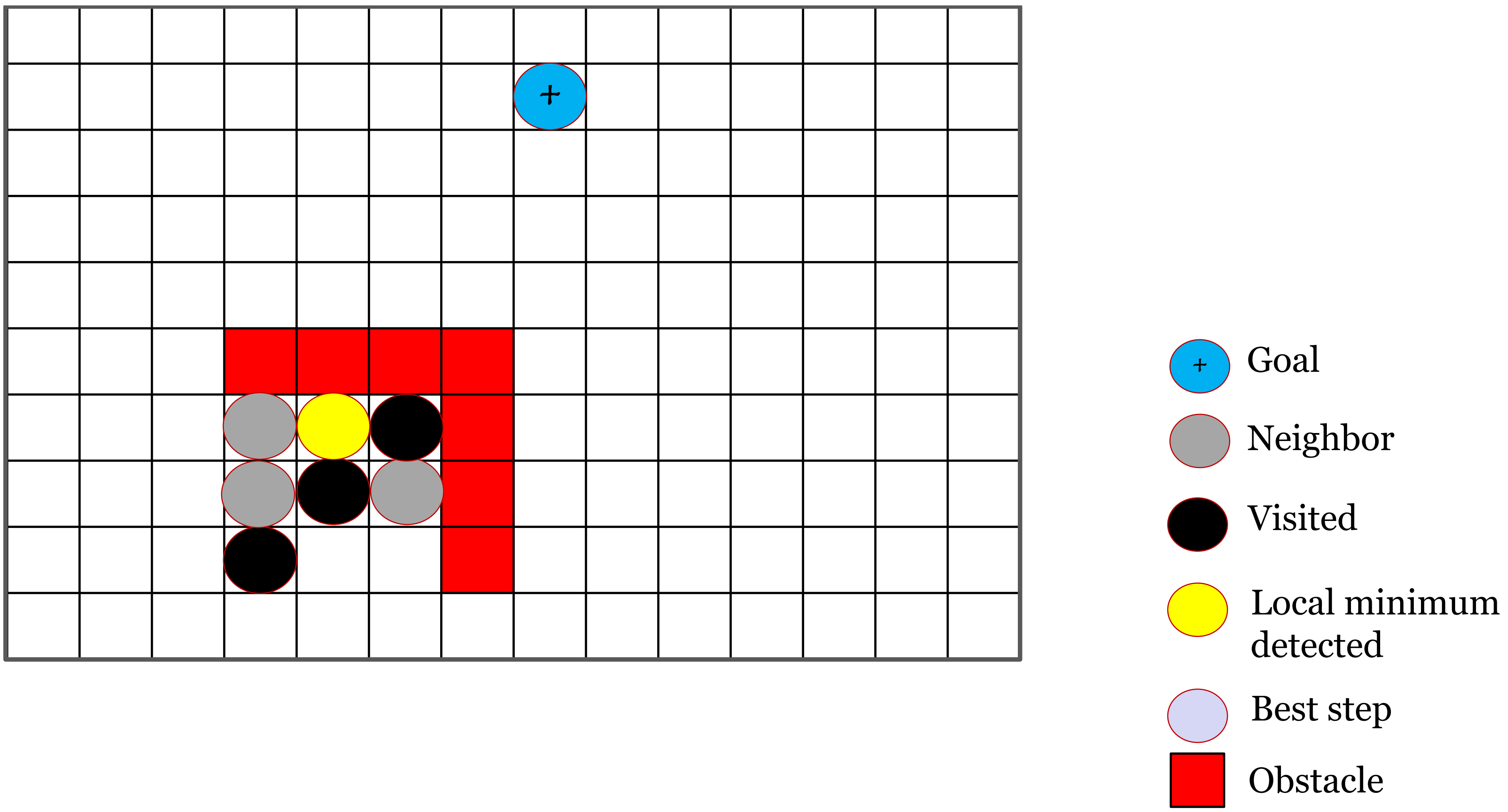
1. **Workspace** discretized into cells
2. **Insert** (x_{init}, y_{init}) into list **OPEN**
3. **Find** all **8-way neighbors** to (x_{init}, y_{init}) that have not been previously visited and insert into OPEN
4. **Sort** neighbors by minimum potential
5. **Form** paths from neighbors to (x_{init}, y_{init})
6. **Delete** (x_{init}, y_{init}) from OPEN
7. (x_{init}, y_{init}) = minPotential(OPEN)
8. **GOTO** 2 until (x_{init}, y_{init})=goal (SUCCESS) or OPEN empty (FAILURE)

Reading Material

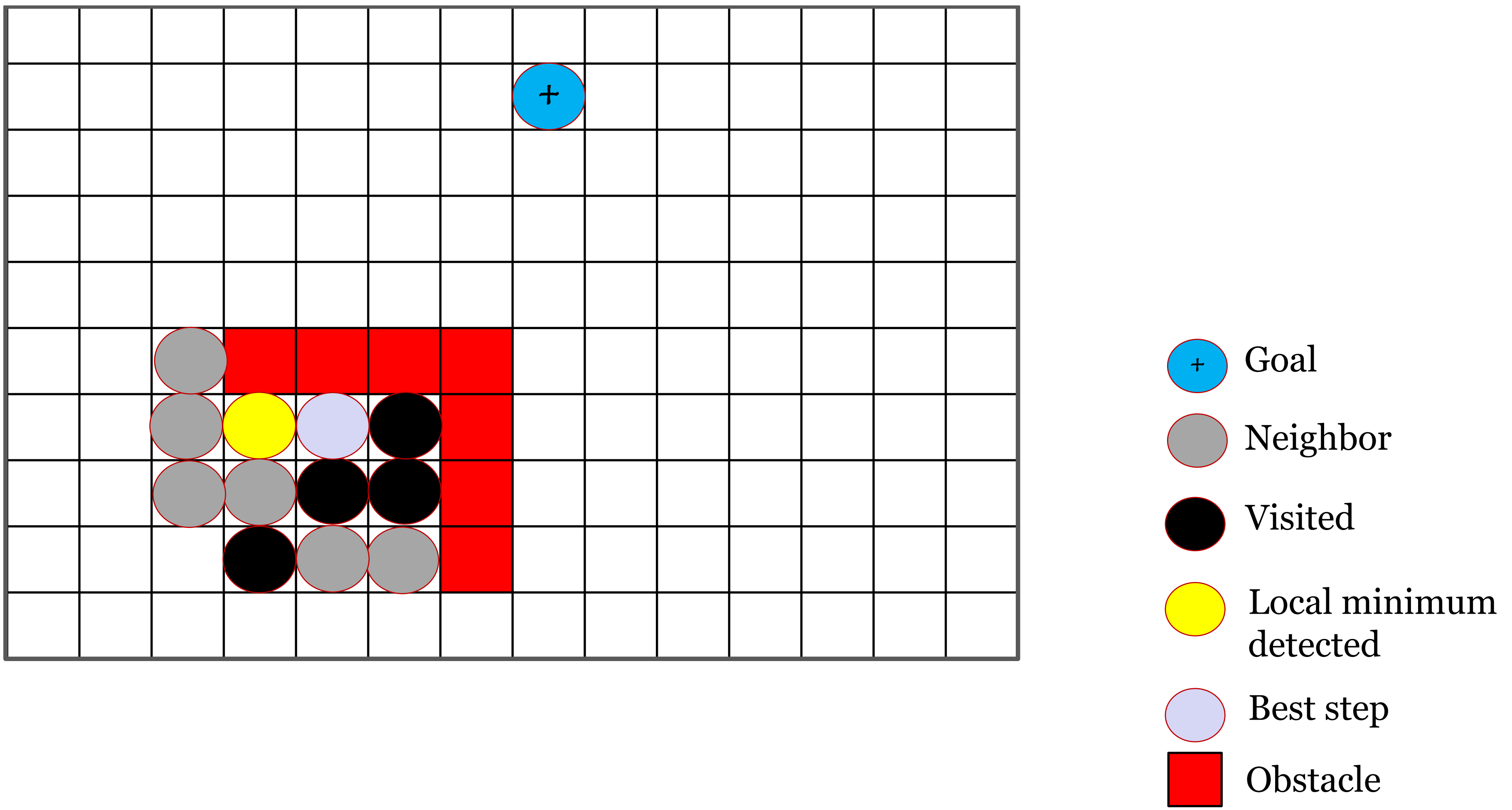
Best-first


+
+
Goal
Neighbor
Visited
Local minimum
detected
Best step
Obstacle

Best-first

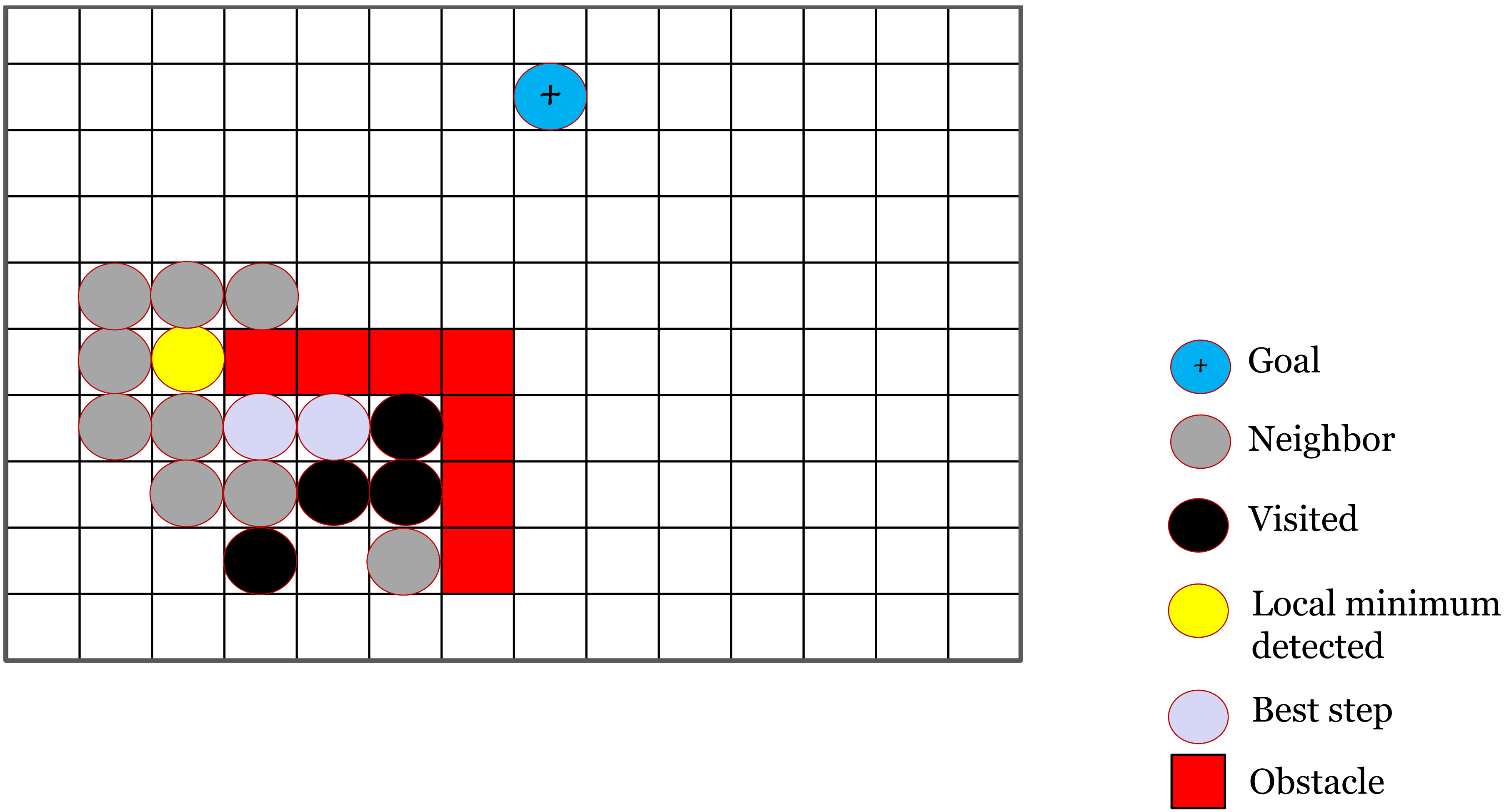

Goal
Neighbor
Visited
Local minimum detected
Best step
Obstacle

Best-first



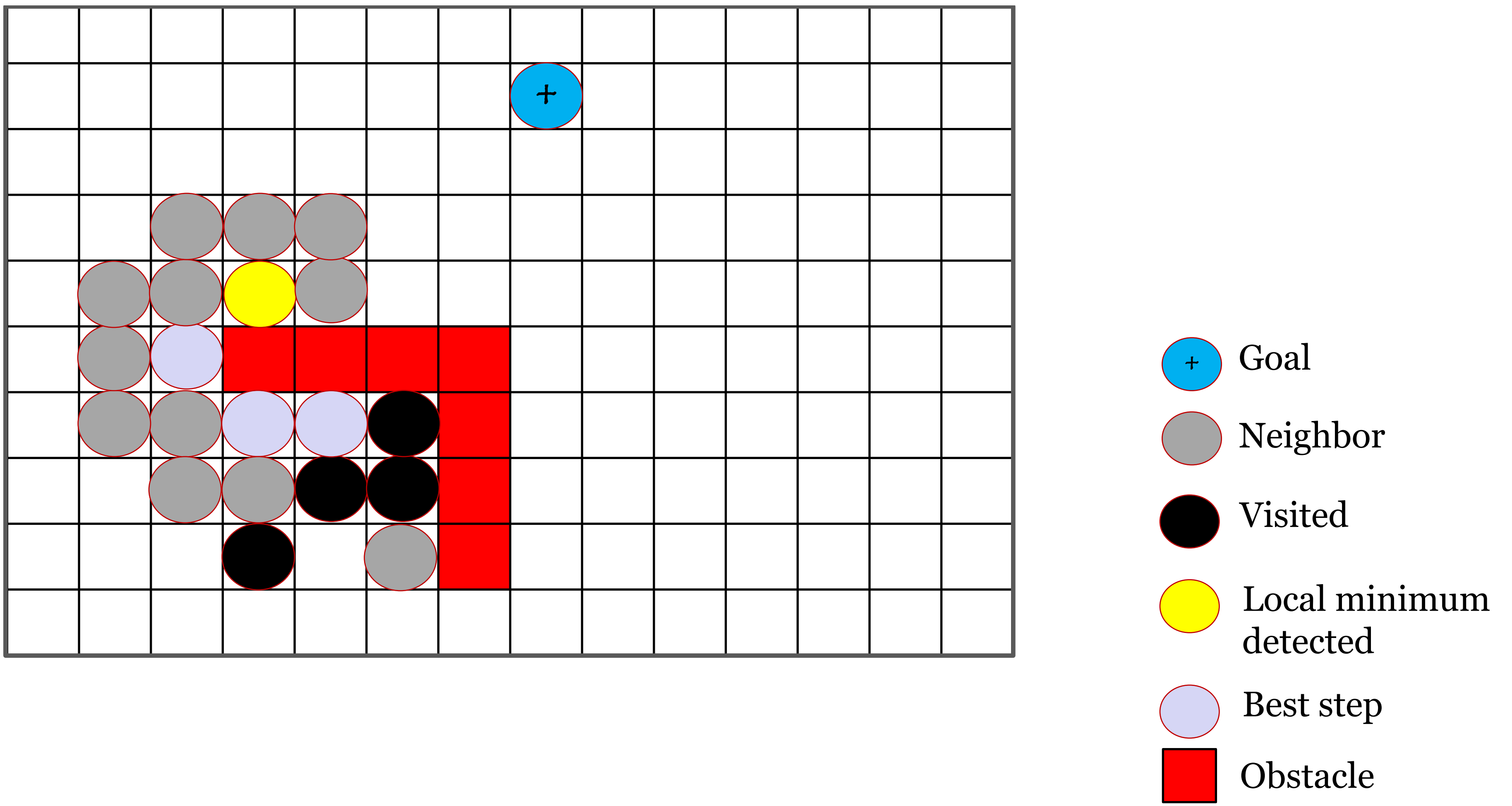
Goal
Neighbor
Visited
Local minimum detected
Best step
Obstacle

Best-first



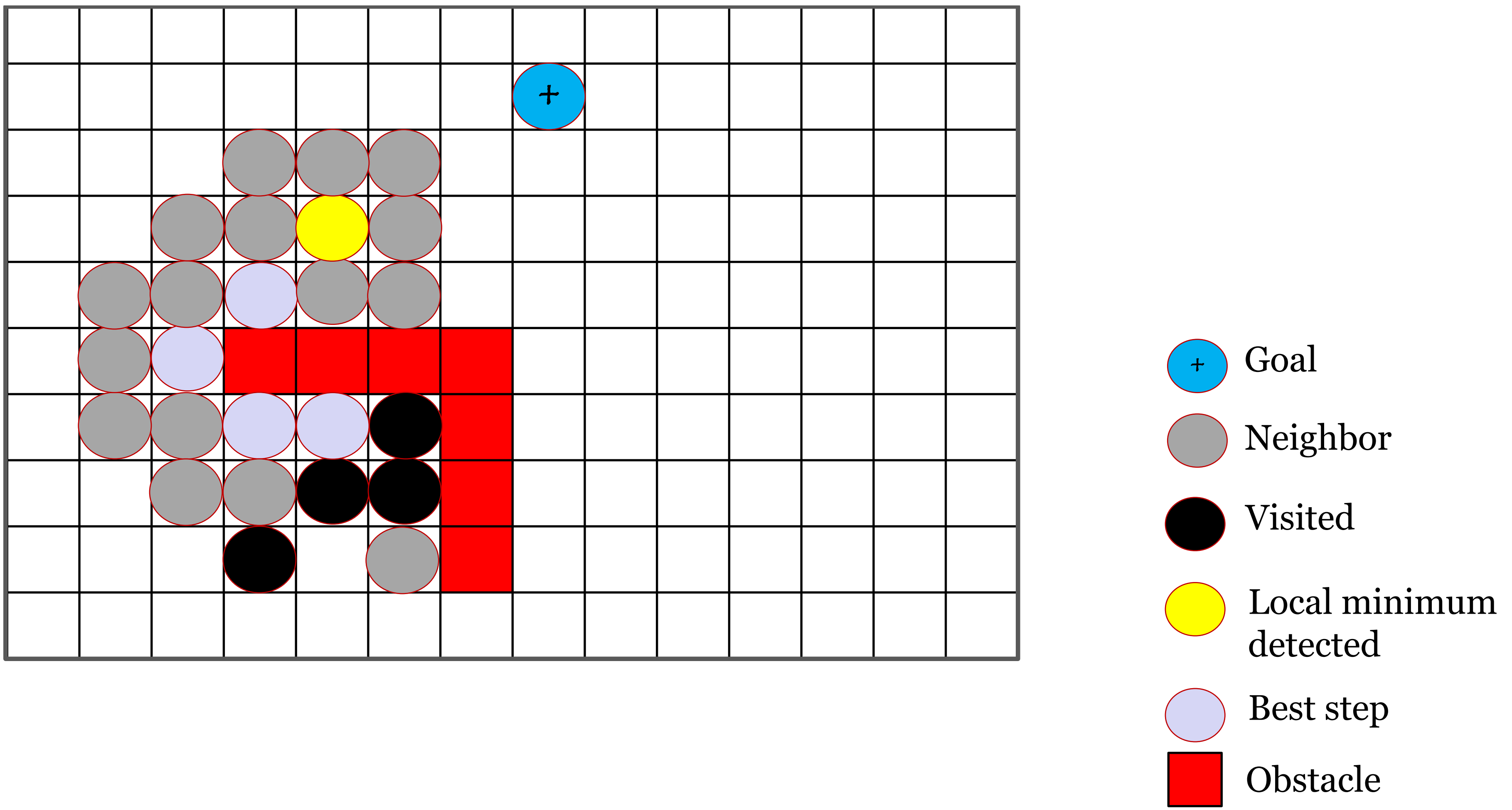
Goal
Neighbor
Visited
Local minimum detected
Best step
Obstacle

Best-first



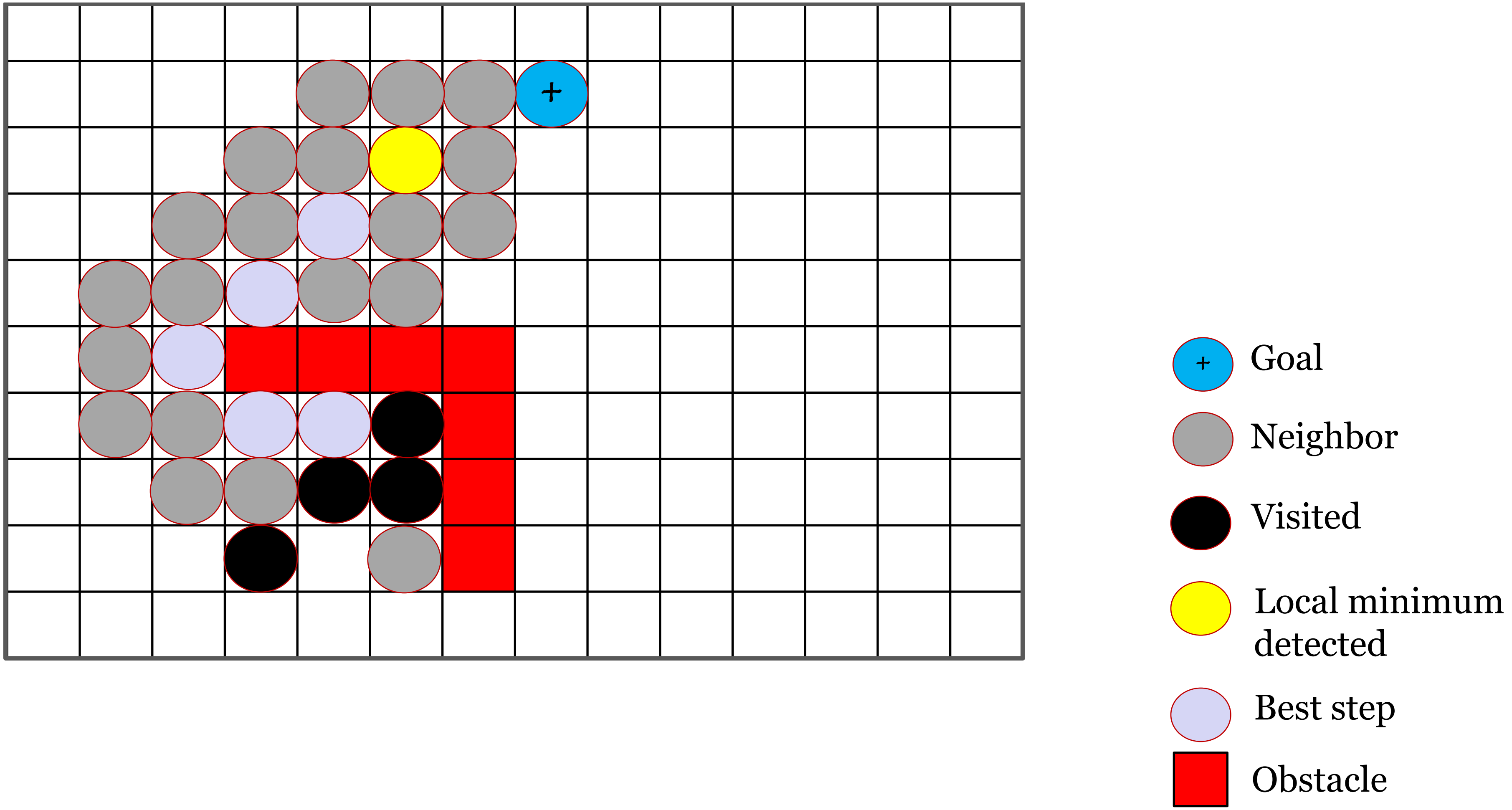
Goal
Neighbor
Visited
Local minimum detected
Best step
Obstacle

Best-first

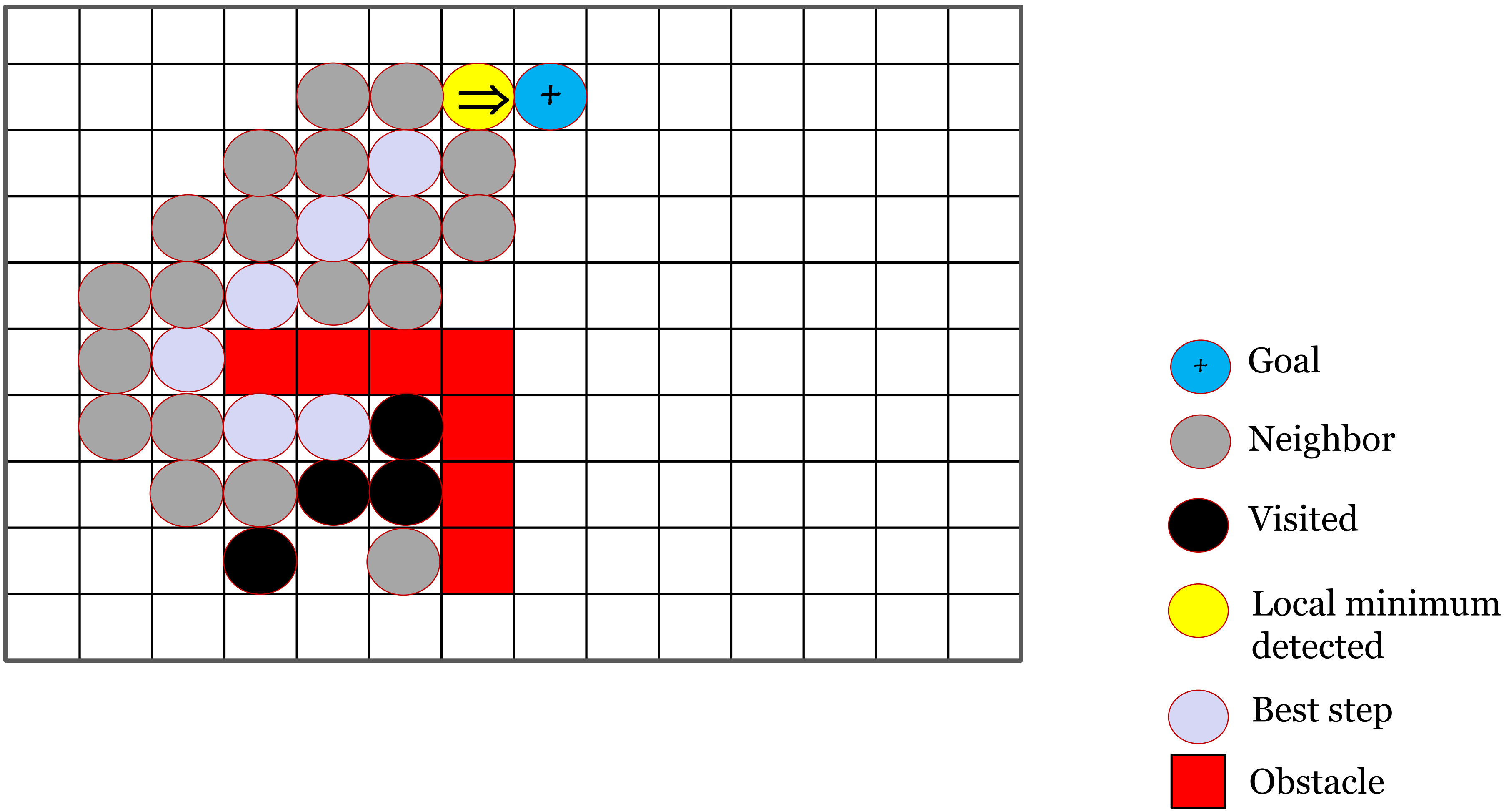


+
Goal
Neighbor
Visited
Local minimum detected
Best step
Obstacle

Best-first

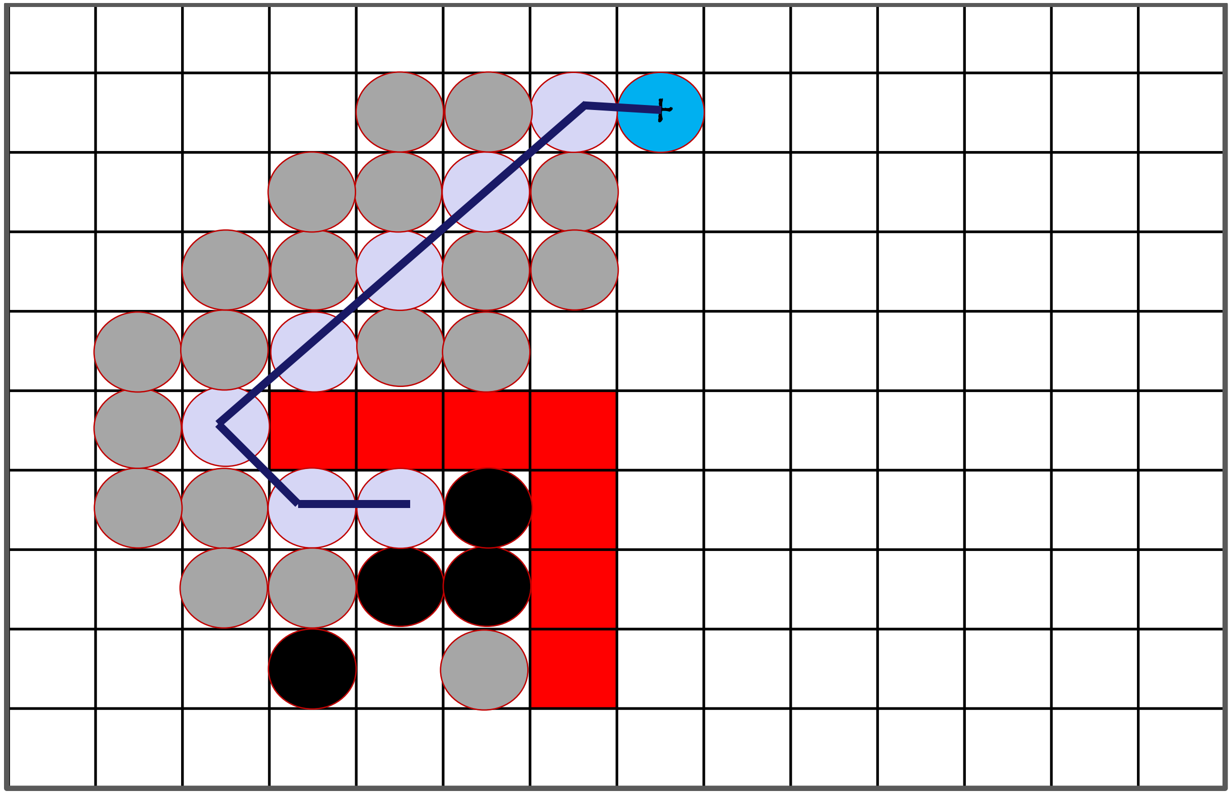

Goal
Neighbor
Visited
Local minimum detected
Best step
Obstacle

Best-first



⇒
+
Goal
Neighbor
Visited
Local minimum detected
Best step
Obstacle

Best-first




+
Goal
Neighbor
Visited
Local minimum detected
Best step
Obstacle

Best-first

◇ It is a kind or **mixed depth and breadth first** search.

◇ Adds the successors of a node to the expand list.

◇ All nodes on the list are sorted according to the **heuristic values**.

◇ Expand **most desirable unexpanded** node.

◇ Special Case: **A***.

Outline

- Introductory Concepts
- Discrete Planning
- Breadth-first Search (BFS)
- Depth-first Search (DFS)
- Dijkstra's Algorithm
- Best-first
- **A* Algorithm**

A* Algorithm

Pronounced "**A-Star**"; heuristic algorithm for computing the **shortest path** from one given **start node** to one given **goal node**.

The A^* search algorithm is a variant of dynamic programming (Dijkstra) that tries to **reduce the total number of states explored** by incorporating a **heuristic estimate** of the cost to get the goal from a given state.

A* Algorithm

1. Use a queue to store all the partially expanded paths.
2. Initialize the queue by adding to the queue a zero length path from the root node to nowhere.
3. **Repeat**

   Examine the first path on the queue.

   If it reaches the goal node then success.

   Else {continue search}

   Remove the first path from the queue.

   Expand the last node on this path by one step.

   Calculate the cost of these new paths.

   Add these new paths to the queue.

Reading Material

A* Algorithm

Sort the queue in ascending order according to the sum of the cost of the expanded path and the estimated cost of the remaining path for each path.

If more than one path reaches a subnode

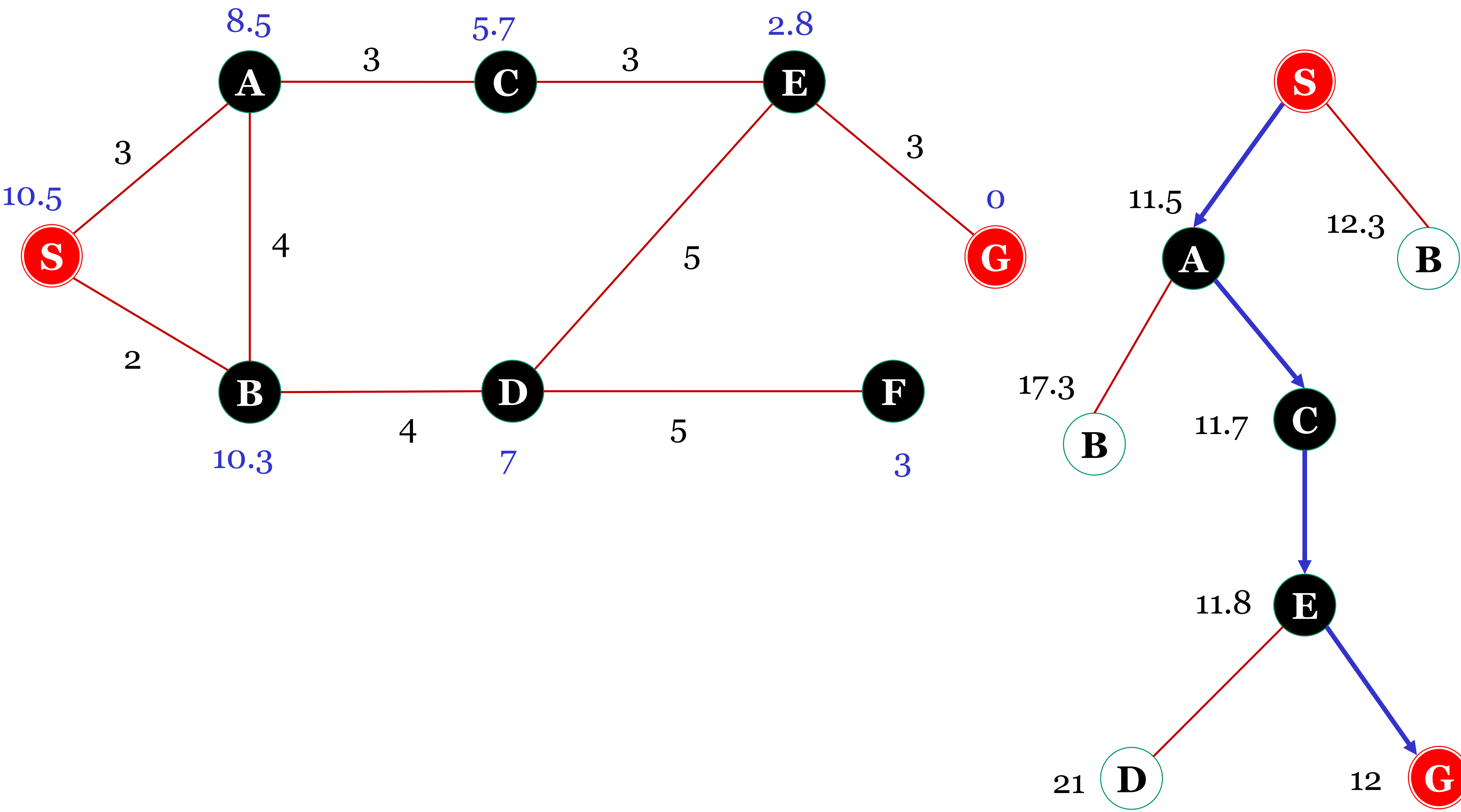
Then delete all but the minimum cost path.

Until the goal has been found or the queue is empty.

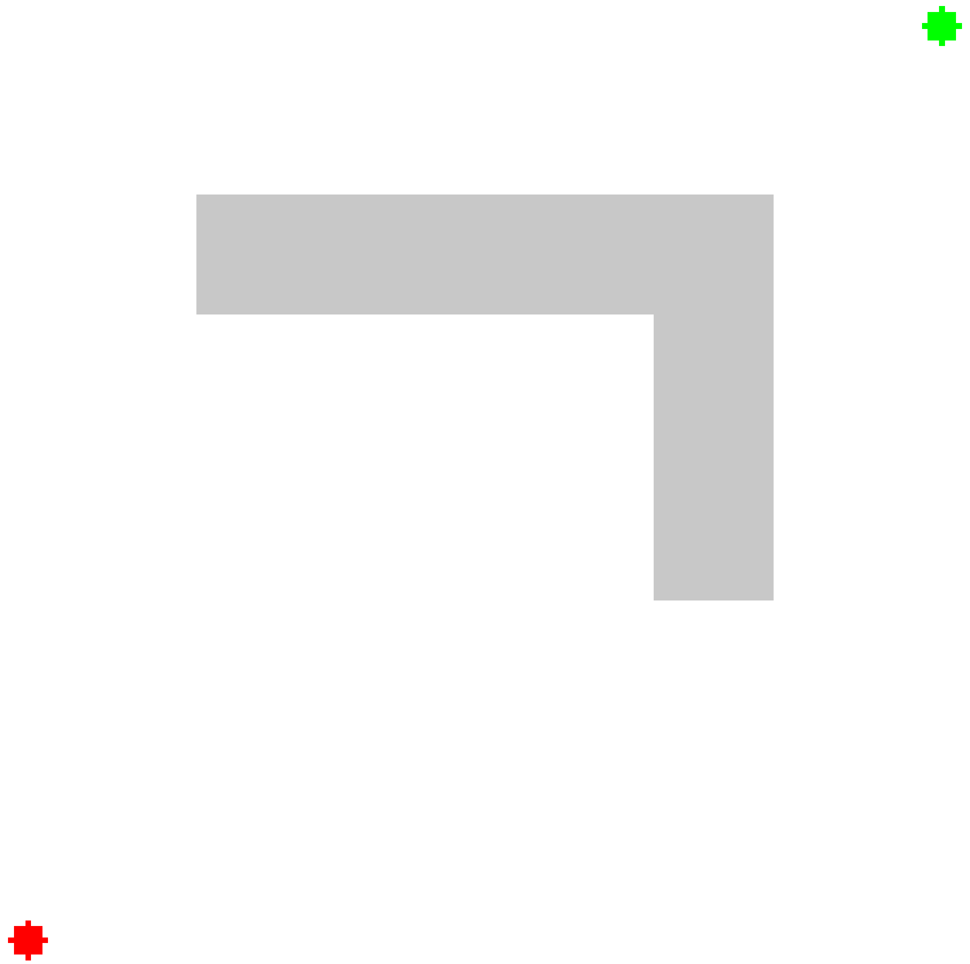
4. **If** the goal has been found return success and the path, otherwise return failure.

Reading Material

A* Algorithm


8.5
5.7
2.8
A
3
C
3
E
S
3
3
10.5
4
5
0
11.5
12.3
S
G
A
B
2
B
4
D
5
F
17.3
C
11.7
10.3
7
3
B
11.8
E
21
D
12
G

A* Algorithm



Source:
http://en.wikipedia.org/wiki/File:Astar_progress_animation.gif

A* Algorithm

- This algorithm may look complex since there seems to be the need to store incomplete paths and their lengths at various places.
- However, using a **recursive best-first search** implementation can solve this problem in an elegant way without the need for explicit path storing.
- The quality of the lower bound goal distance from each node greatly influences the timing complexity of the algorithm.
- The **closer the given lower bound is to the true distance**, the **shorter the execution time**.

References

1. LaValle, S. M. (2006). Planning algorithms. Cambridge university press.
2. van Wezel, W., Jorna, R. J., & Meystel, A. M. (Eds.). (2006). *Planning in Intelligent Systems: Aspects, motivations, and methods*. John Wiley & Sons.
3. Spong, M. W., Hutchinson, S., & Vidyasagar, M. *Robot modeling and control*. John Wiley & Sons, 2006.
4. Siegwart, R., Nourbakhsh, I. R., & Scaramuzza, D. (2011). *Introduction to autonomous mobile robots*. MIT press.
5. Crina Grosan and Ajith Abraham. *Intelligent Systems: A Modern Approach*. Springer, 2011.
6. http://dasl.mem.drexel.edu/Hing/BFSDFSTutorial.htm, Last accessed: November 20, 2016.