CSCI6650: Discrete Search

Instructor: Abdullah Al Redwan Newaz, PhD

Assistant Professor

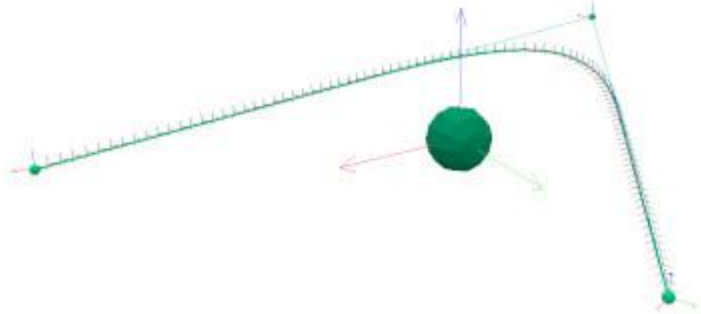
Department of Computer Science
Mathematics Building #345

Agenda

- **Path Finding**
  - Forward Search Template
  - Backward Search Template
  - Bidirectional Search Template
- **Algorithms**
  - Breadth First Search
  - Depth First Search
  - Dijkstra Algorithm
  - A* Algorithm
  - Bidirectional A* Algorithm
  - D* Algorithm
  - D* Lite Algorithm

What Is a Path Finding Problem?

- Given,
  - Initial position of a robot
  - Goal position of a robot
  - A disjoint set of obstacles



- Find
  - A set of geometric points on the workspace that connect the robot’s initial position to the goal position.

CoppeliSim API

- https://manual.coppeliarobotics.com/en/pathsAndTrajectories.htm
- https://manual.coppeliarobotics.com/en/regularApi/simCreatePath.htm

sim.createPath

Creates a path.

Synopsis

Python | Lua — CoppeliaSim script (Python)

```
int pathHandle = sim.createPath(list ctrlPts, int options = 0, int subdiv = 100,
                                float smoothness = 1.0, int orientationMode = 0,
                                list upVector = [0, 0, 1])
```

Arguments

- **ctrlPts**: control points, specified in row-major order, with *[x y z qx qy qz qw]* values for each path point
- **options**: bit-coded:
  - bit0 set (1): path is hidden during simulation
  - bit1 set (2): path is closed
  - bit2 set (4): generates an extruded shape
  - bit3 set (8): show individual path points
  - bit4 set (16): the path points' orientation is computed according to the orientationMode below
- **subdiv**: number of individual path points
- **smoothness**: value between 0.0 (linear interpolation) and 1.0 (100% Bezier interpolation)
- **orientationMode**: value specifiying how the individual path points are oriented along the path, if bit16 of options is set: 0: x along path, y is up, 1: x along path, z is up, 2: y along path, x is up, 3: y along path, z is up, 4: z along path, x is up, 5: z along path, y is up
- **upVector**: up vector, used for generating an extruded shape and for computing individual path point orientations

Return values

- **pathHandle**: handle of the created path object

Forward Search

At any point during the forward search, there will be three kinds of states:

- **Unvisited:** States that have not been visited yet. Initially, this is every state except x_I.
- **Dead:** States that have been visited, and for which every possible next state has also been visited. A next state of x is a state x' for which there exists a $u \in U(x)$ such that $x' = f(x,u)$. In a sense, these states are dead because there is nothing more that they can contribute to the search; there are no new leads that could help in finding a feasible plan. a variant in which dead states can become alive again in an effort to obtain optimal plans.
- **Alive:** States that have been encountered and possibly some adjacent states that have not been visited. These are considered alive. Initially, the only alive state is x_I .

Forward Search Template

FORWARD_SEARCH

$Q.Insert(x_I)$ and mark x_I as visited
while Q not empty **do**
    $x \leftarrow Q.GetFirst()$
    if $x \in X_G$
        return SUCCESS
    forall $u \in U(x)$
        $x' \leftarrow f(x, u)$
        if x' not visited
            Mark x' as visited
            $Q.Insert(x')$
        else
            Resolve duplicate x'
return FAILURE

Need to construct a backward version of state transition function f

- Define state-action pairs

$$U^{-1}(x') = \{(x, u) \in U^{-1} \mid x' = f(x, u)\}.$$

- Define backward action space

$$U^{-1} = \{(x, u) \in X \times U \mid x \in X, u \in U(x)\}$$

- Define a backward state transition equation

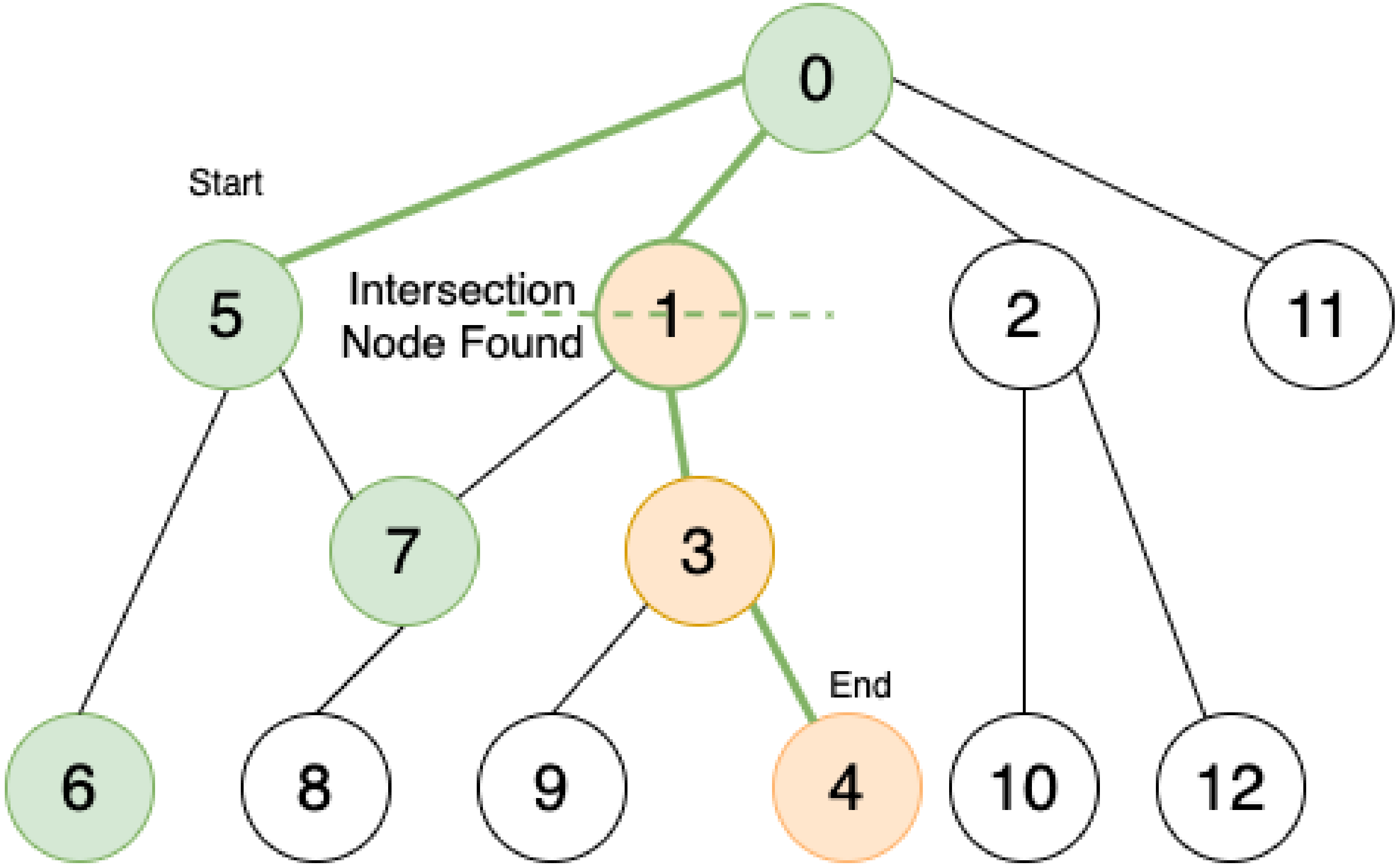
$$x = f^{-1}(x', u^{-1}),$$

Backward Search Template

BACKWARD_SEARCH

$Q.Insert(x_G)$ and mark x_G as visited
while Q not empty **do**
    $x' \leftarrow Q.GetFirst()$
    if $x = x_I$
        return SUCCESS
    forall $u^{-1} \in U^{-1}(x)$
        $x \leftarrow f^{-1}(x', u^{-1})$
        if x not visited
            Mark x as visited
            $Q.Insert(x)$
        else
            Resolve duplicate x
return FAILURE

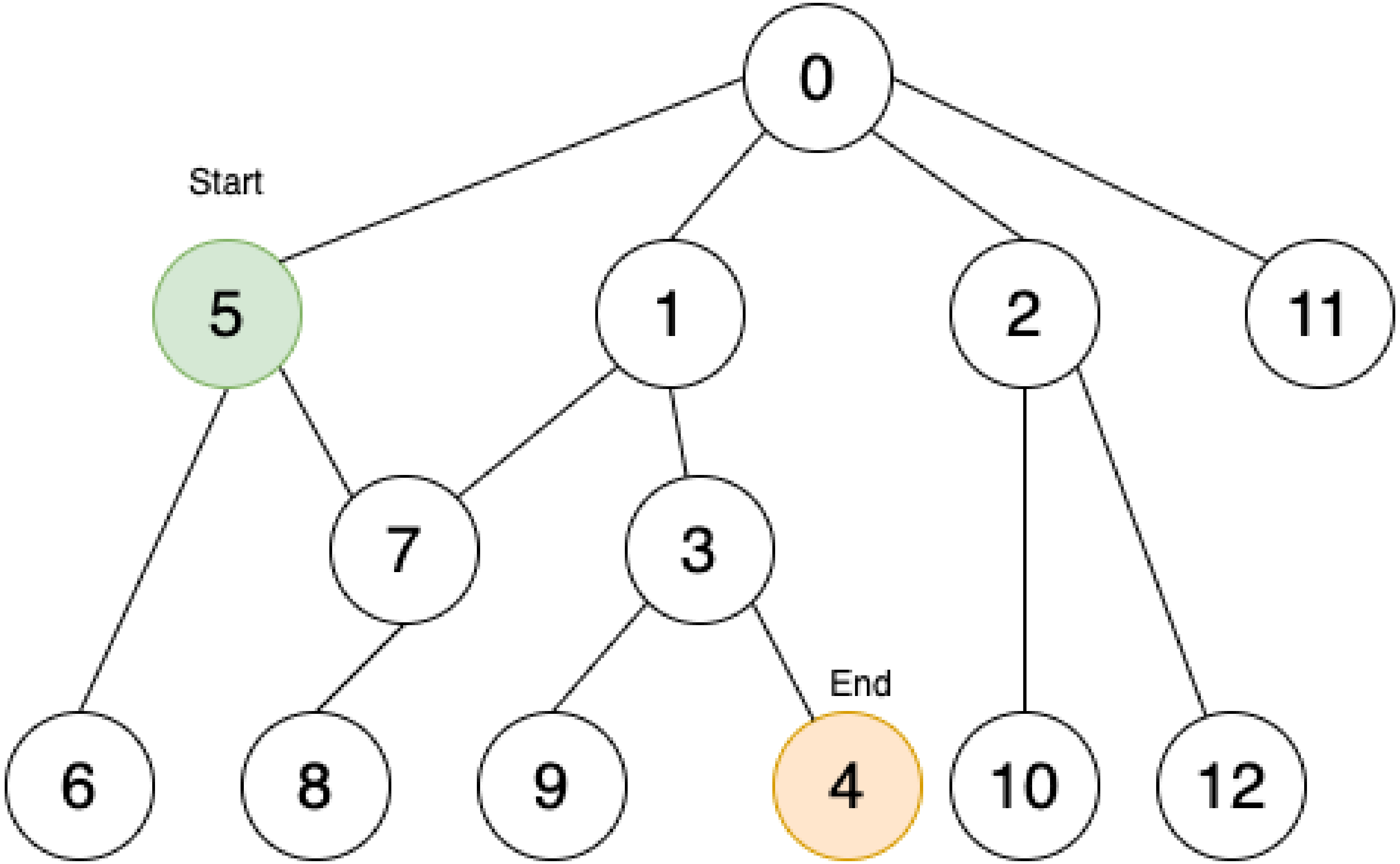
Bidirectional Search



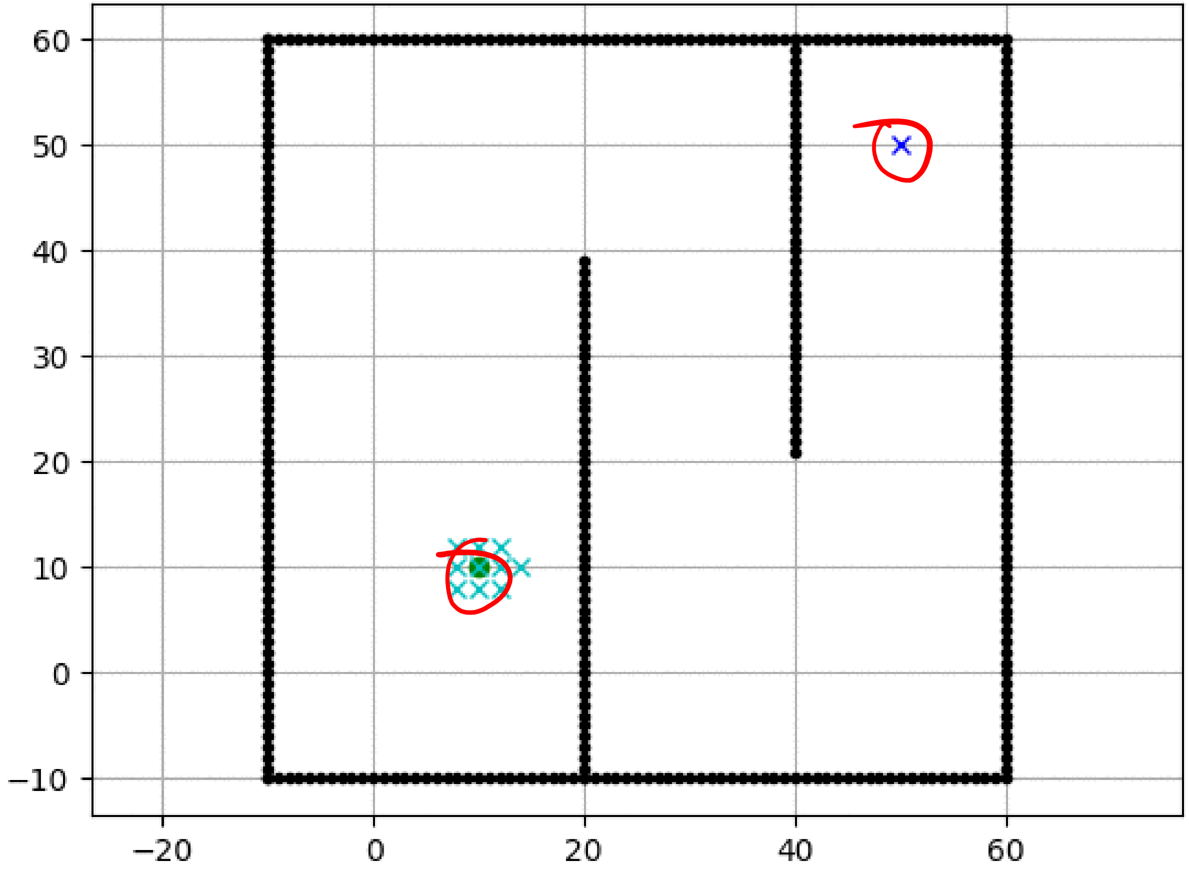
BIDIRECTIONAL_SEARCH

```
Q_I.Insert(x_I) and mark x_I as visited
Q_G.Insert(x_G) and mark x_G as visited
while Q_I not empty and Q_G not empty do
    if Q_I not empty
        x ← Q_I.GetFirst()
        if x = x_G or x ∈ Q_G
            return SUCCESS
        forall u ∈ U(x)
            x' ← f(x, u)
            if x' not visited
                Mark x' as visited
                Q_I.Insert(x')
            else
                Resolve duplicate x'
    if Q_G not empty
        x' ← Q_G.GetFirst()
        if x' = x_I or x' ∈ Q_I
            return SUCCESS
        forall u^{-1} ∈ U^{-1}(x')
            x ← f^{-1}(x', u^{-1})
            if x not visited
                Mark x as visited
                Q_G.Insert(x)
            else
                Resolve duplicate x
return FAILURE
```

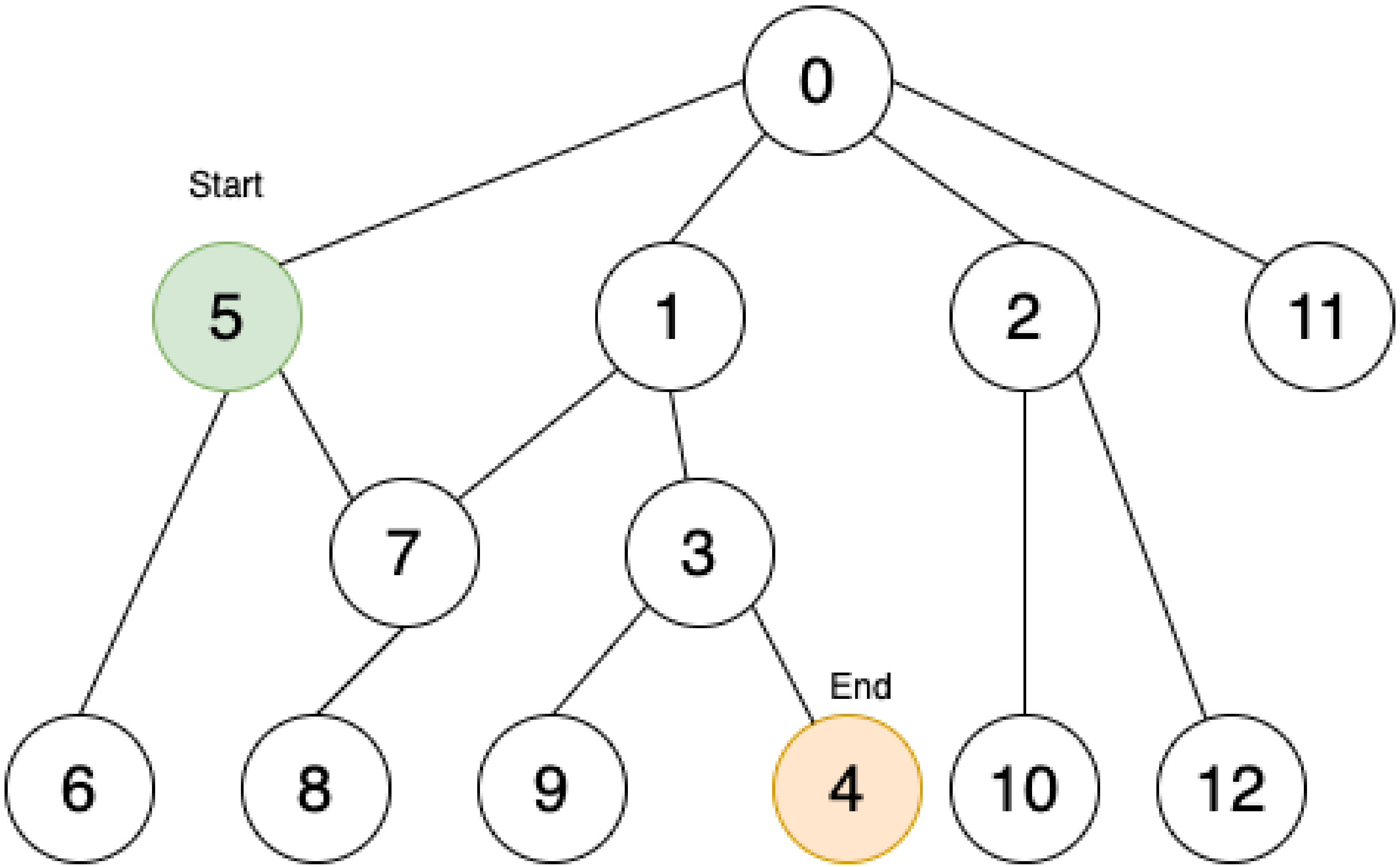
Breadth First Search



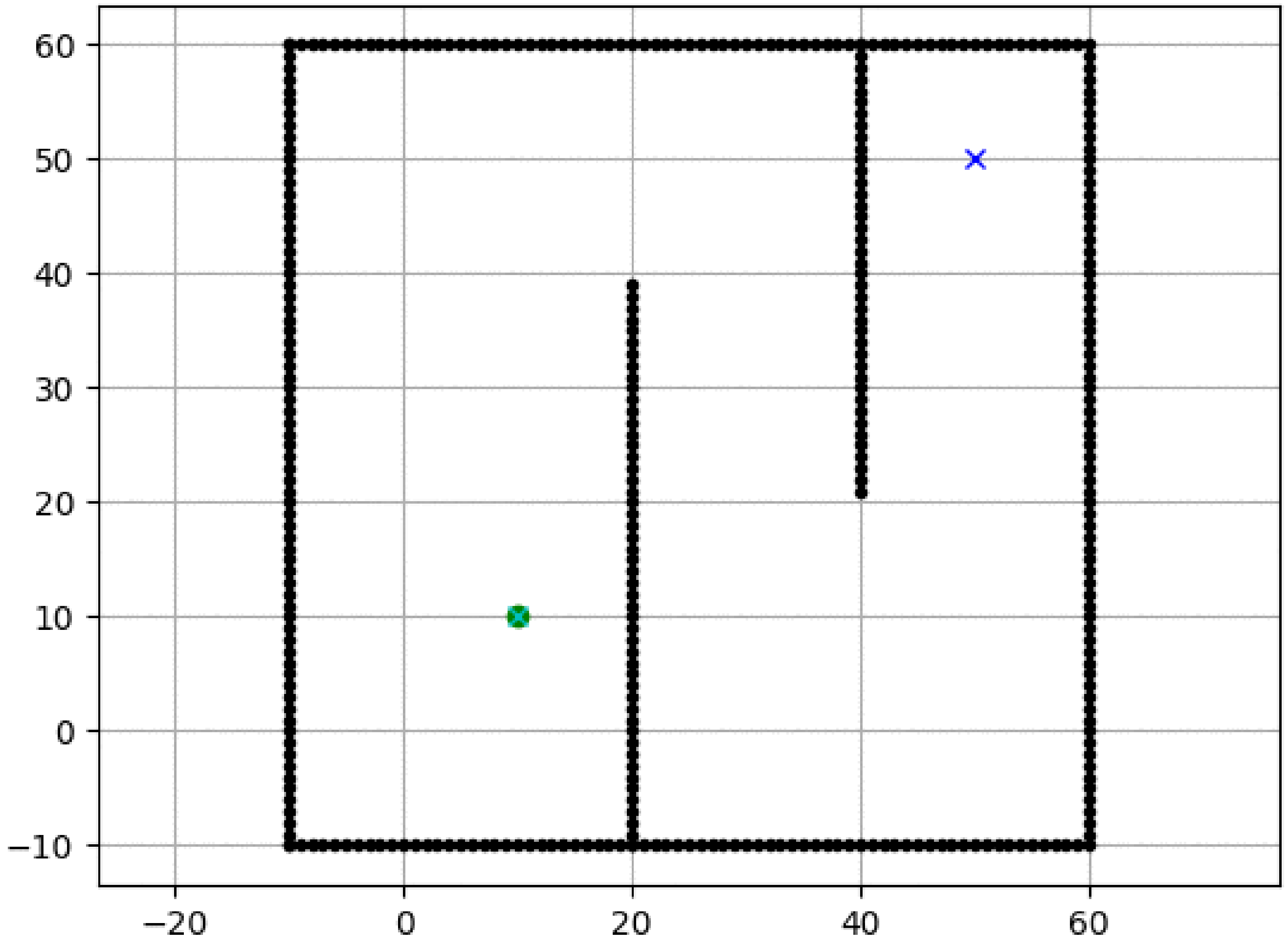
Breadth First Search



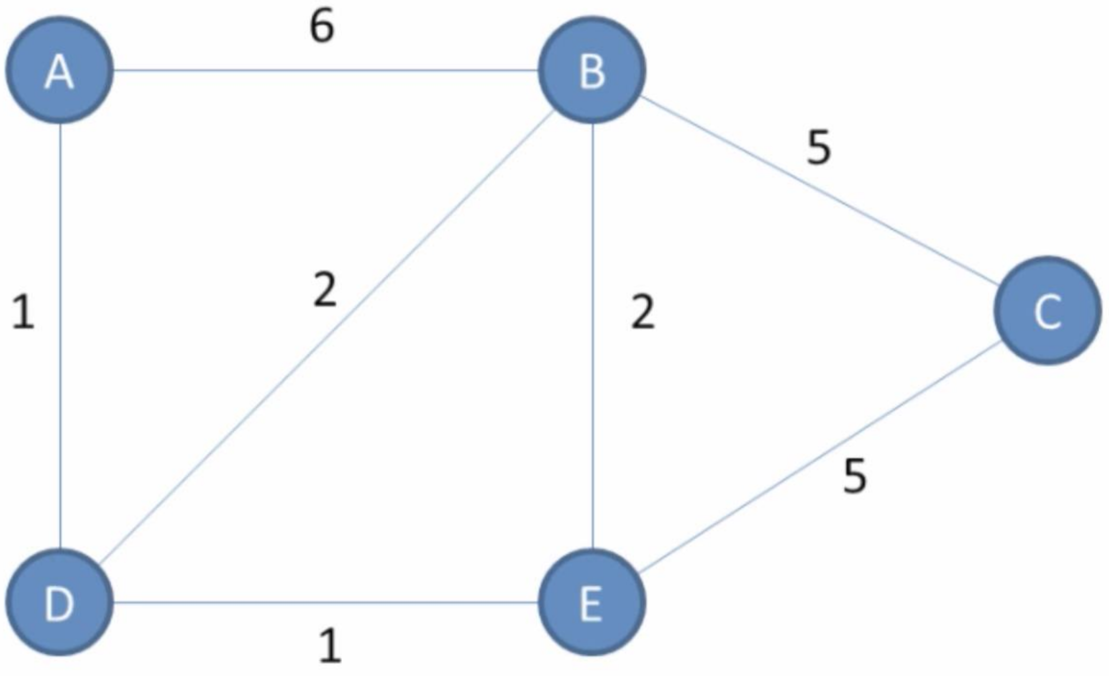
Depth First Search



Depth First Search



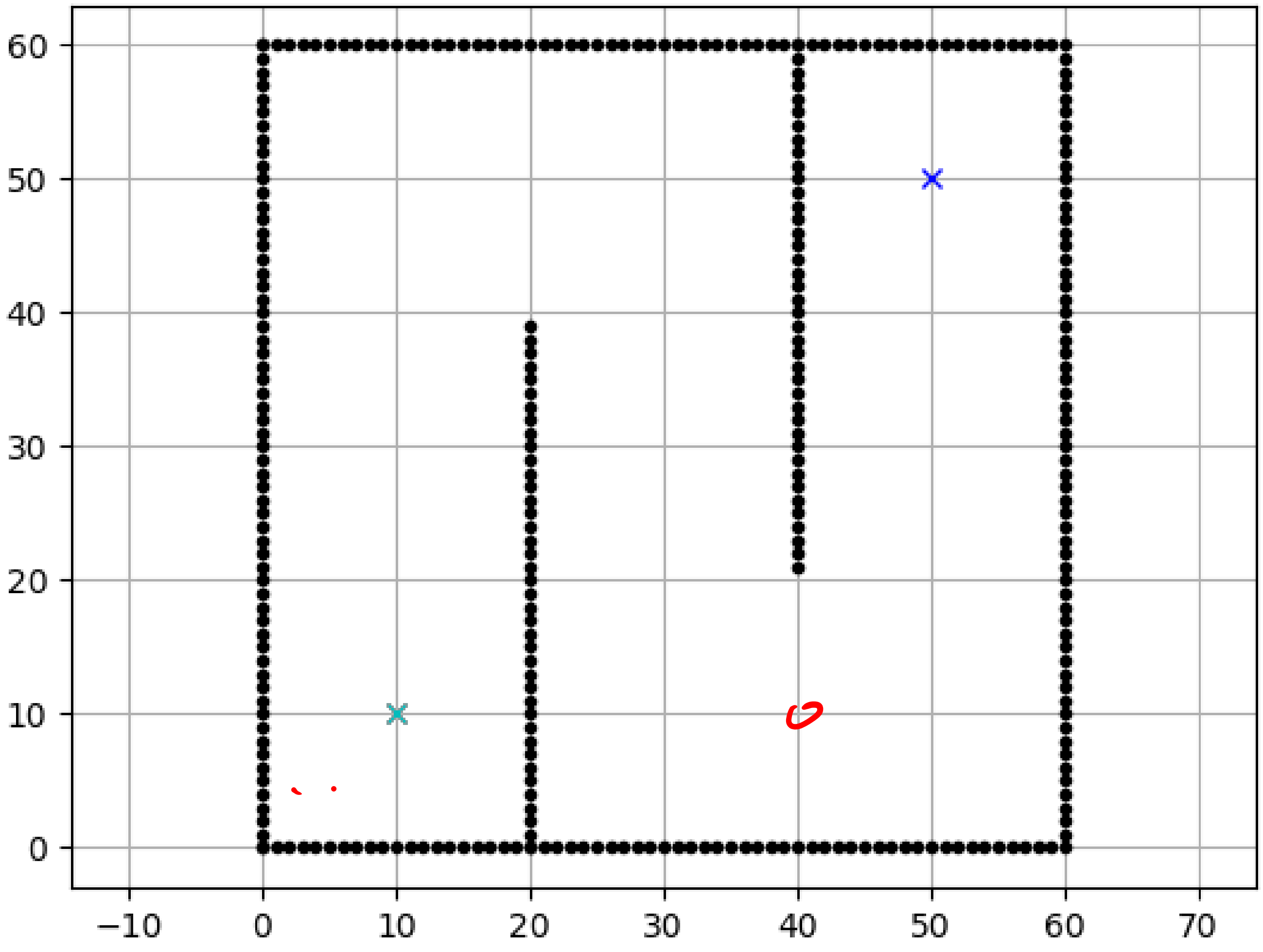
Dijkstra Algorithm



Vertex	Shortest distance from A	Previous vertex
A	0	
B	6	A
C	∞	
D	1	A
E	∞	

Visited = [A] Unvisited = [B, C, D, E]

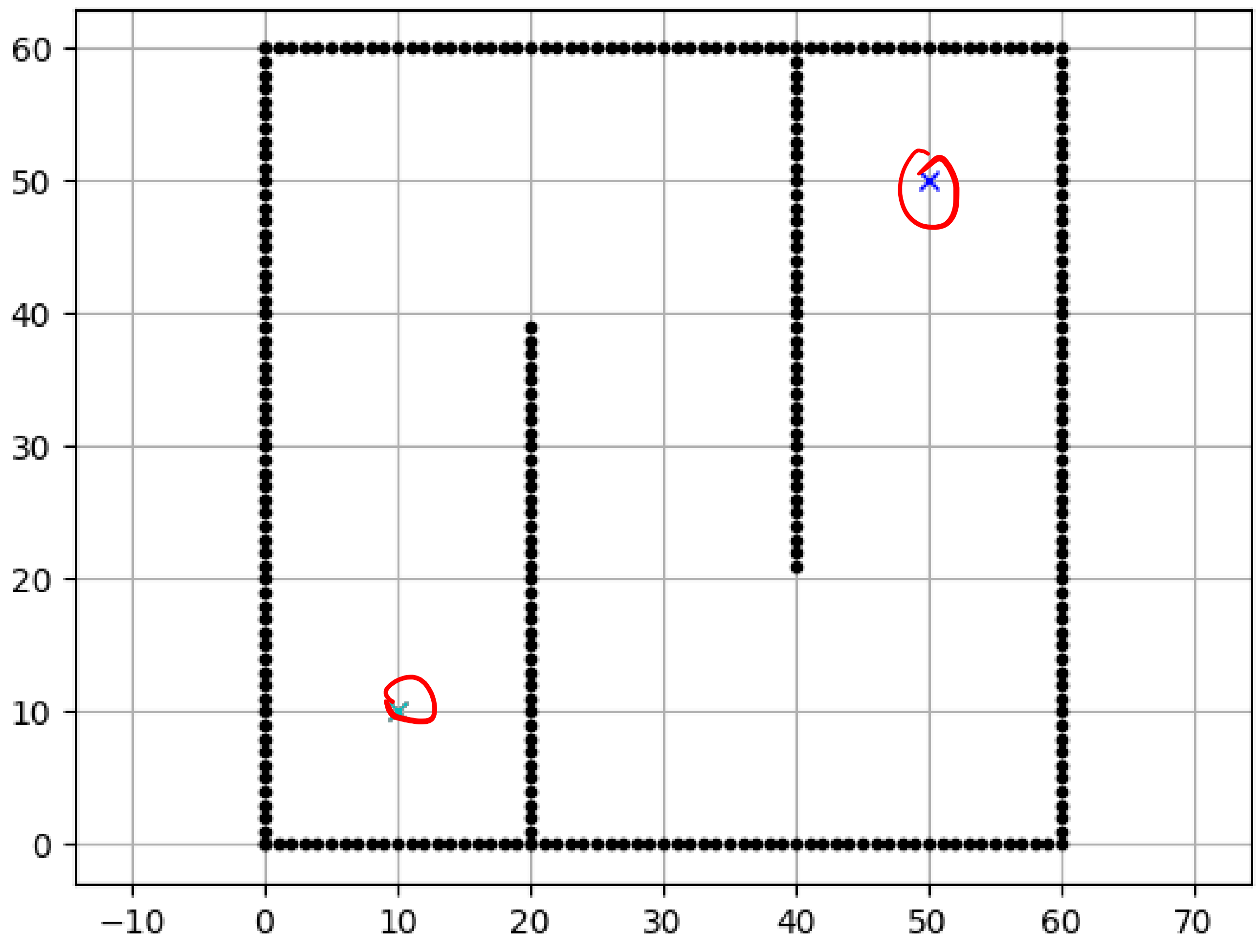
Dijkstra Algorithm



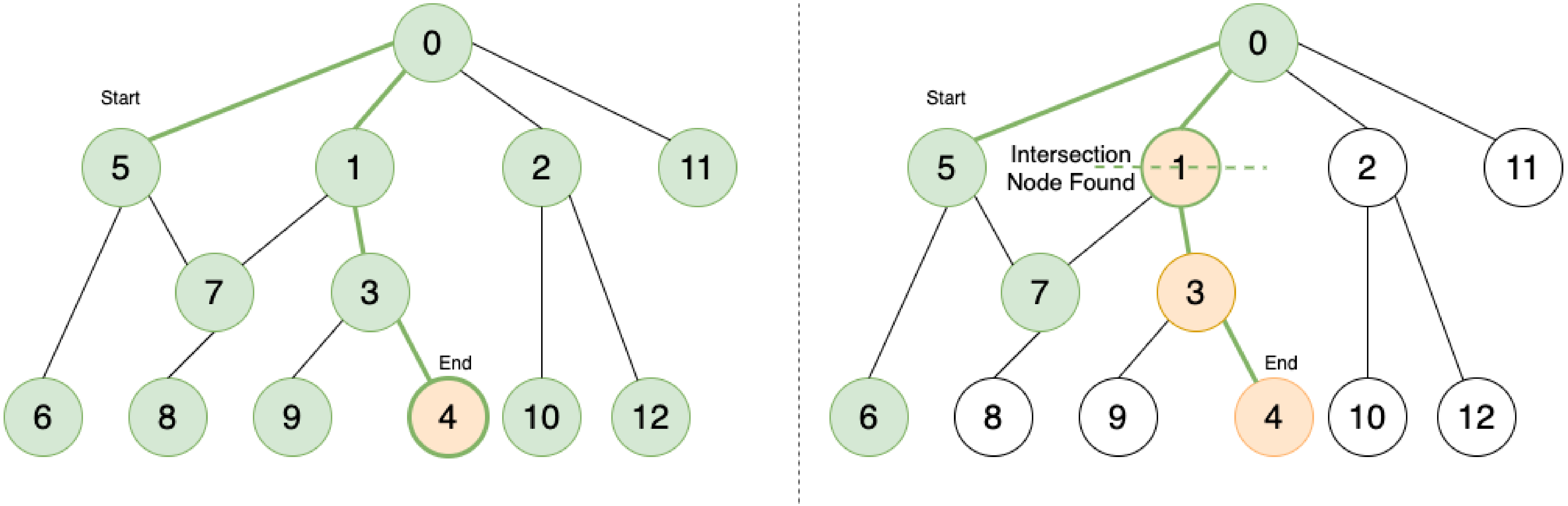
```
Procedure A* algorithm
Add start to open
While open is not empty
    Let n=first nod on open
    Drop n from open & add it to closed
    Add adjacent walkable squares to open
    Compute scores
    Select next square
    Add parent to closed
    If parent=destination
        Search is succeed
    Else if open is empty and destination is not find
        Search is fail
    End
End
```

Fig.3 The proposed A* algorithm pseudocode

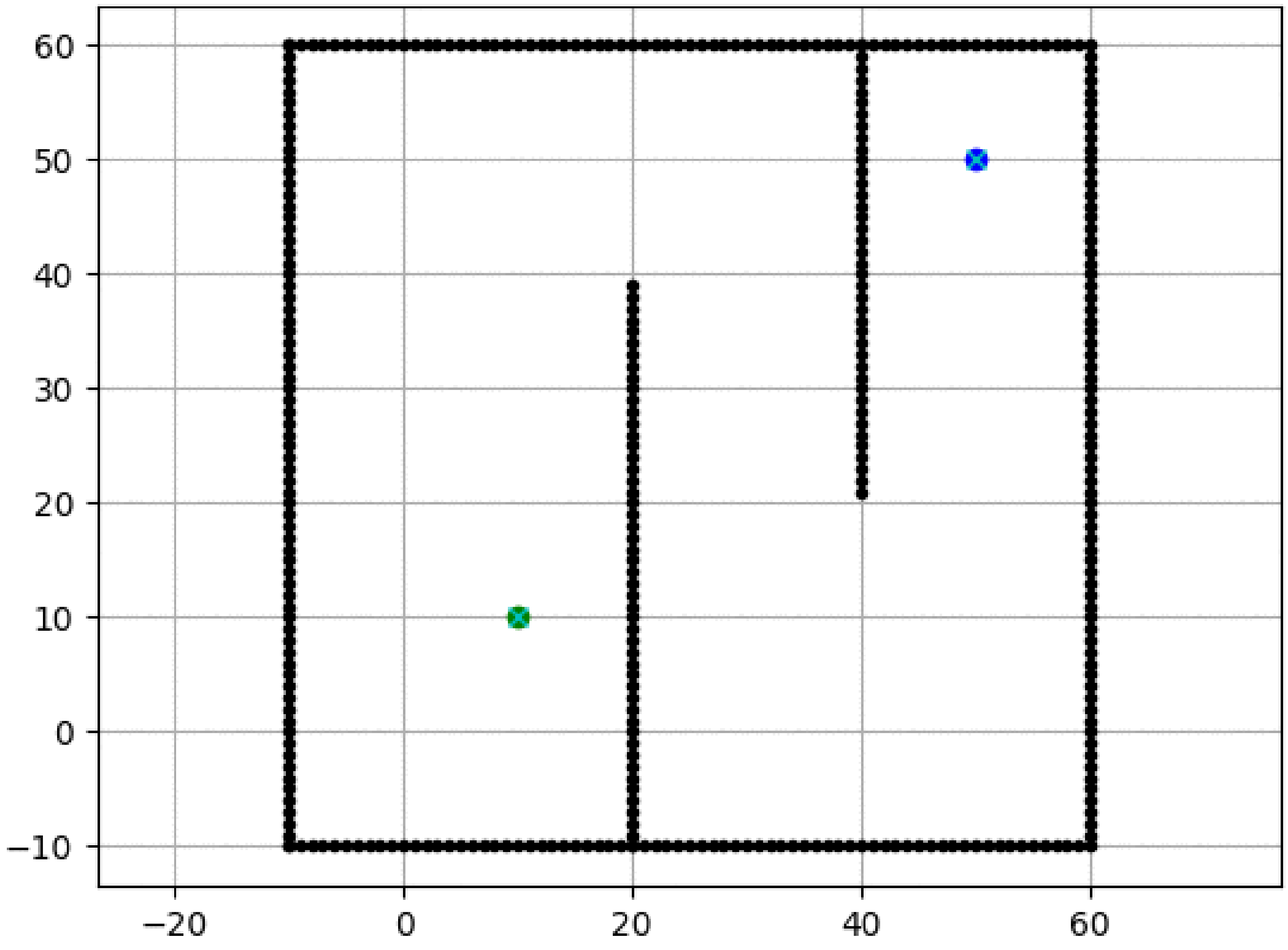
A* Algorithm



BFS vs Bidirectional Search



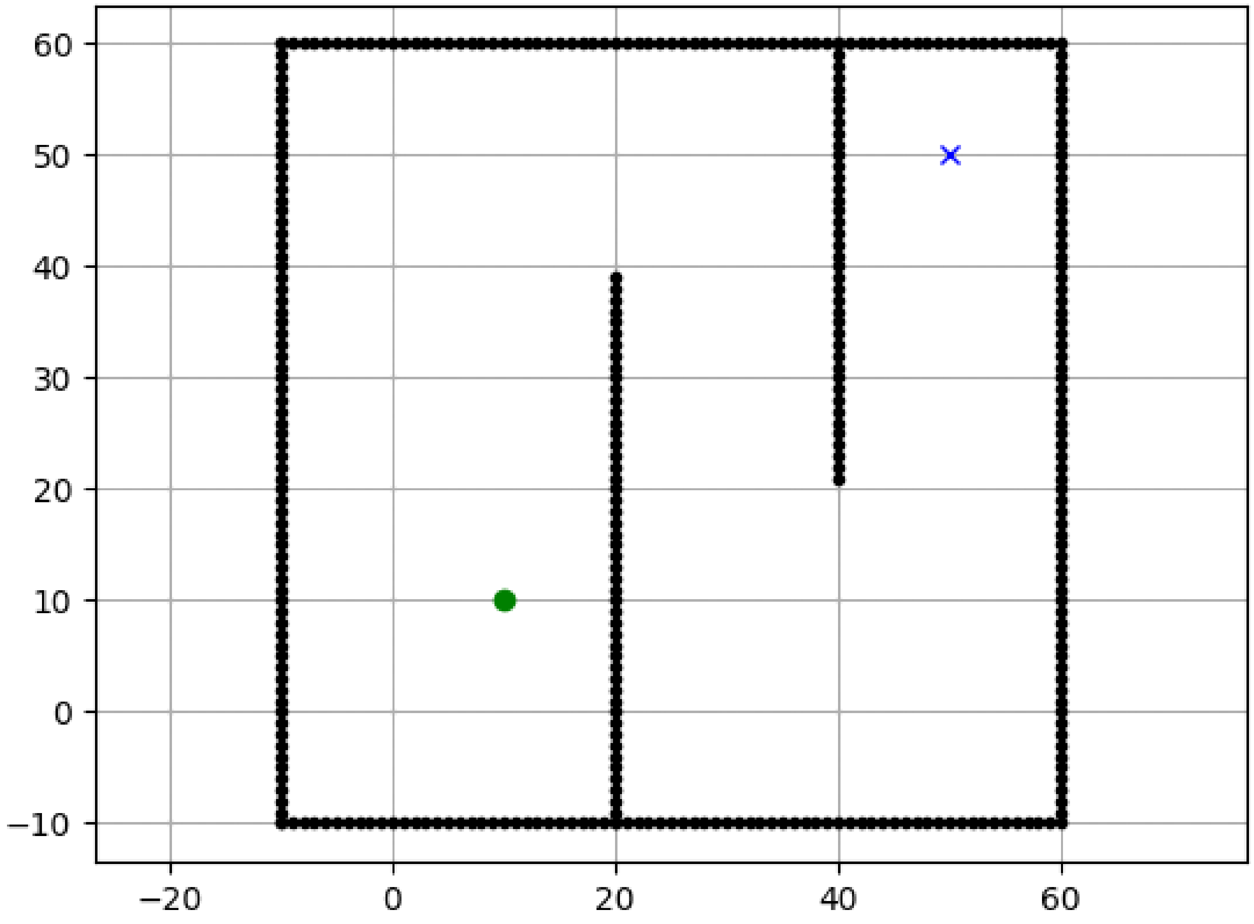
Bidirectional A* Algorithm



D* Algorithm

- Operations
  - NEW, meaning it has never been placed on the OPEN list
  - OPEN, meaning it is currently on the OPEN list
  - CLOSED, meaning it is no longer on the OPEN list
  - RAISE, indicating its cost is higher than the last time it was on the OPEN list
  - LOWER, indicating its cost is lower than the last time it was on the OPEN list
- Iteratively selecting a node from the OPEN list and evaluating it.
- D* begins by searching backwards from the goal node
- Each expanded node has a backpointer which refers to the next node leading to the target, and
- Each node knows the exact cost to the target

D* Algorithm



Algorithm 3: D* Lite

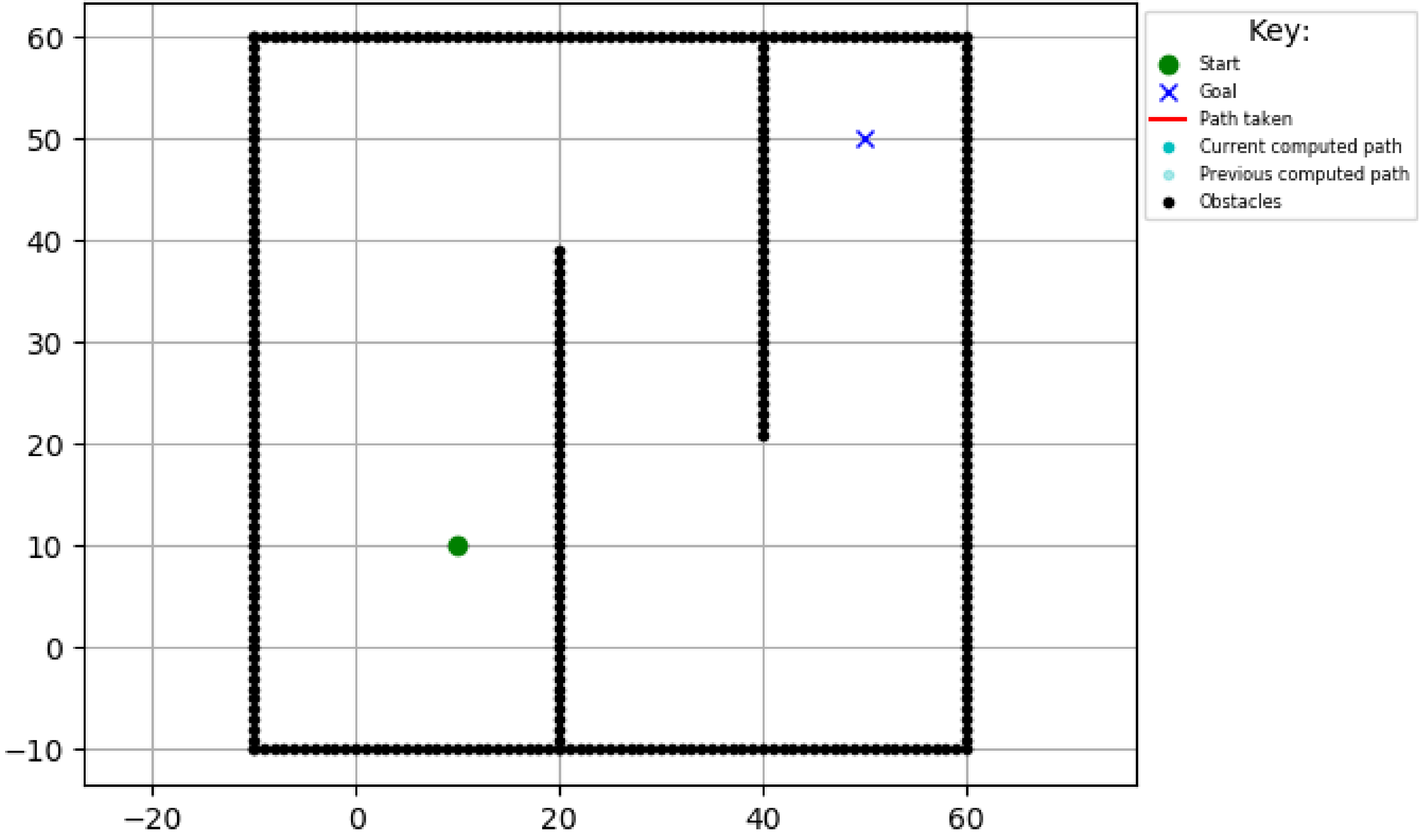
Function *Key(s)*:
  return $[min(g(s), rhs(s)) + h(s_{start}, s) + k_m; min(g(s), rhs(s))]$

Function *UpdateVertex(s)*:
  if $s \neq s_{goal}$ **then**
    $rhs(s) = min_{s' \in Succ(s)}(cost(s, s') + g(s'))$
  end
  if $s \in OPEN$ **then**
    $OPEN.remove(s)$
  end
  if $g(s) \neq rhs(s)$ **then**
    $OPEN.insert(s, Key(s))$
  end

Function *ComputePath()*:
  while $OPEN.TopKey() < Key(s_{start})$ OR $rhs(s_{start}) \neq g(s_{start})$ **do**
    $k_{old} = OPEN.TopKey()$
    $s = OPEN.Pop()$
    if $k_{old} < Key(s)$ **then**
      $OPEN.insert(s, Key(s))$
    else if $g(s) > rhs(s)$ **then**
      $g(s) = rhs(s)$
      forall $s' \in Pred(s)$ **do**
        UpdateVertex(s')
      end
    else
    $g(s) = \infty$
    forall $s' \in Pred(s) \cup \{s\}$ **do**
      UpdateVertex(s')
    end
  end

Function *Main()*:
  forall $s \in S$ **do**
    $rhs(s) = g(s) = \infty$
  end
  $s_{last} = s_{start}$
  $OPEN = \emptyset$
  $rhs(s_{goal}) = 0$; $k_m = 0$
  $OPEN.insert(s_{goal}, Key(s_{goal}))$
  ComputePath()
  while $s_{start} \neq s_{goal}$ **do**
    $s_{start} = argmin_{s' \in Succ(s_{start})}(cost(s_{start}, s') + g(s'))$
    Move to s_{start}
    Scan for cell changes in environment (e.g. sensor ranges)
    if *Cell changes detected* **then**
      $k_m = k_m + h(s_{last}, s_{start})$
      $s_{last} = s_{start}$
      forall $s \in CHANGES$ **do**
        Update cell s state
        forall $s' \in Pred(s) \cup \{s\}$ **do**
          UpdateVertex(s')
        end
      end
      ComputePath()
    end
  end

D* Lite Algorithm



Conclusion

- It is often convenient to express planning problem as a directed state transition graph.
- It is important not to explicitly represent the entire state transition graph.
- Instead, it is revealed incrementally in the planning process
- Heuristic algorithms provide quick solution but sacrifice performance guarantee.
- Optimal algorithms provide performance guarantee but sacrifice scalability.