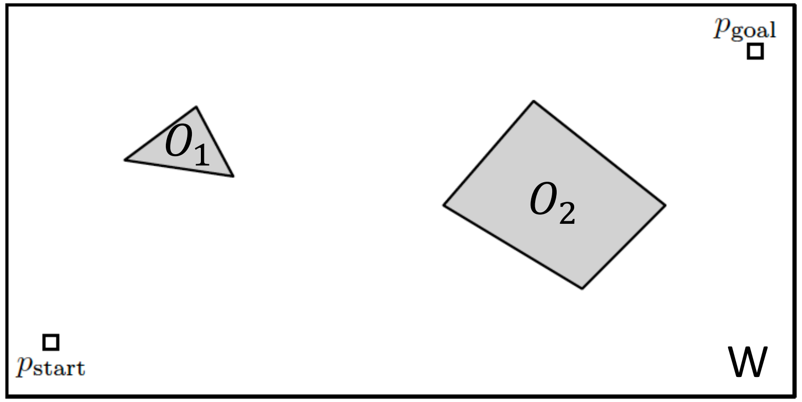
Introduction to Robot Motion Planning & Navigation
Module 2

Solmaz S. Kia (solmaz.eng.uci.edu)
solmaz@uci.edu
Mechanical and Aerospace Engineering Department
University of California Irvine

Chapter 2: Motion Planning via Decomposition and Search

- study techniques for decomposing the continuous robot workspace into convex regions,
- define roadmaps, which encode the decomposed workspace, and
- introduce graph algorithms for computing point-to-point paths in roadmaps.

Problem Setup and Modeling Assumptions



- a robot described by a moving point
- A workspace $W \subset R^2$;
- Some obstacles $O_1, O_2, \cdots, O_n$;
- $W_{free} = W \backslash (O_1 \cup O_2 \cup \cdots \cup O_n)$
- A start point p_{start} and a goal point p_{goal};

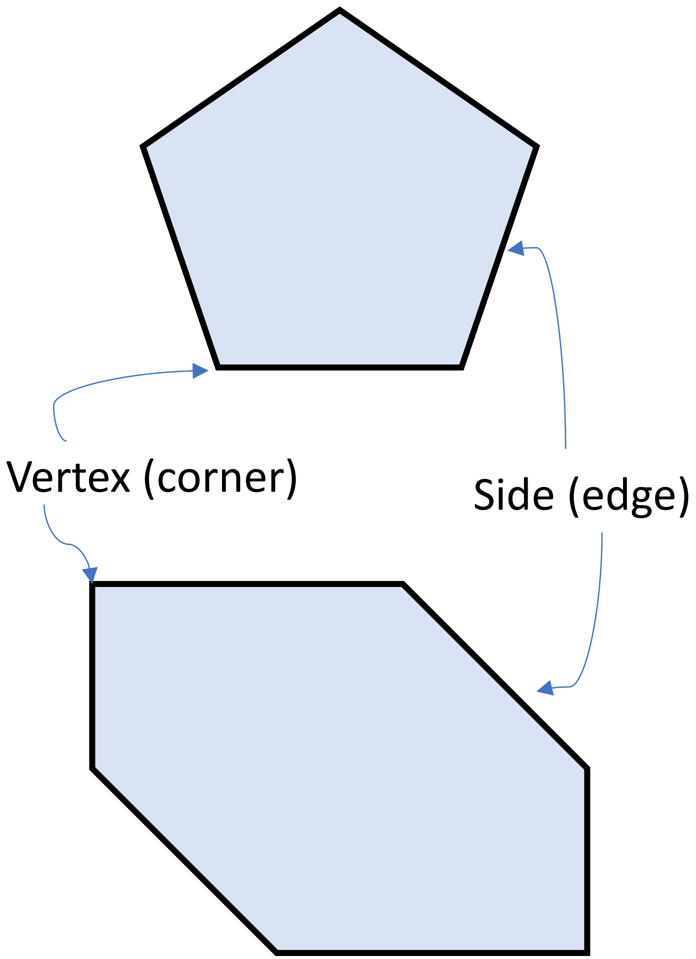
Assumptions on the capabilities and knowledge of the robot

- knows the start and goal locations, and
- knows the workspace and obstacles.
- the robot's motion is omni-directional (i.e., the robot can move in every possible direction)

Environment Assumptions

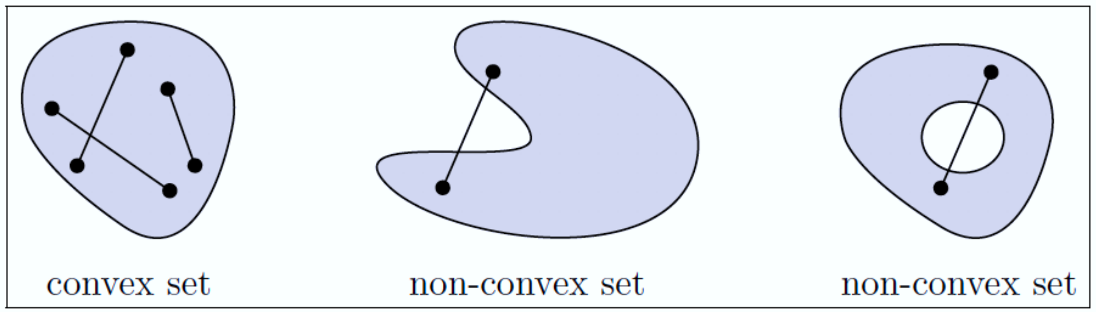
- the workspace is a bounded polygon,
- there are only a finite number of obstacles that are polygons inside the workspace, and
- the start and goal points are inside the workspace and outside all obstacles.

Polygons



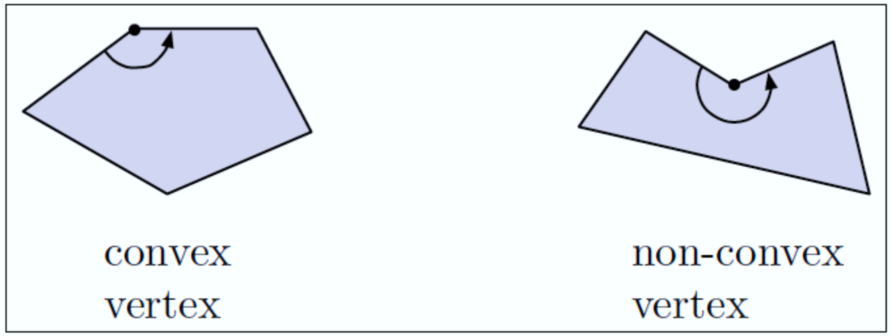
Convex set

A set S is convex if for any two points p and q in S, the entire segment $\overline{pq}$ is also contained in S. Examples of convex and non-convex sets:



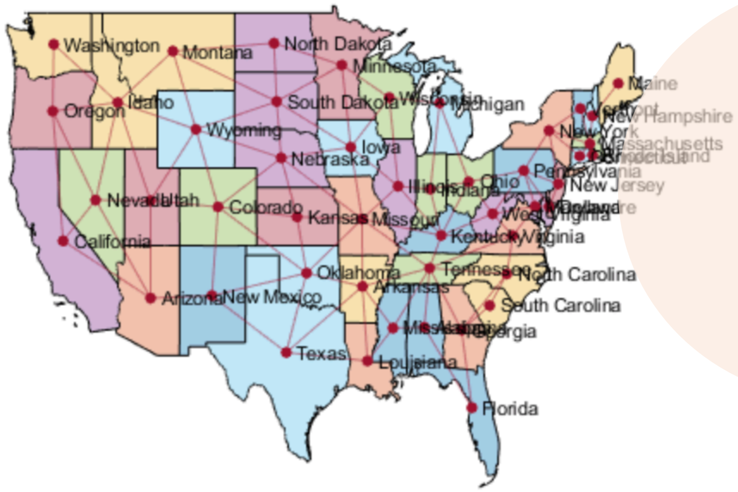
For polygons, convexity is related to the interior angles at each vertex of the polygon (each vertex of a polygon has an interior and an exterior angle): a polygonal set is convex if and only if each
vertex is convex, i.e., it has an interior angle less than π. A vertex is instead called non-convex if its interior angle is larger than π.

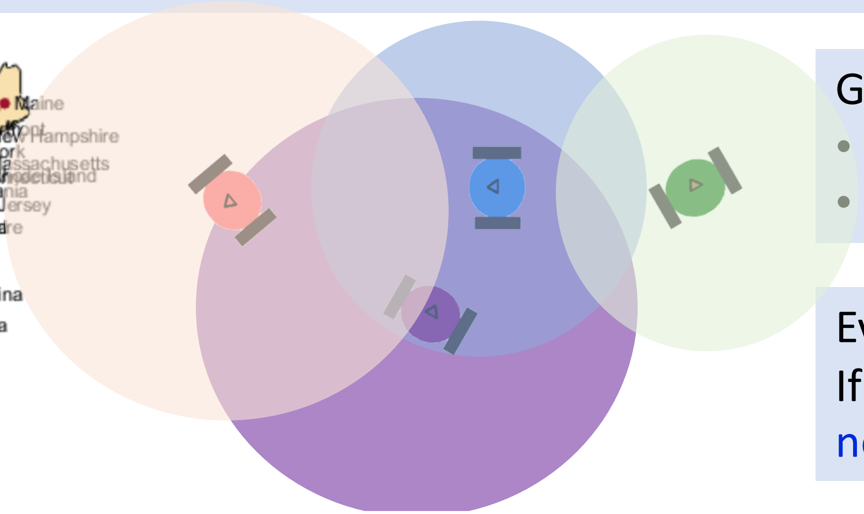
.



Graphs

In mathematics, especially **graph theory, graphs** are mathematical structures used to model pairwise relations between objects.



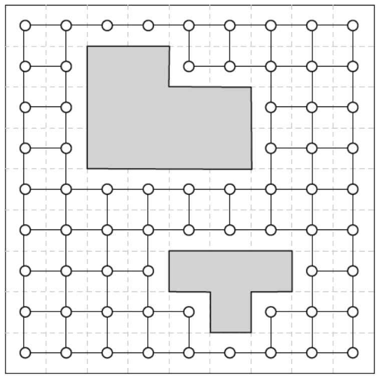


Graph: $G(V;E)$ where

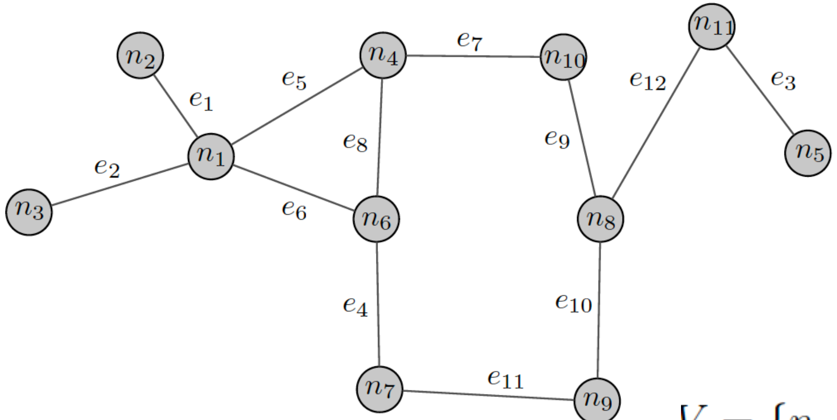
- V is a set of nodes (also called vertices)
- E is a set of edges (also called links or arcs).

Every edge is a pair of nodes.
If $\{u,v\}$ is an edge, then u and v are said to be neighbors.

Graphs as defined here are sometimes referred to as unweighted and undirected graphs.



(d) A prototypical robotic roadmap generated by a grid



(e) A sample graph from equation (2.1)

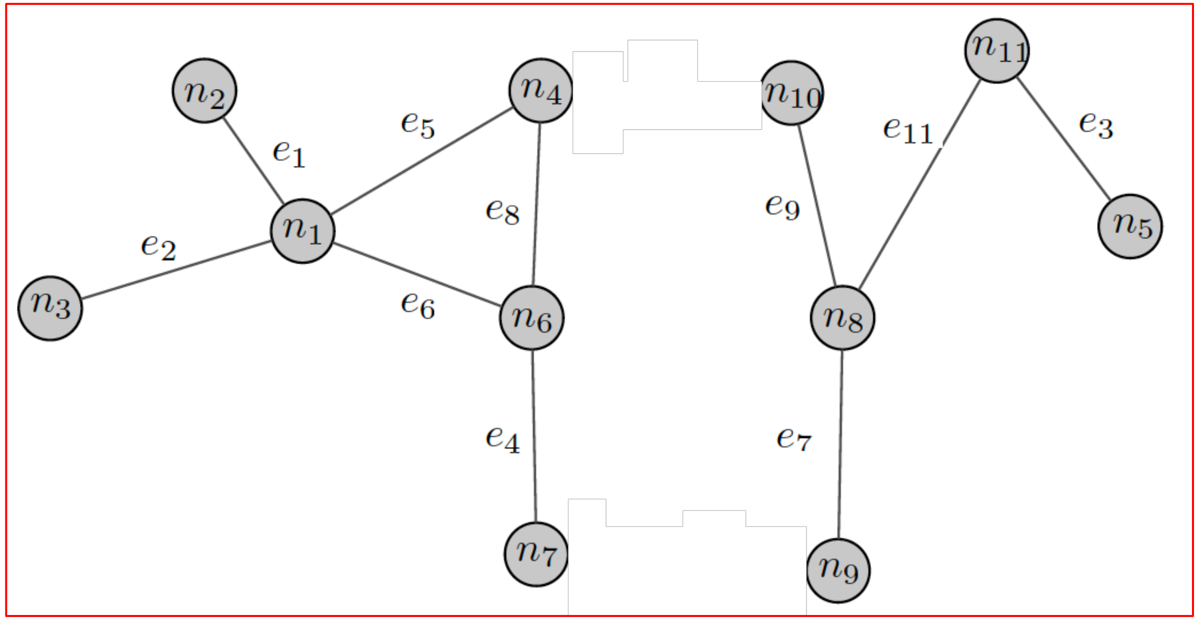
$$V = \{n_1, \ldots, n_{11}\}$$

$$E = \{e_1 = \{n_1, n_2\}, e_2 = \{n_1, n_3\}, e_3 = \{n_{11}, n_5\}, e_4 = \{n_6, n_7\},$$
$$e_5 = \{n_1, n_4\}, e_6 = \{n_1, n_6\}, e_7 = \{n_4, n_{10}\}, e_8 = \{n_4, n_6\},$$
$$e_9 = \{n_8, n_{10}\}, e_{10} = \{n_8, n_9\}, e_{11} = \{n_7, n_9\}, e_{12} = \{n_8, n_{11}\}\}.$$

Graphs

- A path is an ordered sequence of nodes such that from each node there is an edge to the next node in the sequence.
- The length of a path is the number of edges in the path from start node to end node.
- Two nodes in a graph are path-connected if there is a path between them.
- A graph is connected if every two nodes are path-connected.
- If a graph is not connected, it is said to have multiple connected components. More precisely, a connected component is a subgraph in which (1) any two nodes are connected to each other and (2) all nodes outside the subgraph are not connected to the subgraph.

- A shortest path between two nodes is a path of minimum length between the two nodes. Note that a shortest path does not need to be unique.
- The distance between two nodes is the length of a shortest path connecting them, i.e., the minimum number of edges required to go from one node to the other.
- A cycle is a path with at last three distinct nodes and with no repeating nodes, except for the first and last node which are the same. A graph that contains no cycles and is connected is called a tree.



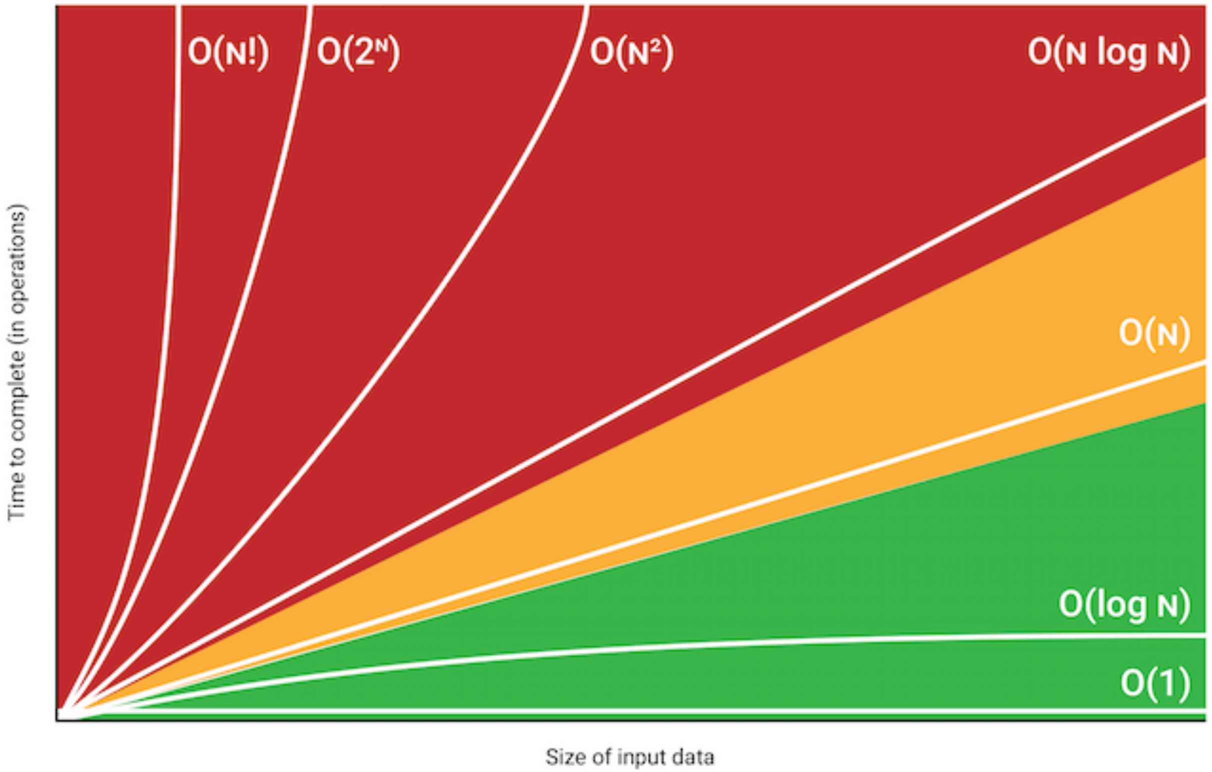
Big O (Time Complexity of Algorithms)

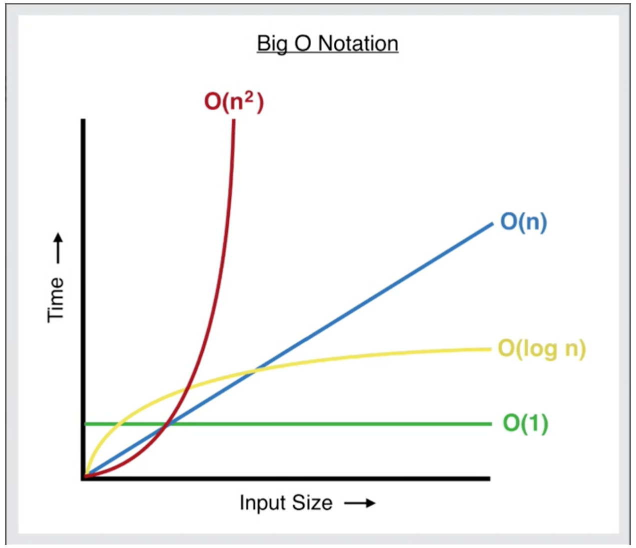
Big O notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity.

- Finding a specific number in an array of n numbers

- Finding the maximum in an array of n numbers

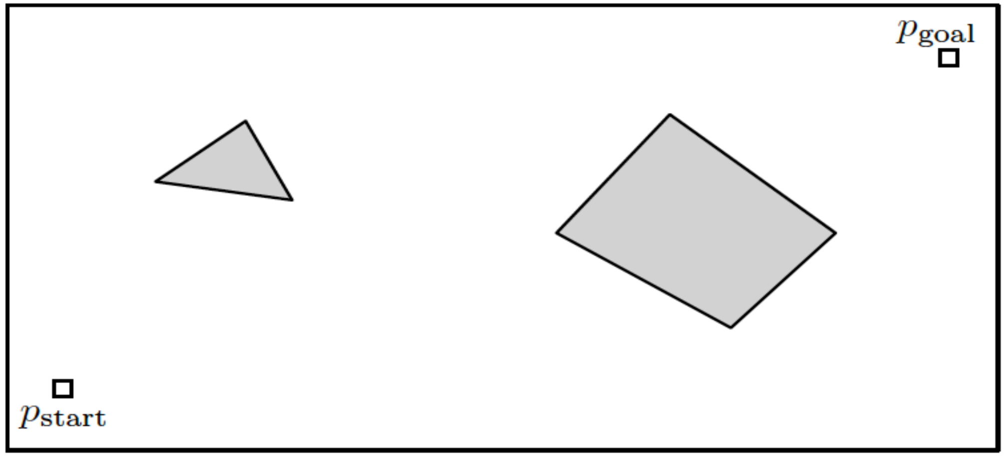
- Traveling salesman problem

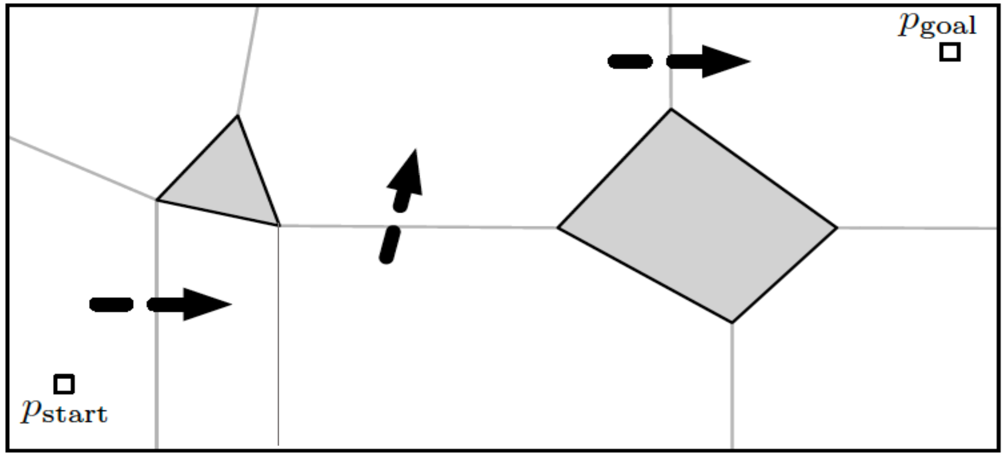


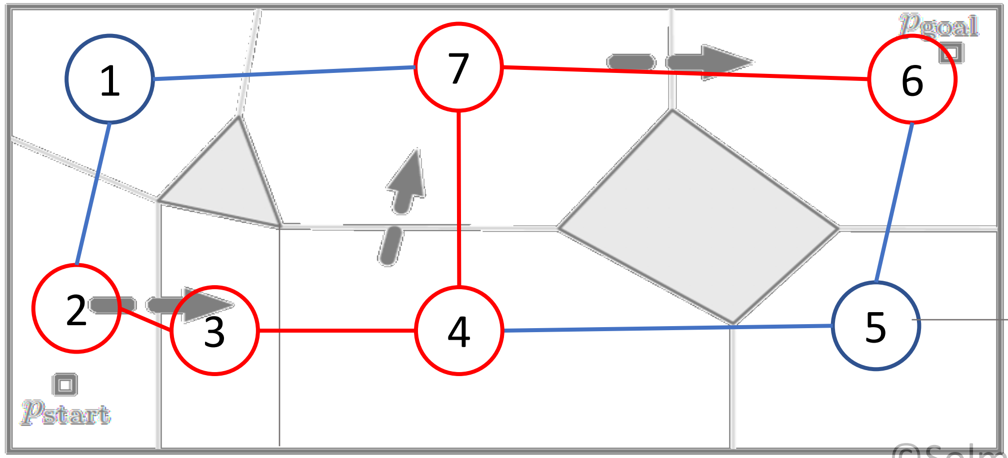


Planning in non-convex sets via convex decompositions

- If p_{start} and p_{goal} are in a convex free workspace, then the line connecting them also will be in free workspace; if not
- Decompose the free workspace to convex cells and traverse through these convex sub spaces of the free workspace according to the motion planning algorithm.







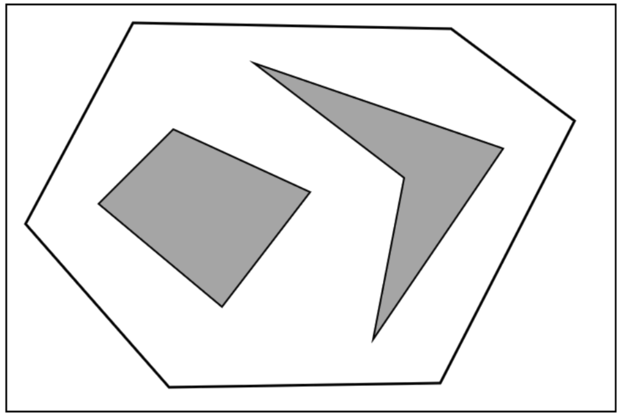
- the triangulation of a polygon is the decomposition of the polygon into a collection of triangles, and
- the trapezoidation of a polygon is the decomposition of the polygon into a collection of trapezoids. (We allow some trapezoids to have a side of zero length and therefore be triangles.)

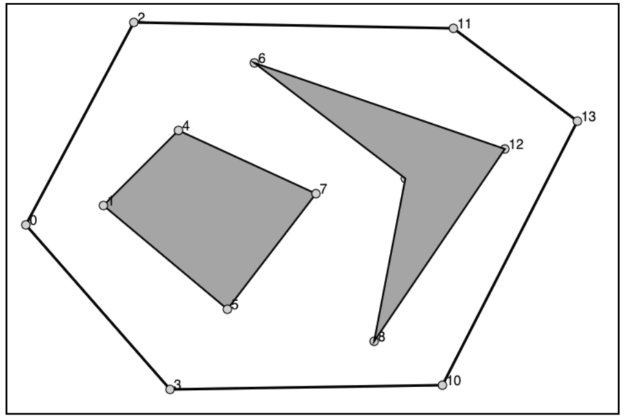
Sweeping Trapezoidation Algorithm

sweeping trapezoidation algorithm

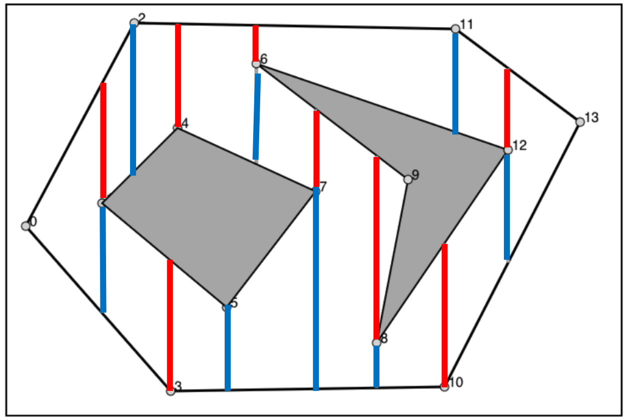
Input: a polygon possibly with polygonal holes
Output: a set of disjoint trapezoids, whose union equals the polygon

initialize an empty list $\mathcal{T}$ of trapezoids
order all vertices (of the obstacles and of the workspace) horizontally from left to right
for each vertex selected in a left-to-right sweeping order :
    extend vertical segments upwards and downwards from the vertex until they intersect an obstacle or the workspace boundary
    add to $\mathcal{T}$ the new trapezoids, if any, generated by these segment(s)

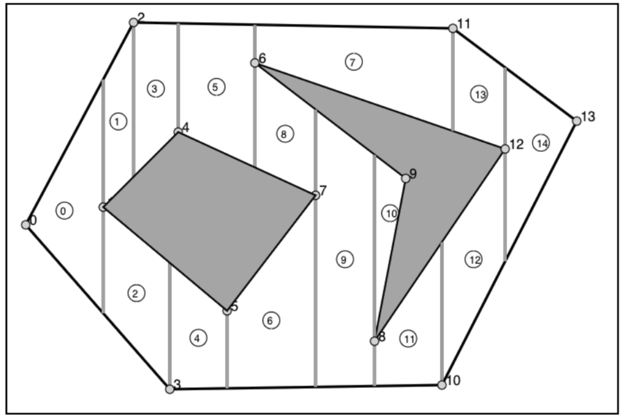




Vertex points of polygons are marked.



Upper and lower edge extensions of a polygon vertex is depicted.



Determined Trapezoidal Cells.

Each trapezoidal cell lies between two successive polygon vertex.

Sweeping Trapezoidation Algorithm: Naïve Implementation

sweeping trapezoidation algorithm

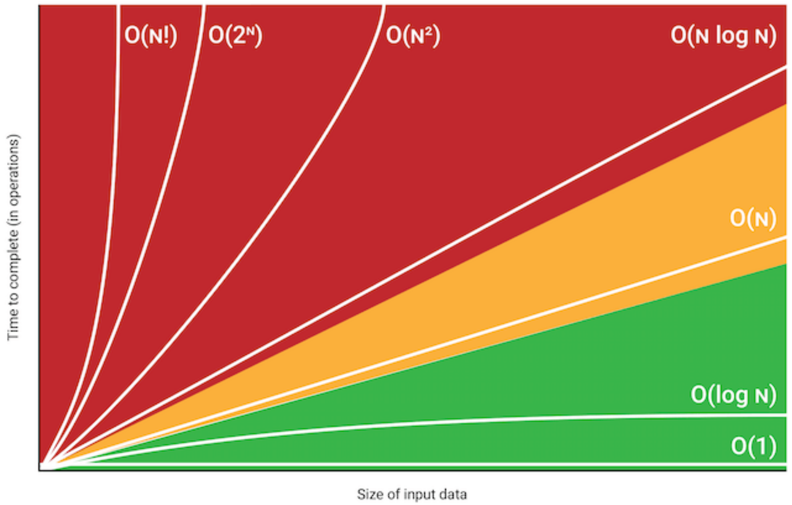
Input: a polygon possibly with polygonal holes

Output: a set of disjoint trapezoids, whose union equals the polygon

```
initialize an empty list 𝒯 of trapezoids
order all vertices (of the obstacles and of the workspace) horizontally from left to right
for each vertex selected in a left-to-right sweeping order :
    extend vertical segments upwards and downwards from the vertex until they intersect an obstacle or the workspace boundary
    add to 𝒯 the new trapezoids, if any, generated by these segment(s)
```

n: number of vertices (or edges)

- Input to the algorithm is a list of polygons, each represented by a list of vertices.
- First step: to sort the vertices based on the x-coordinate of each vertex
  - Many sorting algorithms like bubble sort, merge sort, quick sort exists. Best of them takes $O(n \log n)$ time and $O(n)$ storage
- Next step: to determine the vertical extensions.
  - For each vertex v_i, a naïve algorithm can intersect a line through v_i with each edge e_j for all j. This takes $O(n^2)$ time to construct the trapezoidal decomposition.

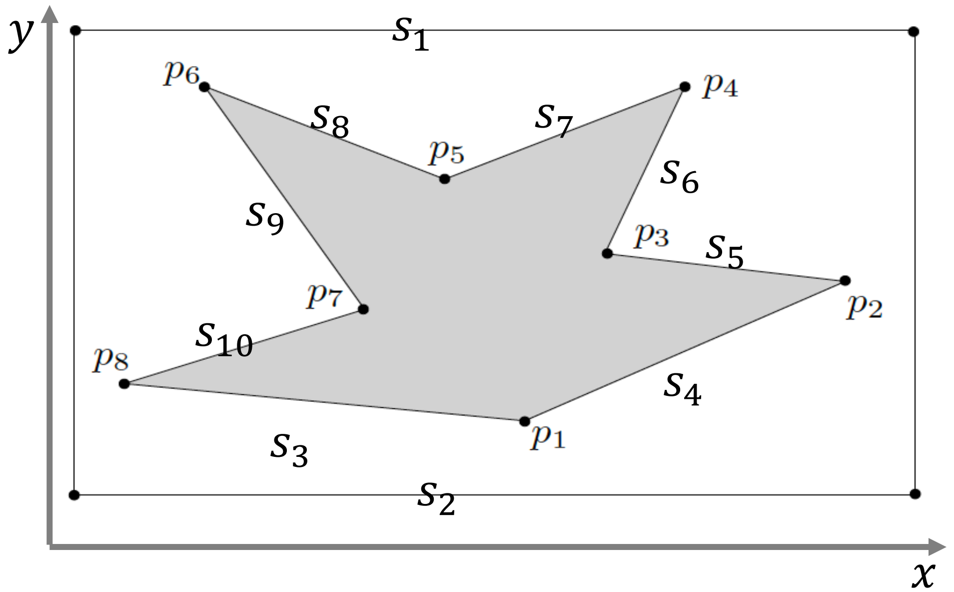


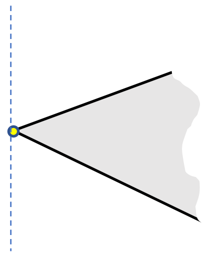
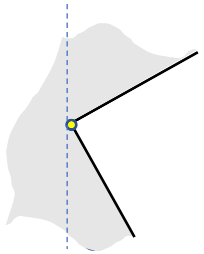
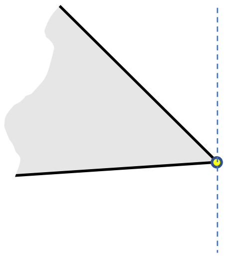
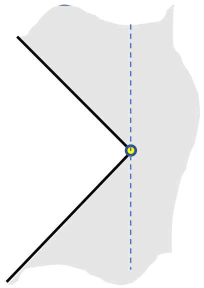
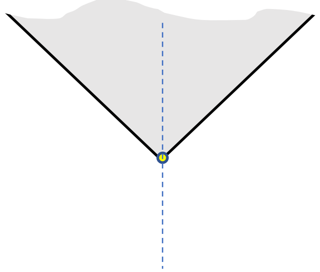
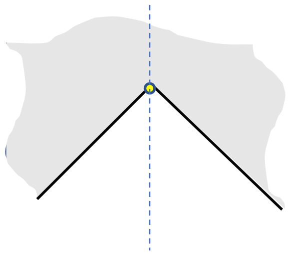
Can we do better?

Sweeping Trapezoidation Algorithm: 'Smarter' Implementation

Consider a workspace in which

- the boundary is an axis-aligned rectangle
- every obstacle vertex has a unique x-coordinate
- Represent each segment with the x coordinate of its left and right end points: $s_i = [\ell_i, r_i]$
- define a sweeping vertical line L moving from left to right
- Determine the vertex type



Type (i): left/left convex	Type (ii): left/left non-convex	Type (iii): right/right convex	Type (iv): right/right non-convex	Type (v): left/right convex	Type (vi): left/right non-convex
					
Example:	Example:	Example:	Example:	Example:	Example:

Sweeping Trapezoidation Algorithm: 'Smarter' Implementation

- Maintain a list $\mathcal{S}$ of the obstacle segments intersected by the sweeping line L.
  - The obstacle segments are stored in decreasing order of their y-coordinates at the intersection point with L.
  - $\mathcal{S}$ changes only when L hits a new vertex.

- when the new vertex v is encountered, steps 4: and 5: update the list of trapezoids $\mathcal{T}$ and the list of obstacle segments $\mathcal{S}$, as follows

  4.1 determine the type of vertex v

  4.2: update $\mathcal{S}$ by adding obstacle segments starting at v and removing obstacle segments ending at v (i.e., add two

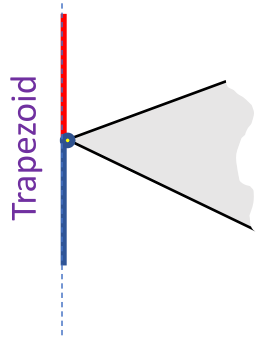
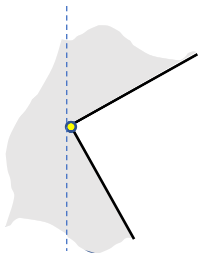
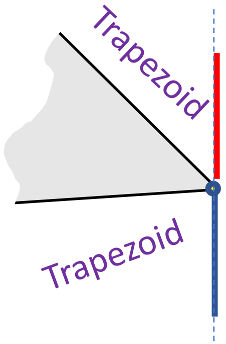
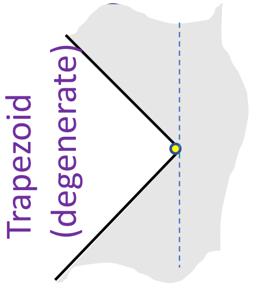
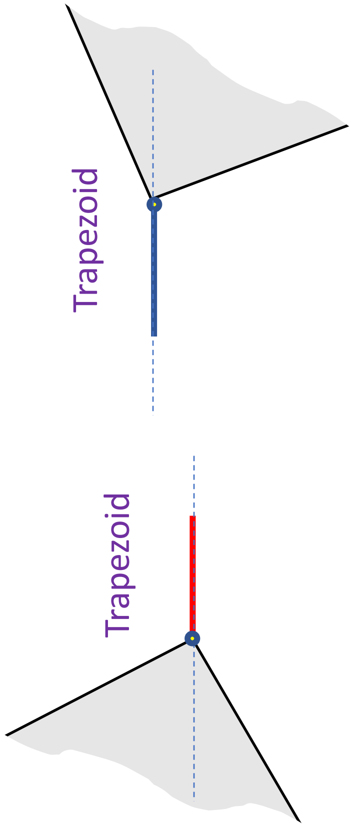
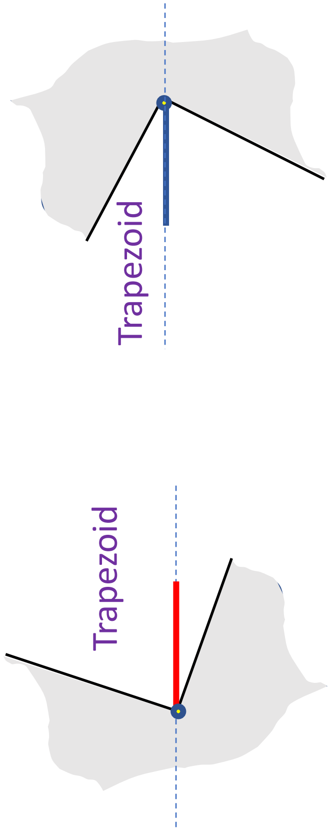
  segments, remove one segment and add one segment, or remove two segments, depending on vertex type as shown in the next page)

  4.3: use $\mathcal{S}$ to extend vertical segments upwards and downwards from v, that is, to find intersection points p_t and p_b above and below v (if any) — more detail on this computation is given in the paragraph below

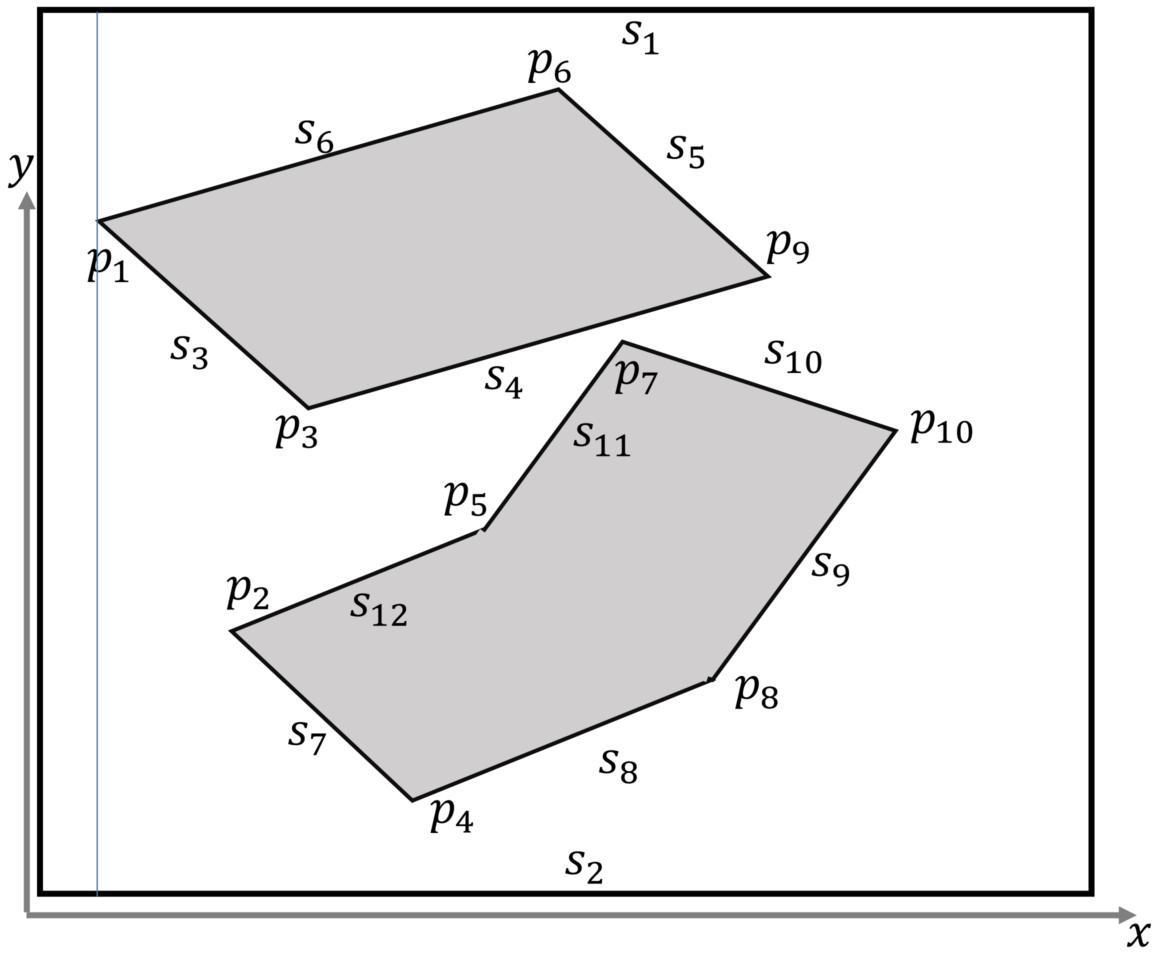
  5.1: add to $\mathcal{T}$ zero, one or two new trapezoids depending on vertex type (see figures in the next page)

  5.2: update the left endpoints of the obstacle segments in $\mathcal{S}$ above and below the vertex v

The type of v can be determined by checking its convexity and looking at the number of obstacle segments in $\mathcal{S}$ that have v as an endpoint.

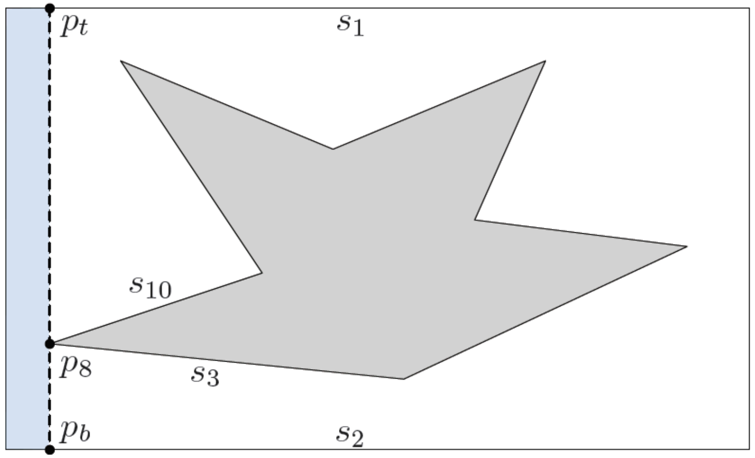
Type (i): left/left convex	Type (ii): left/left non-convex	Type (iii): right/right convex	Type (iv): right/right non-convex	Type (v): left/right convex	Type (vi): left/right non-convex
					
Have p_t *and* p_b	Have neither p_t *nor* p_b	Have p_t *and* p_b	Have neither p_t *nor* p_b	Have either p_t *or* p_b	Have either p_t *or* p_b
Both sides containing the vertex are present in current $\mathcal{S}$.	Both sides containing the vertex are present in current $\mathcal{S}$.	Zero side containing the vertex are present in current $\mathcal{S}$.	Zero side containing the vertex are present in current $\mathcal{S}$.	one side containing the vertex are present in current $\mathcal{S}$.	one side containing the vertex are present in current $\mathcal{S}$.

Sweeping Trapezoidation Algorithm: ‘Smarter’ Implementation



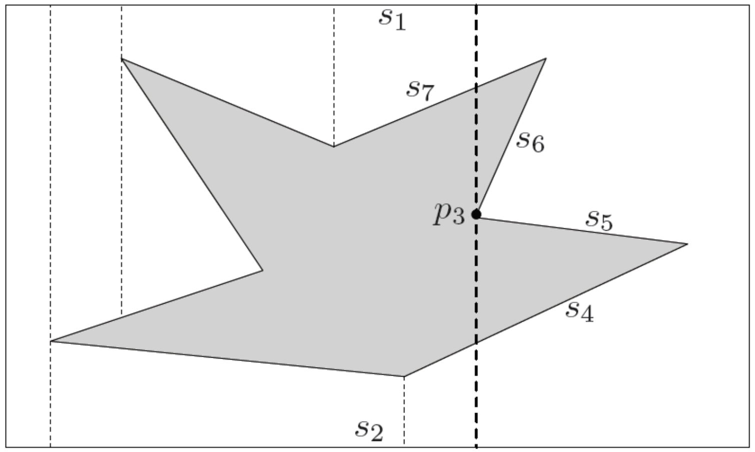
Sweeping Trapezoidation Algorithm: ‘Smarter’ Implementation (Example from textbook)

Type (i): left/left convex vertex



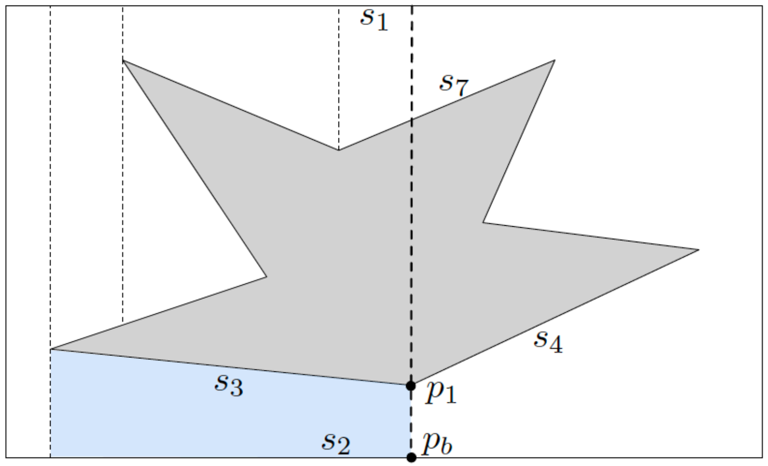
Update $\mathcal{S}$: $[s_1, s_2]$ to $[s_1, s_{10}, s_3, s_2]$
Add to $\mathcal{T}$: $[p_t, \ell_1, \ell_2, p_b]$
Update segment endpoints: $\ell_1 := p_t$, and $\ell_2 := p_b$

Type (ii): left/left non-convex vertex



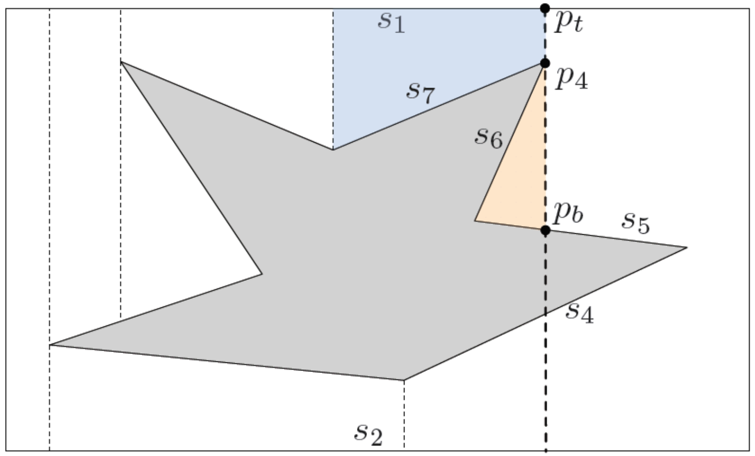
Update $\mathcal{S}$: $[s_1, s_7, s_4, s_2]$ to $[s_1, s_7, s_6, s_5, s_4, s_2]$
Add to $\mathcal{T}$: None
Update segment endpoints: None

Type (v): left/right convex vertex



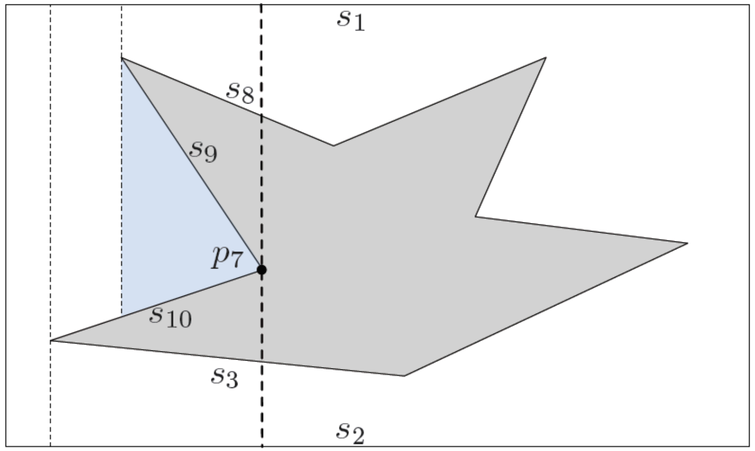
Update $\mathcal{S}$: $[s_1, s_7, s_3, s_2]$ to $[s_1, s_9, s_3, s_2]$
Add to $\mathcal{T}$: $[p_1, \ell_3, \ell_2, p_b]$
Update segment endpoints: $\ell_2 := p_b$

Type (iii): right/right convex vertex



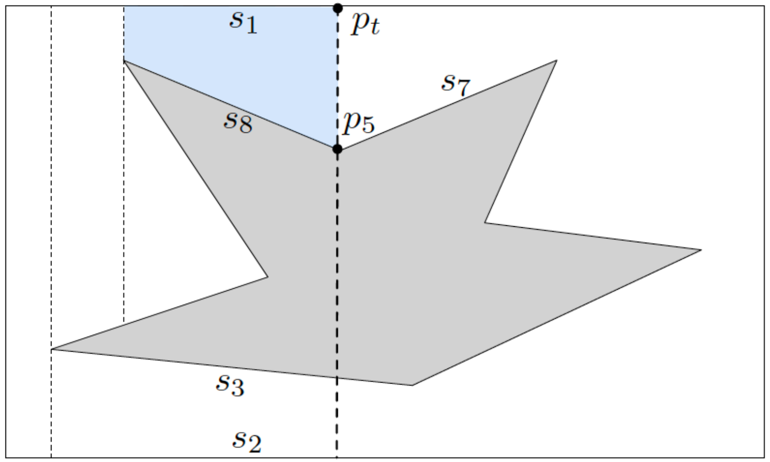
Update $\mathcal{S}$: $[s_1, s_7, s_6, s_5, s_4, s_2]$ to $[s_1, s_5, s_4, s_2]$
Add to $\mathcal{T}$: $[p_t, \ell_1, \ell_7, p_4]$, and $[p_4, \ell_6, p_b]$
Update segment endpoints: $\ell_1 := p_t$, and $\ell_5 := p_b$

Type (iv): right/right non-convex vertex



Update $\mathcal{S}$: $[s_1, s_8, s_9, s_{10}, s_3, s_2]$ to $[s_1, s_8, s_3, s_2]$
Add to $\mathcal{T}$: $[p_7, \ell_9, \ell_{10}]$
Update segment endpoints: None

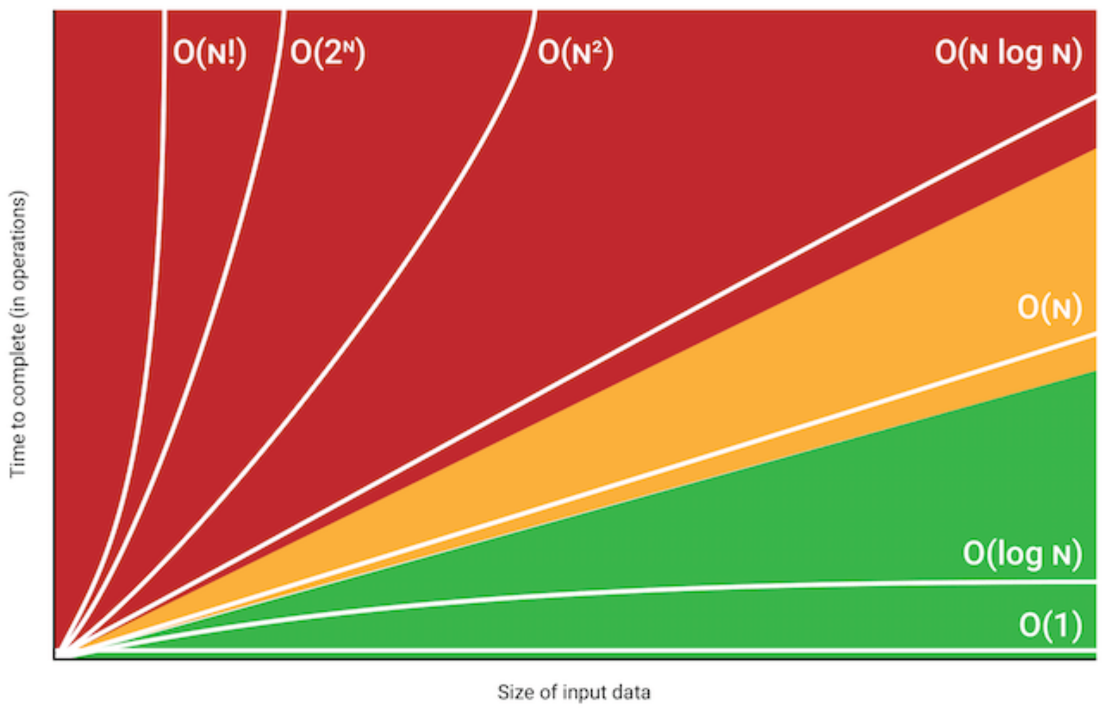
Type (vi): left/right non-convex vertex



Update $\mathcal{S}$: $[s_1, s_8, s_3, s_2]$ to $[s_1, s_7, s_3, s_2]$
Add to $\mathcal{T}$: $[p_t, \ell_1, \ell_8, p_5]$
Update segment endpoints: $\ell_1 := p_t$

Sweeping Trapezoidation Algorithm: Time Complexity

by using a more sophisticated data structure for S that allows us to insert and delete segments more quickly. In particular, a binary search tree can be used to maintain the ordered segments in S. A segment can be inserted/deleted in O(log(n)), instead of O(n) time for the simple array implementation. With a binary tree, the sweeping decomposition algorithm can be implemented with a run-time belonging to O(n log(n)) for a free workspace with n vertices.



Navigation on Roadmaps

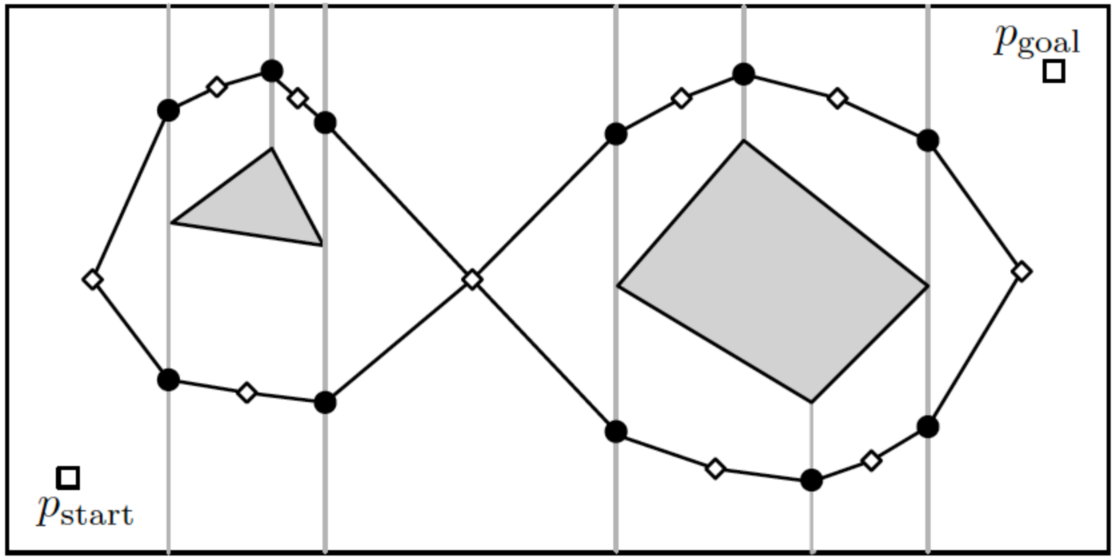
roadmap-from-decomposition algorithm

Input: the trapezoidation of a polygon (possibly with holes)
Output: a roadmap

1: label the center of each trapezoid with the symbol $\diamond$
2: label the midpoint of each vertical separating segment with the symbol $\bullet$
3: **for** each trapezoid :
4: connect the center to all the midpoints in the trapezoid
5: **return** the roadmap consisting of centers and connections between them through midpoints

As a result of this algorithm we obtain a roadmap specified as follows:

(1) a collection of center points (one for each trapezoid), and
(2) a collection of paths connecting center points (each path being composed of 2 segments, connecting a center to a midpoint and the same midpoint to a distinct center).



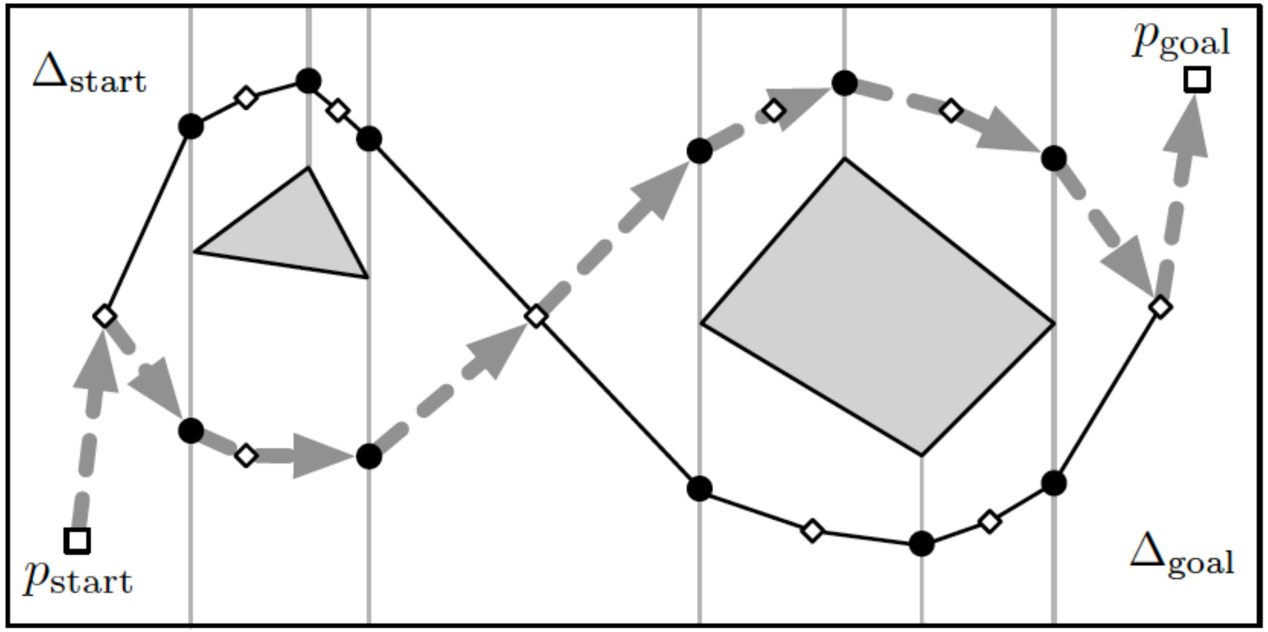
Navigation on Roadmaps

planning-via-decomposition+search algorithm

Input: free workspace W_{free}, start point p_{start} and goal point p_{goal}

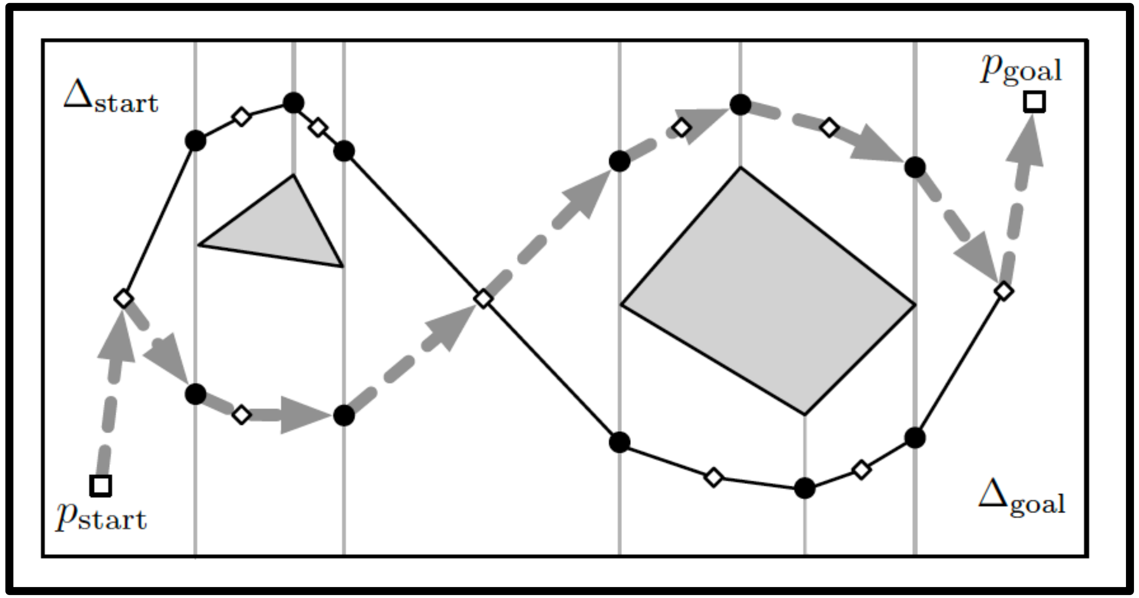
Output: a path from p_{start} to p_{goal} if it exists, otherwise a failure notice. Either outcome is obtained in finite time.

compute a decomposition of W_{free} and the corresponding roadmap
in the decomposition, find the start trapezoid Δ_{start} containing p_{start} and the goal trapezoid Δ_{goal} containing p_{goal}
in the roadmap, search for a path from Δ_{start} to Δ_{goal}
if no path exists from Δ_{start} to Δ_{goal} :
    return *failure* notice
else
    return path by concatenating:
        the segment from p_{start} to the center of Δ_{start},
        the path from the Δ_{start} to Δ_{goal}, and
        the segment from the center of Δ_{goal} to p_{goal}.

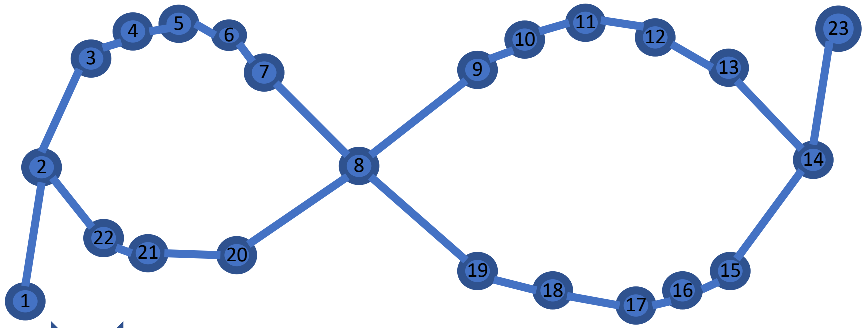


Navigation on Roadmaps: Optimality

- Multiple paths might exist from start to goal. Which one to choose?
  - Path with shortest length
    - But how to find that path



Obtain the dual graph of the road map

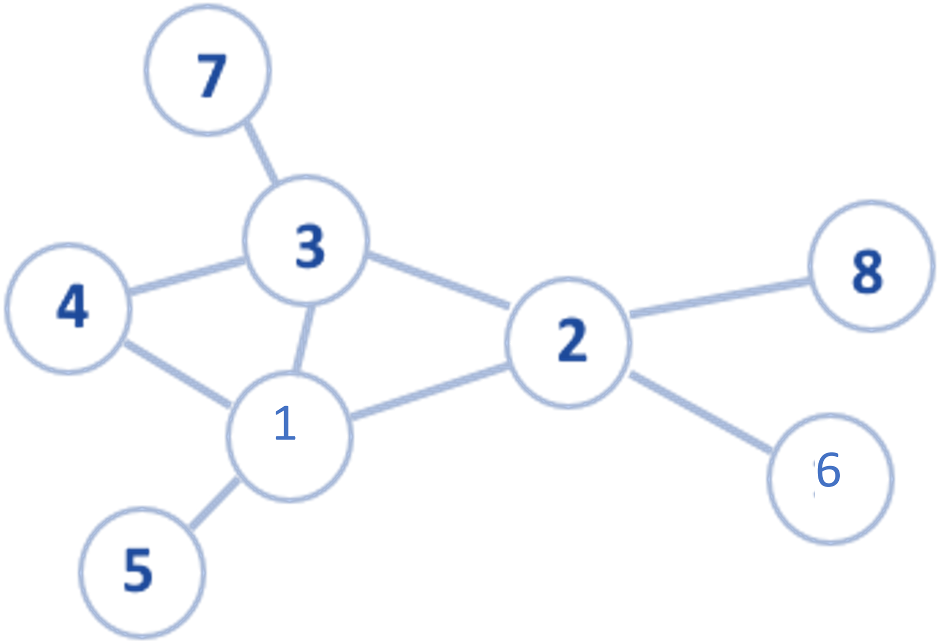


Solve the shortest-path problem:
Given a graph with a start node and a goal node, find a shortest path from the start node to the goal node.

We will use the breadth-first search (BFS) algorithm

BFS Algorithm

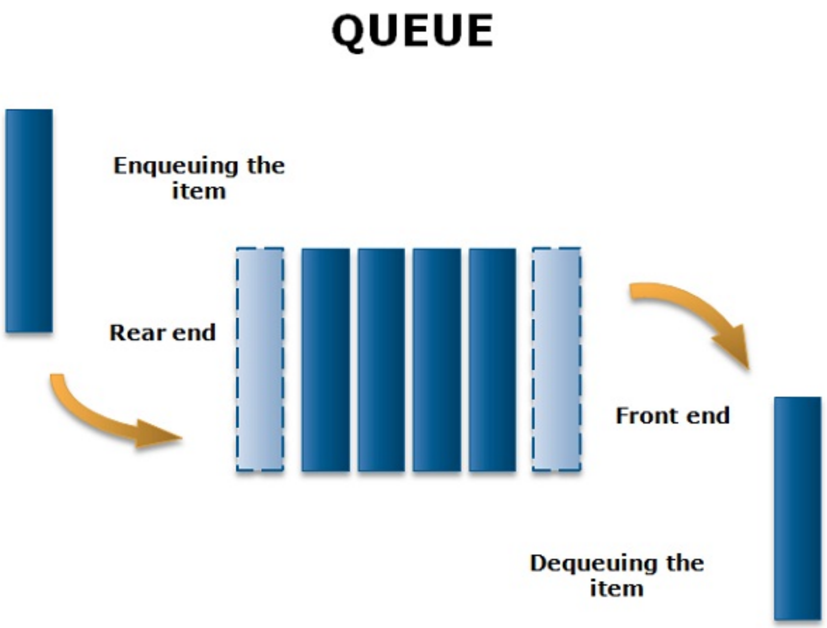
begin with the start node and mark as visited // *The start node forms Layer 0*
for each unvisited neighbor u of the start node :
    mark u as visited, and set the start node as the parent of u. // *The nodes u form Layer 1*
for each unvisited neighbor v of the nodes in Layer 1 :
    mark v as visited and record the neighbor from Layer 1 as the parent of v
repeat the process until you reach a layer that has no unvisited neighbors
if the goal node has been visited :
    follow the parent values back to the start node, and return this sequence of vertices as the shortest path from start to goal
else
    return a failure notice (i.e., the start and goal node are not path-connected)



BFS Algorithm: The Queue Solution Approach

A queue (also called first-in-first-out (FIFO) queue) is a variable-size data container

- Two operations
  - ➢ $insert(Q, v)$: inserts an item into the back of the queue
  - ➢ $retrieve(Q)$: returns the item that sits at the front of the queue



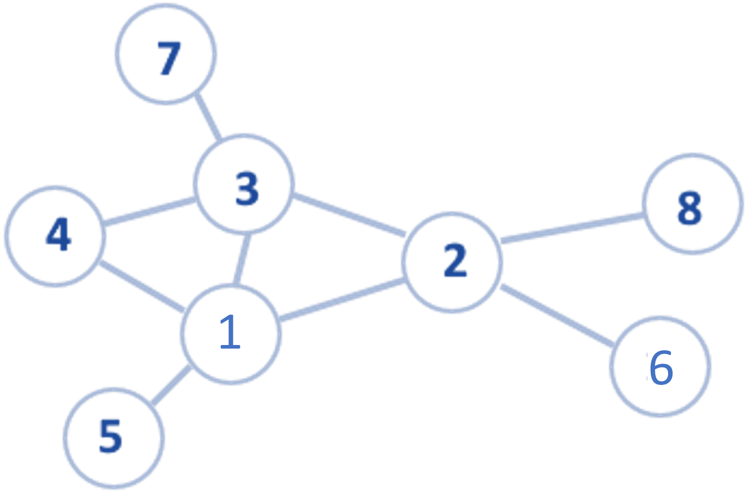
- Can be run such that each insert and retrieve runs in O(1) time

BFS Algorithm: The Queue Solution Approach

breadth-first search (BFS) algorithm

Input: a graph G, a start node v_{start} and goal node v_{goal}
Output: a path from v_{start} to v_{goal} if it exists, otherwise a failure notice

for each node v in G :
    $\texttt{parent}(v) := \text{NONE}$
$\texttt{parent}(v_{\text{start}}) := \text{SELF}$
create an empty queue Q and $\text{insert}(Q, v_{\text{start}})$
while Q is not empty :
    $v := \text{retrieve}(Q)$
    for each node u connected to v by an edge :
        if $\texttt{parent}(u) == \text{NONE}$:
            set $\texttt{parent}(u) := v$ and $\text{insert}(Q, u)$
        if $u == v_{\text{goal}}$:
            run *extract-path* algorithm to compute the path from start to goal
            return *success* and the path from start to goal
return *failure* notice along with the parent values.

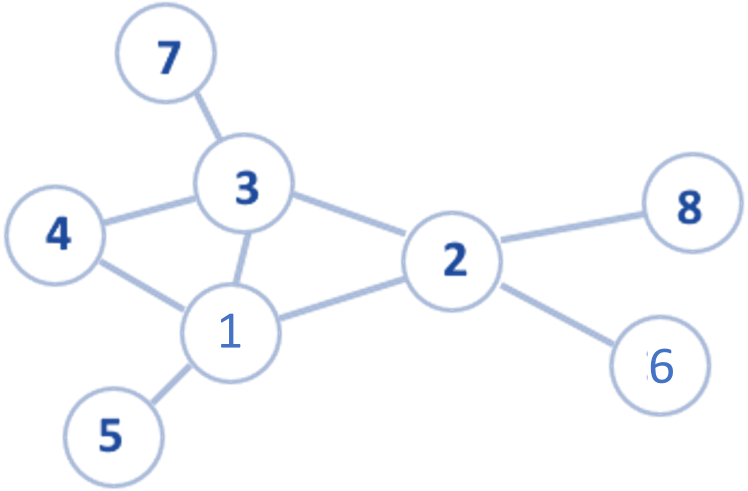


extract-path algorithm

Input: a goal node v_{goal}, and the parent values
Output: a path from v_{start} to v_{goal}

create an array $P := [v_{\text{goal}}]$
set $u := v_{\text{goal}}$
while $\texttt{parent}(u) \neq \text{SELF}$:
    $u := \texttt{parent}(u)$
    insert u at the beginning of P
return P

Representing a Graph



Representation #1 (Adjacency Table/List): a lookup table, that is, an array whose elements are lists of varying length: the i-th entry is a list of all neighbors of node i.

$AdjTable[1] =$

$AdjTable[2] =$

$AdjTable[3] =$

$AdjTable[4] =$

$AdjTable[5] =$

$AdjTable[6] =$

$AdjTable[7] =$

$AdjTable[8] =$

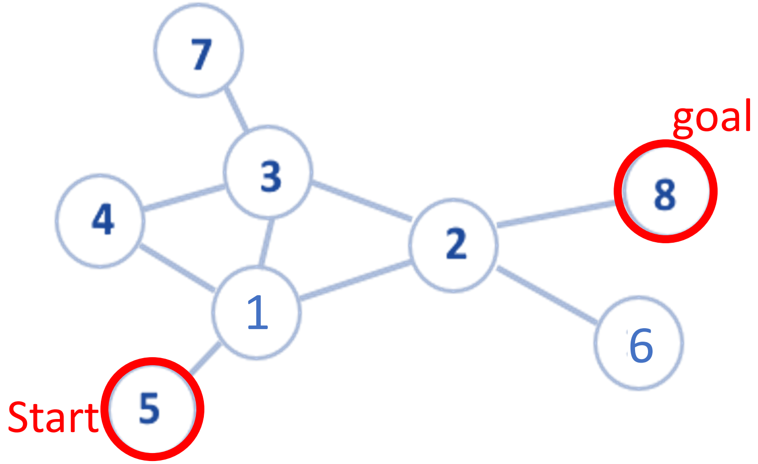
Representation #2 (Adjacency Matrix): a symmetric matrix whose (i,j) entry is equal to 1 if the graph contains the edge {i,j} and is equal to 0 otherwise.

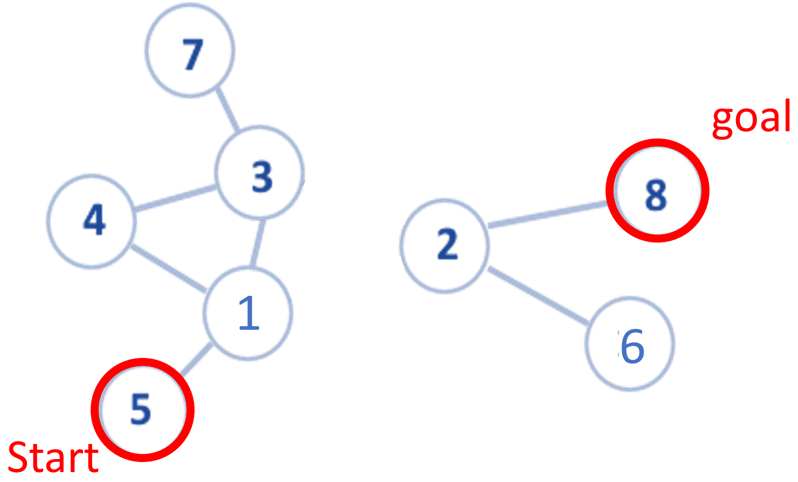
$$A = \begin{bmatrix} \quad \end{bmatrix}$$

Representation #3 (Edge List): an array, where each entry is an edge in the graph. This representation of edges is called an edge list.

Runtime of BFS

➢ Is BFS algorithm complete?

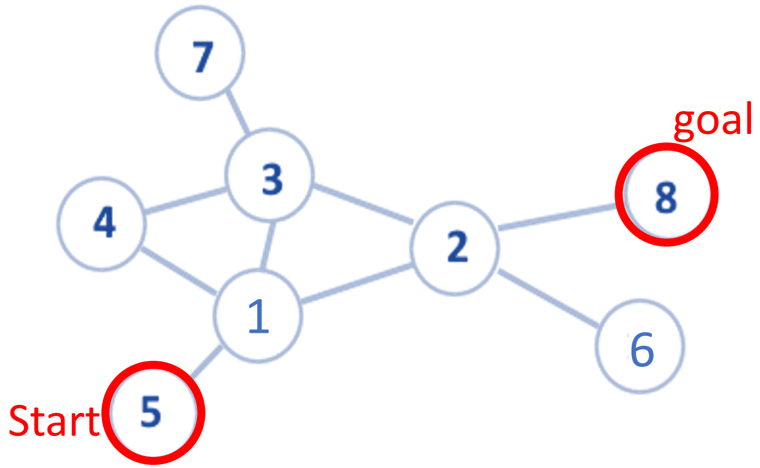




Runtime of BFS

➢ How quickly does it run?

Input to the algorithm: a graph G with node set V (|V|=n) and edge set E (|E|=m).



Theorem (Run-time of the BFS algorithm) Consider a graph G = (V;E) with n vertices and m edges, along with a start
and goal node. Then the runtime of the breadth-first-search algorithm is

- $O(n + m)$ if G is represented as an adjacency table,
- $O(n^2)$ if G is represented as an adjacency matrix, and
- $O(n.m)$ if G is represented as an edge list.

Runtime of BFS

breadth-first search (BFS) algorithm

Input: a graph G, a start node v_{start} and goal node v_{goal}
Output: a path from v_{start} to v_{goal} if it exists, otherwise a failure notice

```
for each node v in G :
    parent(v) := NONE
parent(v_start) := SELF
create an empty queue Q and insert(Q, v_start)
while Q is not empty :
    v := retrieve(Q)
    for each node u connected to v by an edge :
        if parent(u) == NONE :
            set parent(u) := v and insert(Q, u)
        if u == v_goal :
            run extract-path algorithm to compute the path from start to goal
            return success and the path from start to goal
return failure notice along with the parent values.
```

Initialization

Outer while loop

Inner for loop

extract-path algorithm

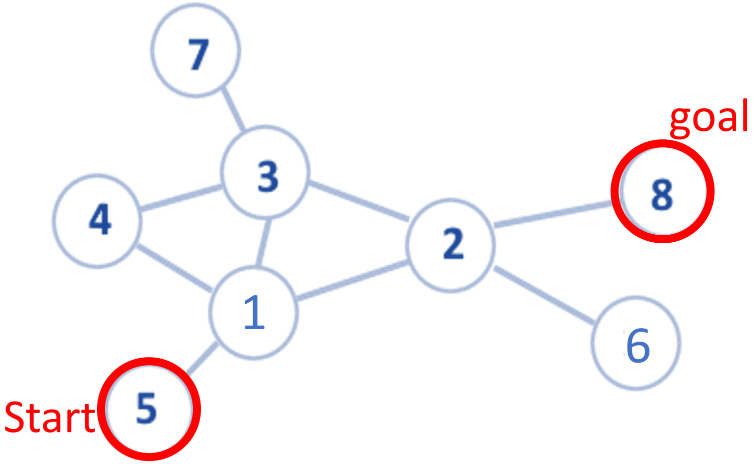
Input: a goal node v_{goal}, and the parent values
Output: a path from v_{start} to v_{goal}

```
create an array P := [v_goal]
set u := v_goal
while parent(u) ≠ SELF :
    u := parent(u)
    insert u at the beginning of P
return P
```

- If a graph is connected, then $m \geq n-1$
- for any undirected graph $m \leq \frac{n(n-1)}{2}$.
- $m \in O(n^2)$

From this we see that the best graph representation for
BFS is an adjacency table, followed by an adjacency matrix, followed by an edge list.

Runtime of BFS



➢ How quickly does it run?

Input to the algorithm: a graph G with node set V (|V|=n) and edge set E (|E|=m).

Theorem (Run-time of the BFS algorithm) Consider a graph G = (V;E) with n vertices and m edges, along with a start
and goal node. Then the runtime of the breadth-first-search algorithm is

- O(n + m) if G is represented as an adjacency table,
- O(n^2) if G is represented as an adjacency matrix, and
- O(n.m) if G is represented as an edge list.

- If a graph is connected, then $m \geq n - 1$
- for any undirected graph $m \leq \frac{n(n-1)}{2}$.
- $m \in O(n^2)$

From this we see that the best graph representation for BFS is an adjacency table, followed by an adjacency matrix, followed by an edge list.

References:

- F. Bullo and S. L. Smith. Lecture notes on robotic planning and kinematics
- H. Choset, K. Lynch, S. Hutchinson, G. Kantor, et al. Principles of Robot Motion, Theory, Algorithms, and Implementations.