(H)

Graph Representation and Basic Search

1) Graphs

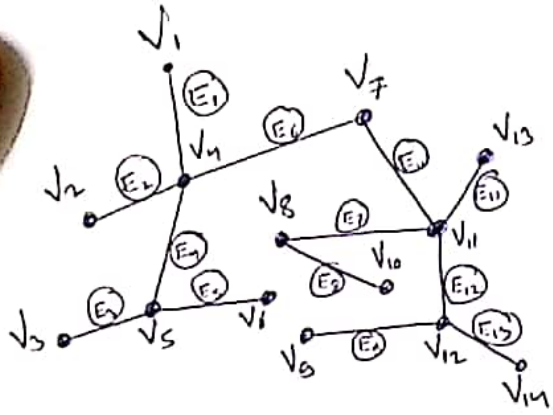
"A graph is a collection of nodes and edges"

$G = (V, E)$

⇒ Edges are either directed or Undirected.

"A tree is a directed graph {without any cycle} with a special node called the root"

→ {Possesses no incoming arc}



{Undirected Graph}

⇒ In motion planning, a node represents a Salient location, and an edge connects two nodes and corresponds to location that have an important relationship.

→ {Relationship could be that the nodes are mutually accessible from each other.}

⇒ Some time, edges are annotated with a non-negative numerical value reflective of the costs of traversing this edge.

↳ Such values are called weights.

→ A path
nodes
V: and

⇒ A graph
in the
V: and

→ A cycle
first

★ Tree

→ A par
Child

→ Root is

→ A node

→ The
the

Depth-f

Breadth

⇒ Best
is close

⇒ A path or walk in a graph is a sequence of nodes $\{V_i\}$ such that for adjacent nodes V_i and V_{i+1}, $E_{i,i+1}$ exist.

⇒ A graph is connected if for all nodes V_i and V_j in the graph, there exists a path connecting V_i and V_j.

⇒ A cycle is a path of n vertices such that first and last nodes are the same.

$$\{V_1 = V_n\}$$

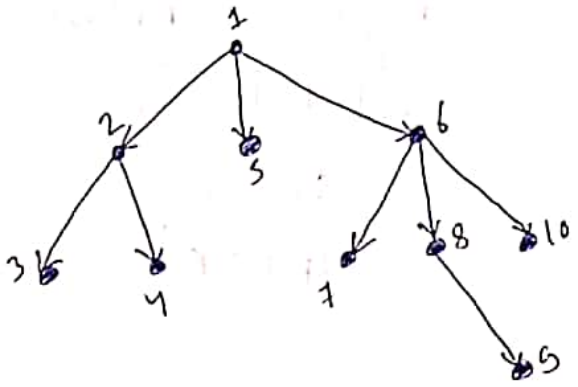
★ Tree

→ A parent node has nodes below it called children.

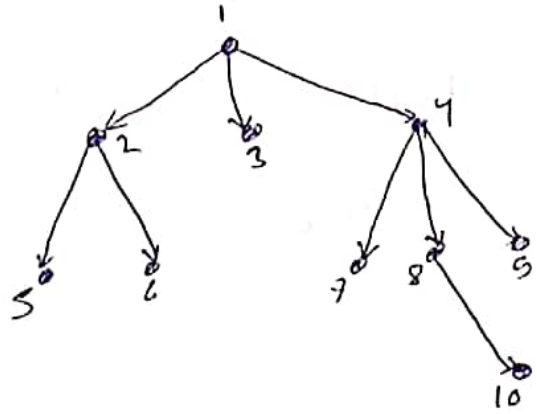
→ Root is a parent node but cannot be child node.

→ A node with no children is called leaf.

→ The removal of any nonleaf node breaks the connectivity of the tree.



Depth-first Search



Breadth-first Search

⇒ Best when target is close to root.

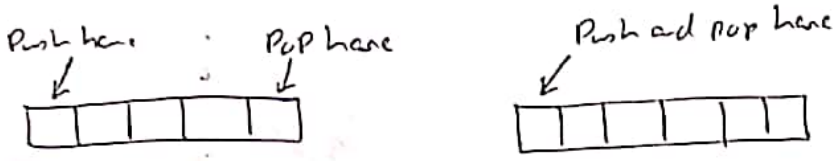
⇒ Typically, one searches a tree for a node with some desired properties such as the good location for the robot.

⇒ A grid induces a graph where each node corresponds to a pixel and an edge connects nodes of pixels that neighbor each other.
↳ (N4 or N8)

⇒ Graph that represents the grid is not a tree, but breadth-first and depth-first search techniques still apply.

⇒ Let the link length be the number of edges in the path of a graph.

⇒ Link length differ from path length in that the weight of the edges are ignored, only the number of edges counts.



Queue (FIFO)

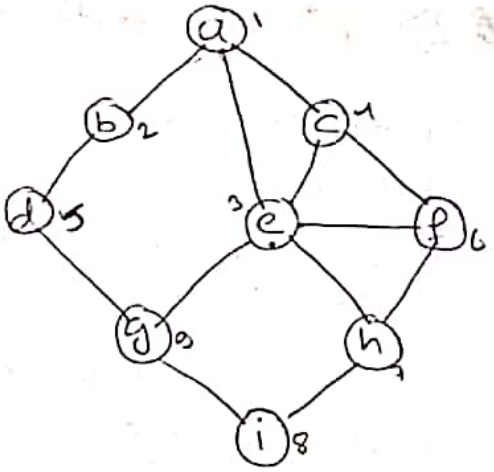
Stack (LIFO)

Breadth first Search ⟶ queue

Depth first Search ⟶ Stack

Greedy Search ⟶ Priority queue

Breadth first Search

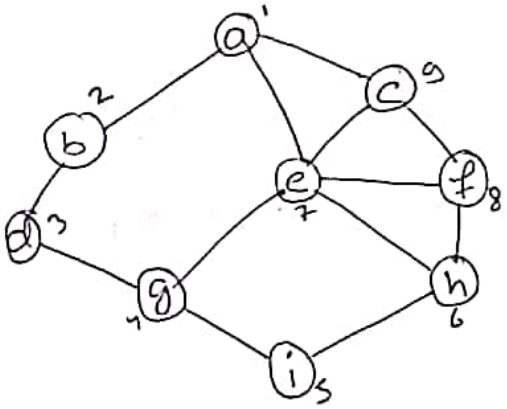


queue: ~~a~~ ~~b~~ ~~e~~ ~~c~~ ~~d~~ ~~f~~ ~~h~~ ~~i~~ ~~g~~

Is visited flag

a	b	c	d	e	f	g	h	i
✓	✓	✓	✓	✓	✓	✓	✓	✓

Depth first Search



Stack: ~~a~~ ~~b~~ ~~d~~ ~~g~~ ~~i~~ ~~h~~ ~~e~~ ~~f~~ ~~c~~

Is visited flag

a	b	c	d	e	f	g	h	i
✓	✓	✓	✓	✓	✓	✓	✓	✓

* Greedy Algorithm

"An algorithmic paradigm that follows the Problem Solving approach of making a locally optimal choice at each stage with the hope of finding a global optimum"

Pros ⇒ Simple, easy to implement, runs fast

Cons ⇒ Very often they dont provide a globally optimum solution.

2> A* Algorithm

⇒ Breadth-first search produces the shortest path to the start node in terms of link length.

⇒ However, shortest-path length is not the only metric we may want to optimize.

⇒ We also want to minimize the number of nodes that have to be visited to locate the goal node subject to our path-optimality criteria.

Optimality → Measure the path

Efficiency → Measure the search {number of nodes visited to determine the path}

⇒ The A* algorithm searches a graph efficiently, with respect to a chosen heuristic.

→ A* will produce an optimal path if its heuristic is optimistic.

⇒ If graph represented a grid, an optimistic heuristic could be the Euclidean distance to the goal.

⇒ The A* search has a priority queue which contains a list of nodes sorted by priority, which is determined by the sum of the distance traveled in the graph thus far from the start node, and the heuristic.

⇒ First no
is natu

⇒ Next w
the sta
nodes t
queue

⇒ The ex
a proces

★ Basic

⇒ The imp

⇒ Edge
have
require
adjace

⇒ The
path,
from

⇒ We
, an

⇒ Star
which

⇒ C(N
N, a

⇒ First node to be put into the priority queue is naturally the start node.

⇒ Next we expand the start node by popping the start node and putting all adjacent nodes to the start node into the priority queue sorted by their corresponding priorities.

⇒ The explicit path through the graph is represented by a series of back pointer.

(represents immediate history of the expansion process)

★ Basic Notation and Assumption

⇒ The input for A* is the graph itself.

⇒ Edge correspond to adjacent nodes and have values corresponding to the cost required to traverse between the adjacent nodes.

⇒ The output of A* algorithm is a back-pointer path, which is a sequence of nodes starting from the goal and going back to the start.

⇒ We will use two additional data structures, an open set O and a closed set C.

(Priority queue) (Contains all processed nodes)

⇒ Star(n) represents the set of nodes which are adjacent to n.

⇒ $C(n_1, n_2)$ is the length of edge connecting n_1 and n_2.

→ $g(n)$ is the total length of a backpointer Path from n to q_{start}.

→ $h(n)$ is the heuristic cost function, which returns estimated cost of shortest path from n to q_{goal}.

→ $f(n) = g(n) + h(n)$ is the estimated cost of shortest path from q_{start} to q_{goal} via n.

A* Algorithm

Input : A graph
Output : A path between start and goal nodes

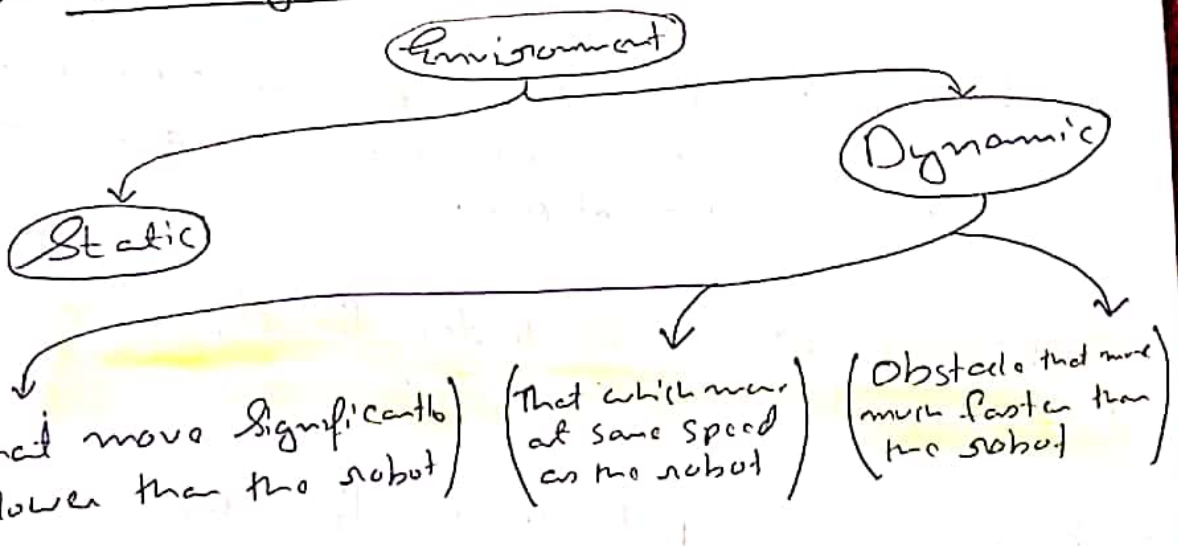
```
1  repeat
2    Pick n_best from O such that f(n_best) ≤ f(n), ∀n∈O
3    Remove n_best from O and add to C
4    If n_best = q_goal, EXIT
5    Expand n_best: for all x ∈ Succ(n_best) that are not in C
6    if x ∉ O then
7      add x to O
8    else if g(n_best) + C(n_best, x) < g(x)
9      Update x's backpointer to point to n_best.
10   end if
11 Until O is empty
```

* Discussion: Completeness, Efficiency and Optimality

Greedy Search
→ Special case of A* where $f(n)=h(n)$

Dijkstra's Algorithm
→ Special case of A* where $f(n)=g(n)$

3) D* Algorithm



⇒ The superfast obstacle case is easy to ignore because the obstacles will be moving so fast that there probably is no need to plan for them because they will either move too fast for the planner to have time to account for them or they will be in and out of the robot's path so quickly that it does not require any consideration.

⇒ D* algorithm can deal with dynamic obstacle.

⇒ D* initially determines a path starting with the goal and working back to the start using a slightly modified Dijkstras algorithm.

↳ The modification involves updating a heuristic and a minimum heuristic function. (h) (k)

⇒ The minimum heuristic values (k) are the estimate of the shortest path length to the goal.

↳ Both h and k values ~~are~~ will vary as the D* search runs, but they are equal upon initialization.

At present D* Lite is standard for dynamic planning for real-world use

⇒ h is estimated path length from the particular cell to the goal, not necessarily the shortest path length to the goal as it was for A*.

⇒ On this approach, the h value do not respect the presence of obstacles when reflecting distance to the goal node.

x_g	x_2	x_3
x_8	x_1	x_4
x_7	x_6	x_5

$C(x_1, x_2) = 1$

$C(x_1, x_5) = 1.4$

$C(x_1, x_8) = 10000$ if x_8 is in obstacle

$C(x_1, x_5) = 10000.4$ if x_5 is in obstacle.

⇒ Pixel with k value is less than h value, such pixel is said to have raised state.

⇒ When a pixel is in a raised state, its back pointer may no longer point to an optimal path.

⇒ This algorithm uses the following notation:-

→ X represents a state.

→ O is the priority queue.

→ L is the list of all state

→ G is the goal state

→ S is the start state.

→ t(X) is value of state with regards to the priority queue.

- t(X) = NEW, if X has never been in O
- t(X) = OPEN if X is currently in O
- t(X) = CLOSED if X was in O but currently is not.

→ C(X, Y) is the estimated path length between adjacent states X and Y.

→ $h(x)$ is the estimated cost of a path from x to Goal (heuristic).

→ $k(x)$ is the estimated cost of the shortest path from x to Goal.

→ $b(x) = y$ implies that y is a parent state of x.

→ $c(x, y)$ is the measured distance (sensor measurement) between adjacent states x and y.

D* Algorithm

Input: List of all states L

Output: The goal state if it is reachable and the list of states L are updated so that the backpointer list describes a path from the start to the goal. If the goal state is not reachable, return NULL.

```
1  for each X ∈ L do
2    t(X) = NEW
3  end for
4  h(G) = 0 = k(G)        ← Addition
5  O = {G}
6  X_c = S
   { The following loop is Dijkstra's search
     for an initial path. }
9  repeat
8    K_min = PROCESS_STATE(O, L)
10 Until (K_min = -1) or (t(X_c) = CLOSED)
```

```
P = GET_BACKPOINTER_LIST(L, Xc, G)
if P = NULL then
    Return (NULL)
end if
Repeat
  for each neighbor Y ∈ L of Xc do
    if σ(Xc, Y) ≠ c(Xc, Y) then
      MODIFY_COST (O, Xc, Y, σ(Xc, Y))
      Repeat
        Kmin = PROCESS_STATE (O, L)
      Until (Kmin ≥ h(Xc)) or (Kmin = -1)
      P = GET_BACKPOINTER_LIST (L, Xc, G)
      if P = NULL then
        Return (NULL)
      end if
    end if
  end for
  Xc = the second element of P {Move to the next state in P}
  P = GET_BACKPOINTER_LIST (L, Xc, G)
Until Xc = G
Return (Xc)
```

GET_BACKPOINTER_LIST (L, S, G)

Input: A list of states L and two states (start & goal)
Output: A list of state from start to goal as described by the backpointers in the list of state L

```
1 if path exist then
2    Return (The list of states).
3 else
4    Return (NULL)
5 endif
```

Algorithm: INSERT (O, X, hnew)

Input: Open list, a state, and an h-value
Output: Open lis is modified.

```
1  if t(x) = NEW then
2     K(x) = hnew
3  else if t(x) = OPEN then
4     K(x) = min (K(x), hnew)
5  else if t(x) = CLOSED then
6     K(x) = min (h(x), hnew)
7  end if
8  h(x) = hnew
9  t(x) = OPEN
10 Sort O based on increasing K values.
```

Algorithm: MODIFY_COST (O, X, Y, cval)

Input: The open list, two states and a value

Output: A K-value and the open list gets updated.

1. c(X,Y) = cval
2. if t(X) = CLOSED then
3. ~~INSERT (O, X, h(X))~~ INSERT(O, Y, h(Y))
4. end if
5. Return GET_KMIN(O)

MLgorithm: MIN_STATE(O)

Input: The open list O

Output: The state with minimum K value in the list & related values.

1. if O = ∅ then
2. Return (-1) {Should be NULL}.
3. else
4. Return (argmin$_{Y \in O}$ K(Y))
5. end if

Algorithm: GET_KMIN(O)

Input: The open list O

Output: Lowest K-value of all states in the open list.

1. if O = ∅ then
2. Return (-1)
3. else
4. Return (min$_{Y \in O}$ K(Y))
5. end if

Algorithm : PROCESS_STATE

Input: List of all states L and the list of all state that are open O.

Output: A k_{min}, an updated list of all states, and an updated open list.

```
1  X = MIN_STATE (O)
2  if X = NULL then
3     Return (-1)
4  else if
5  Kold = GET_KMIN (O)
6  DELET (X)
7  if Kold < h(x) then
8     for each neighbor Y ∈ L of X do
9        if h(Y) ≤ Kold and h(X) > h(Y) + c(Y,X) then
10          b(X) = Y
11          h(x) = h(Y) + c(Y,X);
12       end if
13    end for
14 else if Kold = h(X) then
15    for each neighbor Y ∈ L of X do
16       if (t(Y) = NEW) or (b(Y) = X and
         h(Y) ≠ h(x) + C(X,Y) or (b(Y) ≠ X
         and h(Y) > h(x) + C(x,Y)) then
17          b(Y) = X
18          INSERT (O,Y, h(x) + C(x,Y))
19       end if
20    end for
21 else
```

```
for each neighbor Y ∈ L of X do
    if (t(Y)=NEW) or (b(Y)=X and h(Y)≠h(x)
     + C(x,Y)) then
        b(Y)=X
        INSERT (O,Y,h(x)+C(x,Y))
    else if b(Y)≠X and h(Y) > h(x) + C(x,Y) then
      INSERT (O, X, h(x))
    else if (b(Y)≠X and h(x) > h(Y) + C(x,Y))
     and (t(Y)=CLOSED) and (h(Y) > Kold) then
        INSERT (O, Y, h(Y))
    endif
endfor
endif
Return GET_KMIN (O)
```

Assumption

DELETE(x) → Remove that from the Priority Queue.
DELETE(x) → Update t(x) as closed