

L02: Discrete Planning

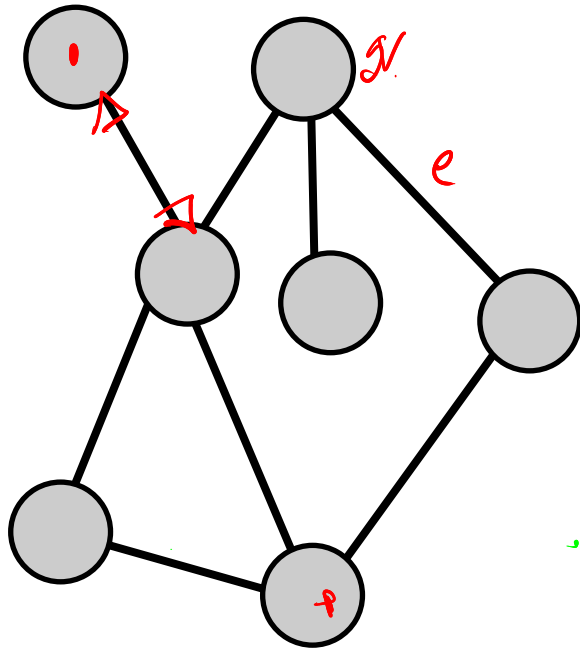
Planning Algorithms in AI and Robotics

Discrete Planning (LaValle 2.1-2.2)

- In this lecture we will discuss one of the most fundamental and straight-forward solutions to planning: transforming the problem to a **graph search** problem.
- On this first approach, we will study discrete spaces only. Why?
 - 1) There are many examples of discrete problems.
 - 2) Many problems accept discretization. Not all of them.
- The objective is to find a feasible solution (plan) to the planning problem, if one exists, it is guaranteed to be found.
- ◆ The negative side of this methods is the curse of dimensionality, for more than 4 dimensions things start getting slow...
- ◆ There are heuristics to alleviate this problem.

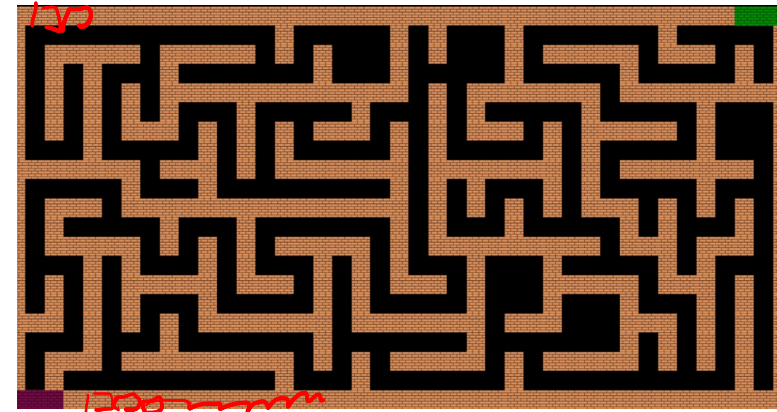
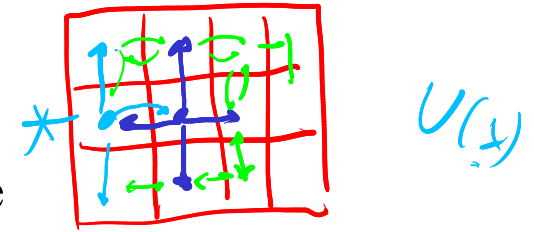
Examples of discrete planning problems

Traversing a graph $G(V, E)$



Given a graph, this is a direct example of graph search. However, the full structure of the graph might not be known *a priori*.

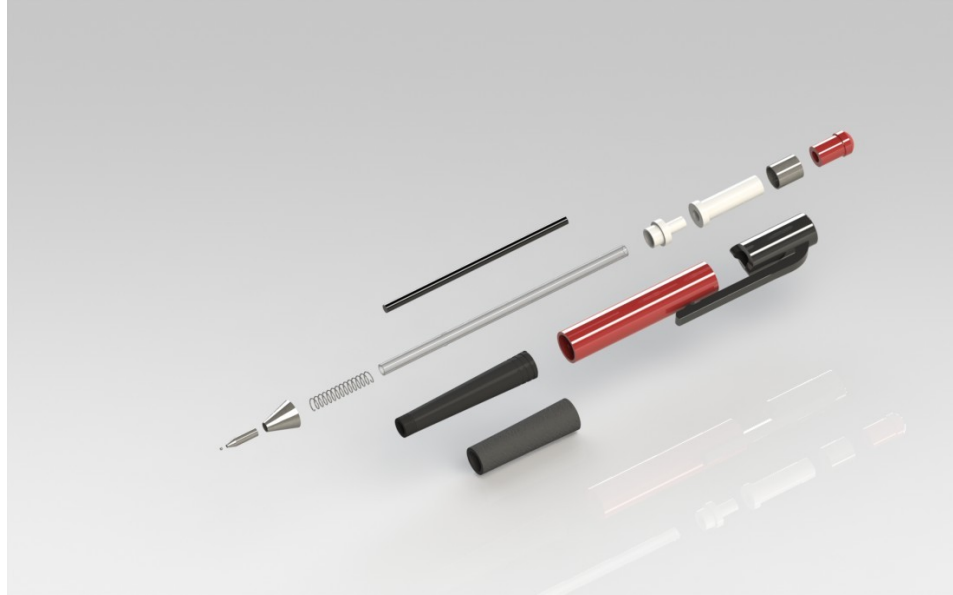
Escaping a maze



The plan will consist of a sequence of connected cells in order to escape. The state transition graph is revealed as we plan.

Examples of discrete planning problems

Logic-based planning

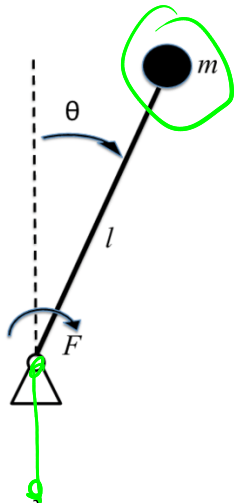


How to assemble the pen? The state space is discrete, but the size of it is gigantic. Combinatorial problem.

We will not focus on this kind of problems, but give a brief hint.

Examples: discretization of dynamics

Inverted pendulum

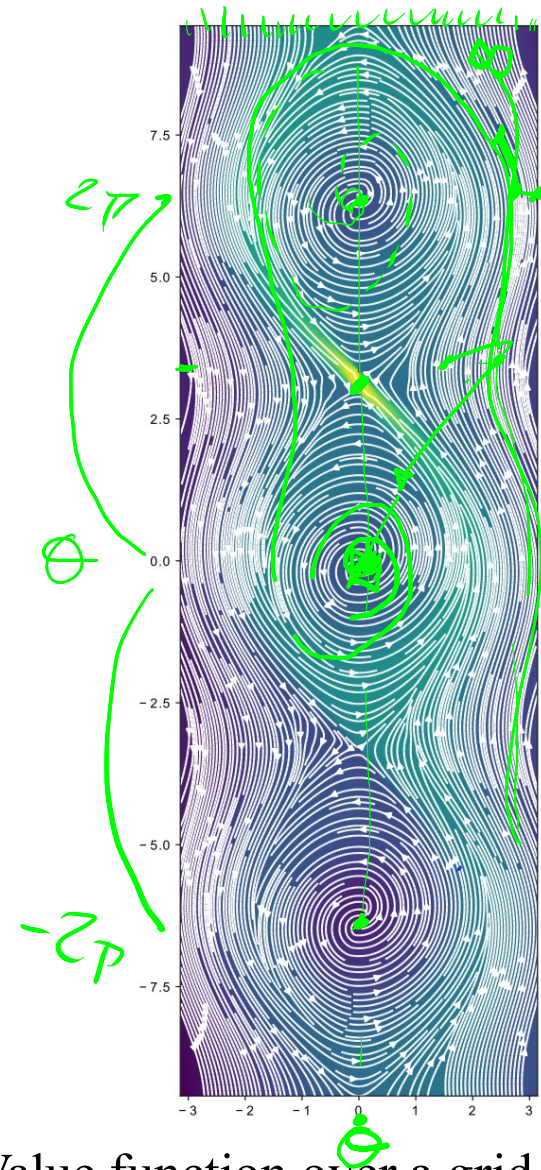


$$x = [\theta, \dot{\theta}]^\top$$

$$u = \ddot{\theta}$$

In this example, we consider dynamics and stochastic perturbations. Still, the state space and action space can be discretized such that a **plan** can be found.

In this course we will not discuss about dynamics or stochastic processes, but on random variables.



Value function over a grid of the discretized state of this system.

-

$$x' = f(x, \underline{u}) \in \mathcal{X}$$

+

[illegible]

$$U(x_1) = \{u, d, l, r\}$$

Definition: An algorithm A is complete if in a finite amount of time, A always finds a solution if a solution exists or otherwise A determines that a solution does not exist.

Discrete planning becomes Graph Search

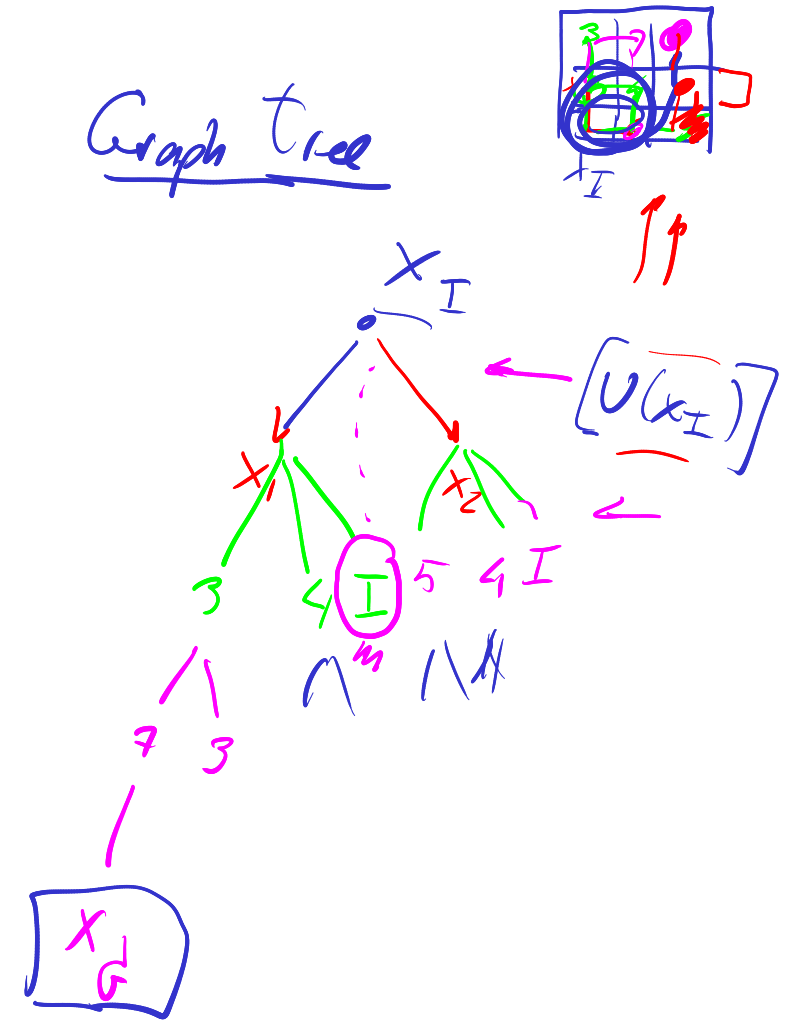
Executing actions creates a directed **graph** (tree) as we search over it.

If all information was known from the beginning, it would be standard graph search.

Graph search is systematic: it calculates in finite time whether or not a solution exists. For this, it keeps track of all visited states.

Some methods focus on improving efficiency by reducing the required visited states before finding a plan.

We will see later the negative side.



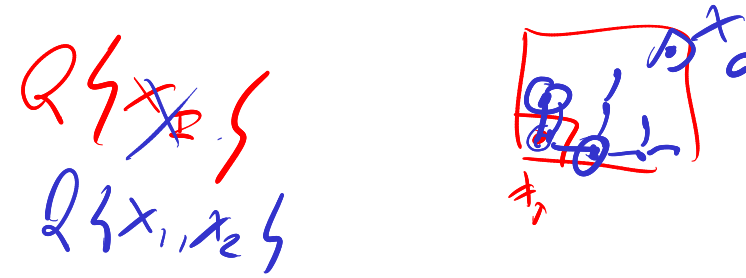
General Forward Search Algorithm

Forward search:

Inputs: x_I , X_G .

Output: {SUCCESS,FAILURE}

```
1  Q = {} common queue
2  Q.insert( $x_I$ ) and mark  $x_I$  as visited
3  while Q not empty:
4       $x \leftarrow Q.get\_first()$  → remove
5      if  $x$  in  $X_G$ :
6          return SUCCESS
7      for  $u$  in  $U(x)$ :
8           $x' \leftarrow f(x,u)$ 
9          if  $x'$  not visited:
10             Mark  $x'$  as visited
11             Q.insert( $x'$ )
12      else:
13          resolve duplicate  $x'$ 
14  return FAILURE
```



- This algorithm is a graph search algorithm where the state transition graph is revealed incrementally after applying actions (line.7).
- The resultant graph is a tree that grows as we explore cells.
- This algorithm does not calculate a plan...but a simple modification will allow.

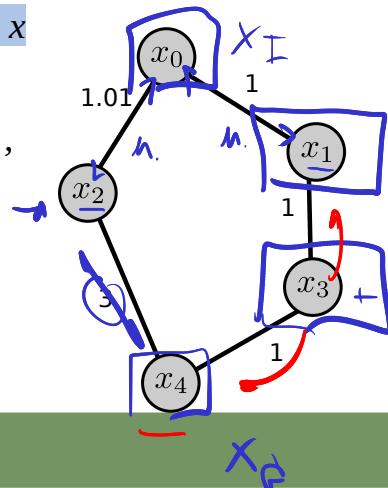
Particular case: Breadth First

Breadth First:

Inputs: x_I, X_G .

Output: {SUCCESS,FAILURE}, plan

```
1  -  $Q = \{\}$  First In First Out (FIFO)
2  -  $Q.insert(x_I)$  and mark  $x_I$  as visited
3  while Q not empty:
4       $x \leftarrow Q.get\_first()$ 
5      if  $x$  in  $X_G$ :
6          return SUCCESS, plan()
7      for  $u$  in  $U(x)$ :
8           $x' \leftarrow f(x,u)$ 
9          if  $x'$  not visited:
10             Mark  $x'$  as visited
11             Parent_table( $x'$ ) =  $x$ 
12              $Q.insert(x')$ 
13     else:
14         resolve duplicate  $x'$ 
15 return FAILURE
```



$$Q = \{x_0\}$$

$$x_I = x_0$$

$$i=1, \quad x = x_0$$

$$Q = \{x_1, x_2\}$$

$$i=2 \quad x = x_1$$

$$Q = \{x_2, x_3\}$$

$$i=3 \quad x = x_2$$

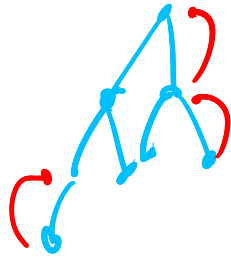
$$Q = \{x_3, x_4\}$$

$$i=4$$

$$x = x_3$$

$$i=5 \quad x = x_4 = x_G //$$

Depth First



Forward search:

Inputs: x_I, X_G .

Output: {SUCCESS,FAILURE}, plan

```

1  Q = {} Last In First Out (LIFO)
2  Q.insert( $x_I$ ) and mark  $x_I$  as visited
3  while Q not empty:
4       $x \leftarrow Q.get\_first()$ 
5      if  $x$  in  $X_G$ :
6          return SUCCESS, plan()
7      for  $u$  in  $U(x)$ :
8           $x' \leftarrow f(x,u)$ 
9          if  $x'$  not visited:
10             Mark  $x'$  as visited
11             Parent_table( $x'$ ) =  $x$ 
12             Q.insert( $x'$ )
13         else:
14             resolve duplicate  $x'$ 
15 return FAILURE
    
```

Parent Table

node	parent
x_0	x_I
x_1	x_0
x_2	x_0
x_3	x_2
x_4	x_2

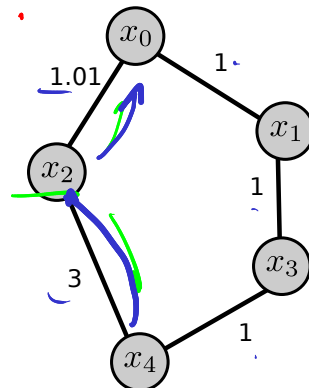
$Q = \{x_0\}$

$i=1$ $x = x_1$, $Q = \{x_1, x_2\}$

$i=2$ $x = x_2$

$Q = \{x_3, x_4\}$

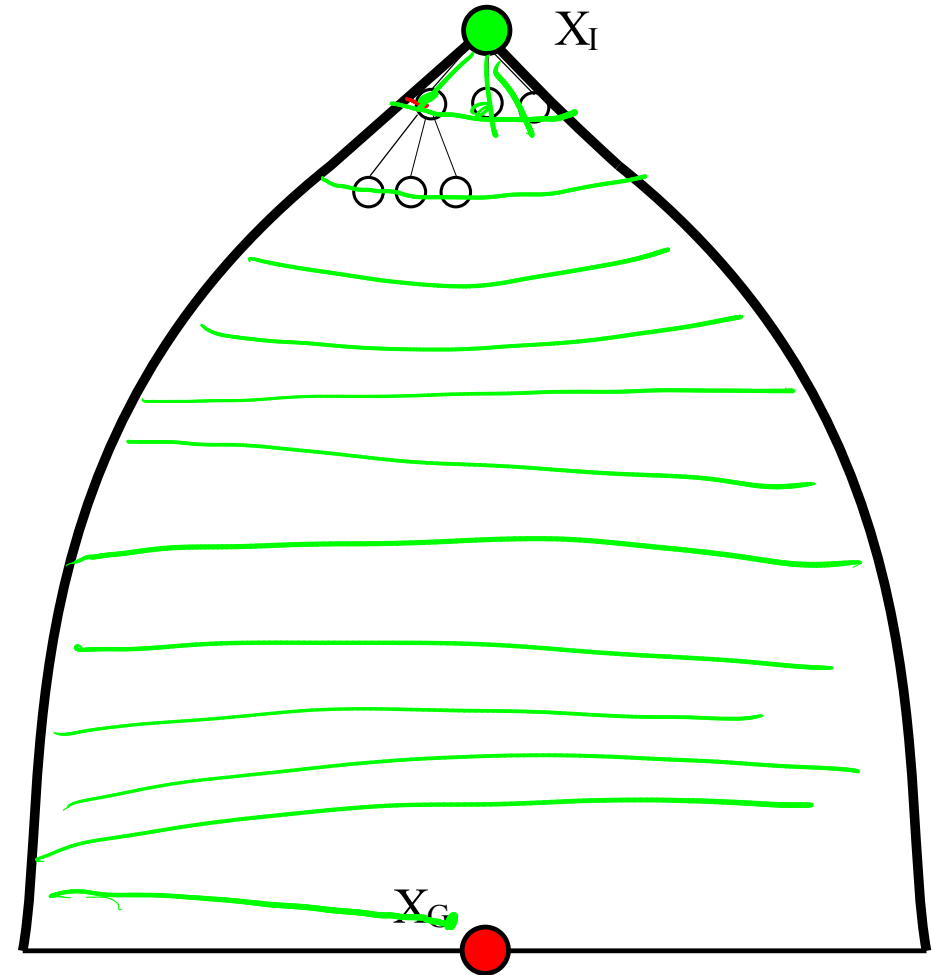
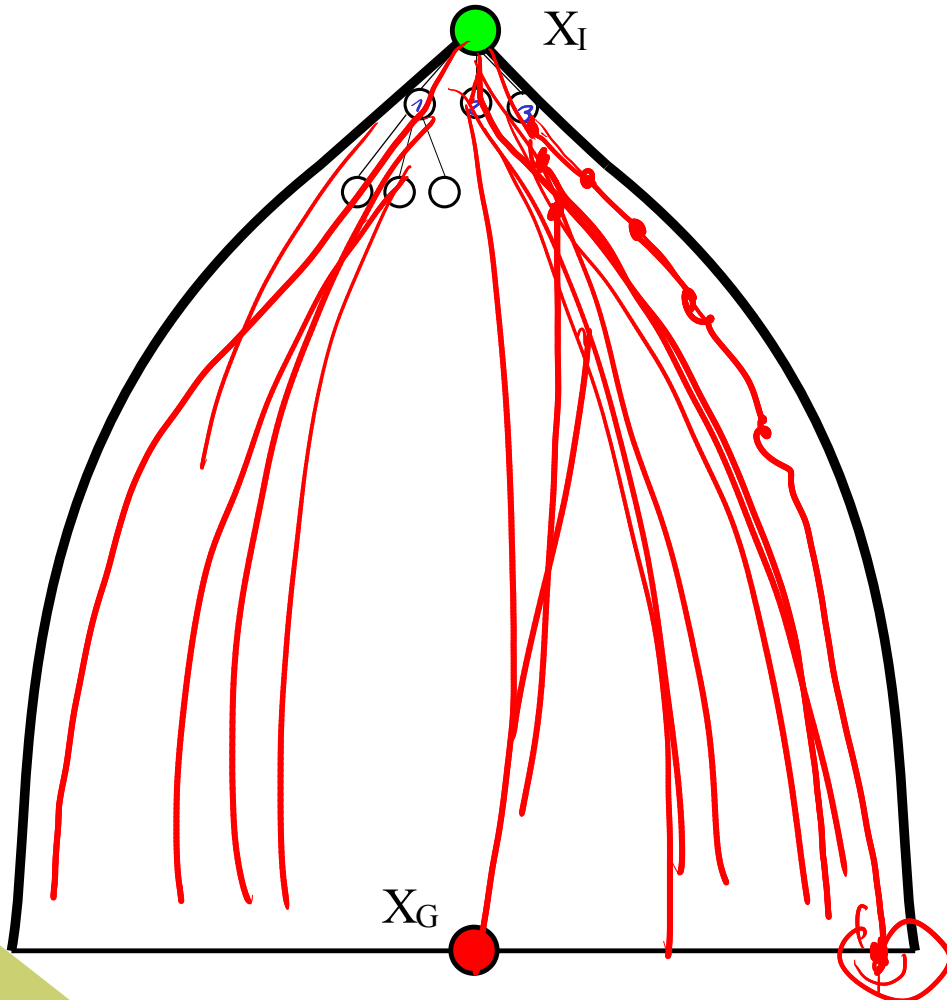
$i=3$ $x = x_4$



Depth First

vs

Breath First



Dijkstra's algorithm

Dijkstra:

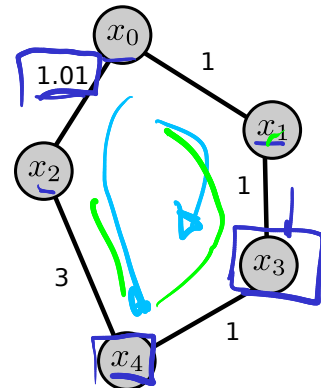
Inputs: x_i, X_G .

Output: {SUCCESS,FAILURE}, plan

```

1  Q = {} Priority Queue
2  Q.insert( $x_i$ ) and mark  $x_i$  as visited
    $C(x_i) = 0$ 
3  while Q not empty:
4       $x \leftarrow Q.get\_first()$ 
5      if  $x$  in  $X_G$ :
6          return SUCCESS, plan()
7      for  $u$  in  $U(x)$ :
8           $x' \leftarrow f(x,u)$ 
9          if  $x'$  not visited:
10             Mark  $x'$  as visited
11             parent_table( $x'$ ) =  $x$ 
12              $C(x') = C(x) + l(x,u)$ 
13             Q.insert( $x', C(x')$ )
14             else:
15                 if  $C(x) + l(x,u) < C(x')$ :
16                      $C(x') = C(x) + l(x,u)$ 
17                     parent_table( $x'$ ) =  $x$ 
18             return FAILURE

```



$$Q = \{ (x_0, 0) \}$$

$$C(x_0) + l(x_0, u)$$

$$i=1, x=x_0, Q = \{ (x_2, 1.01), (x_3, 1) \}$$

$$i=2, x=x_1, Q = \{ (x_2, 1.01), (x_3, C(x_0) + l(x_1, u)) \}$$

$$i=3, x=x_2, Q = \{ (x_3, 2), (x_4, 4) \}$$

$$C(x_3) + l(x_3, u) < C(x_4) = 4 \quad l > 0$$

$C(x)$ can be seen as a **cost-to-come** to the current state from the initial state.

$$Q = \{ (x_4, 3) \}$$

$$i=5, x=x_4$$

A*

A-star:

Inputs: x_I, X_G .

Output: {SUCCESS,FAILURE}, plan

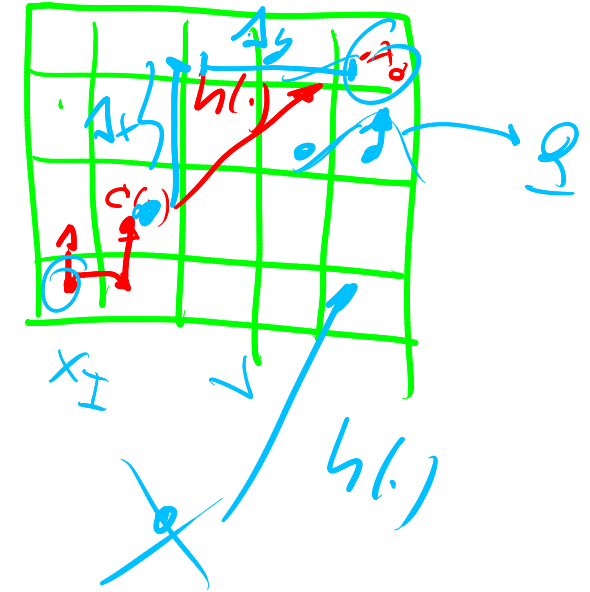
```
1  Q = {} Priority Queue ✓
2  Q.insert( $x_I$ ) and mark  $x_I$  as visited
    $C(x_I) = 0$  ✓
3  while Q not empty:
4       $x \leftarrow Q.get\_first()$ 
5      if  $x$  in  $X_G$ :
6          return SUCCESS, plan()
7      for  $u$  in  $U(x)$ :
8           $x' \leftarrow f(x,u)$ 
9          if  $x'$  not visited:
10             Mark  $x'$  as visited
11             parent_table( $x'$ ) =  $x$ 
12              $C(x') = C(x) + l(x,u)$  ✓
13              $Q.insert(x', C(x') + h(x', x_G))$  ✓
14         else:
15             if  $C(x) + l(x,u) < C(x')$ :
16                  $C(x') = C(x) + l(x,u)$  ✓
17                 parent_table( $x'$ ) =  $x$  ✓
18 return FAILURE
```

$h = 0$.

2D Manhattan distance

$$d(x, x_G) = (x_{1x} - x_{2x}) + (x_{3x} - x_{2y})$$

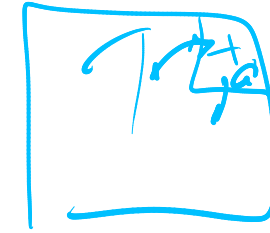
- The function $h(x, x_G)$ is an heuristic estimation of the cost-to-go to the goal.
- For guaranteeing optimal paths, $h()$ must be an underestimate of the real cost-to-go distance.
- The design of heuristic functions is not easy.
- For the 2D case, a good example is L1 norm (Manhattan distance)



A*, choosing good heuristics

- A correct heuristic function can speed up the graph search dramatically.
- The problem is when designing heuristic functions for non-trivial state spaces. —
- The effect on heuristics functions can alter the solution, obtaining a non-optimal path. ~
- We will retake the discussion on the relation to the cost-to-come and cost-to-go when talking about dynamic programming (L05).)

Backward Search



Backward search:

Inputs: x_I, x_G .

Output: {SUCCESS,FAILURE}

```
1  Q = {} common queue
2  Q.insert( $x_G$ ) and mark  $x_G$  as visited
3  while Q not empty:
4       $x' \leftarrow Q.get\_first()$ 
5      if  $x'$  in  $x_I$ :
6          return SUCCESS
7      for  $u^{-1}$  in  $U^{-1}(x')$ :
8           $x \leftarrow f^{-1}(x', u^{-1})$ 
9          if  $x'$  not visited:
10             Mark  $x$  as visited
11             Q.insert( $x$ )
12      else:
13          resolve duplicate  $x$ 
14  return FAILURE
```

- The definition of the inverse action space:

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

- The definition of backward function:

— $x = f^{-1}(x', u^{-1})$

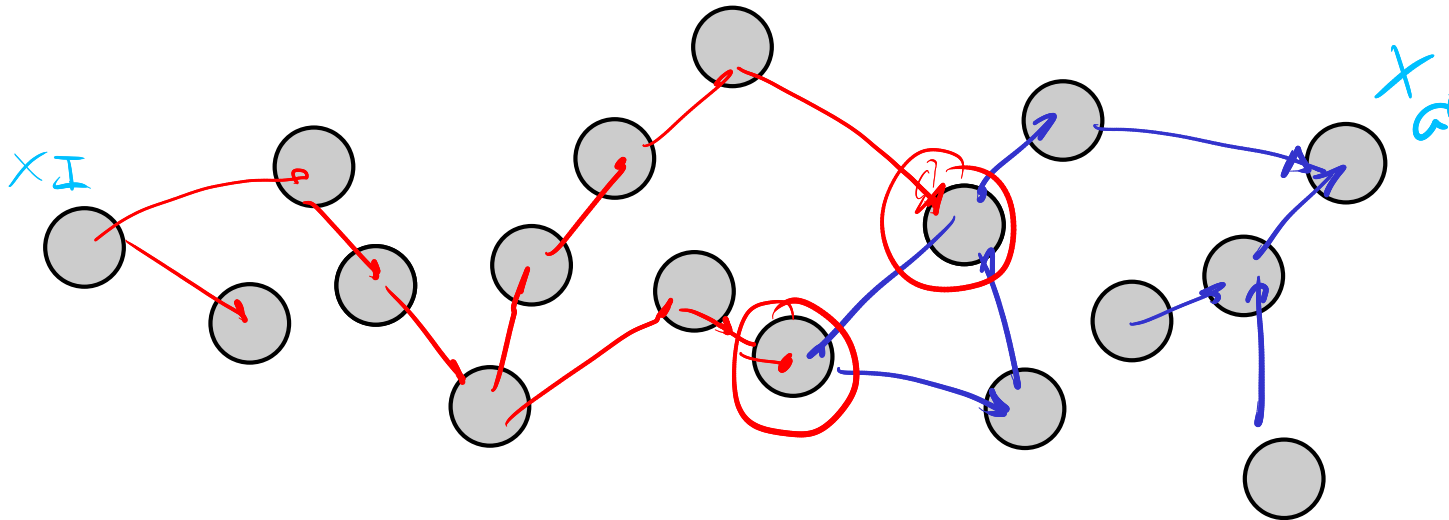
- Some systems may be more convenient to work on reverse mode.

Bidirectional search

Both backward and forward search can be combined, for instance alternating both searches.

The search terminates when both trees meet.

Example:



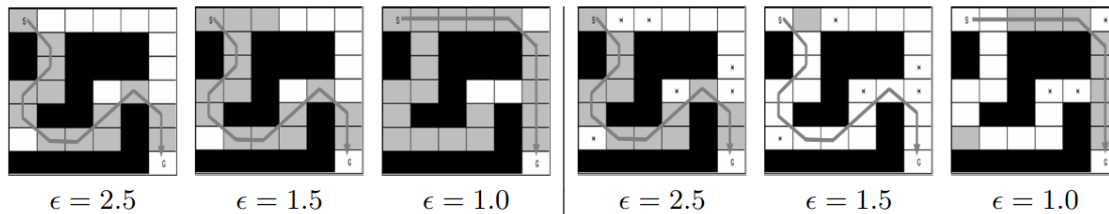
Anytime algorithms

Some algorithms admit an incremental solution, that could be refined over time.

The idea is to find a solution fast, and spend the remaining computational budget to improve it.





This way, one can obtain a solution anytime, and the quality will depend on the computational time spent.

For instance, the ARA* [1] searches with decreasing weight $C(x') + \epsilon \cdot h(x', x_G)$ which are provably guaranteed to find the optimal solution.

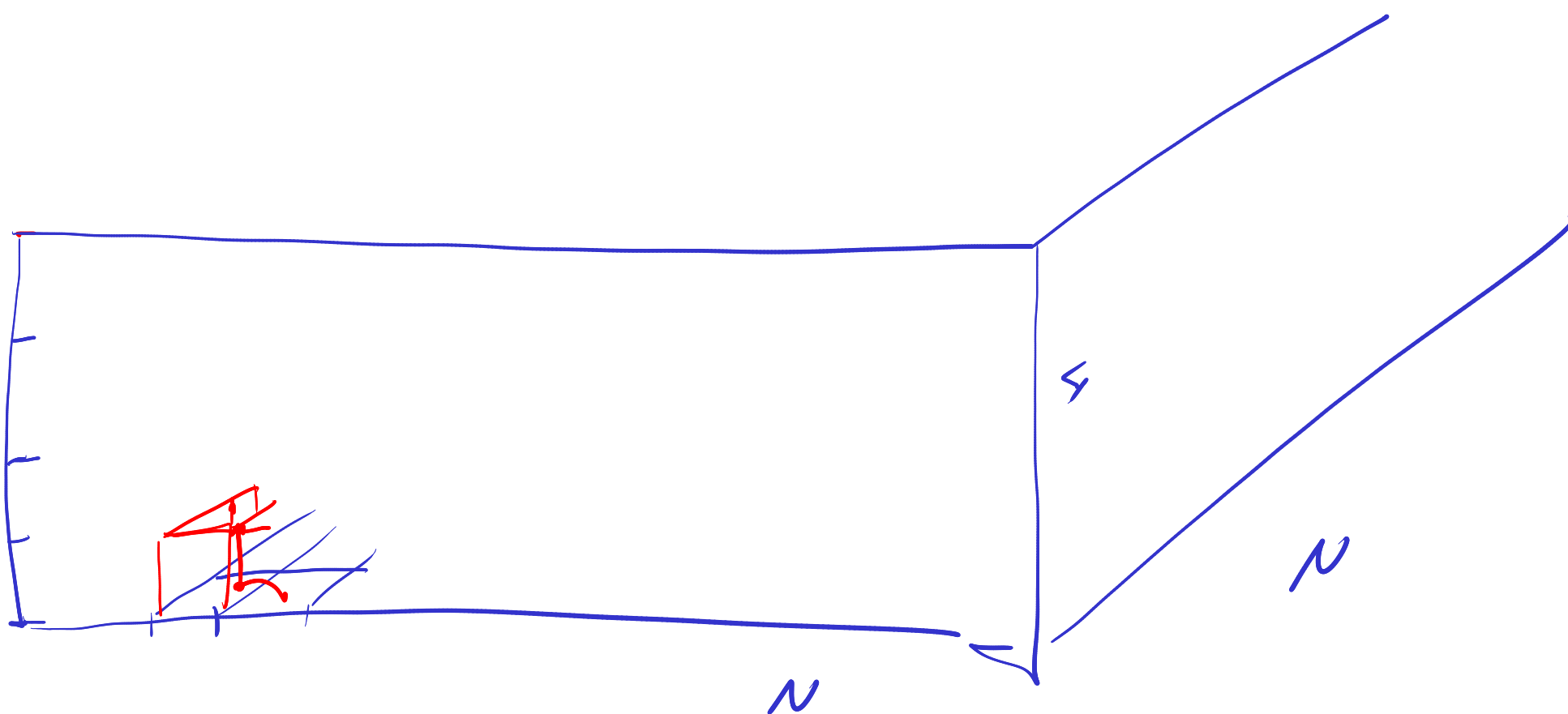


[1] Likhachev, M., Gordon, G. J., & Thrun, S. (2003). ARA*: Anytime A* with provable bounds on sub-optimality. Advances in neural information processing systems, 16, 767-774.

About PS1

$\theta = 0$ 
45 
90 
135 

Conf. space.



$N \times N \times 4$