L02: Discrete Planning

Planning Algorithms in AI and Robotics

Prof. Gonzalo Ferrer,

Skoltech, 31 October 2025

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?

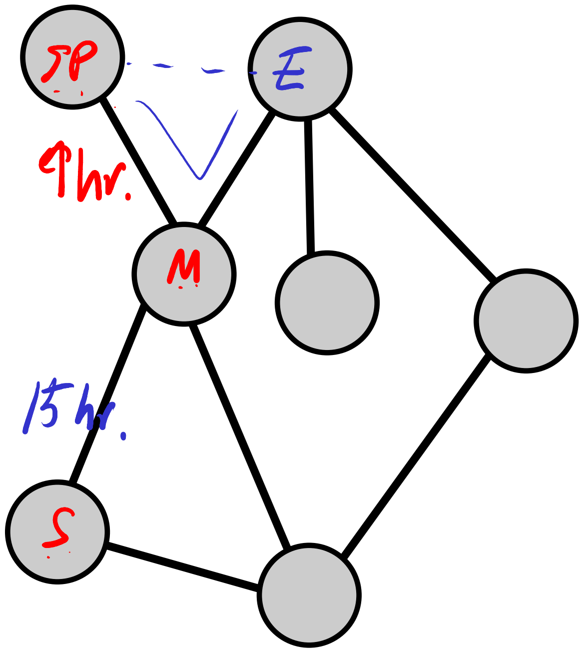
  1) There are many examples of discrete problems.

  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.

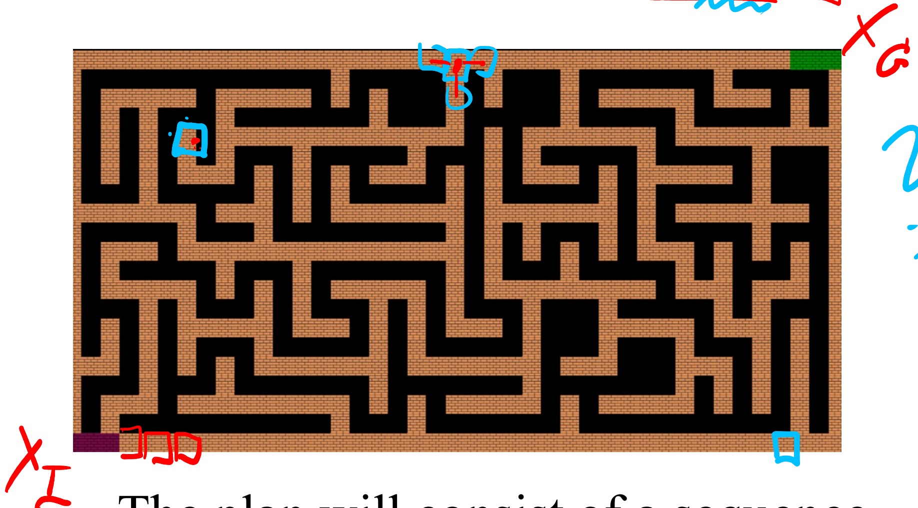
Examples of discrete planning problems

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. A set of cells with values {free, obstacle}. $X = \{x_i\}_N$



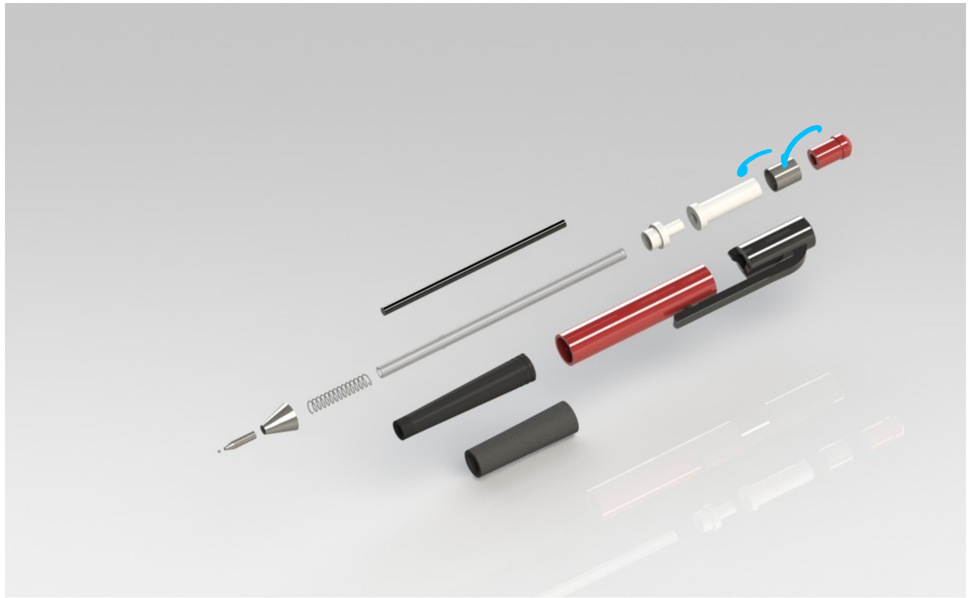
$U(x_j) = \{a_1, a_2, a_3\}$

The plan will consist of a sequence of connected cells in order to escape.

The **state transition graph** is revealed as <u>we plan</u>.

Examples of discrete planning problems

Logic-based planning

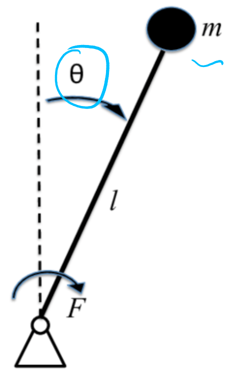


How to assemble the pen? The state space is discrete, but the size of it is gigantic. Combinatorial problem.

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

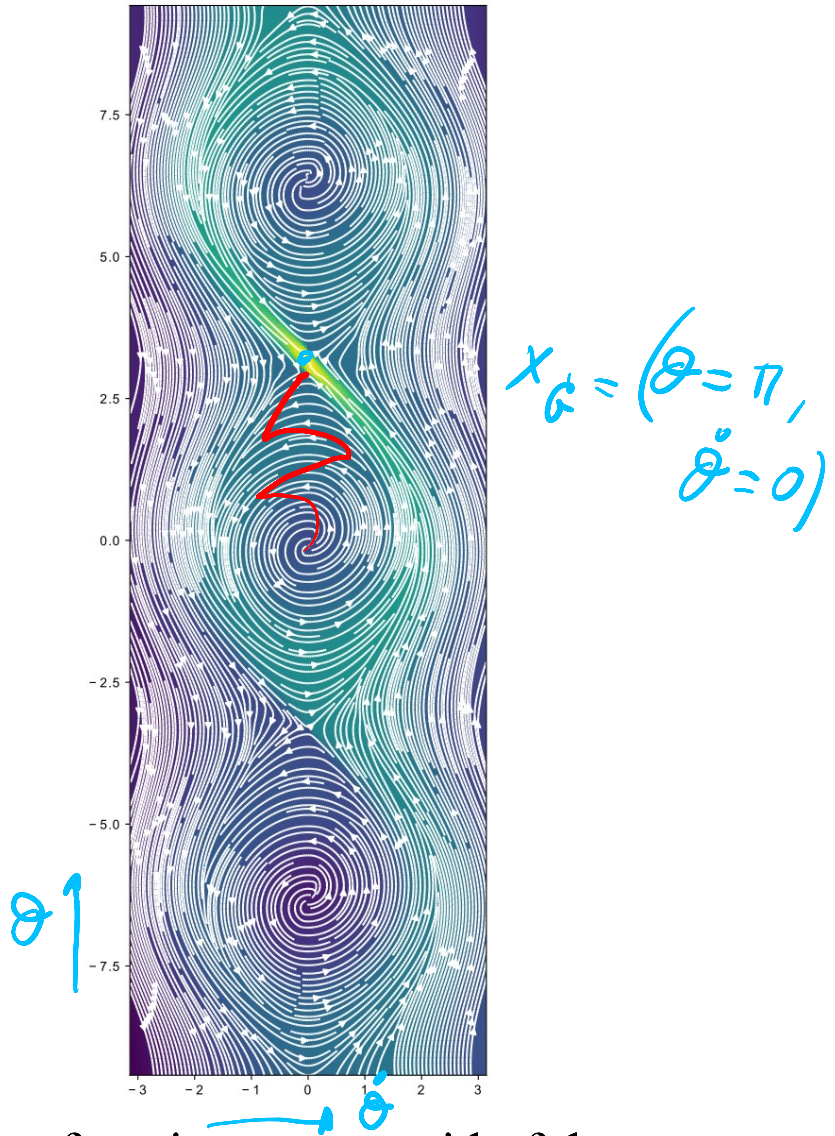
Examples: discretization of dynamics

Inverted pendulum



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

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



Value function over a grid of the discretized state of this system. X-axis angular velocity and Y-axis - angle.

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.

AI Boyko, IV Oseledets, G Ferrer. “TT-QI: Faster Value Iteration in Tensor Train Format for Stochastic Optimal Control.” CMMP. 2021

Problem formulation

1. A non-empty **state space** X of finite elements (or countably infinite*)
2. For each state $x \in X$, a finite **action space** $U(x)$
3. A **state transition function** *f( )* that produces a state:

$$x' = f(x, u)$$

4. An **initial state** $x_I \in X$
5. A **goal set** $X_G \subset X$

If these conditions are met, then the discrete planning algorithm is **complete**

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

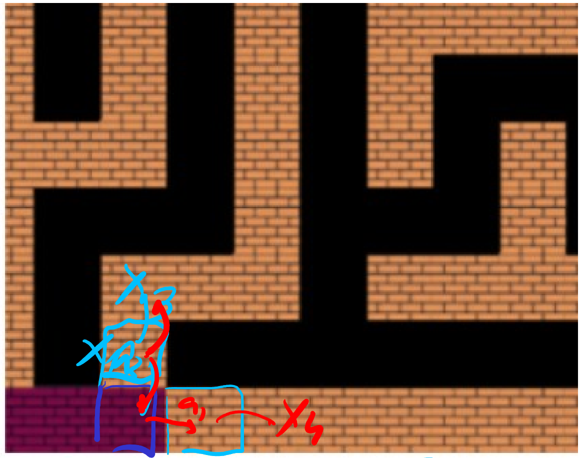
- Executing actions creates a state transition graph or a directed **graph** (tree) as we search over it.

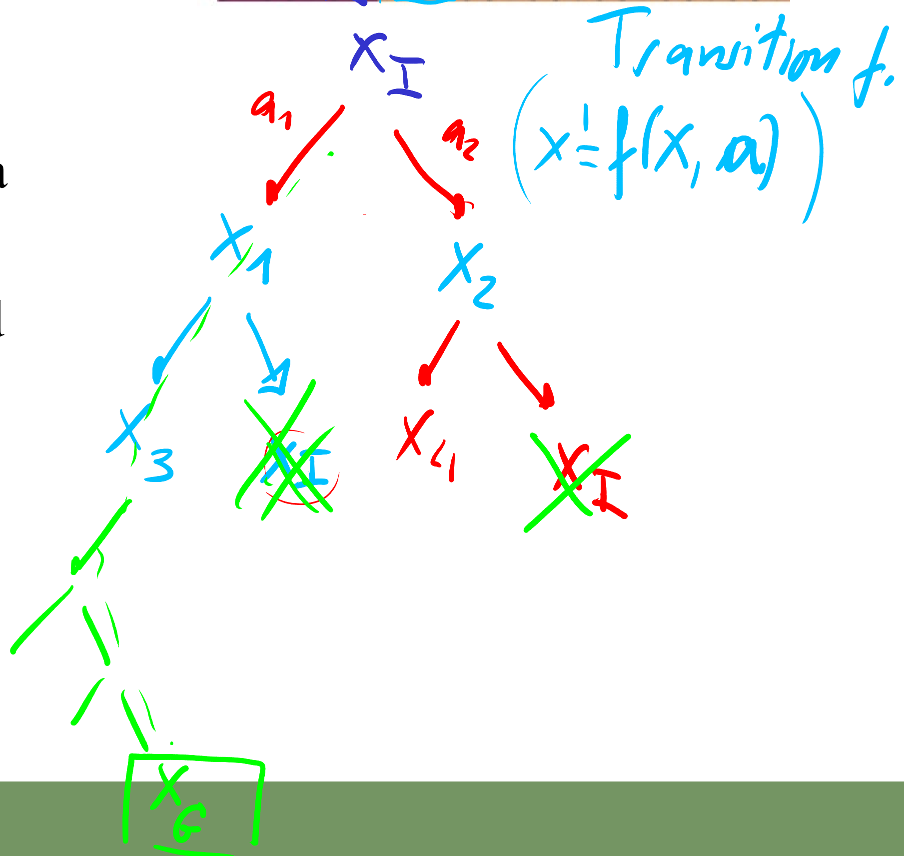
  If all information was known from the beginning, it would be standard graph search.

  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.





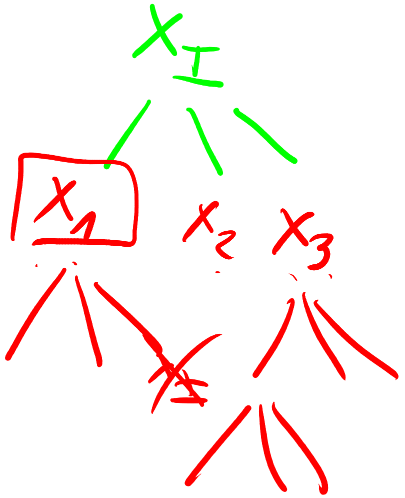
General Forward Search Algorithm

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 ← Q.get_first()                  1
5        if x in X_G:
6            return SUCCESS
7        for u in U(x):
8            x' ← f(x,u)
9            if x' not visited:
10               Mark x' as visited
11               Q.insert(x')              2
12           else:
13               resolve duplicate x'
14   return FAILURE
```



$Q = \{x_I\}$

1 $Q = \{\}$

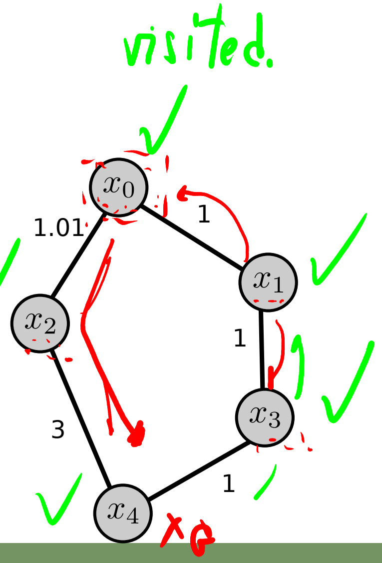
2 $Q = \{x_1, x_2, x_3\}$

- 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 Search (BFS)

```
Breadth First Search:
   Inputs: x_I, X_G.
   Output: {SUCCESS,FAILURE}, plan
      Q = {} First In First Out (FIFO)
      Q.insert(x_I) and mark x_I as visited
      while Q not empty:
            x ← Q.get_first()
            if x in X_G:
                  return SUCCESS, plan()
            for u in U(x):
                  x' ← f(x,u)
                  if x' not visited:
                        Mark x' as visited
                        Parent_table(x') = x
                        Q.insert(x')
                  else:
                        resolve duplicate x'
      return FAILURE
```

(do nothing)



$i=0 \quad Q=\{x_I\}$

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

$i=2, \quad x=x_1, \quad f(U(x_1), x_1)=\{x_0, x_3\}$

$Q=\{x_2, x_3\}$

$i=3, \quad x=x_2, \quad Q=\{x_3, x_4\}$ (x_G)

$i=4, \quad x=x_3, \quad Q=\{x_4\}$

$i=5, \quad x=x_4 \rightarrow$ success!

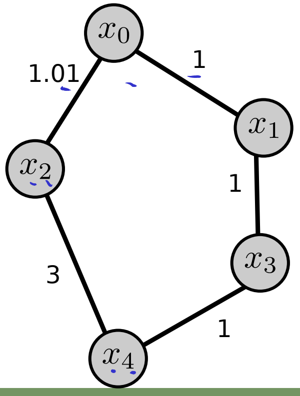
Depth First Search (DFS)

Depth First Search:

Inputs: x_I, X_G.

Output: {SUCCESS,FAILURE}, plan

```
Q = {} Last In First Out (LIFO)
Q.insert(x_I) and mark x_I as visited
while Q not empty:
    x ← Q.get_first()
    if x in X_G:
        return SUCCESS, plan()
    for u in U(x):
        x' ← f(x,u)
        if x' not visited:
            Mark x' as visited
            Parent_table(x') = x
            Q.insert(x')
        else:
            resolve duplicate x'
return FAILURE
```



$i=0$, $Q=\{x_I\}$

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

$i=2$, $x=x_2$, $Q=\{x_1, x_4\}$

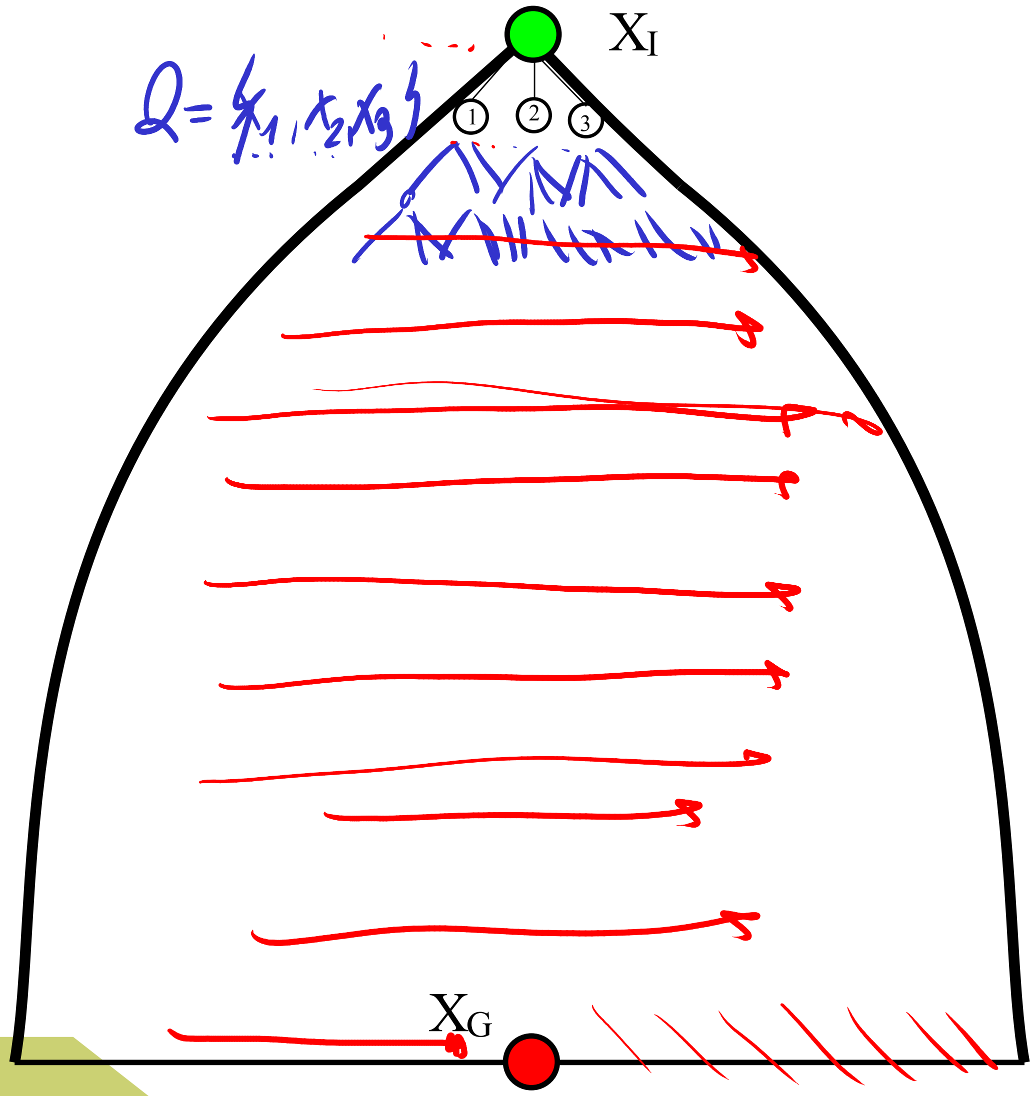
$i=3$, $x=x_4$ SUCC.

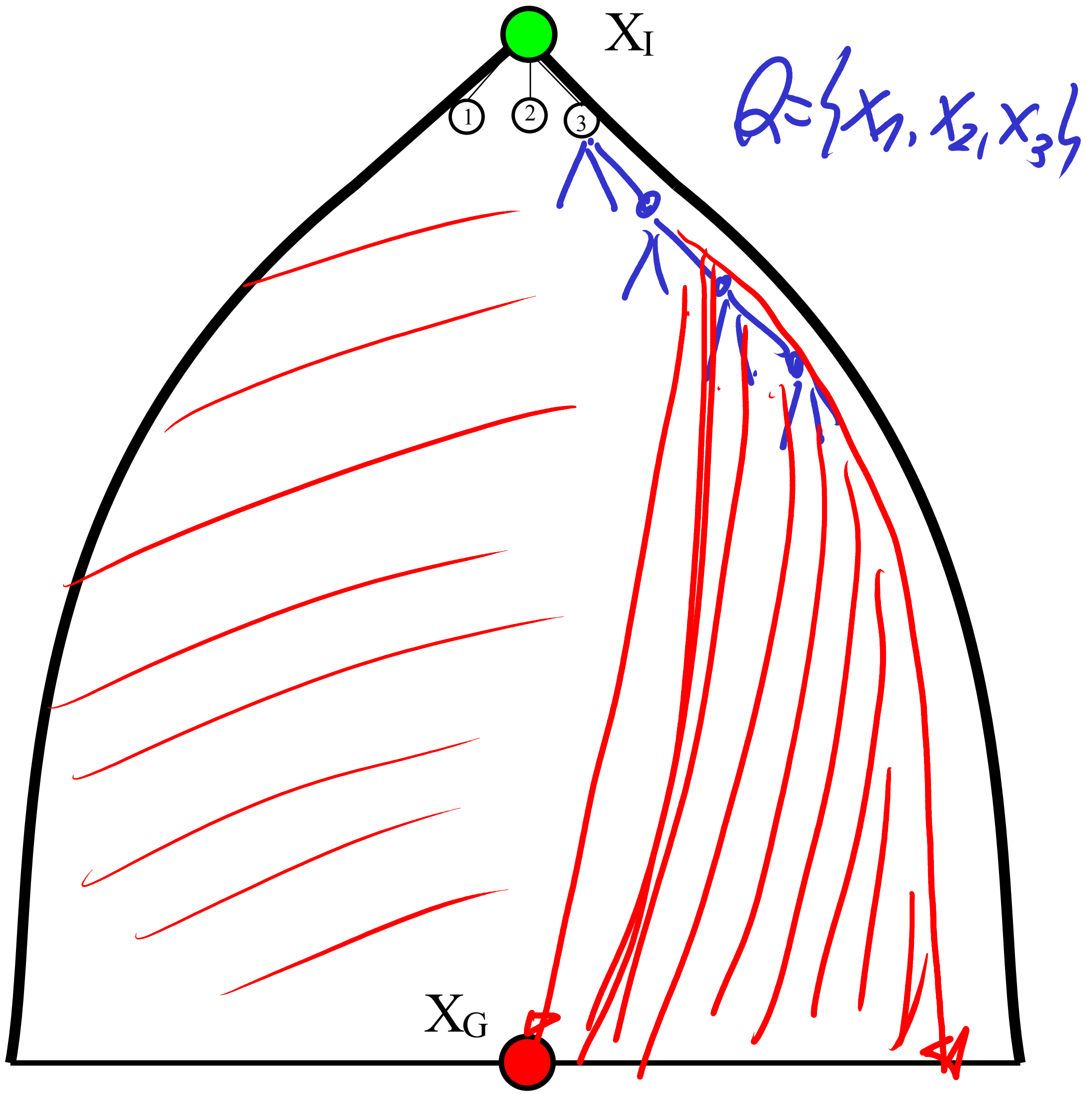
parent	child
x_0, x_I	x_1
$\boxed{x_0, x_I}$	x_2
x_2	$\boxed{x_4\ x_G}$ ✓

$$\text{reverse}(x_4, x_2, x_0) =$$

$$= \boxed{x_0, x_2, x_4}$$

Breadth First (FIFO) vs Depth First (LIFO)





Some remarks about BFS and DFS

- We are studying a particular kind of graphs: state transtion graph and we are making sure that search is systematic (keep track of states). For this, BFS and DFS are complete algorithms.

- In general, for any kind of graph, BFS is guaranteed to be complete, by expanding systematically all nodes until a solution is found. (Perhaps requiring more nodes to be visited)
- In general, DFS is not considered complete since it can explore an infinite path due to cycles or loops. For our problems, this will not be the case, but it is important to take into account.

Can we do better? In the sense of reducing the search and the plan?

- Greedy Best-First Search (GBFS). Some heuristic is used to choose next node. Incomplete Alg.
- **Dijkstra** Algorithm: search by minimizing some cost en each edge. Complete algorithm.

Dijkstra's algorithm

Dijstra:

Inputs: x_I, X_G.

Output: {SUCCESS,FAILURE}, plan

```
Q = {} Priority Queue
Q.insert(x_I) and mark x_I as visited
C(x_I)= 0
while Q not empty:
    x ← Q.get_first()
    if x in X_G:
        return SUCCESS, plan()
    for u in U(x):
        x' ← f(x,u)
        if x' not visited:
            Mark x' as visited
            parent_table(x') = x
            C(x') = C(x) + l(x,u)
            Q.insert(x', C(x'))
        else:
            if C(x) + l(x,u) < C(x'):
                C(x') = C(x) + l(x,u)
                parent_table(x') = x
return FAILURE
```

$l \geq 0$

$i=0$, $Q = \{(x_I, C(x_I)=0)\}$

$i=1$, $Q = \{(x_1, \underbrace{C(x_I)}_{0} + \underbrace{l(x_I, u_{x_1})}_{1}), (x_2, 1.01)\}$

$i=2$, $x = x_1$ $(C(x_1) \leq C(x_i), i=2)$

$(x_3, \underbrace{C(x_1)}_{1} + 1 = 2)$

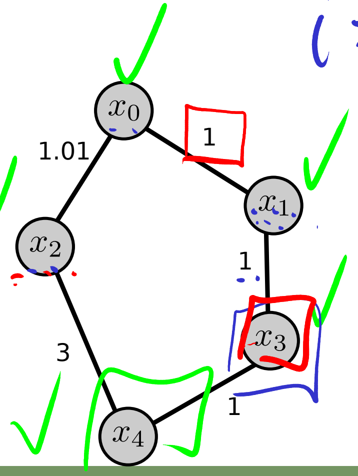
$Q = \{(x_2, 1.01), (x_3, 2)\}$

$i=3$, $x = x_2$, $Q = \{(x_3, 2), (x_4, 4.01)\}$

$i=4$, $x = x_3$

$C(x_4') = C(x_3) + l(x_3, u_{x_3}) = 3$

$Q = \{(x_4, 3), (x_6, 4)\}$



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

A*

$$h(x_1, x_2) = L_1(x_1, x_2) = |x_{1x} - x_{2x}| + |x_{1y} - x_{2y}|$$


```
A-star:
   Inputs: x_I, X_G.
   Output: {SUCCESS,FAILURE}, plan
      Q = {} Priority Queue
      Q.insert(x_I) and mark x_I as visited
      C(x_I)= 0
      while Q not empty:
         x ← Q.get_first()
         if x in X_G:
               return SUCCESS, plan()
         for u in U(x):
               x' ← f(x,u)
               if x' not visited:
                  Mark x' as visited
                  parent_table(x') = x
                  C(x') =C(x) + l(x,u)
                  Q.insert(x', C(x')+h(x',x_G))
               else:
                  if C(x) + l(x,u) < C(x'):
                     C(x') =C(x) + l(x,u)
                     parent_table(x') = x
      return FAILURE
```

- 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

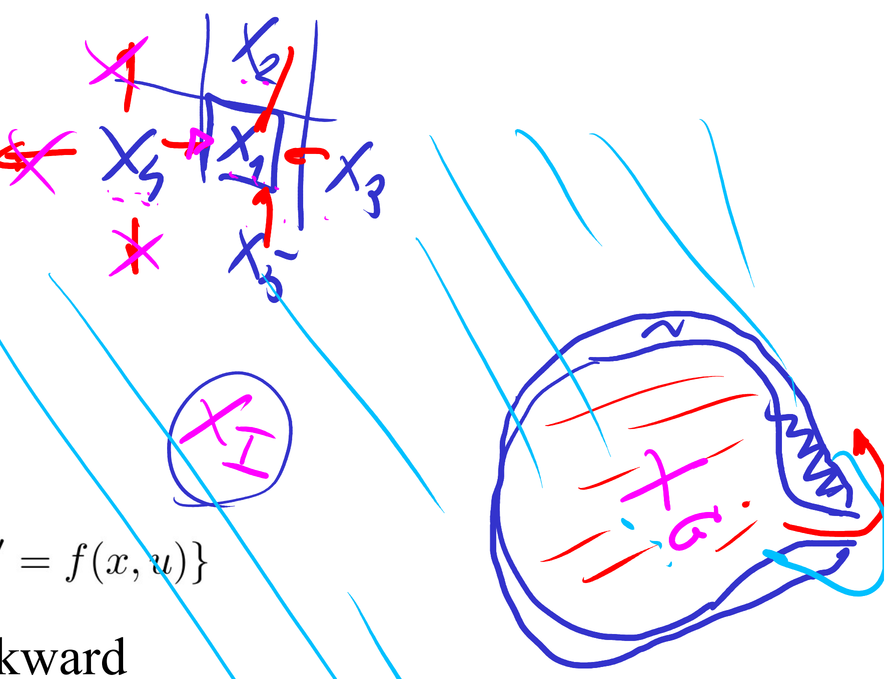
- A correct heuristic function can speed up the graph search dramatically.
- The problem is when designing heuristic functions for non-trivial state spaces.
- The effect on heuristics functions can alter the solution, obtaining a non-optimal path.
- We will retake the discussion on the relation to the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

Backward Search

$f(x,u)$ $U(x)$ x

Backward search:
  Inputs: x_I, x_G.
  Output: {SUCCESS,FAILURE}

```
    Q = {} common queue
    Q.insert(x_G) and mark x_G as visited
    while Q not empty:
        x' ← Q.get_first()
        if x' in x_I:
            return SUCCESS
        for u^-1 in U^-1(x):
            x ← f^-1(x',u^-1)
            if x' not visited:
                Mark x as visited
                Q.insert(x)
            else:
                resolve duplicate x
    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:

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

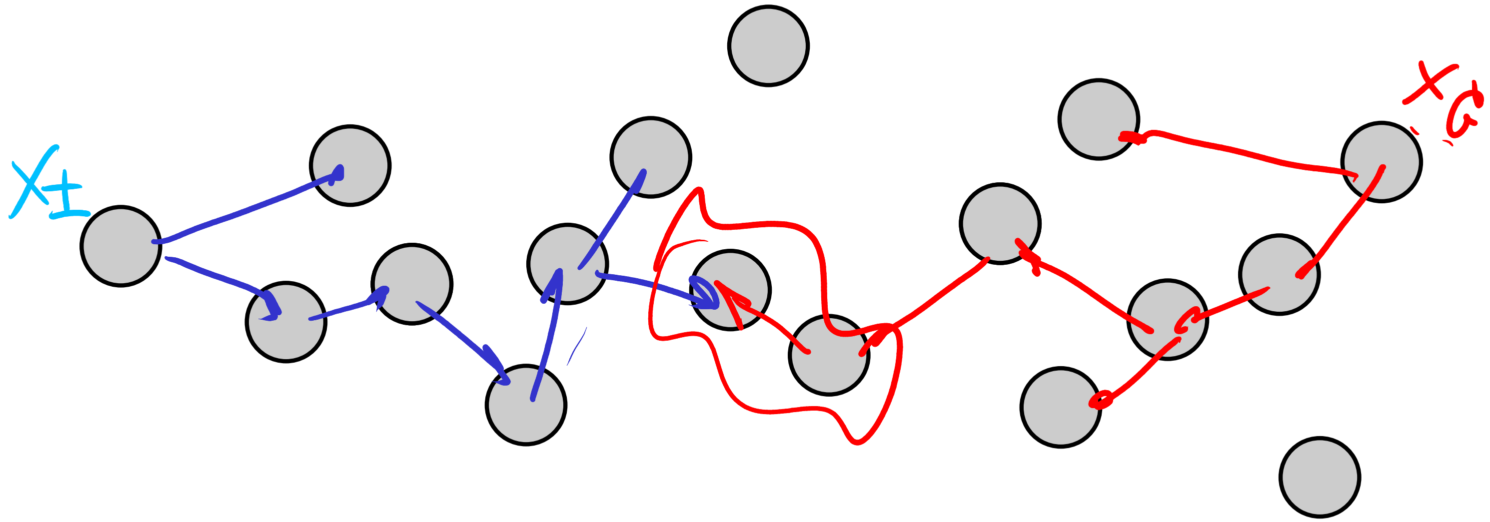
- Some systems may be more convenient to work on reverse mode.

Bidirectional search

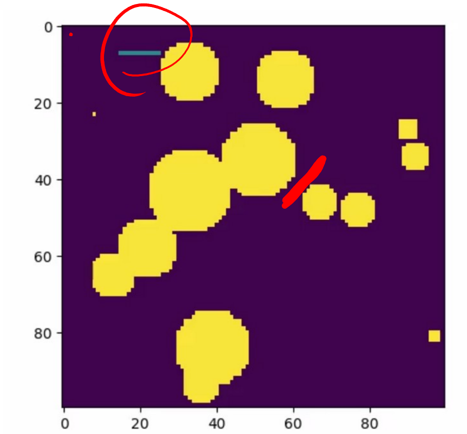
Both backward and forward search can be combined, for instance alternating both searches.

The search terminates when both trees meet.

Example:



About PS1



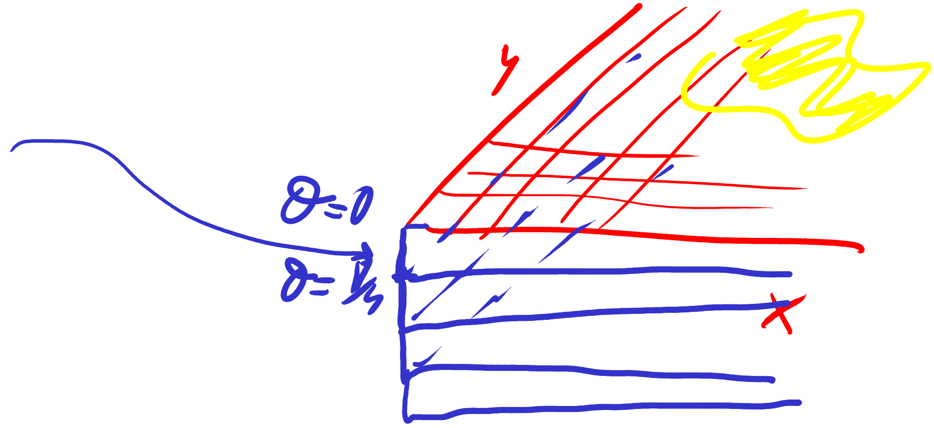
workspace.

$x(\theta=0)$

$x(\theta=\pi/4)$

$x(\theta=\pi/2)$

$x(\theta=3\pi/4)$



configuration sp.

L3

Selected Readings

Dynamic and Incremental search:

[1] Stenz. A. Optimal and Efficient Path Planning for Partially-Known Environments. ICRA 1994 D*

[2] Koenig, S., & Likhachev, M. The D Lite Algorithm for Real-Time Path Planning. ICRA 2002

[3] 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.

[4] S Koenig, M Likhachev, D FurcyLifelong planning A*. Artificial Intelligence 2004

[5] E Groshev, M Goldstein, A Tamar, S Srivastava, P Abbeel. Learning Generalized Reactive Policies Using Deep Neural Networks. AAAI 2018