Introduction to MAPF

Ali Czech

Multi-Agent Pathfinding

- Goal: Assign collision-free paths on graphs to each agent.
- The fundamental building block of multi-robot coordination
- Real-world Applications
  - Fleet operations in automated warehouses
  - Requires solving instances with thousands of agents in real-time
  - Examples
    - Brightpick - real world execution
    - Not perfect - failure case: collision/livelock

MAPF Formulation

- MAPF instance
  - Graph $G = (V,E)$
  - Agents $A = \{1, . . . , n\}$
  - Starts $S_{start} = (s_1, . . . , s_n)$
  - Goals $S_{goal} = (g_1, . . . , g_n)$
  - $s_i \neq s_j$ and $g_i \neq g_j$ for all $i \neq j$; $s_i, g_i \in V$

- Configuration
  - A tuple of locations for all agents at a given timestep.

- Given MAPF instance
  - Location of an agent i at discrete time t: $\pi_i[t] \in V$
  - Path: Sequence of locations of an agent

- Valid moves
  - At each discrete timestep t, an agent moves to an adjacent vertex or stays

$$\pi_i[t+1] \in neigh(\pi_i[t]) \cup \{\pi_i[t]\}$$

- Collisions to avoid (for all $i \neq j$)
  - Vertex conflict: $\pi_i[t] = \pi_j[t]$
  - Swap conflict: $\pi_i[t] = \pi_j[t+1] \land \pi_i[t+1] = \pi_j[t]$

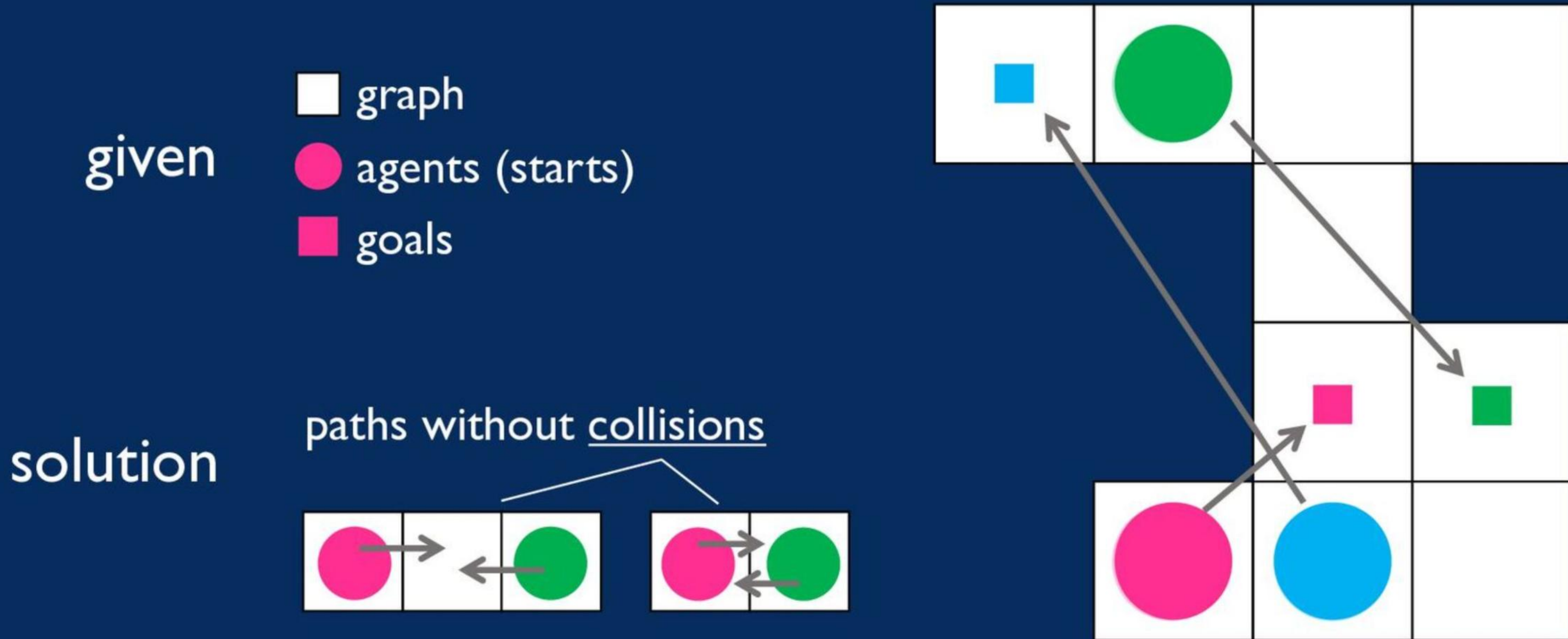
- Objective
  - T_i is the earliest timestep when agent i reaches its goal g_i and stays there

$$\pi_i[T_i] = \pi_i[T_i+1] = \ldots = g_i$$

  - Find a collision-free solution minimizing the Sum-of-Costs (SOC):

$$\Sigma_\{i \in A\}\ T_i$$

MAPF: Multi-Agent Path Finding



optimization is intractable in various criteria

[Yu+ AAAI-13, Ma+ AAAI-16, Banfi+ RA-L-17, Geft+ AAMAS-22]

The Complexity Bottleneck

- State Space Explosion
  - The total number of possible configurations is $O(|V|^{|A|})$.
- Massive Branching Factor
  - A standard search algorithm (e.g., A*) generates $O(\Delta^{|A|})$ configurations from a single node, where Δ is the maximum degree of the graph.
- Intractability
  - Solving MAPF optimally is NP-hard in various criteria (Yu and LaValle 2013)
- Finding optimal paths for hundreds of agents in real-time is computationally intractable. We have no choice but to rely on sub-optimal algorithms for large instances.

complete
optimal

quick
scalable

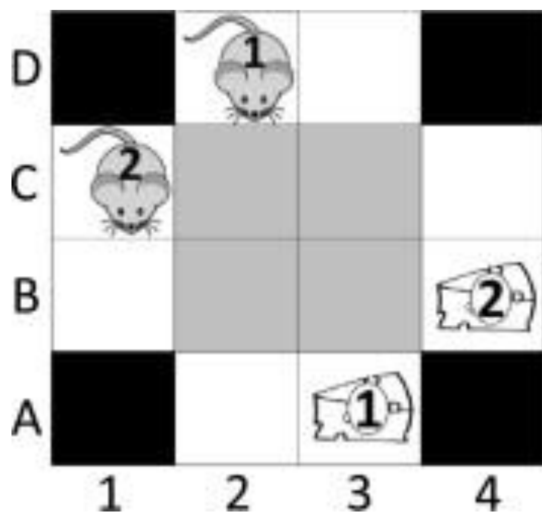
Search-Based Solvers: Conflict-Based Search (CBS)

- Two-level search architecture
- Low-level search
  - Single-agent pathfinding
  - Finds paths consistent with a given set of constraints.
- High-level search
  - Builds a Conflict Tree (CT) to resolve collisions
  - Core mechanism: Uses *negative constraints*
    -> Agent i cannot be at vertex v at time t: (a_i, v, t)
- Properties: Complete and optimal, but the constraint tree branches exponentially in dense scenarios

Optimal Path Routes

1: D2>D3>C3>B3>A3
D2>C2>C3>B3>A3
D2>C2>B2>B3>A3
D2>C2>B2>A2>A3

2: C1>B1>B2>B3>B4
C1>C2>B2>B3>B4
C1>C2>C3>B3>B4
C1>C2>C3>C4>B4



Con: {} | **Cost**: 8
Sol
1- D2,C2,B2,A2,A3
2- C1,B1,B2,B3,B4

Con: {1,B2,2} | **Cost**: 8
Sol
1- D2,D3,C3,B3,A3
2- C1,B1,B2,B3,B4

Con: {2,B2,2} | **Cost**: 8
Sol
1- D2,C2,B2,A2,A3
2- C1,C2,C3,C4,B4

Con: {1,B2,2} {1,B3,3} | **Cost**: 9
Sol
1- D2,C2,C2,B2,A2,A3
2- C1,B1,B2,B3,B4
GOAL

Con: {1,B2,2} {2,B3,3} | **Cost**: 8
Sol
1- D2,D3,C3,B3,A3
2- C1,C2,C3,C4,B4

Con: {2,B2,2} {1,C2,1} | **Cost**: 8
Sol
1- D2,D3,C3,B3,A3
2- C1,C2,C3,C4,B4

Con: {2,B2,2} {2,C2,1} | **Cost**: 9
Sol
1- D2,C2,B2,A2,A3
2- C1,C1,C2,C3,C4,B4
GOAL

Con: {1,B2,2} {2,B3,3} {1,C3,2} | **Cost**: 9
Sol
1- D2,D3,D3,C3,B3,A3
2- C1,C2,C3,C4,B4
GOAL

Con: {1,B2,2} {2,B3,3} {2,C3,2} | **Cost**: 9
Sol
1- D2,D3,C3,B3,A3
2- C1,C2,C2,C3,C4,B4
GOAL

Con: {2,B2,2} {1,C2,1} {1,C3,2} | **Cost**: 9
Sol
1- D2,D3,D3,C3,B3,A3
2- C1,C2,C3,C4,B4
GOAL

Con: {2,B2,2} {1,C2,1} {2,C3,2} | **Cost**: 9
Sol
1- D2,D3,C3,B3,A3
2- C1,C1,C2,C3,C4,B4
GOAL

CBS vs. A* Performance Factors

- Narrow Corridors / Bottlenecks
  - CBS wins - few conflicts, constraints are powerful
- Open Grids / High Density
  - A* can win – high conflict count leads to CT explosion
- CMS - O($\textit{branching_factor}^{\textit{number of conflicts}}$)
- A* - O($\textit{branching_factor}^{\textit{number of agents}}$)

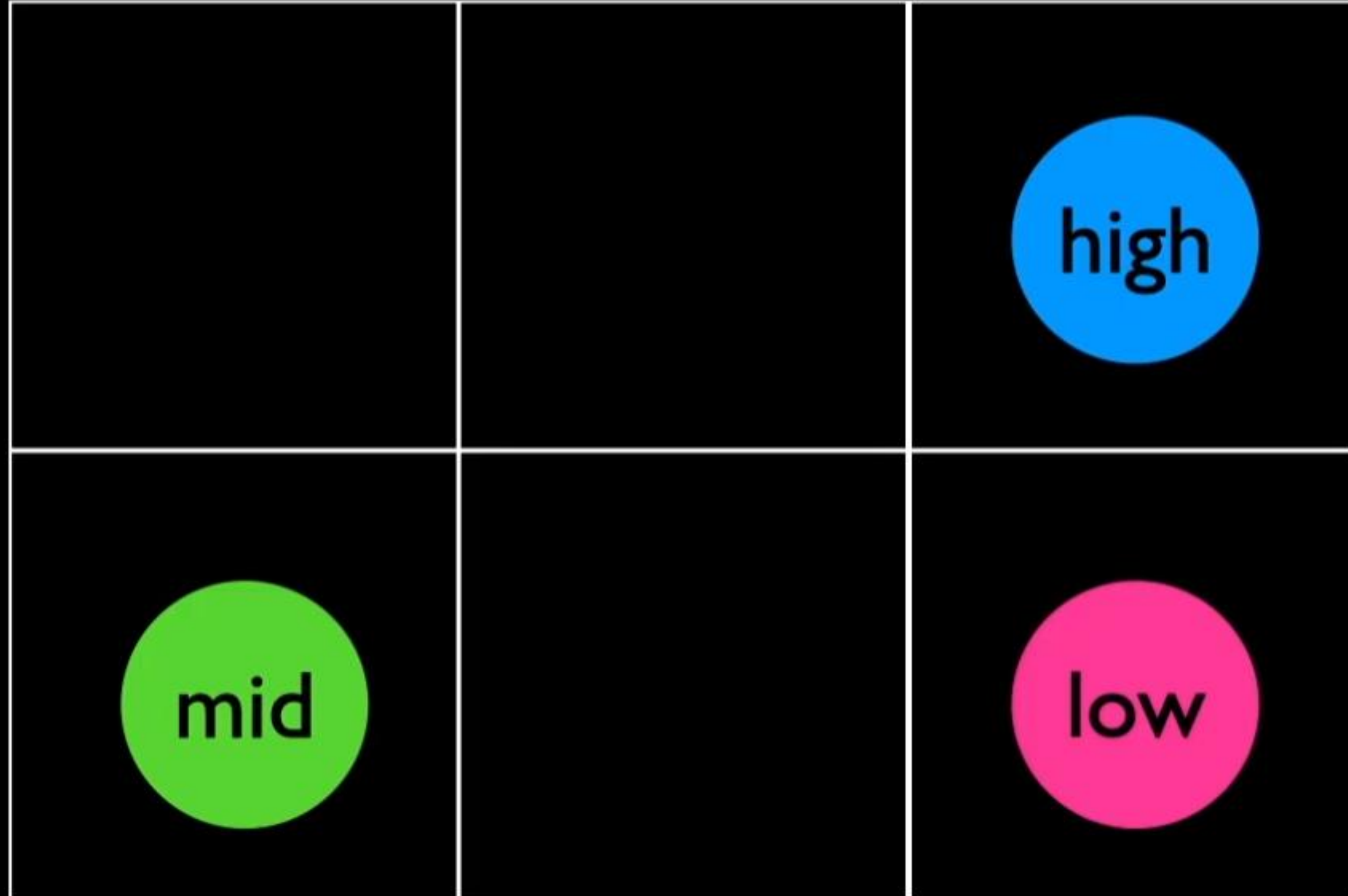
Algorithm 1: high-level of CBS

```
   Input: MAPF instance
1  R.constraints = ∅
2  R.solution = find individual paths using the
   low-level()
3  R.cost = SIC(R.solution)
4  insert R to OPEN
5  while OPEN not empty do
6      P ← best node from OPEN // lowest solution cost
7      Validate the paths in P until a conflict occurs.
8      if P has no conflict then
9          return P.solution // P is goal
10     C ← first conflict (a_i, a_j, v, t) in P
11     foreach agent a_i in C do
12         A ← new node
13         A.constriants ← P.constriants + (a_i, s, t)
14         A.solution ← P.solution.
15         Update A.solution by invoking low-level(a_i)
16         A.cost = SIC(A.solution)
17         Insert A to OPEN
```

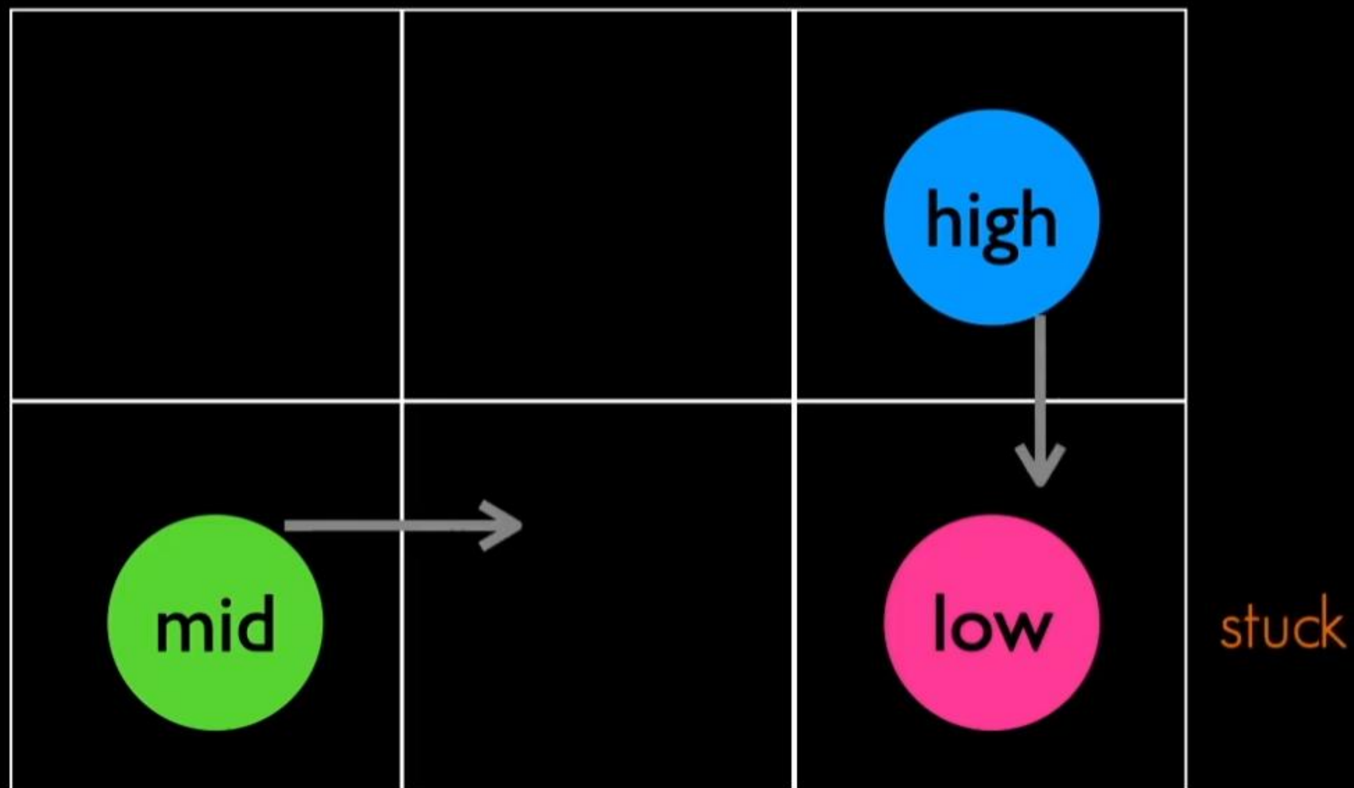
PIBT

- Priority Inheritance with Backtracking
- A rule-based, prioritized algorithm designed for massive scale
- How it works
  - Assigns a priority to each agent (often based on distance to goal)
  - In each timestep, agents pick their best move in order of priority
  - If a higher-priority agent is blocked, it "inherits" its priority to the blocking agent to push them out of the way
- Pros: Extremely fast: $O(|A| * \Delta)$, can handle 10000+ agents
- Cons: Incomplete (can get stuck in livelocks) and sub-optimal

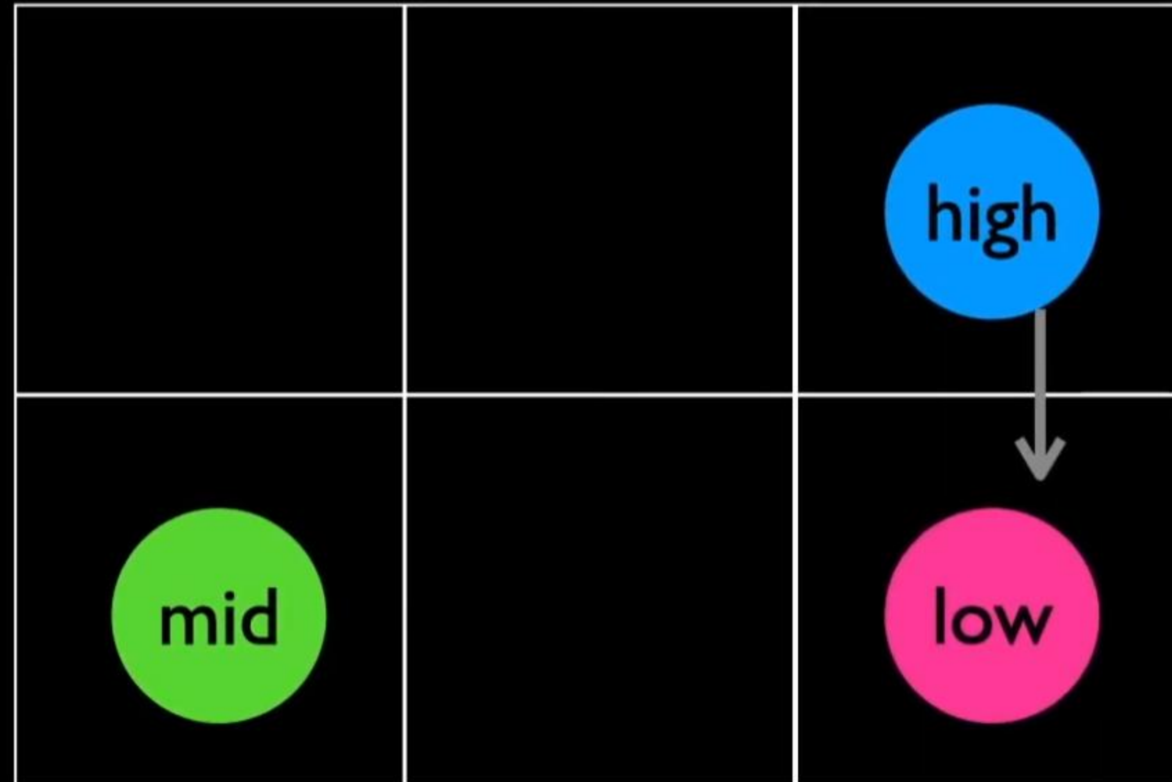
simple one-timestep prioritized planning is efficient but incomplete

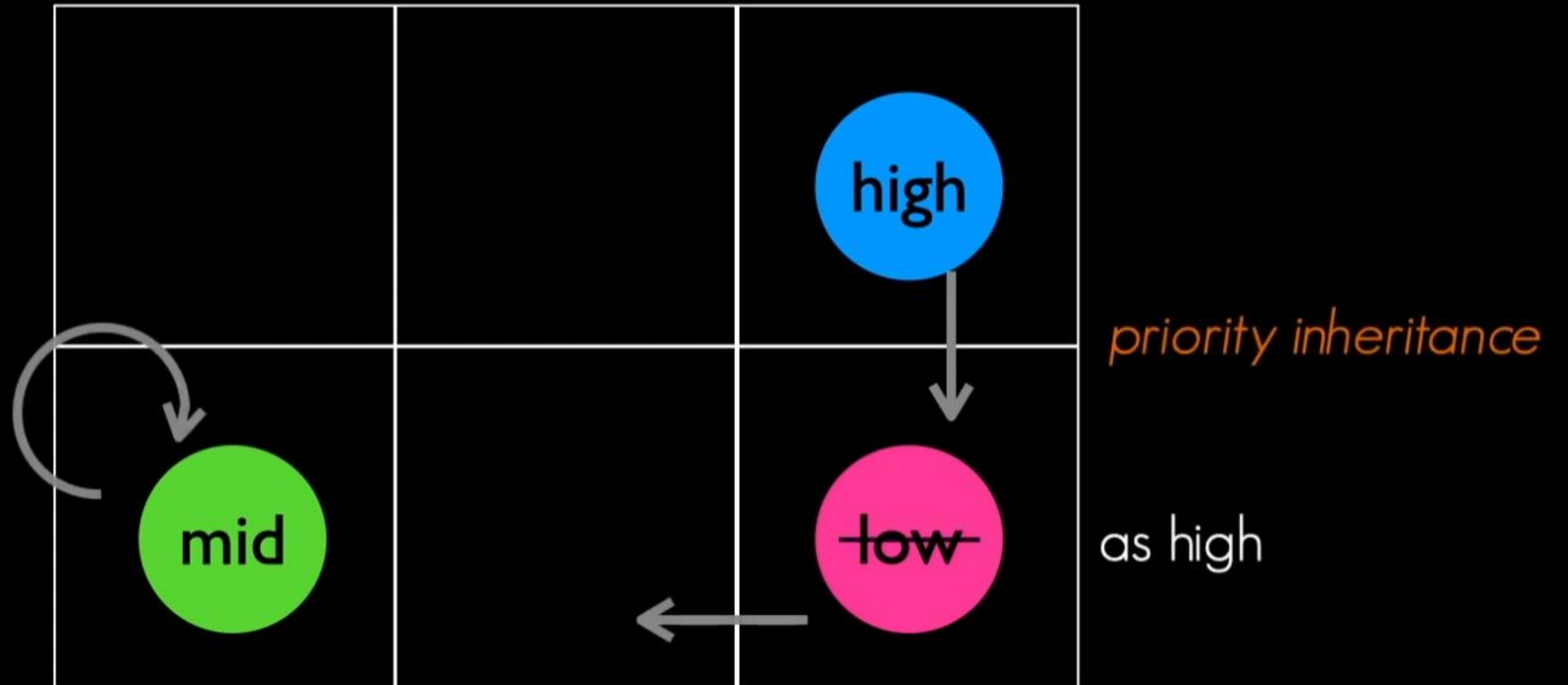


simple one-timestep prioritized planning is efficient but incomplete

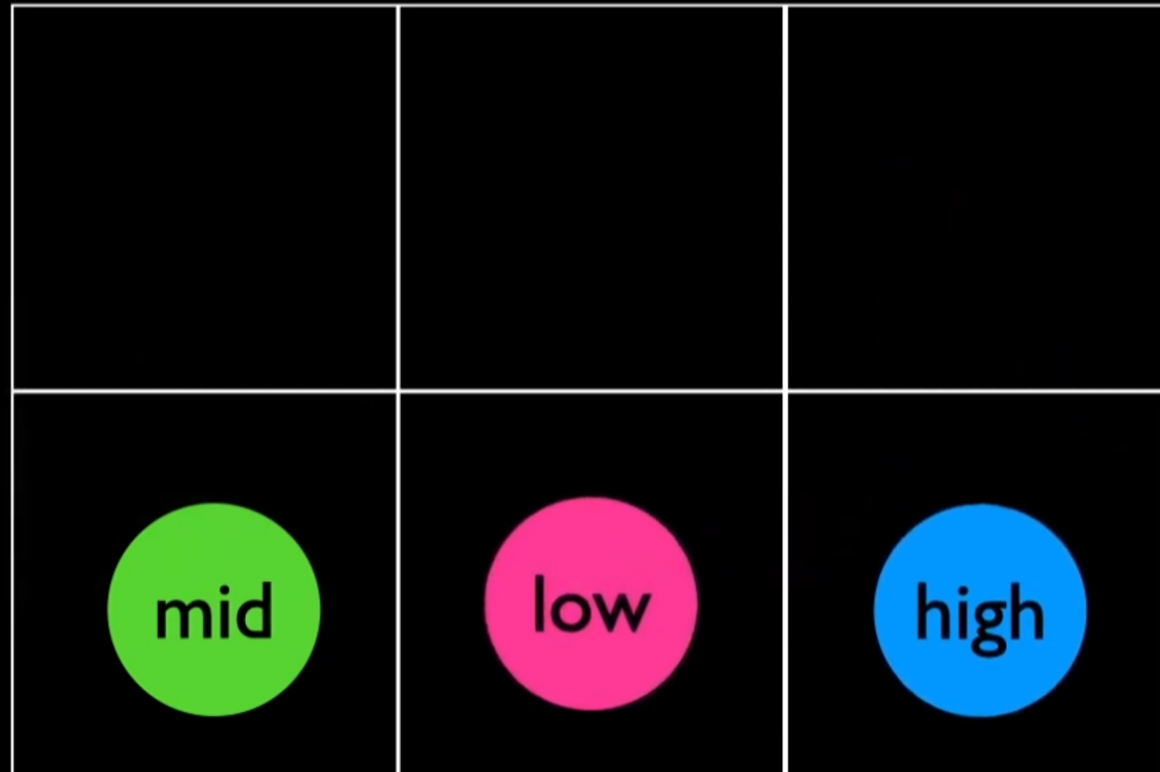


priority inheritance can deal with this situation

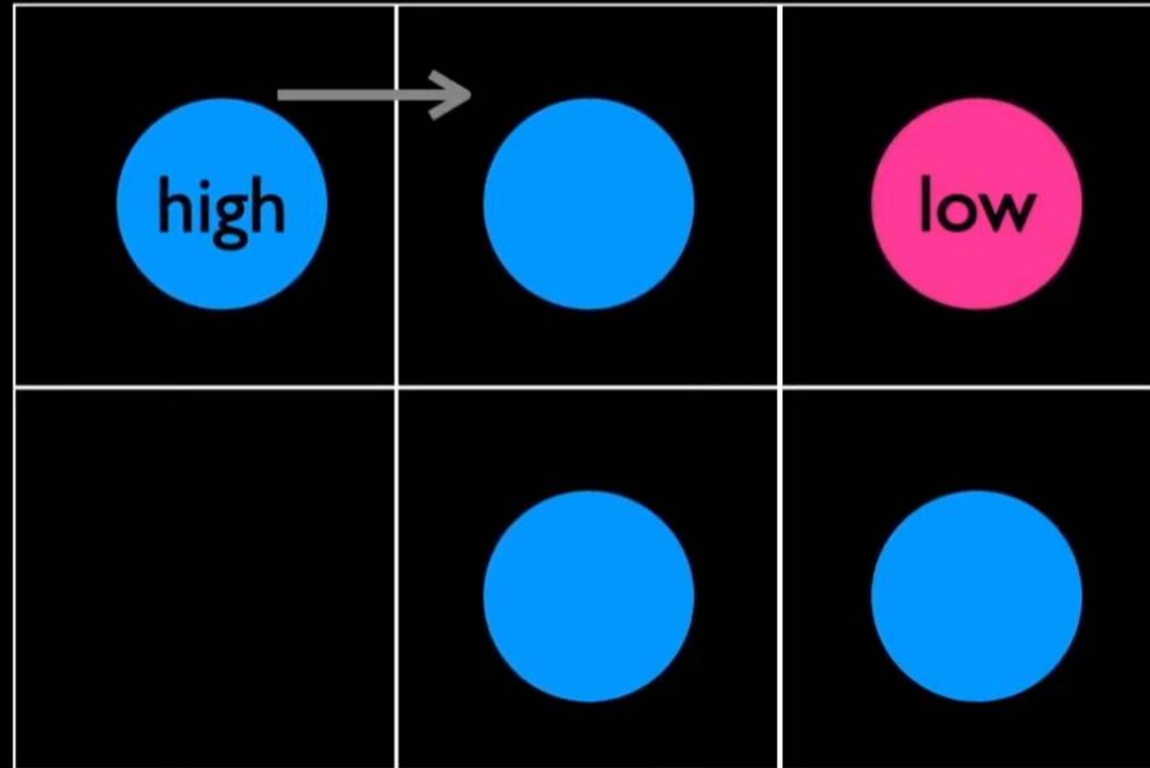



priority inheritance can deal with this situation
high
priority inheritance
mid
low
as high

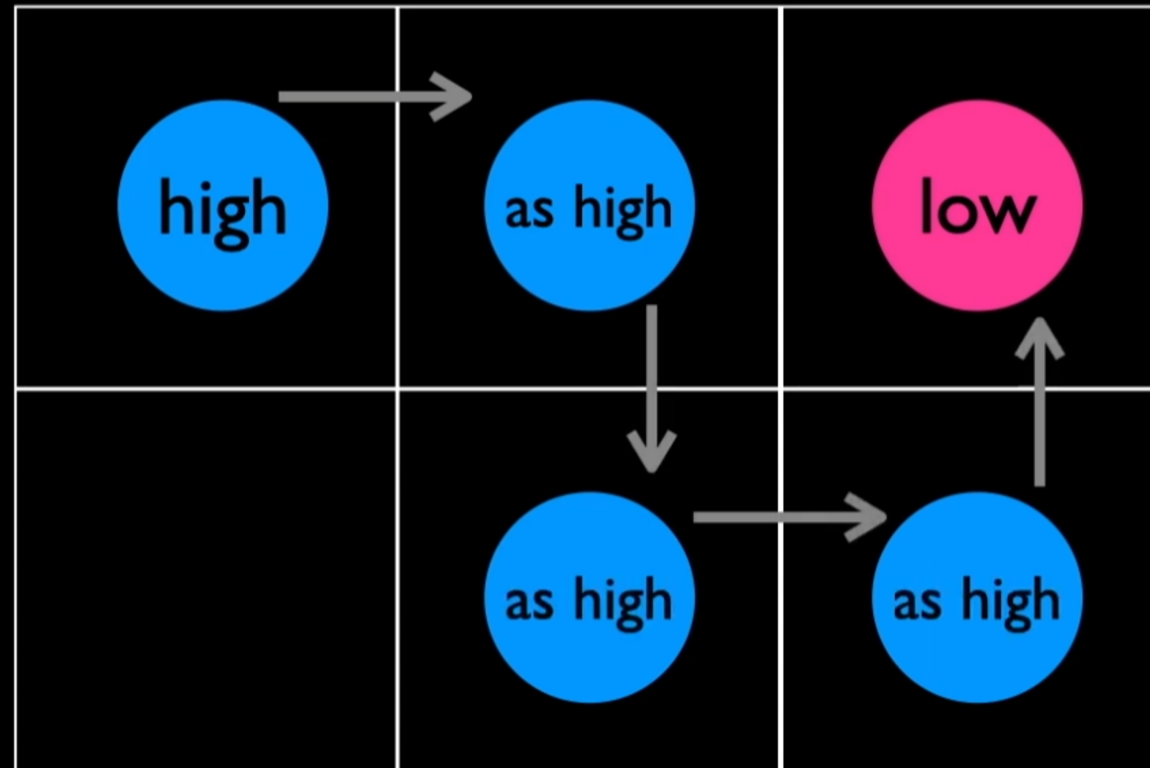
priority inheritance can deal with this situation



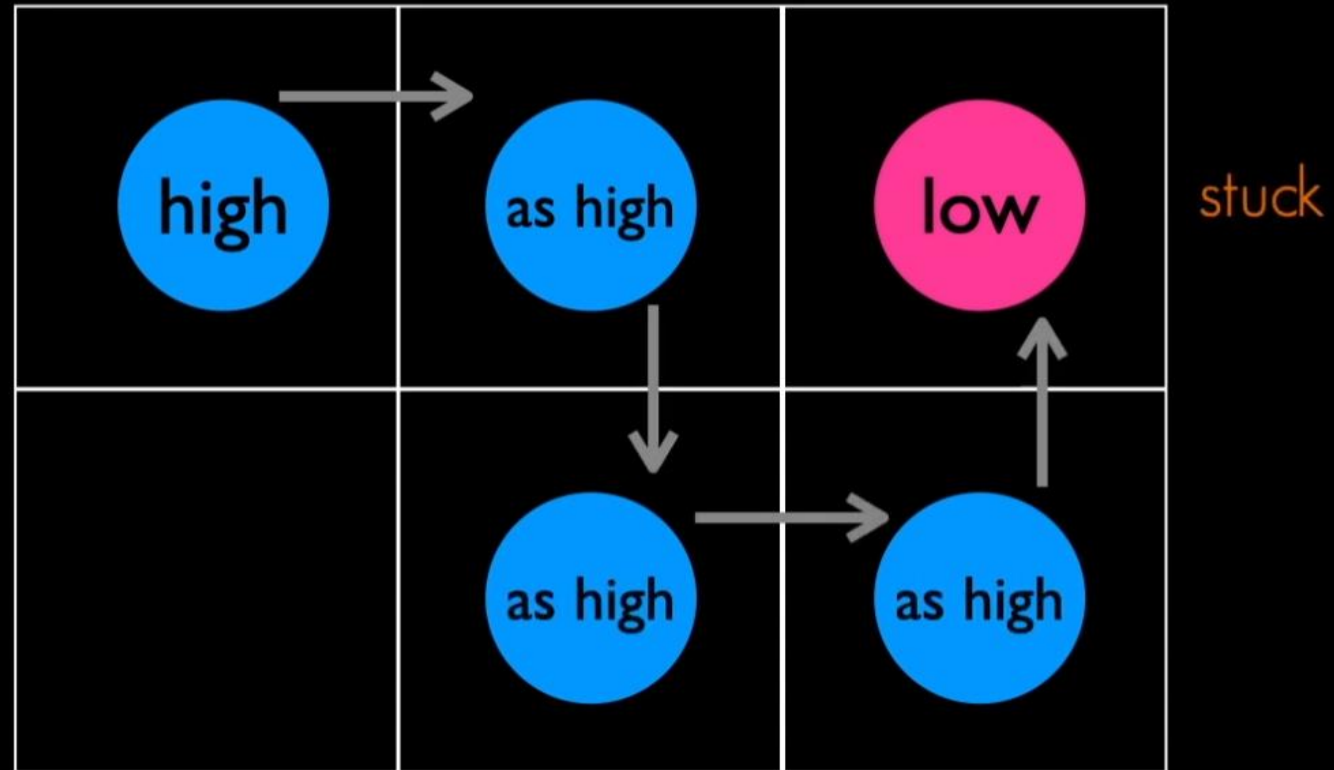
but still not deadlock-free



but still not deadlock-free



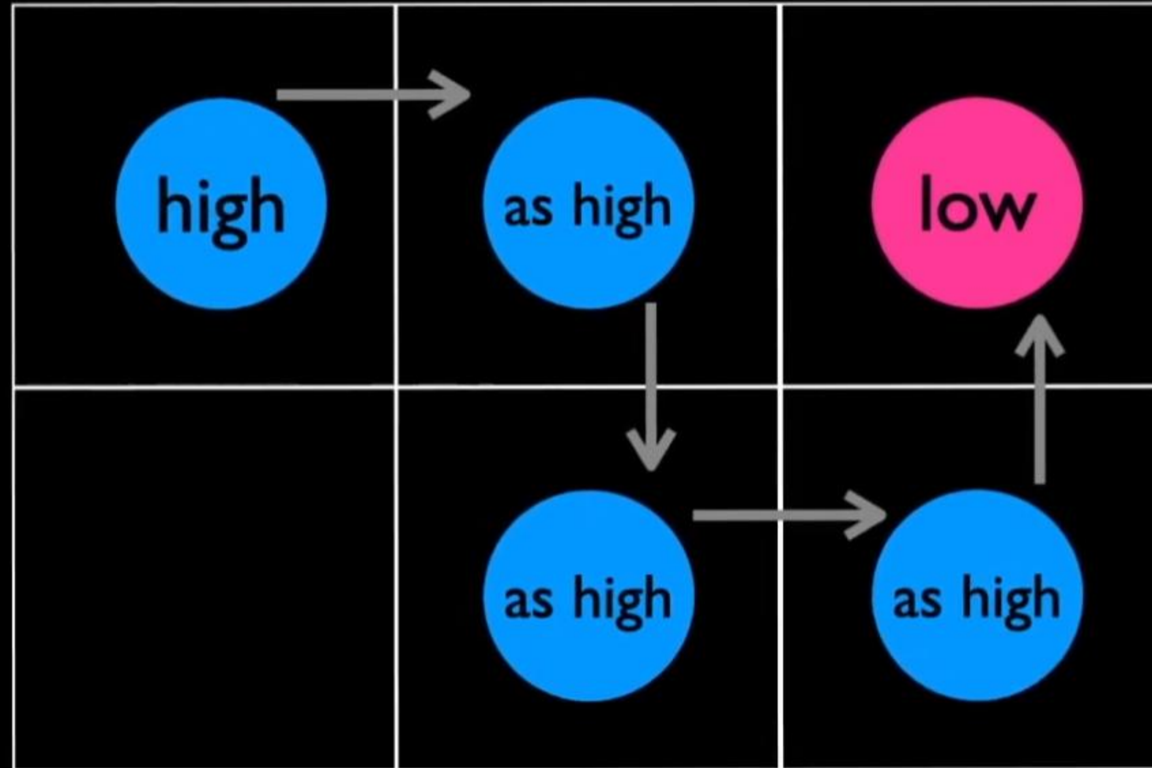
but still not deadlock-free



introduce *backtracking*

valid *"You can move"*

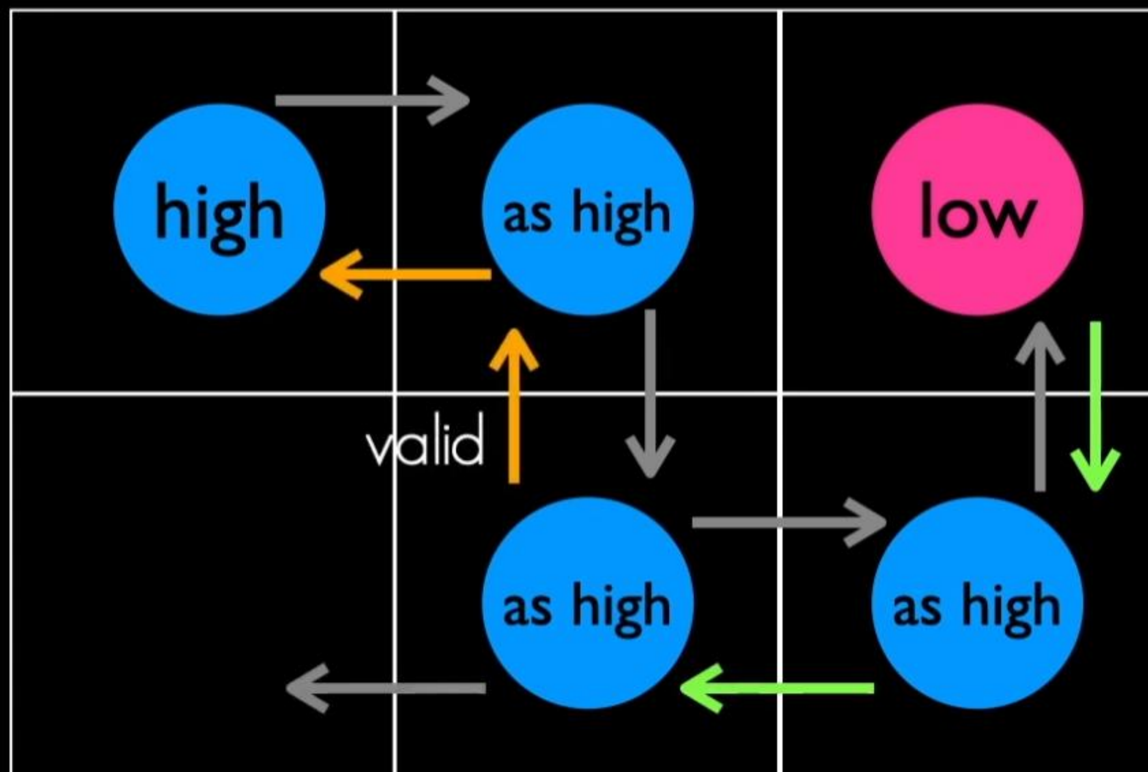
invalid *"You must re-plan, I will stay"*



introduce *backtracking*

valid "*You can move*"

invalid "*You must re-plan, I will stay*"



introduce *backtracking*

valid "*You can move*"

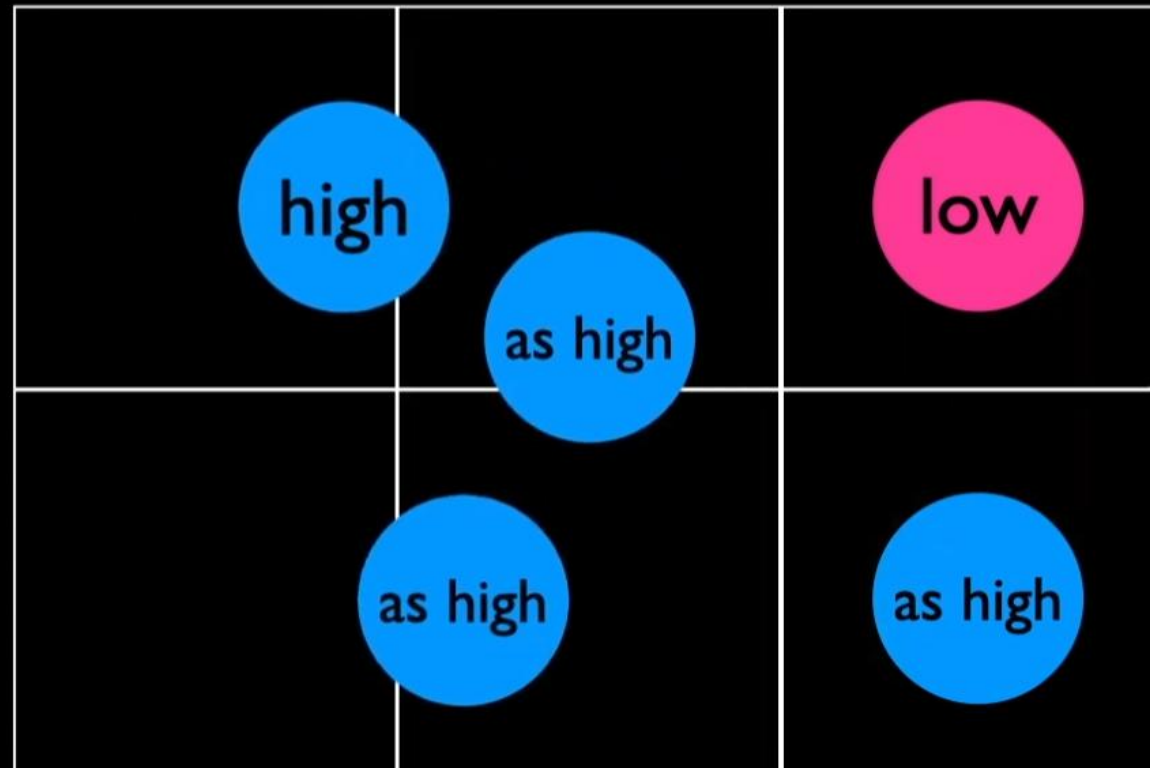
invalid "*You must re-plan, I will stay*"

high	as high	low
	as high	as high

introduce *backtracking*

valid "You can move"

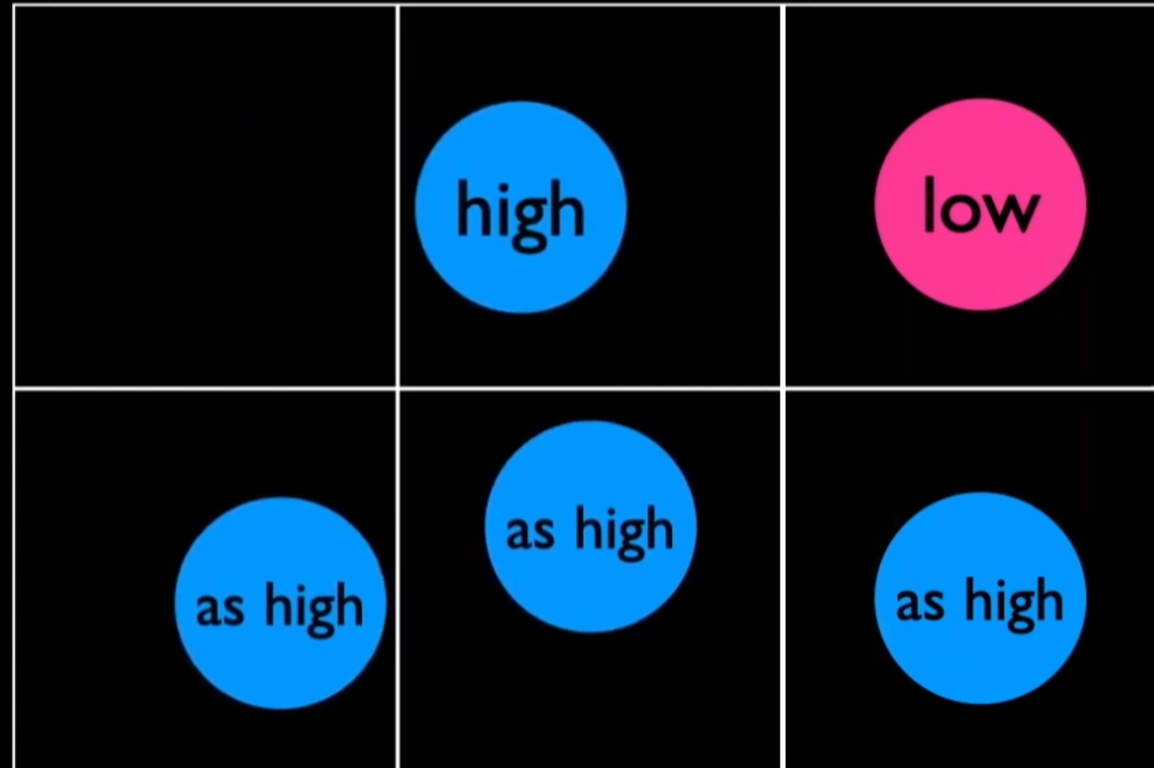
invalid "You must re-plan, I will stay"



introduce *backtracking*

valid "*You can move*"

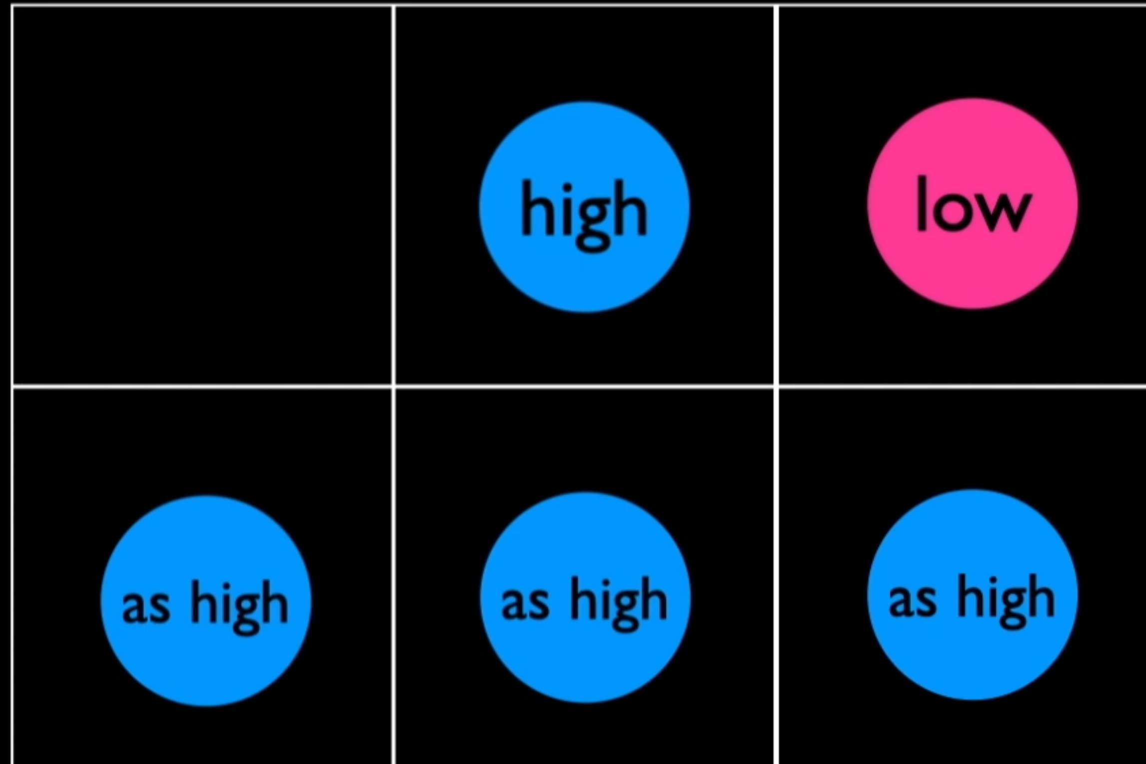
invalid "*You must re-plan, I will stay*"



introduce *backtracking*

valid *"You can move"*

invalid *"You must re-plan, I will stay"*



Thank you!