TOPIC **Search Problem - Basics** DATE . .

rational := maximize expected utility/pre-defined goals

Central Problem in AI

In artificial intelligence, the central problem at hand is that of the creation of a rational **agent**, an entity that has goals or preferences and tries to perform a series of **actions** that yield the best/optimal expected outcome given these goals. Rational agents exist in an **environment**, which is specific to the given instantiation of the agent. As a very simple example, the environment for a checkers agent is the virtual checkers board on which it plays against opponents, where piece moves are actions. Together, an environment and the agents that reside within it create a **world**.

Reflex Agent

based on current percept makes a decision but without consideration of the consequences of their actions

can be rational:
- Pac-Man in a simple content
- 类比人在遇到危险时的本能反应，可能是optimal solution

can be unrational:
- Pac-Man in a slightly complicated context → hit the wall and lose points

Planning Agent

asks "what ... if ..."

must formulate a goal (test)

must have a model of how the world evolves in response to actions

decides based on hypothesized consequences of actions.

in order to create a rational planning agent

↓

need a way to mathematically express the given environment in which the agent will exist.

↓

must formally express a **Search Problem**

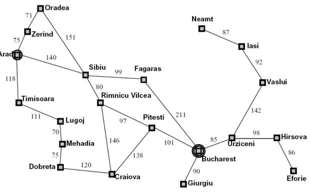
i.e. Given our agent's current state (its configuration within its environment, how can we arrive at a new state that satisfies its goals in the best possible way?

Fundamentally, a search problem is solved by first considering the start state, then exploring the state space using the successor function, iteratively computing successors of various states until we arrive at a goal state, at which point we will have determined a path from the start state to the goal state (typically called a plan). The order in which states are considered is determined using a predetermined strategy.

↓

4 Key Elements for formulating "Search Problem":

Example: Traveling in Romania

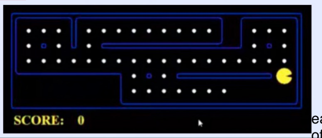


1. State space:
   - Cities
2. Successor function:
   - Roads: Go to adjacent city with cost = distance
3. Start state:
   - Arad
4. Goal test:
   - Is state == Bucharest?
- Solution?

What's in a State Space?

contains all information about a given state

The world state includes every last detail of the environment



A world state may contain more information still, potentially encoding information about things like total distance traveled by Pacman or all positions visited by Pacman on top of its current (x,y) location and dot booleans.

eat all dots attempts to solve the problem of consuming all food pellets in the maze in the shortest time possible.

A search state keeps only the details needed for planning (abstraction)

primarily for space efficiency reasons

- Problem: Pathing
  - States: (x,y) location
  - Actions: NSEW
  - Successor: update location only
  - Goal test: is (x,y)=END

Pathing attempts to solve the problem of getting from position (x1, y1) to position (x2, y2) in the maze optimally.

- Problem: Eat-All-Dots
  - States: {(x,y), dot booleans}
  - Actions: NSEW
  - Successor: update location and possibly a dot boolean
  - Goal test: dots all false

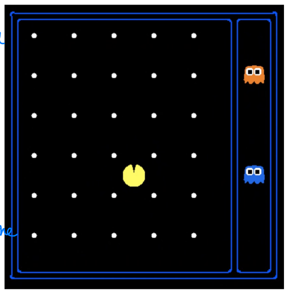
- A **state space** - The set of all possible states that are possible in your given world

For Pac-Man, the state space is a set of possible configurations of where Pac-Man is and where the dots are.

- A **successor function** - A function that takes in a state and an action and computes the cost of performing that action as well as the **successor state**, the state the world would be in if the given agent performed that action
- A **start state** - The state in which an agent exists initially
- A **goal test** - A function that takes a state as input, and determines whether it is $\boxed{\text{a goal state}}$: the condition that you want your agent to meet

State Space Sizes? : An important question that often comes up while estimating the computational runtime of solving a search problem. This is done almost exclusively with the fundamental counting principle, which states that if there are n variable objects in a given world which can take on $x_1, x_2, \cdots, x_n$ different values respectively, then the total number of $x_1, x_2, \cdots, x_n$.

- World state:
  - Agent positions: 120
  - Food count: 30 — 30 locations that could be food or no food.
  - Ghost positions: 12 — 12^2 for both ghosts
  - Agent facing: NSEW
- How many
  - World states? $120 \times (2^{30}) \times (12^2) \times 4$
  - States for pathing? 120 — Only need to know the agent position.
  - States for eat-all-dots? $120 \times (2^{30})$

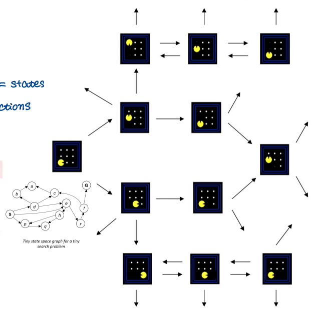


* Why don't we just keep track of only the number of food pallets left rather than going into all details of the locations? — won't be able to do planning.

State Space Graphs / 状态空间图 & Search Trees

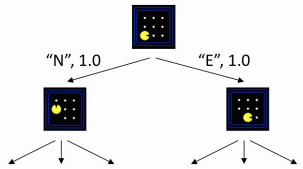
State Space Graphs

- State space graph: A mathematical representation of a search problem
  - Nodes are (abstracted) world configurations — nodes := states
  - Arcs represent successors (action results) — edges := actions
  - The goal test is a set of goal nodes (maybe only one)
- In a state space graph, each state occurs only once!
- We can rarely build this full graph in memory (it's too big), but it's a useful idea



Search Trees

have no restrictions on the number of times a state can appear



This is now / start

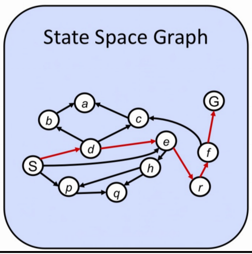
Possible futures

- A search tree:
  - A "what if" tree of plans and their outcomes
  - The start state is the root node
  - Children correspond to successors
  - Nodes show states, but correspond to PLANS that achieve those states
  - For most problems, we can never actually build the whole tree

When we think about a node in the search tree, we actually think of it as a sequence of states that have happened.

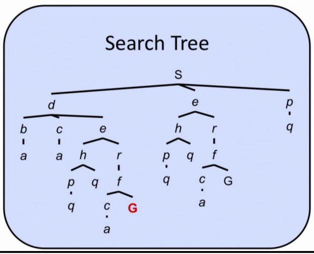
State Space Graphs vs. Search Trees

The highlighted path (S → d → e → r → f → G) in the given state space graph is represented in the corresponding search tree by following the path in the tree from the start state S to the highlighted goal state G.



Each NODE in in the search tree is an entire PATH in the state space graph.

We construct both on demand – and we construct as little as possible.



Each and every path from the start node to any other node is represented in the search tree by a path from the root S to some descendant of the root corresponding to the other node. Since there often exist multiple ways to get from one state to another, states tend to show up multiple times in search trees.

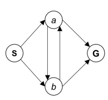
As a result, search trees are greater than or equal to their corresponding state space graph in size.

⇒ Q: How can we perform useful computation on these structures if they're too big to represent in memory?
— The answer lies in successor functions.

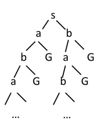
we only store states we're immediately working with, and compute new ones on-demand using the corresponding successor function. Typically, search problems are solved using search trees, where we very carefully store a select few nodes to observe at a time, iteratively replacing nodes with their successors until we arrive at a goal state. There exist various methods by which to decide the order in which to conduct this iterative replacement of search tree nodes.

Quiz: State Space Graphs vs. Search Trees

Consider this 4-state graph:



How big is its search tree (from S)?



∞

Important: Lots of repeated structure in the search tree!

Consider the "**water-jug puzzle**":

There is a 3-liter water jug and a 4-liter water jug. At the beginning, both are empty. At the end, the 4-liter jug shall contain exactly 2 liter. A jug can be emptied or filled with water (completely). Water can be poured from one jug into the other. This must be done exactly until one jug is empty or full.

- Formalize this as a search problem where the costs of all actions are 1.

Initial State: $(0, 0)$

Actions:
- empty
  - e3: $(x, y) \to (0, y)$ for $x \neq 0$
  - e4: $(x, y) \to (x, 0)$ for $y \neq 0$
- full
  - f3: $(x, y) \to (3, y)$ for $x \neq 3$
  - f4: $(x, y) \to (x, 4)$ for $y \neq 4$
- pour
  - p3: $(x, y) \to (x-z, y+z)$ for $x \neq 0 \wedge y \neq 4 \wedge z = \min\{x, 4-y\}$
  - p4: $(x, y) \to (x+z, y-z)$ for $y \neq 0 \wedge x \neq 3 \wedge z = \min\{3-x, y\}$

State Space: Set of all states reachable from the initial state.

$$\{0,3\} \times \{1,2,3,4\} \cup \{1,2\} \times \{0,4\}$$
$$= \{0,1,2,3\} \times \{0,1,2,3,4\} \setminus \{1,2\} \times \{1,2,3\}$$

Goal Test: (x, y) is a goal iff $y = 2$

Path Cost: Length of path

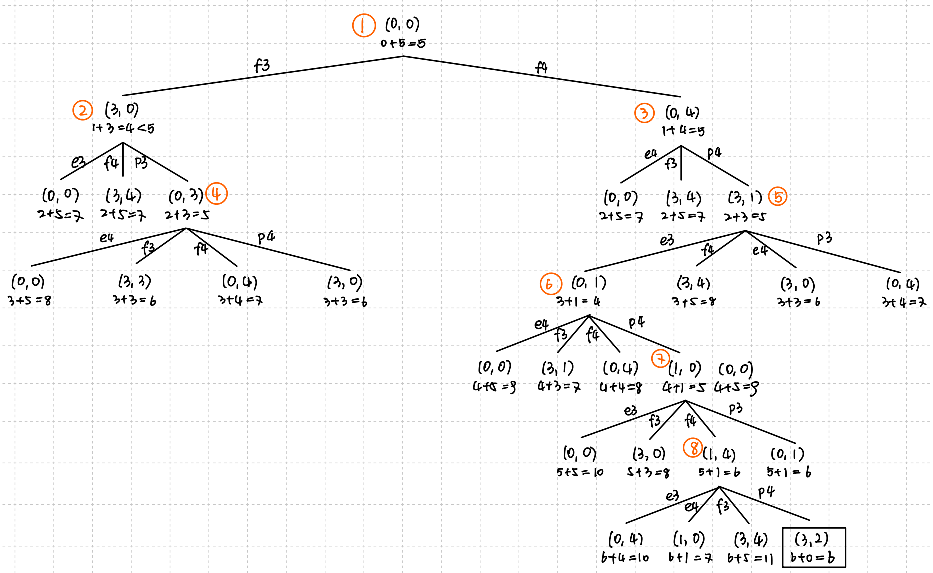
- The following is true for that problem in every state except goal states:

  If the 3-liter jug is full, then at least 3 steps are necessary to reach a goal state. If the 3-liter jug is empty and the 4-liter jug contains x liter, then at least x steps are necessary to reach a goal state. If both jugs are full or both jugs are empty, then at least 5 steps are necessary to reach a goal state.

  Use this (and only this[2]) information to find an admissible heuristic that is as good as possible.

state $[n]$	$h(n)$
$(x, 2)$	0
$(0, 0)$	5
$(3, 4)$	5
$(3, y)$, $y \notin \{2, 4\}$	3
$(0, y)$, $y \notin \{0, 2\}$	y
else	1

- Solve the problem with A* search using your heuristic and draw the A* search tree. Label each node with the corresponding state and the estimated cost of the cheapest solution path through it. Additionally, mark in the tree the order of the expansion of the nodes.



Uninformed Search

- Main question: which fringe nodes to explore?

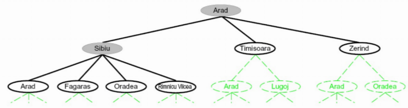
The standard protocol for finding a plan to get from the start state to a goal state is to maintain an outer **fringe** of partial plans derived from the search tree. We continually **expand** our fringe by removing a node (which is selected using our given **strategy**) corresponding to a partial plan from the fringe, and replacing it on the fringe with all its children. Removing and replacing an element on the fringe with its children corresponds to discarding a single length n plan and bringing all length $(n+1)$ plans that stem from it into consideration. We continue this until eventually removing a goal state off the fringe, at which point we conclude the partial plan corresponding to the removed goal state is in fact a path to get from the start state to the goal state. Practically, most implementations of such algorithms will encode information about the parent node, distance to node, and the state inside the node object. This procedure we have just outlined is known as **tree search**, and the pseudocode for it is presented below:

```
function TREE-SEARCH(problem, fringe) return a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        for child-node in EXPAND(STATE[node], problem) do
            fringe ← INSERT(child-node, fringe)
        end
    end
```

When we have no knowledge of the location of goal states in our search tree, we are forced to select our strategy for tree search from one of the techniques that falls under the umbrella of **uninformed search**. We'll now cover three such strategies in succession: **depth-first search**, **breadth-first search**, and **uniform cost search**. Along with each strategy, some rudimentary properties of the strategy are presented as well, in terms of the following:

- The **completeness** of each search strategy - if there exists a solution to the search problem, is the strategy guaranteed to find it given infinite computational resources?
- The **optimality** of each search strategy - is the strategy guaranteed to find the lowest cost path to a goal state?
- The **branching factor** b - The increase in the number of nodes on the fringe each time a fringe node is dequeued and replaced with its children is $O(b)$. At depth k in the search tree, there exists $O(b^k)$ nodes.
- The maximum depth m.
- The depth of the shallowest solution s.

Searching with a Search Tree



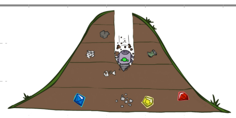
- Search:
  - Expand out potential plans (tree nodes)
  - Maintain a fringe of partial plans under consideration
  - Try to expand as few tree nodes as possible

fringe / 边缘 := a data structure used to store all the possible states (nodes) that you can go from the current states.

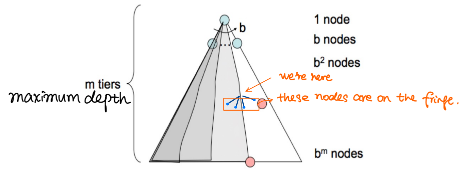
- A search algorithm is an algorithm that systematically builds a search tree (hopefully only fraction of entire search tree). It has to choose an ordering of what to currently expand (ready to be expanded is called the fringe, but it has to choose which one to expand first)

An optimal search algorithm is the one that finds least-cost plans.

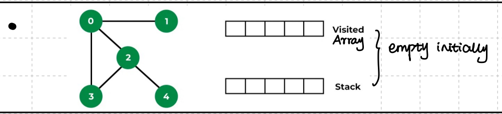
Depth-First Search



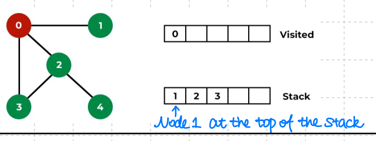
- *Description* - Depth-first search (DFS) is a strategy for exploration that always selects the *deepest* fringe node from the start node for expansion.
- *Fringe representation* - Removing the deepest node and replacing it on the fringe with its children necessarily means the children are now the new deepest nodes - their depth is one greater than the depth of the previous deepest node. This implies that to implement DFS, we require a structure that always gives the most recently added objects highest priority. A last-in, first-out (LIFO) stack does exactly this, and is what is traditionally used to represent the fringe when implementing DFS.



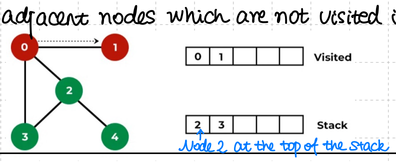
- *Completeness* - Depth-first search is not complete. If there exist cycles in the state space graph, this inevitably means that the corresponding search tree will be infinite in depth. Hence, there exists the possibility that DFS will faithfully yet tragically get "stuck" searching for the deepest node in an infinite-sized search tree, doomed to never find a solution.
- *Optimality* - Depth-first search simply finds the "leftmost" solution in the search tree without regard for path costs, and so is not optimal. nor depth
- *Time Complexity* - In the worst case, depth first search may end up exploring the entire search tree. Hence, given a tree with maximum depth m, the runtime of DFS is $O(b^m)$.
- *Space Complexity* - In the worst case, DFS maintains b nodes at each of m depth levels on the fringe. This is a simple consequence of the fact that once b children of some parent are enqueued, the nature of DFS allows only one of the subtrees of any of these children to be explored at any given point in time. Hence, the space complexity of BFS is $O(bm)$. 对于路径上的m个节点，需为m个节点中的每个节点额外存储b个节点.



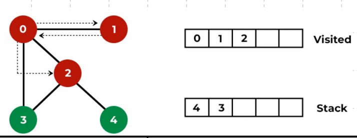
- Visit 0 and put its adjacent nodes which are not visited yet into the stack.

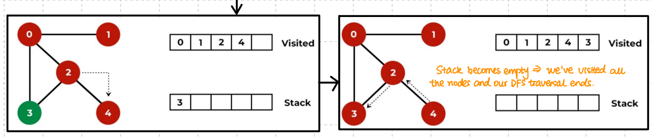


- Visit node 1, pop node 1 from the stack and put all of its adjacent nodes which are not visited in the stack.

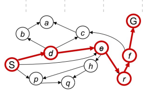


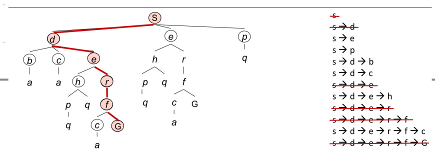
- Visit node 2, pop it from the stack and put all of its adjacent nodes which are not visited (i.e. 3, 4) in the stack.



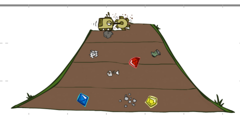


- The space that the fringe takes contains only siblings on path to root. i.e. O(bm)

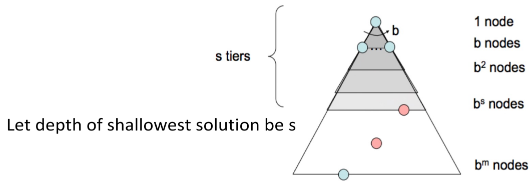




Breadth-First Search

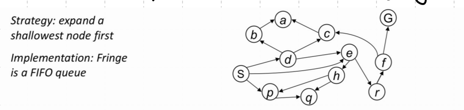


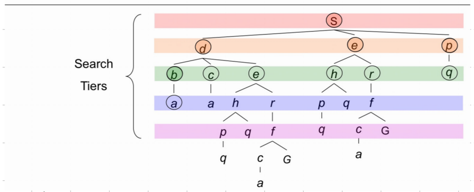
- *Description* - Breadth-first search is a strategy for exploration that always selects the *shallowest* fringe node from the start node for expansion.
- *Fringe representation* - If we want to visit shallower nodes before deeper nodes, we must visit nodes in their order of insertion. Hence, we desire a structure that outputs the oldest enqueued object to represent our fringe. For this, BFS uses a first-in, first-out (FIFO) queue, which does exactly this.



- *Completeness* - If a solution exists, then the depth of the shallowest node s must be finite, so BFS must eventually search this depth. Hence, it's complete.
- *Optimality* - BFS is generally not optimal because it simply does not take costs into consideration when determining which node to replace on the fringe. The special case where BFS is guaranteed to be optimal is if all edge costs are equivalent, because this reduces BFS to a special case of uniform cost search, which is discussed below.
- *Time Complexity* - We must search $1+b+b^2+\ldots+b^s$ nodes in the worst case, since we go through all nodes at every depth from 1 to s. Hence, the time complexity is $O(b^s)$.
- *Space Complexity* - The fringe, in the worst case, contains all the nodes in the level corresponding to the shallowest solution. Since the shallowest solution is located at depth s, there are $O(b^s)$ nodes at this depth.

- The space that the fringe takes contains roughly the last tier, so $O(b^s)$



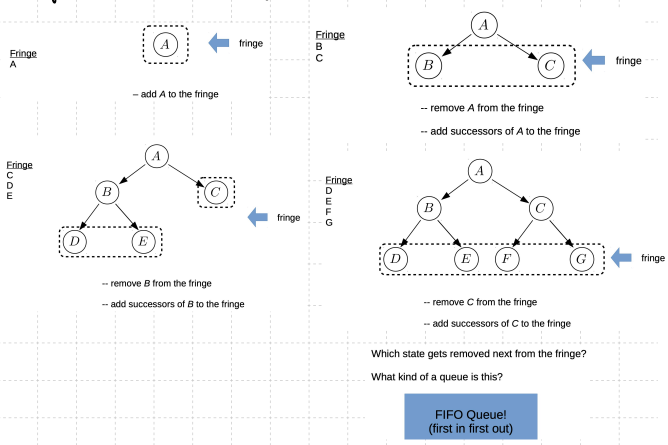


Number of expansions:

- $b + b^2 + \cdots + b^{d-1} + b^d + b^{d+1} - b \subset O(b^d)$

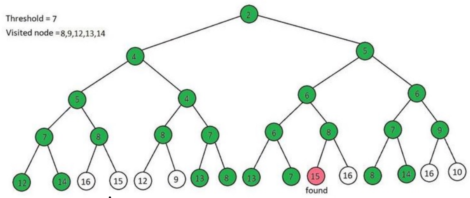
- BFS is a complete search algorithm, which means that it doesn't stop immediately upon finding a goal node.
若目标节点位于深度d，那么会在queue中存储(但不一定完全探索)深度为d+1处的b^{d+1}个节点.
当找到目标节点后，实际上还需继续explore目标节点的子节点.(d个)
因此公式为 $+ b^{d+1} - b$

- Fringe (Initialize fringe as an empty queue)



TOPIC DATE . .

(5)



Threshold = 7 > current node value 2

S3: explore its children, 2 children one by one.

1° children $\boxed{\text{Set current node = 4}}$ → $\boxed{< \text{Threshold}}$ → $\boxed{\text{explore its children}}$

(i) 5. < Threshold
explore its children
(i)$_1$ 7 == threshold
explore its children
(i)$_{11}$ 12 > threashold, prune
(i)$_{12}$ 14 > threashold, prune
(i)$_2$ 8 > threashold, prune

(ii) 4 < Threshold
explore its children
(ii)$_1$ 8 > threshold, prune
(ii)$_2$ 7 == threshold
explore its children
(ii)$_{21}$ 13 > threashold, prune
(ii)$_{22}$ 8 > threashold, prune

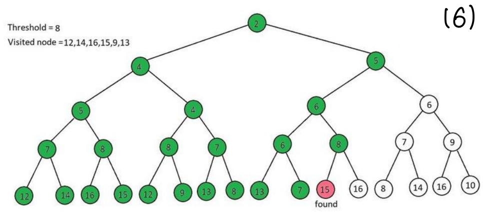
2° children $\boxed{\text{Set current node = 5}}$ → $\boxed{< \text{Threshold}}$ → $\boxed{\text{explore its children}}$

(a) 6 < threshold
explore its children
(a)$_1$ 6 < threshold
(a)$_{11}$ 13 > threshold, prune
(a)$_{12}$ 7 == threshold
doesn't have children
(a)$_2$ 8 > threshold, prune

(b) 6 < threshold
explore its children
(b)$_1$ 7 == threshold
explore its children
(b)$_{11}$ 8 > threashold, prune
(b)$_{12}$ 14 > threashold, prune
(b)$_2$ 9 > threshold, prune

Pruned Value: 12, 14, 8, 13, 9

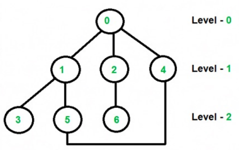
(6)



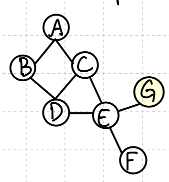
在打展(a)$_2$时找到 Goal Node
The goal path: 2 → 5 → 6 → 8 → 15

IDDFS

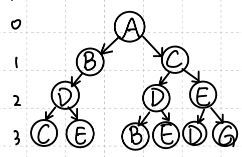
Depth	Iterative Deepening Depth First Search
0	0
1	0 1 2 4
2	0 1 3 5 2 6 4 5
3	0 1 3 5 4 2 6 4 5 1



trace the path from A to G



Depth Limit = 3:



Time Complexity

- Number of expansions: (Depth = d)

depth-0 iteration: 1
depth-1 iteration: 1 + b
depth-2 iteration: $1 + b + b^2$
⋮
depth-d iteration: $1 + b + b^2 + \cdots + b^d$

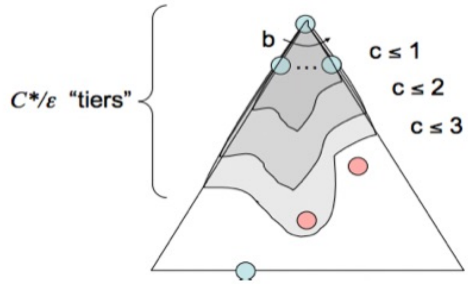
与: $d \cdot b + (d-1) \cdot b^2 + \cdots + 3 \cdot b^{d-2} + 2 \cdot b^{d-1} + b^d \subset \boxed{O(b^d)}$

Space Complexity: $O(b \cdot d)$

Uniform Cost Search 统一代价搜索 / Cheapest First Search

- *Description* - Uniform cost search (UCS), our last strategy, is a strategy for exploration that always selects the *lowest cost* fringe node from the start node for expansion.

- *Fringe representation* - To represent the fringe for UCS, the choice is usually a heap-based priority queue, where the weight for a given enqueued node v is the path cost from the start node to v, or the *backward cost* of v. Intuitively, a priority queue constructed in this manner simply reshuffles itself to maintain the desired ordering by path cost as we remove the current minimum cost path and replace it with its children.



- *Completeness* - Uniform cost search is complete. If a goal state exists, it must have some finite length shortest path; hence, UCS must eventually find this shortest length path.

- *Optimality* - UCS is also optimal if we assume all edge costs are nonnegative. By construction, since we explore nodes in order of increasing path cost, we're guaranteed to find the lowest-cost path to a goal state. The strategy employed in Uniform Cost Search is identical to that of Dijkstra's algorithm, and the chief difference is that UCS terminates upon finding a solution state instead of finding the shortest path to all states. Note that having negative edge costs in our graph can make nodes on a path have decreasing length, ruining our guarantee of optimality. (See Bellman-Ford algorithm for a slower algorithm that handles this possibility)

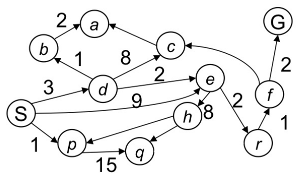
- *Time Complexity* - Let us define the optimal path cost as C^* and the minimal cost between two nodes in the state space graph as ε. Then, we must roughly explore all nodes at depths ranging from 1 to C^*/ε, leading to an runtime of $O(b^{C^*/\varepsilon})$. each individual step costs us ε

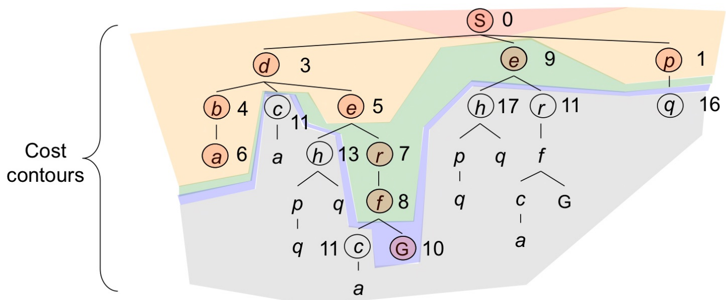
- *Space Complexity* - Roughly, the fringe will contain all nodes at the level of the cheapest solution, so the space complexity of UCS is estimated as $O(b^{C^*/\varepsilon})$.

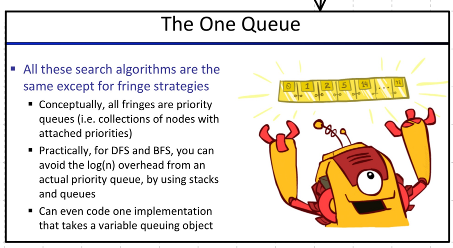
As a parting note about uninformed search, it's critical to note that the three strategies outlined above are fundamentally the same - differing only in expansion strategy, with their similarities being captured by the tree search pseudocode presented above.

Strategy: expand a cheapest node first:

Fringe is a priority queue (priority: cumulative cost)







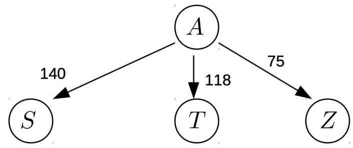
UCS explores increasing cost contours.

- The bad:
  - Explores options in every "direction"
  - No information about goal location

Fringe	Path Cost
A	0

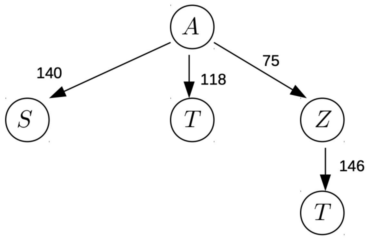
Explored set:

Fringe	Path Cost
~~A~~	~~0~~
S	140
T	118
Z	75



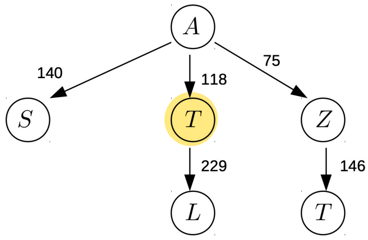
Explored set: A

Fringe	Path Cost
~~A~~	~~0~~
S	140
T	118
~~Z~~	~~75~~
T	146



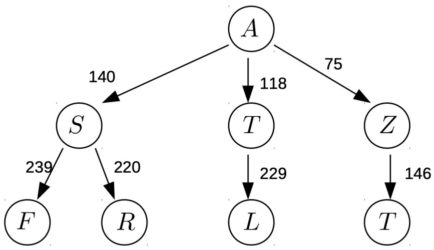
Explored set: A, Z

Fringe	Path Cost
~~A~~	~~0~~
S	140
~~T~~	~~118~~
~~Z~~	~~75~~
T	146
L	229



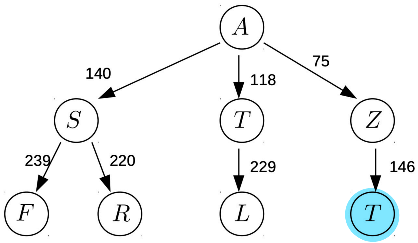
Explored set: A, Z, T

Fringe	Path Cost
~~A~~	~~0~~
~~S~~	~~140~~
~~T~~	~~118~~
~~Z~~	~~75~~
T	146
L	229
F	239
R	220



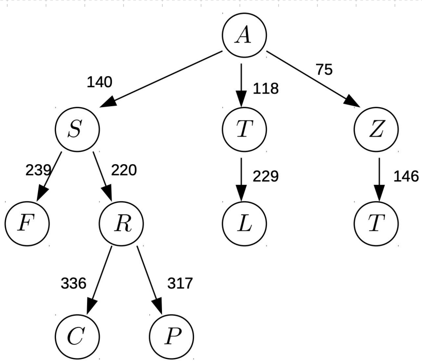
Explored set: A, Z, T, S

Fringe	Path Cost
~~A~~	~~0~~
~~S~~	~~140~~
~~T~~	~~118~~
~~Z~~	~~75~~
~~T~~	~~146~~
L	229
F	239
R	220



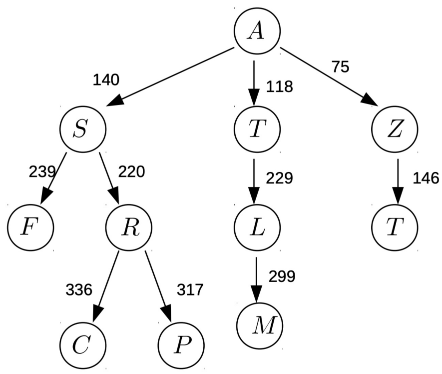
Explored set: A, Z, T, S

Fringe	Path Cost
~~A~~	~~0~~
~~S~~	~~140~~
~~T~~	~~118~~
~~Z~~	~~75~~
~~T~~	~~146~~
L	229
F	239
~~R~~	~~220~~
C	336
P	317



Explored set: A, Z, T, S, R

Fringe	Path Cost
~~A~~	~~0~~
~~S~~	~~140~~
~~T~~	~~118~~
~~Z~~	~~75~~
~~T~~	~~146~~
~~L~~	~~229~~
F	239
~~R~~	~~220~~
C	336
P	317
M	299



Explored set: A, Z, T, S, R, L

When does this end?
- when the goal state is removed from the queue
- NOT when the goal state is expanded

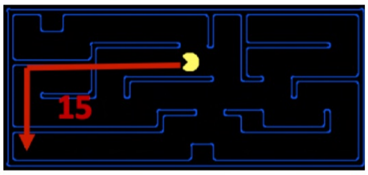
∘ Informed Search

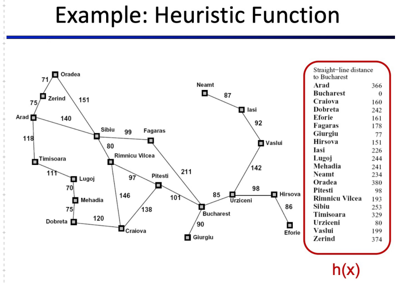
Uniform cost search is good because it's both complete and optimal, but it can be fairly slow because it expands in every direction from the start state while searching for a goal. If we have some notion of the direction in which we should focus our search, we can significantly improve performance and "hone in" on a goal much more quickly. This is exactly the focus of **informed search**.

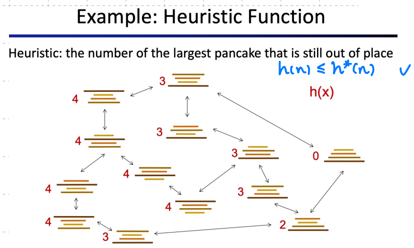
Heuristic /启发式算法

Heuristics are the driving force that allow estimation of distance to goal states - they're functions that take in a state as input and output a corresponding estimate. The computation performed by such a function is specific to the search problem being solved. For reasons that we'll see in A* search, below, we usually want heuristic functions to be a lower bound on this remaining distance to the goal, and so heuristics are typically solutions to **relaxed problems** (where some of the constraints of the original problem have been removed). Turning to our Pacman example, let's consider the pathing problem described earlier. A common heuristic that's used to solve this problem is the **Manhattan distance**, which for two points (x_1, y_1) and (x_2, y_2) is defined as follows:

$$Manhattan(x_1, y_1, x_2, y_2) = |x_1 - x_2| + |y_1 - y_2|$$







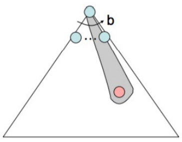
The above visualization shows the relaxed problem that the Manhattan distance helps solve - assuming Pacman desires to get to the bottom left corner of the maze, it computes the distance from Pacman's current location to Pacman's desired location *assuming a lack of walls in the maze*. This distance is the *exact* goal distance in the relaxed search problem, and correspondingly is the *estimated* goal distance in the actual search problem. With heuristics, it becomes very easy to implement logic in our agent that enables them to "prefer" expanding states that are estimated to be closer to goal states when deciding which action to perform. This concept of preference is very powerful, and is utilized by the following two search algorithms that implement heuristic functions: greedy search and A*.

Greedy Search

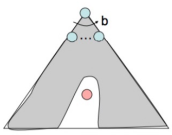
- *Description* - Greedy search is a strategy for exploration that always selects the fringe node with the *lowest heuristic value* for expansion, which corresponds to the state it believes is nearest to a goal.

i.e. expand a node that "heuristic" says it's the closest to a goal state

- *Fringe representation* - Greedy search operates identically to UCS, with a priority queue fringe representation. The difference is that instead of using *computed backward cost* (the sum of edge weights in the path to the state) to assign priority, greedy search uses *estimated forward cost* in the form of heuristic values.
- *Completeness and Optimality* - Greedy search is not guaranteed to find a goal state if one exists, nor is it optimal, particularly in cases where a very bad heuristic function is selected. It generally acts fairly unpredictably from scenario to scenario, and can range from going straight to a goal state to acting like a badly-guided DFS and exploring all the wrong areas.



(a) Greedy search on a good day :)



(b) Greedy search on a bad day :(

b := the branching factor, which indicates how many successors are there from any given node

The effective branching factor (有效分支因子) is defined as : $N = b^* + (b^*)^2 + (b^*)^3 + \dots + (b^*)^d$

N := the number of nodes (i.e. the size of fringe + the size of explored) $N^{\frac{1}{d+1}} \leq b^* \leq N^{\frac{1}{d}}$ (N := the total number of nodes searched)

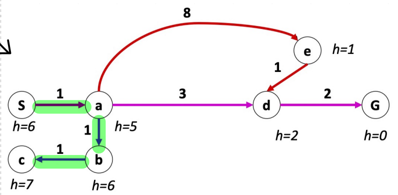
b^* := effective branching factor (to find), it finds the "average" branching factor of a tree. (smaller branching = less Searching)

d := depth of solution i.e. Search depth.

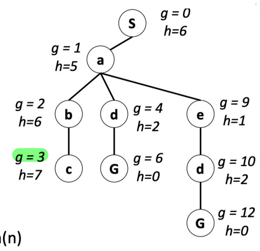
A*

- *Description* - A* search is a strategy for exploration that always selects the fringe node with the *lowest estimated total cost* for expansion, where total cost is the entire cost from the start node to the goal node.
- *Fringe representation* - Just like greedy search and UCS, A* search also uses a priority queue to represent its fringe. Again, the only difference is the method of priority selection. A* combines the total backward cost (sum of edge weights in the path to the state) used by UCS with the estimated forward cost (heuristic value) used by greedy search by adding these two values, effectively yielding an *estimated total cost* from start to goal. Given that we want to minimize the total cost from start to goal, this is an excellent choice.
- *Completeness and Optimality* - A* search is both complete and optimal, given an appropriate heuristic (which we'll cover in a minute). It's a combination of the good from all the other search strategies we've covered so far, incorporating the generally high speed of greedy search with the optimality and completeness of UCS!
  - $g(n)$ - The function representing total backwards cost computed by UCS.
  - $h(n)$ - The *heuristic value* function, or estimated forward cost, used by greedy search.
  - $f(n)$ - The function representing estimated total cost, used by A* search. $f(n) = g(n) + h(n)$.

- Two lists are used:
  - An **open list**, implemented as a priority queue, which stores the next nodes to be explored. Because this is a priority queue, the most promising candidate node (the one with the lowest value from the evaluation function) is always at the top. Initially, the only node in this list is the start node **S**.
  - A **closed list** which stores the nodes that have already been evaluated. When a node is in the closed list, it means that the lowest-cost path to that node has been found.



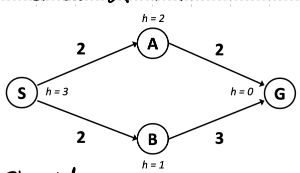
- A* Search orders by the sum: f(n) = g(n) + h(n)

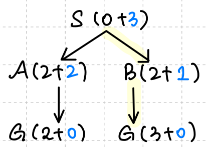


Example: Teg Grenager

- When should A* terminate?

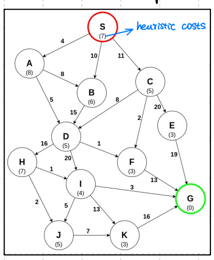
e.g.

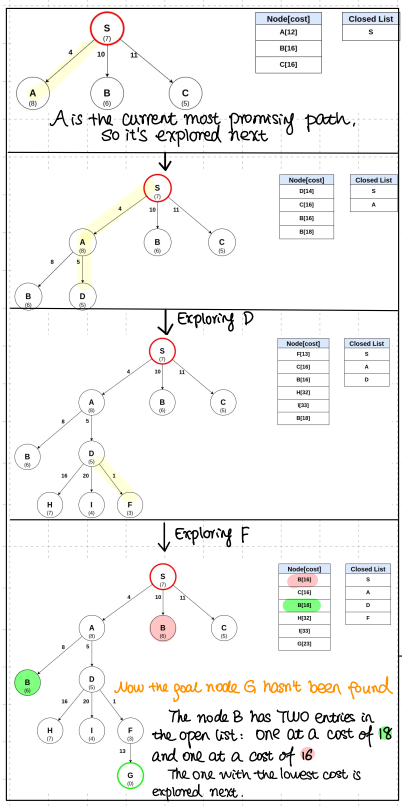




Should we stop when we enqueue a goal? NO. ONLY stop when we dequeue a goal.

- Concrete Implementation: find the shortest path from S to G in the following graph:





A is the current most promising path, so it's explored next

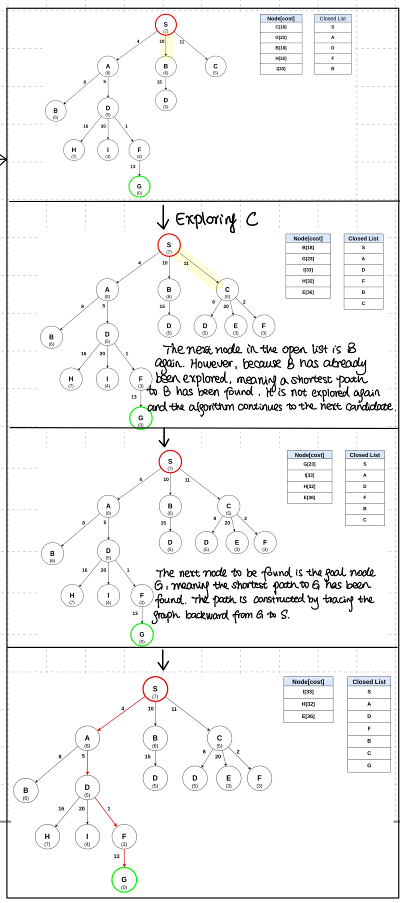
Exploring D

Exploring F

Now the goal node G hasn't been found

The node B has TWO entries in the open list: one at a cost of 18 and one at a cost of 16. The one with the lowest cost is explored next.

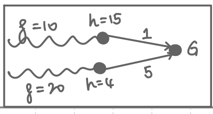
The algorithm continues because there may be a shorter path to G



Exploring C

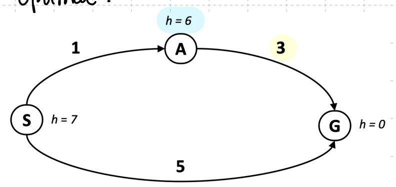
The next node in the open list is B again. However, because B has already been explored, meaning a shortest path to B has been found. It is not explored again and the algorithm continues to the next candidate.

The next node to be found is the goal node G, meaning the shortest path to G has been found. The path is constructed by tracing the graph backward from G to S.



$g :=$ the accumulative cost so far.

- Is A^* optimal?



- What went wrong?
- Actual bad goal cost < estimated good goal cost
- We need estimates to be less than actual costs!

/ Consider the special case in which heuristic function $h(n) = 1 - g(n)$

$\Rightarrow f(n) = g(n) + h(n) = g(n) + 1 - g(n) = 1$

$\Rightarrow$ Such a heuristic reduces A^* search to BFS (f(n)不偏好同一层级的任一节点)

BFS is not guaranteed to be optimal in the general case where edge weights are not constant. (BFS在扩展时不考虑边的权重，可能会错过权重更低的较短路径)

$\Rightarrow A^*$ in its special case is not optimal.

- Admissable / 可采纳 and Consistency

(1) The condition required for optimality when using A* tree search is known as **admissibility**. The admissibility constraint states that the value estimated by an admissible heuristic is neither negative nor an overestimate. Defining $h^*(n)$ as the true optimal forward cost to reach a goal state from a given node n, we can formulate the admissibility constraint mathematically as follows:

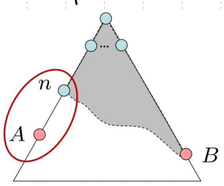
$$\forall n,\ 0 \le \boxed{h(n)} \le \boxed{h^*(n)}$$

Coming up with admissible heuristics is most of what's involved in using A^* in practice.

Normally we can't access this value. But in Pac-Man

In Pac-Man using "Manhatten Distance" { with wall : not true cost. without wall : $h(n) = h^*(n)$

(2) Informal & Formal proof the optimality of A^* with admissibility.



Assume: A is an optimal goal node $\wedge$ B is a suboptimal goal node $\wedge$ h is admissible

Claim: A will exit the fringe before B.

Proof: Imagine B is on the fringe $\wedge$ Some ancestor n of A is on the fringe, too (maybe A!)

$\Rightarrow$ Claim: n will be expand before B. < n将在B之前被扩展 >

(i) f(n) is less or equal to f(A)

$f(n) = g(n) + h(n)$ < Definition of f-cost >

$f(n) \le g(A)$ < Admissability of h >

$g(A) = f(A)$ < h = 0 at a goal >

h being admissable := h underestimates how much it will cost to get to the optimal goal.

A encodes a sequence of actions, as well as a sequence of states that you traverse, that get you from S (start) to a goal state. There could be MANY goal states, and could be many paths to each of the goal states. The optimal one is the one that's shortest from the start to ANY of the goal state; "suboptimal" means it's A PATH to A GOAL STATE, however not as short as the path encoded in A.

从起始点到A的实际路径成本 < 到B的成本

(ii) f(A) is less than f(B)

$g(A) < g(B)$ < B is suboptimal >

$f(A) < f(B)$ < h = 0 at a goal >

(iii) n expands before B

$f(n) \le f(A) < f(B)$

All ancestors of A expand before B $\Rightarrow$ A expands before B $\Rightarrow$ A^* search optimal.

为何不在(ii)之后结束证明?

When B is on the fringe, A might be not on the fringe. (e.g. 在expand过程中较早发现B节点并被加入fringe: A*搜索是动态的，随着搜索的深入和对各节点实际成本的累积，最初看似有前景的节点可能因为累积成本过高而变得less attractive). When A is not on the fringe, we know an ancestor of A has to be on the fringe. And then we say, we are guaranteed that that ancestor will be expanded before b, such that we can get A on the fringe, before b gets expanded.

(或见下页有类似案例)

Theorem. For a given search problem, if the admissibility constraint is satisfied by a heuristic function h, using A* tree search with h on that search problem will yield an optimal solution.

Proof. Assume two reachable goal states are located in the search tree for a given search problem, an optimal goal A and a suboptimal goal B. Some ancestor n of A (including perhaps A itself) must currently be on the fringe, since A is reachable from the start state. We claim n will be selected for expansion before B, using the following three statements:

1. $g(A) < g(B)$. Because A is given to be optimal and B is given to be suboptimal, we can conclude that A has a lower backwards cost to the start state than B.

2. $h(A) = h(B) = 0$, because we are given that our heuristic satisfies the admissibility constraint. Since both A and B are both goal states, the true optimal cost to a goal state from A or B is simply $h^*(n) = 0$; hence $0 \le h(n) \le 0$.

3. $f(n) \le f(A)$, because, through admissibility of h, $f(n) = g(n) + h(n) \le g(n) + h^*(n) = g(A) = f(A)$. The total cost through node n is at most the true backward cost of A, which is also the total cost of A.

We can combine statements 1. and 2. to conclude that $f(A) < f(B)$ as follows:

$$f(A) = g(A) + h(A) = g(A) < g(B) = g(B) + h(B) = f(B)$$

A simple consequence of combining the above derived inequality with statement 3. is the following:

$$f(n) \le f(A) \wedge f(A) < f(B) \Longrightarrow f(n) < f(B)$$

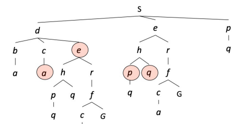
Hence, we can conclude that n is expanded before B. Because we have proven this for arbitrary n, we can conclude that *all* ancestors of A (including A itself) expand before B. □

(3) 优化 ⇒ 图搜索

One problem we found above with tree search was that in some cases it could fail to ever find a solution, getting stuck searching the same cycle in the state space graph infinitely. Even in situations where our search technique doesn't involve such an infinite loop, it's often the case that we revisit the same node multiple times because there's multiple ways to get to that same node. This leads to exponentially more work, and the natural solution is to simply keep track of which states you've already expanded, and never expand them again. More explicitly, maintain a "closed" set of expanded nodes while utilizing your search method of choice. Then, ensure that each node isn't already in the set before expansion and add it to the set after expansion if it's not. Tree search with this added optimization is known as **graph search**, and the pseudocode for it is presented below:

在状态空间图中无限循环搜索同一周期或多次访问同一节点

In BFS, for example, we shouldn't bother expanding the circled nodes (why?)



BFS是按层级顺序访问节点的，当一个节点被访问时，我们可能已经通过可能的最短路径到达了它，不可能通过更长路径找到更优的解。

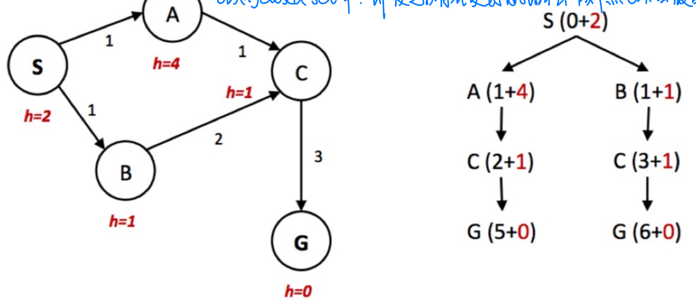
```
function GRAPH-SEARCH(problem, fringe) return a solution, or failure
  closed ← an empty set
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node ← REMOVE-FRONT(fringe)
    if GOAL-TEST(problem, STATE[node]) then return node
    if STATE[node] is not in closed then
      add STATE[node] to closed
      for child-node in EXPAND(STATE[node], problem) do
        fringe ← INSERT(child-node, fringe)
      end
  end
```

Note that in implementation, it's critically important to store the closed set as a disjoint set and not a list. Storing it as a list requires costs $O(n)$ operations to check for membership, which eliminates the performance improvement graph search is intended to provide. An additional caveat of graph search is that it tends to ruin the optimality of A*, even under admissible heuristics.

(4) Consistency / 一致性

Consider the following simple state space graph and corresponding search tree, annotated with weights and heuristic values:

h(A) >> h(B)，C先沿着次优路径S→B→C→G被扩展后，被加入到closed set中，即使之后存在更好的路径，该节点也不会被重新考虑



In the above example, it's clear that the optimal route is to follow $S \to A \to C \to G$, yielding a total path cost of $1+1+3=5$. The only other path to the goal, $S \to B \to C \to G$ has a path cost of $1+2+3=6$. However, because the heuristic value of node A is so much larger than the heuristic value of node B, node C is first expanded along the second, suboptimal path as a child of node B. It's then placed into the "closed" set, and so A* graph search fails to reexpand it when it visits it as a child of A, so it never finds the optimal solution. Hence, to maintain completeness and optimality under A* graph search, we need an even stronger property than admissibility, **consistency**. The central idea of consistency is that we enforce not only that a heuristic underestimates the *total* distance to a goal from any given node, but also the cost/weight of each edge in the graph. The cost of an edge as measured by the heuristic function is simply the difference in heuristic values for two connected nodes. Mathematically, the consistency constraint can be expressed as follows:

不仅要低估从任何给定节点到目标的总距离；还要低估图中每条边的权重
两个相连节点的启发值之差

$$\forall A, C \quad h(A) - h(C) \le cost(A, C)$$

Theorem. For a given search problem, if the consistency constraint is satisfied by a heuristic function h, using A* graph search with h on that search problem will yield an optimal solution.

Proof. In order to prove the above theorem, we first prove that when running A* graph search with a consistent heuristic, whenever we remove a node for expansion, we've found the optimal path to that node.

沿任何路径的节点f(n)值是非递减的

Using the consistency constraint, we can show that the values of $f(n)$ for nodes along any plan are nondecreasing. Define two nodes, n and n', where n' is a successor of n. Then:

$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + cost(n, n') + h(n') \\ &\ge g(n) + h(n) \\ &= f(n) \end{aligned}$$

If for every parent-child pair (n, n') along a path, $f(n') \ge f(n)$, then it must be the case that the values of $f(n)$ are nondecreasing along that path. We can check that the above graph violates this rule between $f(A)$ and $f(C)$. With this information, we can now show that whenever a node n is removed for expansion, its optimal path has been found. Assume towards a contradiction that this is false - that when n is removed from the fringe, the path found to n is suboptimal. This means that there must be some ancestor of n, n'', on the fringe that was never expanded but is on the optimal path to n. Contradiction! We've already shown that values of f along a path are nondecreasing, and so n'' would have been removed for expansion before n.

All we have left to show to complete our proof is that an optimal goal A will always be removed for expansion and returned before any suboptimal goal B. This is trivial, since $h(A) = h(B) = 0$, so

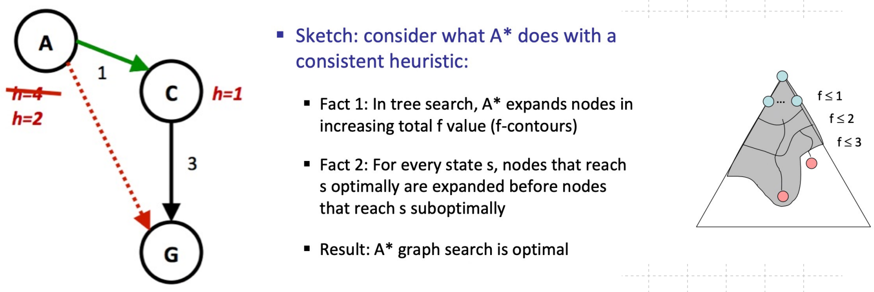
$$f(A) = g(A) < g(B) = f(B)$$

just as in our proof of optimality of A* tree search under the admissibility constraint. Hence, we can conclude that A* graph search is optimal under a consistent heuristic. □

A couple of important highlights from the discussion above before we proceed: for heuristics that are either admissible/consistent to be valid, it must by definition be the case that $h(G) = 0$ for any goal state G. Additionally, consistency is not just a stronger constraint than admissibility, consistency *implies* admissibility. This stems simply from the fact that if no edge costs are overestimates (as guaranteed by consistency), the total estimated cost from any node to a goal will also fail to be an overestimate.

没有边的成本被高估
⇒ 从任何节点到目标的总估计成本也不会是一个高估.

Consider the following three-node network for an example of an admissible but inconsistent heuristic:



The red dotted line corresponds to the total estimated goal distance. If $h(A) = 4$, then the heuristic is admissible, as the distance from A to the goal is $4 \geq h(A)$, and same for $h(C) = 1 \leq 3$. However, the heuristic cost from A to C is $h(A) - h(C) = 4 - 1 = 3$. Our heuristic estimates the cost of the edge between A and C to be 3 while the true value is $cost(A,C) = 1$, a smaller value. Since $h(A) - h(C) \nleq cost(A,C)$, this heuristic is not consistent. Running the same computation for $h(A) = 2$, however, yields $h(A) - h(C) = 2 - 1 = 1 \leq cost(A,C)$. Thus, using $h(A) = 2$ makes our heuristic consistent.

◦ Dominance

Now that we've established the properties of admissibility and consistency and their roles in maintaining the optimality of A* search, we can return to our original problem of creating "good" heuristics, and how to tell if one heuristic is better than another. The standard metric for this is that of **dominance**. If heuristic a is dominant over heuristic b, then the estimated goal distance for a is greater than the estimated goal distance for b for every node in the state space graph. Mathematically,

怎样判断一个启发式是否比另一个更好？

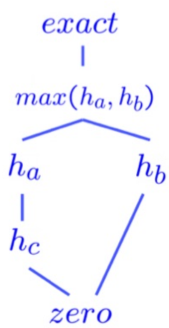
- h_a优于h_b 〈h_a总能更准确地估计从任何给定状态到目标的距离〉

  := 对于状态空间图的每个节点，

  a的估计目标距离都大于b的估计目标距离

$$\forall n : h_a(n) \geq h_b(n)$$

Dominance very intuitively captures the idea of one heuristic being better than another - if one admissible/consistent heuristic is dominant over another, it must be better because it will always more closely estimate the distance to a goal from any given state. Additionally, the **trivial heuristic** is defined as $h(n) = 0$, and using it reduces A* search to UCS. All admissible heuristics dominate the trivial heuristic. The trivial heuristic is often incorporated at the base of a **semi-lattice** for a search problem, a dominance hierarchy of which it is located at the bottom. Below is an example of a semi-lattice that incorporates various heuristics h_a, h_b, and h_c ranging from the trivial heuristic at the bottom to the exact goal distance at the top:



As a general rule, the max function applied to multiple admissible heuristics will also always be admissible. This is simply a consequence of all values output by the heuristics for any given state being constrained by the admissibility condition, $0 \leq h(n) \leq h^*(n)$. The maximum of numbers in this range must also fall in the same range. The same can be shown easily for multiple consistent heuristics as well. It's common practice to generate multiple admissible/consistent heuristics for any given search problem and compute the max over the values output by them to generate a heuristic that dominates (and hence is better than) all of them individually.

当我们有多个符合A*可接纳的启发式函数时，(i.e. 每个方法都不会高估实际从当前状态到目标状态的成本)，我们可以采取策略来创造一个新的启发式函数：对于Search Problem中的每个状态，我们比较每个启发式函数给出的成本估计值，并取这些值中的max值作为新启发式方法的估计值

∘ Heuristics for 8 Puzzle

Manhattan Distance Heuristic

A tile can move from square A to squre B, if A is adjacent to B.

- Given a particular state
  Consider every non-empty tile: calculate the Manhattan Distance between the current position of the tile and the goal position of the tile.
  Add this value for all the non-empty tiles together.
-

Initial State

5	3	
8	7	6
2	4	1

Goal State

1	2	3
4	5	6
7	8	

e.g. $\underset{\text{Tile1}}{4} + 3 + 1 + 2 + 2 + 0 + 2 + 2 = 16$

Misplaced Tile Heuristic

A tile can move from square A to squre B

- Given a particular state, count the number of non-empty tiles that are not in their goal positions.

i.e., if a tile is not in its goal position, we can move it to its goal position in one step.
上例中的"Misplaced Tile Heuristic Value"为7.

Gaschnig's Heuristic

- Any tile can be moved to the blank square directly, count the number of swaps.

e.g.

Goal State

1	2	3
8		4
7	6	5

Initial State

2	1	3
8		4
6	7	5

$h=0$

Initial State

6	2	8
	3	5
4	1	7

$h=8$

→

6	2	3
8		5
4	1	7

$h=7$

→

1	2	3
8		5
4	6	7

$h=4$

→

1	2	3
8		4
7	6	5

$h=0$

- Suggest a way to calculate Gaschnig's heuristic efficiently:

e.g.

Initial State

6	2	8
0	3	5
4	1	7

Goal State

1	2	3
8	0	4
7	6	5

Goal: $\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 3 & 6 & 2 & 8 & 5 & 7 & 1 & 4 & 0 \end{pmatrix}$

$\Leftrightarrow (0\ 3\ 8)(1\ 6)(2)(4\ 5\ 7)$ "Cycle Decomposition"

We calculate Gaschnig's heuristic by cyclic decomposition:

1. Select a square A which has not been reached yet.
2. Find the square A' where the tile (or blank) in square A should locate, then again find the location of tile in A'. Repeat until we reach the square A again.
3. Record the number of tile (if it is blank then we record B) reached in step 2 and build a circle γ. Then back to step 1 until all squares are reached.

Suppose that $\gamma_1, \cdots, \gamma_m$ are all the cycles generated
Let $S(\gamma) :=$ numbers of Gaschnig's moves needed for cycle γ.
$|\gamma| :=$ length of cycle γ.

Then: $S(\gamma) = \begin{cases} 0 & \text{if } |\gamma| = 1 \\ |\gamma|-1 & \text{if } |\gamma|>1 \text{ and Blank} \in \gamma \\ |\gamma|+1 & \text{if } |\gamma|>1 \text{ and Blank} \notin \gamma \end{cases}$ (需要额外的移动将Blank移入permutation并移出)

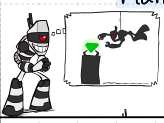
TOPIC **Constraint Satisfaction Problems (CSPs) / 约束满足问题**

◦ Intro

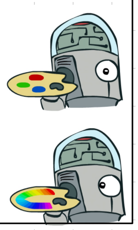
- 区分 Search Problems & CSPs

In the previous note, we learned how to find optimal solutions to search problems, a type of **planning problem**. Now, we'll learn about solving a related class of problems, **constraint satisfaction problems** (CSPs). Unlike search problems, CSPs are a type of **identification problem**, problems in which we must simply identify whether a state is a goal state or not, with no regard to how we arrive at that goal. CSPs are defined by three factors:

1. *Variables* - CSPs possess a set of N variables $X_1, ..., X_N$ that can each take on a single value from some defined set of values. Variables usually represent some quantities of abstractions that we try to reason about.
2. *Domain* - A set $\{x_1, ..., x_d\}$ representing all possible values that a CSP variable can take on. a set of values
3. *Constraints* - Constraints define restrictions on the values of variables, potentially with regard to other variables.

	Planning 规划	Identification 识别
	 Heuristics give problem-specific guidance.	
Key	The path (i.e. Sequences of actions) to the goal	The goal itself (i.e. assignments to variables), not the path.
Paths	Paths have various costs, depth.	All paths at the same depth (for some formulations)
	Standard Search Problems Assumptions about the world: • a single agent (You)	CSP := a specialised class of identification problems := a special subset of search problems
	• ∧ fully observable states State itself is a black box, the only things that you can do to a state are ".getSuccessor( );" & ".isGoal( );"	State (i.e. partial assignment) is defined by variables X_i with values from a domain D (sometimes D depends on i) <we can peek inside the state>
	• ∧ deterministic actions Successor function can be anything and Goal Test can be any function over states	Successor function is "assign a new variable" Goal Test is {constraints} specifying allowable combinations of values for subsets of variables.
	• ∧ discrete state space	
		Allows for useful general-purpose algorithms with more power than standard search algorithms.

- **Varieties of CSPs**
  - Discrete Variables
    - Finite domains
      - Size d means $O(d^n)$ complete assignments
      - E.g., Boolean CSPs, including Boolean satisfiability (NP-complete)
    - Infinite domains (integers, strings, etc.)
      - E.g., job scheduling, variables are start/end times for each job
      - Linear constraints solvable, nonlinear undecidable
  - Continuous variables
    - E.g., start/end times for Hubble Telescope observations
    - Linear constraints solvable in polynomial time by LP methods (see cs170 for a bit of this theory)



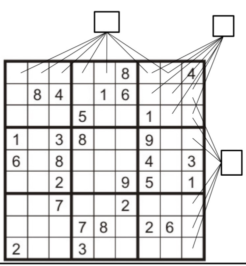
Varieties of Constraints

绝对约束（任何违反规则的都排除在解之外）
- *Unary Constraints* - Unary constraints involve a single variable in the CSP. They are not represented in constraint graphs, instead simply being used to prune the domain of the variable they constrain when necessary.
- *Binary Constraints* - Binary constraints involve two variables. They're represented in constraint graphs as traditional graph edges.
- *Higher-order Constraints* - Constraints involving three or more variables can also be represented with edges in a CSP graph, they just look slightly unconventional.

- Preferences (soft constraints): 偏好约束（哪些解更preferable）
  - E.g., red is better than green
  - Often representable by a cost for each variable assignment
  - Gives constrained optimization problems
  - (We'll ignore these until we get to Bayes' nets)

◦ CSP Example

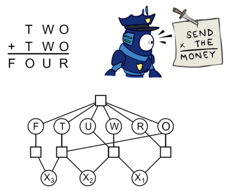
Example: Sudoku



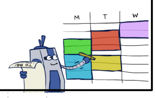
- Variables:
  - Each (open) square
- Domains:
  - {1,2,…,9}
- Constraints:
  - 9-way alldiff for each column
  - 9-way alldiff for each row
  - 9-way alldiff for each region
  - (or can have a bunch of pairwise inequality constraints)

Example: Cryptarithmetic

- Variables:
  $F\ T\ U\ W\ R\ O\ X_1\ X_2\ X_3$
- Domains:
  $\{0,1,2,3,4,5,6,7,8,9\}$
- Constraints:
  alldiff(F,T,U,W,R,O)
  $O+O=R+10\cdot X_1$
  ...



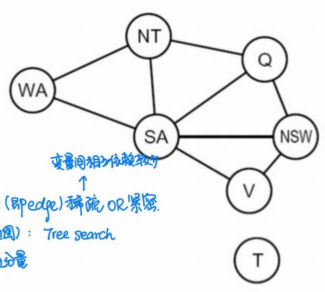
Real-World CSPs
- Scheduling problems: e.g., when can we all meet?
- Timetabling problems: e.g., which class is offered when and where?
- Assignment problems: e.g., who teaches what class
- Hardware configuration
- Transportation scheduling
- Factory scheduling
- Circuit layout
- Fault diagnosis
- … lots more!



Map Coloring { Given: a set of colors
Problem: Color a map that no two adjacent states (or regions) have the same color.

- CSPs are often represented as Constraint Graph / 约束图 { nodes := variables; edges := constraint between nodes.

The constraints in this problem are simply that no two adjacent states can be the same color. As a result, by drawing an edge between every pair of states that are adjacent to one another, we can generate the constraint graph for the map coloring of Australia as follows:



约束图反映出的信息:
- 连接度: 变量之间的连接(即edge)稀疏 OR 紧密. ↑ 变量间相互依赖较少
- { 树结构 (没有环的连通图): Tree search
  非树结构: 寻找弱连通分量
- 约束紧密度.
- ...

The value of constraint graphs is that we can use them to extract valuable information about the structure of the CSPs we are solving. By analyzing the graph of a CSP, we can determine things about it like whether it's sparsely or densely connected/constrained and whether or not it's tree-structured. We'll cover this more in depth as we discuss solving constraint satisfaction problems in more detail.

- Variables: WA, NT, Q, NSW, V, SA, T
- Domains: $D = \{\text{red}, \text{green}, \text{blue}\}$
- Constraints: adjacent regions must have different colors
  - Implicit: $WA \neq NT$
  - Explicit: $(WA, NT) \in \{(\text{red}, \text{green}), (\text{red}, \text{blue}), \ldots\}$
- Solutions are assignments satisfying all constraints, e.g.:
  - {WA=red, NT=green, Q=red, NSW=green, V=red, SA=blue, T=green}

∘ Solving CSPs

- Standard Search Formulation of CSPs
  - States defined by the values assigned so far (partial assignments)
    - Initial state: the empty assignment, {}
    - Successor function: assign a value to an unassigned variable
    - Goal test: the current assignment is complete and satisfies all constraints

- Method 1 (naïve) — 回溯搜索

Constraint satisfaction problems are traditionally solved using a search algorithm known as **backtracking search**. Backtracking search is an optimization on depth first search used specifically for the problem of constraint satisfaction, with improvements coming from two main principles:

1. Fix an ordering for variables, and select values for variables in this order. Because assignments are commutative (e.g. assigning $WA = Red$, $NT = Green$ is identical to $NT = Green$, $WA = Red$), this is valid.

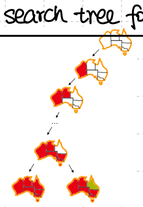
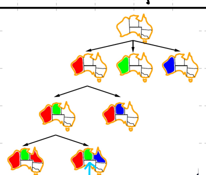
2. When selecting values for a variable, only select values that don't conflict with any previously assigned values. If no such values exist, backtrack and return to the previous variable, changing its value.

The pseudocode for how recursive backtracking works is presented below:

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment given CONSTRAINTS[csp] then
            add {var = value} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove {var = value} from assignment
    return failure
```

SELECT-UNASSIGNED-VARIABLE: 从尚未赋值的变量中选择一个变量 → 选什么? (按什么顺序选择?)
ORDER-DOMAIN-VALUES: 为选定的变量提供一个值的顺序. → 按什么顺序尝试value?
RECURSIVE-BACKTRACKING: 递归调用自身来为下一个变量赋值
if result ≠ failure then return result: 如果递归调用返回的不是失败, 则返回这个解
remove {var = value} from assignment: 如果在某一层给变量赋值失败, 则撤销这次赋值, 回溯到上一步.

partial search tree for DFS	partial search tree for Backtracking
	

1° 选择可能是给每个区域都涂上红色

forward checking 的power体现在这一块

Backtracking 是一种特殊的DFS: DFS 只有在涂完所有区域并检查颜色时, 才会意识到有些相邻区域颜色相同, 然后它不得不回溯并修改颜色配置; Backtracking 在给每个区域上色之前都会先检查这个颜色是否会违反约束. (若违反则尝试另一种颜色; 若所有颜色都不能满足条件, 则回溯到前一个区域, 给上一个区域尝试另一种颜色)

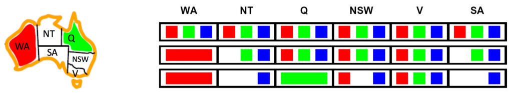
- Method2 — Filtering: Can we detect inevitable failure early?
  Keep track of domains for unassigned variables and cross off bad options.

Naive method for Filtering — Forward Checking / 向前检验

Cross off values that violate a constraint when added to the existing assignment.

(1) Idea: 回溯不会提前检查赋值为未来带来的影响，只有在当前赋值无以为继时回溯（是一种试错的方法）。Forward Checking 会检查当前赋值对尚未赋值的变量会有哪些影响. Whenever a value is assigned to a variable X_i, Forward Checking prunes the domains of the unassigned variables that share a constraint with X_i that would violate the constraint if assigned.

(2) e.g.



assign WA = red ⇒ the size of the domains for NT and SA decreases

assign Q = green ⇒ the size of the domains for NT, SA and NSW decreases

但是 Forward Checking 无法提前检测出矛盾，经过两次 Forward Checking 后，NT 和 SA 的值域都只剩蓝色但二者相邻.

⇒ We need to reason from constraint to constraint (constraint propagation)

Arc Consistency / 弧相容.

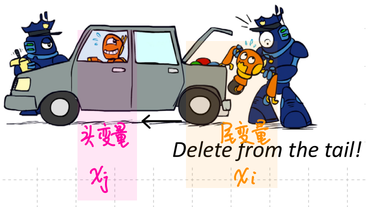
(1) Idea: 将 CSP 约束图中的每条 undirected edge (无向边) 理解为 two directed edges pointing in opposite directions (两条指向相反的有向边)
每一条这样的有向边叫作 arc (弧)

(2) Arc Consistency Algorithm:

An arc X → Y is consistent iff for *every* x in the tail there is *some* y in the head which could be assigned without violating a constraint

A simple form of propagation makes sure all arcs are simultaneously consistent:

- Begin by storing all arcs in the constraint graph for the CSP in a queue Q.
- Iteratively remove arcs from Q and enforce the condition that in each removed arc $X_i \longrightarrow X_j$, for every remaining value v for the tail variable X_i, there is at least one remaining value w for the head variable X_j such that $X_i = v, X_j = w$ does not violate any constraints. If some value v for X_i would not work with any of the remaining values for X_j, we remove v from the set of possible values for X_i.
- If at least one value is removed for X_i when enforcing arc consistency for an arc $X_i \longrightarrow X_j$, add arcs of the form $X_k \longrightarrow X_i$ to Q, for all unassigned variables X_k. If an arc $X_k \longrightarrow X_i$ is already in Q during this step, it doesn't need to be added again.
- Continue until Q is empty, or the domain of some variable is empty and triggers a backtrack. 队列 Q 为空或某个变量的域成为空集则触发回溯.



- 迭代处理.
  对于尾变量 x_i 所有可能的值 v，必须在头变量 x_j 中至少存在一个可能的值 w，使得 $x_i = v$, $x_j = w$ 的赋值不违反任何约束；
  如果对于 x_i 的某个值 v，没有任何 x_j 的值 w 能够满足约束，那么 v 就会被从 x_i 的可能值域中移除.
  ⇒ If this is the case，那么所有与 x_i 相连的其它未赋值变量 x_k，形如 $x_k \to x_i$ 的弧会被加入队列 Q；如果队列中已存在弧 $x_k \to x_i$，则无需再次添加.
  (尾 x_k, 头 x_i)

```
function AC-3(csp) returns the CSP, possibly with reduced domains
    inputs: csp, a binary CSP with variables {X1, X2, ..., Xn}
    local variables: queue, a queue of arcs, initially all the arcs in csp

    while queue is not empty do
        (Xi, Xj) ← REMOVE-FIRST(queue)
        if REMOVE-INCONSISTENT-VALUES(Xi, Xj) then
            for each Xk in NEIGHBORS[Xi] do
                add (Xk, Xi) to queue
```

tail head → e

ed

```
function REMOVE-INCONSISTENT-VALUES(Xi, Xj) returns true iff succeeds
    removed ← false
    for each x in DOMAIN[Xi] do
        if no value y in DOMAIN[Xj] allows (x,y) to satisfy the constraint Xi ↔ Xj
            then delete x from DOMAIN[Xi]; removed ← true
    return removed
```

处理的是变量的值域

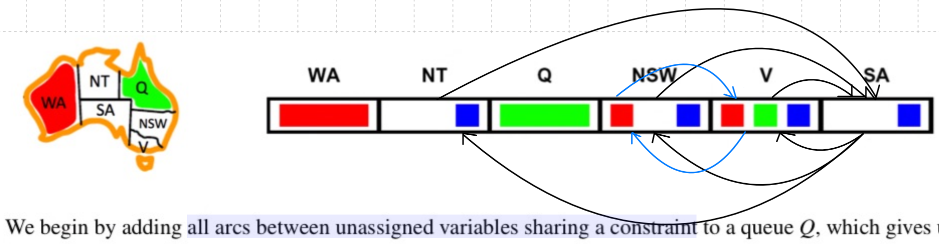
Remove-Inconsistant-Values 方法最差会被调用多少次？
起初 Q 中有所有的弧，每调用一次方法，就会从 Q 中删除一条弧
这相当于调用了 e 次方法；但同时我们也会将弧添加回 Q：
只有当 x_i 的域中的某一个值被删除时，我们才会将弧添加回 Q. (i.e. 只能按照域中值的数量添加 arcs) 因此在最差情况下，我们要将每个 arc 添加回队列 d 次
⇒ 方法最多被调用 e+ed 次

每次从队列中取出一个弧进行检查时，最坏情况下可能要考虑 d 个可能的值
↓
对于每个可能的值，可能需要检查与之相连的变量的所有 d 个可能值来确保一致性.

→

The AC-3 algorithm has a worst case time complexity of $O(ed^3)$, where e is the number of arcs (directed edges) and d is the size of the largest domain. Overall, arc consistency is more holistic of a domain pruning technique than forward checking and leads to fewer backtracks, but requires running significantly more computation in order to enforce. Accordingly, it's important to take into account this tradeoff when deciding which filtering technique to implement for the CSP you're attempting to solve.

(3) e.g.



We begin by adding all arcs between unassigned variables sharing a constraint to a queue Q, which gives us

$$Q = [\boxed{SA \to V}, \boxed{V \to SA}, SA \to NSW, NSW \to SA, SA \to NT, NT \to SA, V \to NSW, NSW \to V]$$

tail head
SA→V OK 即删
V→SA 删除V中blue, 即删

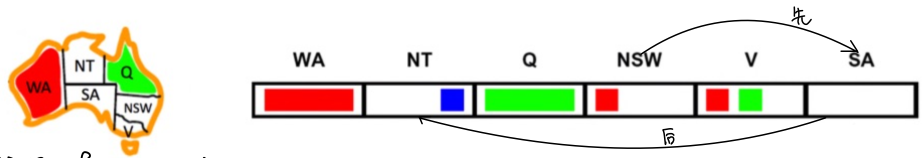
For our first arc, $SA \to V$, we see that for every value in the domain of SA, $\{blue\}$, there is *at least* one value in the domain of V, $\{red, green, blue\}$, that violates no constraints, and so no values need to be pruned from SA's domain. However, for our next arc $V \to SA$, if we set $V = blue$ we see that SA will have no remaining values that violate no constraints, and so we prune *blue* from V's domain.

所有以V为head的弧加入Q. 即SA→V, NSW→V. 由于NSW→V已在Q中, 所以只需加入SA→V

Because we pruned a value from the domain of V, we need to enqueue all arcs with V at the head - $SA \to V$, $NSW \to V$. Since $NSW \to V$ is already in Q, we only need to add $SA \to V$, leaving us with our updated queue

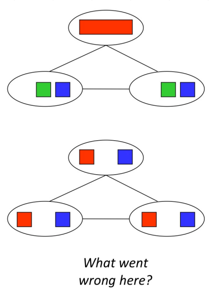
$$Q = [SA \to NSW, NSW \to SA, SA \to NT, NT \to SA, V \to NSW, NSW \to V, \boxed{SA \to V}]$$

We can continue this process until we eventually remove the arc $SA \to NT$ from Q. Enforcing arc consistency on this arc removes *blue* from SA's domain, leaving it empty and triggering a backtrack. Note that the arc $NSW \to SA$ appears before $SA \to NT$ in Q and that enforcing consistency on this arc removes *blue* from the domain of NSW.



(4) Limitations of arc consistency.

- After enforcing arc consistency:
  - Can have one solution left
  - Can have multiple solutions left
  - Can have no solutions left (and not know it)
- Arc consistency still runs inside a backtracking search!



(5) Improvement: K-Consistency (K-相容)

Arc Consistency is a subset of a more generalized notion of K-consistency

Increasing degrees of consistency

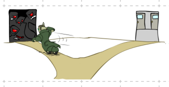
- 1-Consistency (Node Consistency): Each single node's domain has a value which meets that node's unary constraints
- 2-Consistency (Arc Consistency): For each pair of nodes, any consistent assignment to one can be extended to the other
- K-Consistency: For each k nodes, any consistent assignment to k-1 can be extended to the k^{th} node.

对于其中k-1个节点组成的子集的赋值都能保证第k个节点能至少有一个满足相容的赋值. higher K is more expensive to compute

- **Method 3 — Ordering/排序**

在实践中，动态选择（on the fly）下一个变量及其对应的值通常比固定顺序更有效.

最小剩余值（MRV）

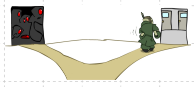


Minimum Remaining Values

MRV选择下一个变量时，会选择剩余有效值最少的unassigned variable.（即最受限制的变量）. 直觉上讲，最受限制的变量最有可能早早用完可用的值，导致回溯.

- "fail fast" ordering

最少约束值（LCV）



Least Constraining Value

在为变量选择具体的值时，select the value that prunes the fewest values from the domains of the remaining unassigned values. 选择能从'剩余未分配的值'的域中剪除最少value的值. 也就是说，这个值在满足当前变量的约束的同时，尽量不要排除其它变量的过多可能性.

- 需要额外计算，因为要评估每个可能的值对其余变量可能取值的影响（running arc consistency / forward checking or other filtering methods for each value to find LCV）但如果正确使用，可减少未来的回溯，总体上提高算法速度

- **Method 4 — Exploit Structure**

(1) Idea

- Extreme case: independent subproblems
  - Example: Tasmania and mainland do not interact
- Independent subproblems are identifiable as connected components of constraint graph
- Suppose a graph of n variables can be broken into subproblems of only c variables:
  - Worst-case solution cost is $O((n/c)(d^c))$, linear in n
  - E.g., n = 80, d = 2, c =20
  - 2^{80} = 4 billion years at 10 million nodes/sec
  - $(4)(2^{20})$ = 0.4 seconds at 10 million nodes/sec

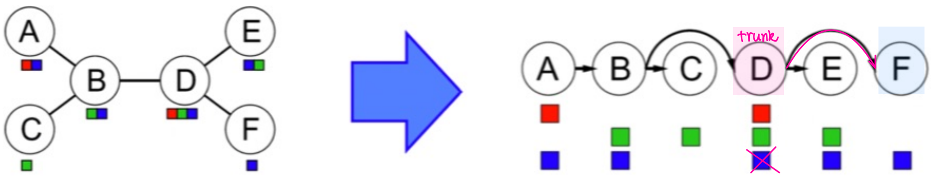
(2) e.g. Tree-Structured CSP, one that has no loop in its constraint graph

- First, pick an arbitrary node in the constraint graph for the CSP to serve as the root of the tree (it doesn't matter which one because basic graph theory tells us any node of a tree can serve as a root).
- Convert all undirected edges in the tree to directed edges that point *away* from the root. Then **linearize** (or **topologically sort**) the resulting directed acyclic graph. In simple terms, this just means order the nodes of the graph such that all edges point rightwards. Noting that we select node A to be our root and direct all edges to point away from A, this process results in the following conversion for the CSP presented below:

将所有无向边转换为从根节点指向外部的有向边

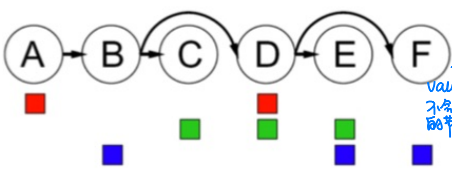
（或称"拓扑排序"）

对有向无环图进行线性化处理



Remove Backward (make the arc pointing to node X_i consistent)

- Perform a **backwards pass** of arc consistency. Iterating from $i = n$ down to $i = 2$, enforce arc consistency for all arcs $Parent(X_i) \longrightarrow \boxed{X_i}$. For the linearized CSP from above, this domain pruning will eliminate a few values, leaving us with the following:



a → b → c →

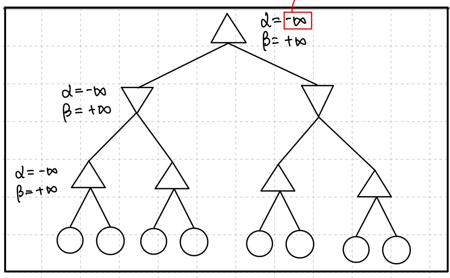
我们总是最先删除"右边"节点中的 unconsistent values，所以随着 propagation 逐渐向左，我们再也不会再去删除"右边"节点中的任何值.（成为truck的节点逐渐左移）

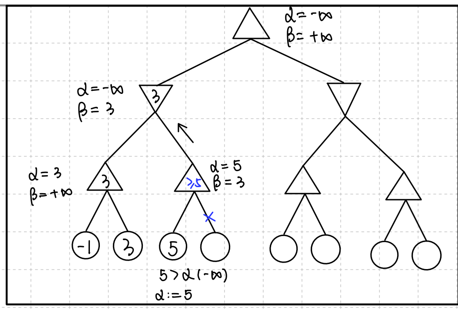
Assign Forward

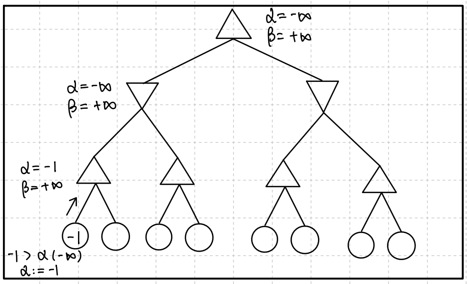
- Finally, perform a **forward assignment**. Starting from X_1 and going to X_n, assign each X_i a value consistent with that of its parent. Because we've enforced arc consistency on all of these arcs, no matter what value we select for any node, we know that its children will each all have at least one consistent value. Hence, this iterative assignment guarantees a correct solution, a fact which can be proven inductively without difficulty.

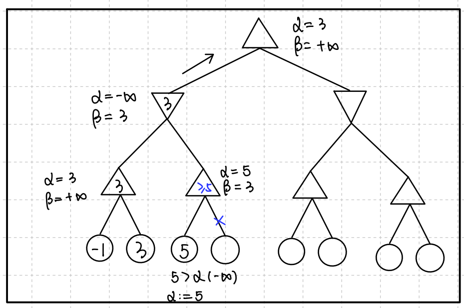
min-max is depth-first search.

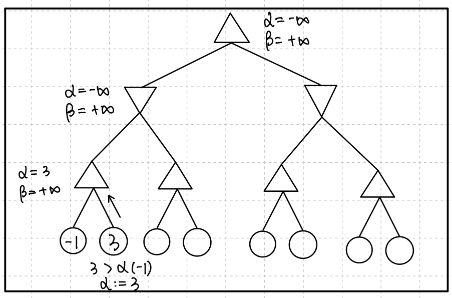
the worst that the maximizer is willing to accept

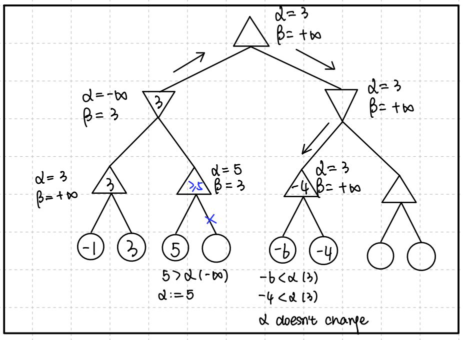


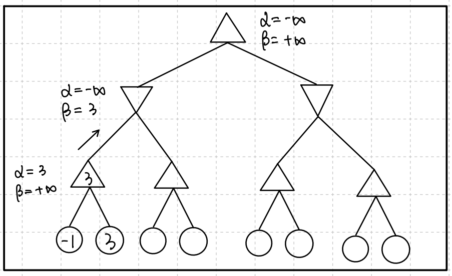


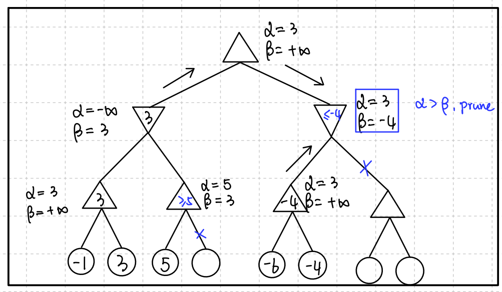




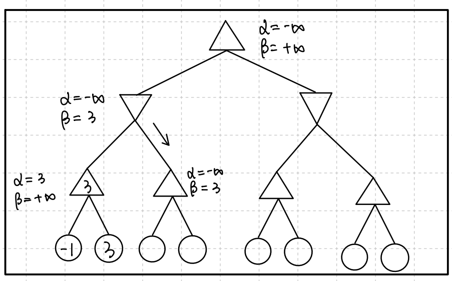


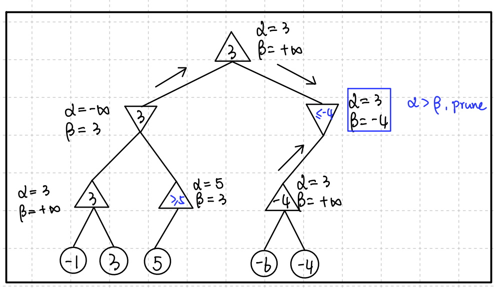




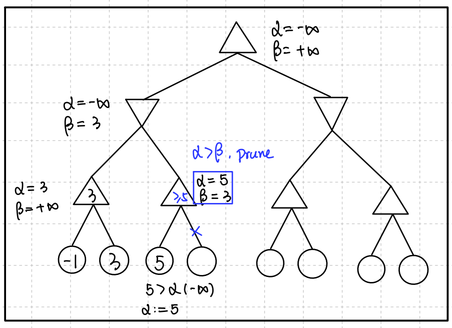


α > β, prune





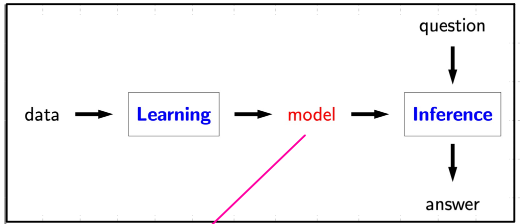
α > β, prune



△ 的叶子节点帮助更新 A 的 α 值. A 的实际value 帮助更新 A 的父节点 ▽ 的 β 值 (▽ 的 β 永远是当前路径的最小值)

若 A 当前 α > β. 由于 β 只能是由 A 的父节点传递下来 } 说明之前 ∃ already a better option

α 是 A 的叶子节点更新的



We should think of KB as carving out a set of models.

Knowledge Base (知识库) := { formulas }

Interface

new sentence asks : questions about what is known.

Ask(α) → KB →
- Yes entailment, $KB \models \alpha$
- No contradiction, $KB \models \neg\alpha$
- I don't know, Contingent.

Tell(α) → KB →
- Already knew that : entailment, $KB \models \alpha$
- Don't believe that : contradiction, $KB \models \neg\alpha$
- Learned something new : update KB Contingent.

Formulas by themselves are just symbols. No meaning (semantics) yet.

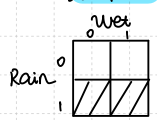
- Ingredients of a logic
  - operates on
  - (1) Syntax — defines a set of valid formulas, just about symbol pushing
  - (2) Semantics — For each formula, specify a set of models $\mathcal{M}(f)$ (assignments / configurations of the world)
  - (3) Inference Rules — Given f, what new formulas g can be added that are guaranteed to follow $\left(\frac{f}{g}\right)$?

- Two goals of a logic language { $\boxed{\text{Represent}}$ knowledge about the world; $\boxed{\text{Reason with}}$ that knowledge }

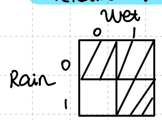
- Intuition : Take a set of formulas ⇒ get a Knowledge Base KB. ⇒ $\mathcal{M}(KB) = \underbrace{\bigcap_{f \in KB} \mathcal{M}(f)}$
  the set of all worlds satisfying these constraints.
  - Each formula = A fact that we know
  - A Formula represents a set of models.

  e.g. Consider two formulas : Rain , Rain → Wet ⇒ KB = {Rain , Rain → Wet} ⇒ $\mathcal{M}(KB)$

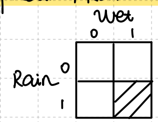
$\mathcal{M}$(Rain)



$\mathcal{M}$(Rain → Wet)



$\mathcal{M}$({Rain , Rain → Wet})



A Model (in the logical sense) represents a possible state of affairs in the world.

- Adding more formulas to KB shrinks the set of models.
  KB → KB ∪ {f} $\mathcal{M}(KB) \to \mathcal{M}(KB) \cap \mathcal{M}(f)$
  相当于 impose another constraint on our world.

Knowledge-Based Systems 知识系统
Start Point : a large KB representing explicit knowledge
↓
Using Inference Methods
↓
Implicit Knowledge

Deductive Inference 演绎推理 is a process : Given : KB, a query α
$\boxed{\text{Compute : whether } KB \models \alpha}$

Correct : For every derivable α : $KB \models \alpha$.

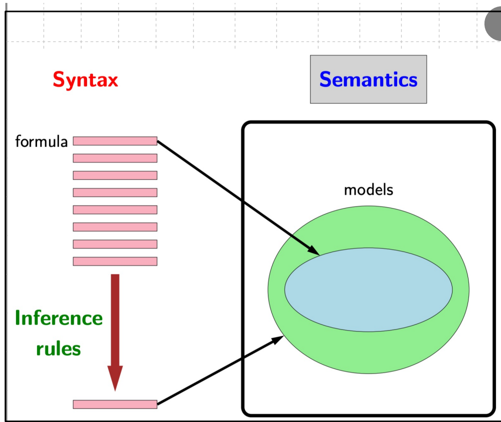
Complete : For every α for which $KB \models \alpha$ holds is derivable

It suffices to do :
(1) Transform ($KB \wedge \neg\alpha$) into CNF (即检查是否有可能出现KB为真，α为假的情况，不使用 $\neg KB \vee \alpha$.)
(2) Test whether this CNF is satisfiable.
不是检查时关注的重点

TOPIC

DATE

NOTE INDEX



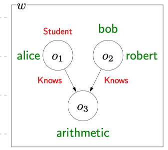
Informal : A model represents a possible situation in the world.

Model in First-Order Logic can be thought of a Graph Representation (In case of unary and binary predicates)

Labeld Nodes := Objects in the world ($o_1, o_2, \cdots$), labeled with constant symbols.

Labeld Edges := Predicates that take two arguments, labeled with predicate symbols.

(* Predicates that take one argument can be visualized as additional node labels, but not so important)



Formal :

Model in First-Order Logic :

Model w maps constant symbols to objects $w(\text{alice}) = o_1$, $w(\text{bob}) = o_2$, $w(\text{arithmetic}) = o_3$

predicate symbols to tuples of objects $w(\text{Knows}) = \{(o_1, o_3), (o_2, o_3) \cdots\}$

(3) Semantics

$I = \langle D, \Phi \rangle$ is an Interpretation

D := Universe of the interpretation / Domain of the discourse (论域)

- A non-emty (infinite or finite) set of arbitrary elements e.g. mathematical objects, students, tables, sentences, etc.

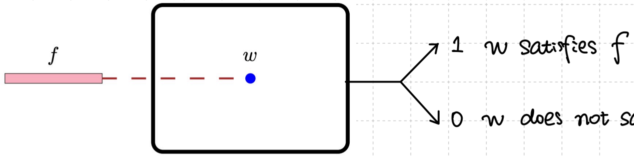
Φ := an Interpretation Function

defined on the sets of Constants, Predicates and Functions such that :

For each $c \in \mathcal{C}$, $\Phi(c) \in D$

For each n-ary Predicate Symbol P, $\Phi(P) : D^n \to \{\text{true}, \text{false}\}$ $\Phi(P) \subseteq D \times \cdots \times D$ an n-ary relation over D

For each n-ary Function Symbol f, $\Phi(f) : D^n \to D$ $\Phi(f) \subseteq [\underbrace{D \times D \times \cdots D}_{n \text{ times}} \to D]$ an n-ary function over D



The semantics is given by an interpretation function, which takes a formula f and a model w, $\Rightarrow$ An interpretation function connects syntax & semantics
and returns whether w satisfies f. (i.e. is f true in w?)

An interpretation is an inference from a specific point of view.

Two people might have the same facts, but with different points of view, they may come to a different solution to the problem.

TOPIC
DATE
NOTE
INDEX

Satisfaction Relation

Variable Assignment

Variables — elements of the universe of Interpretation

Given an Interpretation $I=\langle D,\Phi\rangle$, a (variable) assignment is a mapping $\mu: V \to D$.

- 在给定的$I=\langle D,\Phi\rangle$下，一个"项"t的解释是由Φ分配给它的D中的元素. 写作 $I\|t\|$，读作 "I of t"

  The denotation of a term := the element of D assigned by $I=\langle D,\Phi\rangle$

  如果这个项t包含自由变元，那么它的解释还取决于这些自由变元的赋值. 写作 $I, v\|t\|$, where $v \in [\text{Variables} \to D]$ defines a variable map.

  The denotation of a term with free variables

- Rules:

  $I, v\|x\| = v(x)$ (for every x)　在解释I和变量赋值v下，变量x的解释就是v对x的赋值.

  $I, v\|\underbrace{f(t_1,\cdots,t_n)}_{\text{函数}f\text{应用于项}t_1,\cdots,t_n\text{的解释}}\| = H(d_1,\cdots,d_n)$, where $H=\Phi(f)$ and $d_i = \underline{I, v\|t_i\|}$ (recursively)

  H是函数f在Φ下的对应函数

  d_i是在解释I和变量赋值v下，变量t_i的解释

Satisfaction Relation in FOL

- An Interpretation $I=\langle D,\Phi\rangle$ satisfies a formula α iff for any assignment $\mu: V \to D$, $\Phi_\mu(\alpha) = \text{true}$.

  一个解释$I=\langle D,\Phi\rangle$满足一个公式α，当且仅当 将任何变量赋值$\mu: V \to D$应用到公式α上时，得到的结果总为真.

  写作: $I, v \models \alpha$ "α is satisfied by I and v"　Recap: $v \in [\text{Variables} \to D]$ defines a variable map.

  $$\begin{cases} I \models \alpha & \text{if } \alpha \text{ is a sentence} \\ I \models S & \text{if } S \text{ is a set of sentences} \end{cases}$$

Rules:

- $I, v \models P(t_1,\cdots,t_n)$ iff $\langle d_1,\cdots,d_n\rangle \in R$, where $R=\Phi(P)$ and $d_i = I, v\|t_i\|$

  An atomic sentence Predicate($\text{term}_1, \cdots, \text{term}_n$) is true
  iff the domain elements that are the interpretations of $\text{term}_1, \cdots, \text{term}_n$
  are in the relation that is the interpretation of Predicate.

- $I, v \models (t_1 = t_2)$ iff $I, v\|t_1\|$ equals $I, v\|t_2\|$
- $I, v \models \neg\alpha$ iff $I, v \not\models \alpha$
- $I, v \models (\alpha \wedge \beta)$ iff $I, v \models \alpha$ and $I, v \models \beta$　　$\alpha[x/t] := \alpha$ with all free occurances of x replaced by t. 写作 α^x_t
- $I, v \models (\alpha \vee \beta)$ iff $I, v \models \alpha$ or $I, v \models \beta$
- $I, v \models \exists x\,\alpha$ iff for some $d \in D$, $I, v^x_d \models \alpha$, where v^x_d is like v except that $v^x_d(x) = d$

  $\exists x\alpha$ is true in a model
  iff there is a domain element d in the model such that:
  α is true in the model when x is interpreted by d.
  $\exists x\alpha$在模型中为真，指在模型的论域中存在一个元素d，使得当我们将x解释为d时，谓词$P(x)$为真

- $I, v \models \forall x\,\alpha$ iff for all $d \in D$, $I, v^x_d \models \alpha$

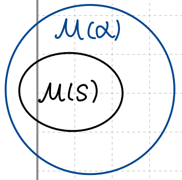
* Rules tell us how the truth value of a sentence depends on the meaning of the nonlogical symbols.
但是并非所有sentences之间的逻辑关联都仅依赖于 → e.g. If α is satisfied by I, then $\neg(\beta \wedge \neg\alpha)$ is true independent of why α is true and what β is.

TOPIC

DATE

NOTE　INDEX

(4) Logical Consequences 逻辑蕴涵 或 Entailment 语义蕴含



S implies/entails α (or α is a logical consequence of S): $S \models \alpha$ iff for all I, if $I \models S$ then $I \models \alpha$

System does NOT have access to the interpretation of the nonlogical symbols as intended by the user! So Use Implication!!

- (In other words) for all I, $I \not\models S \cup \{\neg\alpha\}$
  or $S \cup \{\neg\alpha\}$ is unsatisfiable.
- Note: ① $\{\alpha_1, \cdots, \alpha_n\} \models \alpha$ iff $\models \underbrace{(\alpha_1 \wedge \cdots \wedge \alpha_n) \supset \alpha}_{\text{命题}}$ 将 finite implication 归约到 validity.
  命题 ↓ 在所有可能情况下都是真的，即它是有效的
  即 没有任何一种情况能使 $\alpha_1, \cdots, \alpha_n$ 同时为真而 α 不为真. 因此 $\{\alpha_1, \cdots, \alpha_n\}$ 蕴含 α.

  ② 逻辑符号在逻辑的每个模型中都有固定的解释，但非逻辑符号并非如此. 它们的解释在不同模型中有所不同 (所以说 model 是逻辑系统中 nonlogical symbol 的特定解释)
  在我们的日常理解中，$Dog(fido) \models Mammal(fido)$. 然而在逻辑系统中，除非我们已经定义过 $Dog(x)$ 和 $Mammal(x)$ 的关系，否则 $\Phi(Dog) \not\subset \Phi(Mammal)$
  修改: $\forall x\,[Dog(x) \supset Mammal(x)]$ is represented in S Then: $S \cup \{Dog(fido)\} \models Mammal(fido)$

  ③ $KB \models \alpha$ → a consequence of one's believes $\qquad \underbrace{\{\alpha \mid KB \models \alpha\}}_{\text{implicit knowledge}}$
  KB: explicit knowledge

A clause is a disjunction 析取 of literals

A CNF is a conjunction of Clauses

(5) Convert "Propositional wff α" to an "α' in $\boxed{\text{Conjunctive Normal Form (CNF)}}$" such that $\models \alpha \equiv \alpha'$
合取范式

- Notation
  $((p \vee \neg q) \wedge r \wedge s)$ is represented as $\{[p, \sim q], [r], [s]\}$

  [clause] [ ] empty clause, represents FALSE
  {formulas} { } empty formula, represents TRUE (没有子句可以使得表达式为假)

- **Conversion rules:**
  - Eliminate $\leftrightarrow$: $\dfrac{f \leftrightarrow g}{(f \to g) \wedge (g \to f)}$
  - Eliminate $\to$: $\dfrac{f \to g}{\neg f \vee g}$ $\quad f \supset g$
  - Move $\neg$ inwards: $\dfrac{\neg(f \wedge g)}{\neg f \vee \neg g}$ $\qquad \dfrac{\neg \forall x \alpha}{\exists x \neg\alpha}$
  - Move $\neg$ inwards: $\dfrac{\neg(f \vee g)}{\neg f \wedge \neg g}$
  - Eliminate double negation: $\dfrac{\neg\neg f}{f}$
  - Distribute $\vee$ over $\wedge$: $\dfrac{f \vee (g \wedge h)}{(f \vee g) \wedge (f \vee h)}$
  - Rename variables to make them syntactically correct
    e.g. $\dfrac{\exists x\,[P(x)] \wedge Q(x)}{\exists z\,[P(z)] \wedge Q(x)}$, z a new variable
  - Move $\forall$'s to the left. Universal quantifiers on variables are simply omitted as understood
    e.g. $\dfrac{\alpha \wedge \forall x \beta}{\forall x\,[\alpha \wedge \beta]}$

- Existentially-quantified variables are replaced by Skolem Functions 存在量化的变量被替换为"斯科伦范式"

Skolem 化的本质是对如下形式的公式的观察

$$\forall x_1 \ldots \forall x_n \exists y R(x_1, \ldots, x_n, y)$$

它在某个模型中是可满足的，在这个模型必定对于所有的

$$x_1, \ldots, x_n$$

有某些点 y 使得

$$R(x_1, \ldots, x_n, y)$$

为真，并且必定存在某个函数(选择函数)

$$y = f(x_1, \ldots, x_n)$$

使得公式

$$\forall x_1 \ldots \forall x_n R(x_1, \ldots, x_n, f(x_1, \ldots, x_n))$$

为真。函数 f 叫做 Skolem 函数。

e.g. $\exists x \forall y\, R(x, y) \Rightarrow \forall y\, R(a, y)$ $\quad a$ 为常数
$\forall x \exists y\, R(x, y) \Leftrightarrow \forall x\, R(x, f(x))$
$\forall x \forall y \exists z\, R(x, y, z) \Leftrightarrow \forall x \forall y\, R(x, y, f(x, y))$

What is "Skolemization" and why is it ok to work with skolemized formulas when applying Resolution? 2 Punkte

Skolemization replaces every existentially quantified variable x by a new function symbol whose arguments are the universally quantified variables, in whose scope x appears. Skolemization is ok because a formula is satisfiable iff its skolemized version is satisfiable, and Resolution is a test for satisfiability.

TOPIC Semantic Entailment

DATE . .

1. Formalizing argument validity: Semantic Entailment

Let $\Sigma = \{p_1, p_2, ..., p_n\}$ be a set of premises and let α be the conclusion that we want to derive.

Σ *semantically entails* α, denoted $\Sigma \vDash \alpha$, if and only if

/ If every model of $\{P_1, \cdots, P_n\}$ is also a model of α

- Whenever all the premises in Σ are true, then the conclusion α is true.
- For any truth valuation t, if every premise in Σ is true under t, then the conclusion α is true under t.
- For any truth valuation t, if t satisfies Σ (denoted $\Sigma^t = \mathrm{T}$), then t satisfies α ($\alpha^t = \mathrm{T}$).
- $(p_1 \wedge p_2 \wedge ... \wedge p_n) \rightarrow \alpha$ is a tautology.

If Σ semantically entails α, then we say that the argument (with the premises in Σ and the conclusion α) is valid.

In general, the question of entailment for first-order logic is only semi-decidable, that is, algorithms exist that say yes to every entailed sentence, but no algorithm exists that also says no to every non-entailed sentence. Resolution is an example of such an algorithm.

存在算法能对每一个由KB确实推导出的语句说"是"
但不存在算法能对每一个非由KB推导出的语句说"不是".
i.e. Resolution能够验证KB推导出的语句,
但不能总是确认一个语句是否不被推导.

- Resolution does not always terminate for first-order logic.
- If KB ⊨ S, then resolution will eventually produce EMPTY (= FALSE).
- If KB ⊭ S, then resolution will not produce EMPTY but might not terminate.
- Thus, if resolution has not terminated after x hours (for any given x), one does not know whether it will eventually produce EMPTY or not. One can't just wait and see either since it might run forever.

2. Proving or disproving Entailment

(1) Proving Σ entails α, denoted $\Sigma \vDash \alpha$

I. ▪ Using a truth table: Consider all rows of the truth table in which all of the formulas in Σ are true. Verify that α is true in all of these rows.

e.g. Let $\Sigma = \{(\neg(p \wedge q)), (p \rightarrow q)\}$, $x = (\neg p)$, and $y = (p \leftrightarrow q)$. Based on the truth table, which of the following statements is true?

a. $\Sigma \vDash x$ and $\Sigma \vDash y$.
b. $\Sigma \vDash x$ and $\Sigma \nvDash y$.
c. $\Sigma \nvDash x$ and $\Sigma \vDash y$.
d. $\Sigma \nvDash x$ and $\Sigma \nvDash y$.

p	q	$(\neg(p \wedge q))$	$(p \rightarrow q)$	$x = (\neg p)$	$y = (p \leftrightarrow q)$
0	0	1	1	1	1
0	1	1	1	1	0
1	0	1	0	0	0
1	1	0	1	0	1

Using truth tables to prove entailment does NOT work for FOL.

使用真值表来证明一阶逻辑的蕴含关系（entailment）不可行，主要原因归结于一阶逻辑的特性和真值表的局限性。以下是几个关键点详细说明为什么真值表方法不适用于一阶逻辑的蕴含证明：

1. **无限域的处理**：一阶逻辑允许量化论域（quantification domain）是无限的，比如自然数、实数等。真值表方法是通过枚举所有可能的真值组合来检验逻辑表达式的有效性，但在面对无限论域时，这种枚举方法无法实施，因为无法生成一个无限大的真值表来覆盖所有可能的情况。
2. **变量和量化的复杂性**：一阶逻辑包括对变量的量化，既包括全称量化（∀，表示“对所有”）和存在量化（∃，表示“存在”）。真值表方法无法直接应对这种量化操作，因为它主要是针对命题逻辑设计的，命题逻辑中的语句没有变量的概念，每个命题要么为真要么为假，而一阶逻辑中的语句真假值可能依赖于量化的变量。
3. **结构的复杂性**：一阶逻辑能够表达更复杂的结构和关系，例如可以表达属性、关系以及这些关系和属性之间的函数。这种复杂性意味着单纯通过检查语句的真假值不足以捕捉语句间的逻辑蕴含关系，因为这些关系可能依赖于具体的对象和它们之间的结构关系。
4. **逻辑推导的深度**：一阶逻辑的推导不仅仅依赖于语句的真值，还涉及到如何通过逻辑规则从前提推导出结论。这种推导过程在一阶逻辑中可能非常复杂，需要考虑的不仅是个别语句的真值，而是它们如何通过逻辑规则相互作用。真值表方法缺乏展现和处理这种逻辑推导过程的能力。

总结来说，真值表方法在命题逻辑中是非常有用的，因为它可以通过枚举所有可能的情况来验证逻辑表达式的有效性。然而，由于一阶逻辑的这些复杂性和特性，真值表方法不足以处理一阶逻辑中的蕴含关系证明，需要采用更为复杂的方法，如自然演绎、序列表或者其他逻辑推导技术。

II. ▪ Direct proof: For every truth valuation under which all of the premises are true, show that the conclusion is also true under this valuation.

e.g. (i) Semantic proof of $\dfrac{\Sigma \vDash A \rightarrow B,\ \Sigma \vDash A}{\Sigma \vDash B} \rightarrow e$

Suppose that $I \vDash \Sigma$,

because of the two assumptions, we have $I \vDash A \rightarrow B$ and $I \vDash A$

By definition, the statement $I \vDash A \rightarrow B$ means that

$I \vDash B$ whenever $I \vDash A$, so $I \vDash B$

(ii) Formalize "*Norma Jeane Baker is a daughter of Marilyn Monroe's parents.*" and "*Norma Jeane is not a sister of Marilyn.*" together with the needed background knowledge on relationships as first-order sentences. Provide a semantical[2] proof that "*Norma Jeane is Marilyn.*" is an entailment thereof.

$I = \langle D, \Phi \rangle$

Φ: Constants: $\Phi(NJ) \in D \quad \Phi(Mar) \in D$

Functions: $\Phi(Parent) \subseteq D \rightarrow D$

Predicate: $\Phi(Sister) \subseteq D \times D$

$\Phi(Daughter) \subseteq D \times D$

$\Phi(Female) \subseteq D$

$I, v \|x\| = v(x)$ (for every x) 在解释 I 和变量赋值 v 下，变量 x 的解释就是 v 对 x 的赋值.

$\underbrace{I, v \|f(t_1, \cdots, t_n)\|}_{\text{函数}f\text{应用于项}t_1,\cdots,t_n\text{的解释}} = H(d_1, \cdots, d_n)$, where $H = \Phi(f)$ and $d_i = \underline{I, v \|t_i\|}$ (recursively)

H 是函数 f 在 Φ 下的对应函数

d_i 是在解释 I 和变量赋值 v 下，变量 t_i 的解释

Premises: ① $Daughter(NJ, Parent(Mar))$

② $\neg Sister(NJ, Mar)$

③ $Daughter(x, y) \equiv Female(x) \wedge y = Parent(x) \wedge y \neq x$

④ $Sister(x, y) \equiv Female(x) \wedge Parent(x) = Parent(y) \wedge y \neq x$

$KB = \{①, ②, ③, ④\}$

Claim: $KB \vDash (NJ = Mar)$

Proof: Let $\alpha = \Phi(NJ)$, $\beta = \Phi(Mar)$

$\gamma: I \| Parent(Mar) \| = \Phi(Parent(\Phi(Mar))) = \Phi(Parent(\beta))$ ⑤

$I \vDash ①$ $\langle \alpha, \gamma \rangle \in \Phi(Daughter)$ ⑥

$I \vDash ②$ $\langle \alpha, \beta \rangle \notin \Phi(Sister)$ ⑦

$I \vDash ③$ $\langle \alpha, \alpha' \rangle \in \Phi(Daughter)$ iff $\alpha \in \Phi(Female) \wedge \alpha' = \Phi(Parent(\alpha)) \wedge \alpha \neq \alpha'$ ⑧

$I \vDash ④$ $\langle \alpha, \alpha' \rangle \in \Phi(Sister)$ iff $\alpha \in \Phi(Female) \wedge \Phi(Parent(\alpha)) = \Phi(Parent(\alpha')) \wedge \alpha \neq \alpha'$ ⑨

From ⑥, ⑧: $\langle \alpha, \gamma \rangle \in \Phi(Daughter)$

$Female(\alpha) \wedge \gamma = Parent(\alpha) \wedge \alpha \neq \gamma$ ⑩

From ⑦: $\langle \alpha, \beta \rangle \notin \Phi(Sister)$

$\neg(Female(\alpha) \wedge Parent(\alpha) = Parent(\beta) \wedge \alpha = \beta)$

$\underbrace{\neg Female(\alpha)}_{\text{does not hold}} \vee Parent(\alpha) \neq Parent(\beta) \vee \alpha \neq \beta$

(iv)

1. α is valid if and only if TRUE $\models \alpha$.
2. For any α, FALSE $\models \alpha$.
3. $\alpha \models \beta$ if and only if the sentence $\alpha \supset \beta$ is valid.
4. α and β are equivalent[1] if and only if the sentence $\alpha \equiv \beta$ is valid.
5. $\alpha \models \beta$ if and only if the sentence $\alpha \wedge \neg\beta$ is unsatisfiable.

Can prove this with truth tables

$\alpha \equiv \beta$ iff $\alpha \Rightarrow \beta$ and $\beta \Rightarrow \alpha$

(Proof by contradiction)

[1]For any sentence ϕ, let $Mod(\phi) = \{I \mid I$ is an interpretation such that $I \models \phi\}$. Two sentences α and β are equivalent iff $Mod(\alpha) = Mod(\beta)$.

(a) α is valid if and only if $True \models \alpha$.

Forward direction: If $True \models \alpha$, then α is valid.

By definition, $True \models \alpha$ means that α is true in all worlds where $True$ is True; this is all worlds. Thus α is true in all worlds, which is exactly the defintion of validity. So α is in this case valid.

Backward direction: If α is valid, then $True \models \alpha$.

By definition, if α is valid then it is true in all worlds. In this case *anything* entails α, so clearly $True$ entails α.

(b) For any α, $False \models \alpha$.

Recall the definition of entailment: $p \models q$ means that in all worlds in which p is true, q is true as well. So, $False \models \alpha$ means that in all worlds in which $False$ is true, α is true. But there are no worlds in which $False$ is true! Clearly if there are no worlds in a set, then that set satisfies the condition that α be true in all worlds in that set.

(c) $\alpha \models \beta$ if and only if the sentence $(\alpha \Rightarrow \beta)$ is valid.

Forward direction: If $\alpha \models \beta$, then the sentence $(\alpha \Rightarrow \beta)$ is valid.

By definition, $\alpha \models \beta$ means that β is true in all worlds in which α is true. Thus, in all worlds in which α is true $\alpha \Rightarrow \beta$ holds because both α and β will be true. We must also consider worlds in which α is false; in these worlds, $\alpha \Rightarrow \beta$ also holds by definition of the falsehood of α.

Backward direction: If the sentence $(\alpha \Rightarrow \beta)$ is valid, then $\alpha \models \beta$.

If the sentence $(\alpha \Rightarrow \beta)$ is valid, then it is true in all worlds. Thus, for every world, it must be the case that either both α and β are true, *or* α is false. This is enough to tell us that in every world in which α is true, β is also true, which is the definition of entailment.

4) $\alpha \equiv \beta$ is valid iff $\underbrace{\alpha \supset \beta}_{\neg\alpha \vee \beta}$ and $\underbrace{\beta \supset \alpha}_{\neg\beta \vee \alpha}$ are valid. $\wedge$

For all interpretations I,

$I \models (\neg\alpha \vee \beta) \;\wedge\; I \models (\neg\beta \vee \alpha)$

$\big((I \models \neg\alpha) \vee (I \models \beta)\big) \wedge \big((I \models \neg\beta) \vee (I \models \alpha)\big)$

$\big((I \models \neg\alpha) \vee (I \models \beta)\big) \wedge (I \models \neg\beta)$

$\vee$

$\big((I \models \neg\alpha) \vee (I \models \beta)\big) \wedge (I \models \alpha)$

$\big((I \models \neg\alpha) \wedge (I \models \neg\beta)\big) \vee \big((I \models \beta) \wedge (I \models \alpha)\big)$

For all interpretations, $I \models \alpha$ iff $I \models \beta$.

$Mod(\alpha) = \{I \mid I$ is an interpretation s.t. $I \models \alpha\}$

$= Mod(\beta) = \{I \mid I$ is an interpretation s.t. $I \models \beta\}$

$Mod(\alpha) = Mod(\beta)$

α and β are equivalent $\square$

5) By definition, $\alpha \wedge \neg\beta$ is unsatisfiable means that there is NO interpretation I, s.t. $I \models \alpha \wedge \neg\beta$

For all interpretations I, $I \not\models \alpha \wedge \neg\beta$ is valid

For all interpretations I, $I \models \neg(\alpha \wedge \neg\beta)$

For all interpretations I, $I \models \neg\alpha \vee \neg\neg\beta$

For all interpretations I, $I \models \neg\alpha \vee \beta$

That means for all interpretations I, $\neg\alpha \vee \beta$ is valid

for all interpretations I, $I \models \alpha \supset \beta$

III.

- Proof by contradiction: Assume that the entailment does not hold, which means that there is a truth valuation under which all of the premises are true and the conclusion is false. Derive a contradiction.

Proving that Σ does not entail α, denoted $\Sigma \not\models \alpha$:

- Find one truth valuation t under which all of the premises in Σ are true and the conclusion α is false.

TOPIC **Inference Rules 推理规则**

not yet specified wehether it's valid

1. Definition

If $f_1, \cdots, f_k, g$ are formulas then the following is an inference rule: $\dfrac{f_1, \cdots f_k}{g}$ premises / conclusion

2. Modus Ponens (肯定前件) Inference Rule : $\boxed{\dfrac{p,\ p \to q}{q}}$ $\dfrac{\text{若p，则q；且p为真}}{\text{故q为真}}$

- For any propositional symbols p and q:
- e.g. Rain: It's raining.
  Rain → Wet : If it's raining, then it's wet.
  Wet: Therefore, it's wet.

$\dfrac{\text{Rain, Rain} \to \text{Wet}}{\text{Wet}}$

More generally: $\dfrac{p_1, \cdots, p_k,\ (p_1 \wedge \cdots \wedge p_k) \to q}{q}$

3. Inference Algorithm:

Algorithm: forward inference

Input: set of inference rules Rules.
Repeat until no changes to KB:
  Choose set of formulas $f_1, \ldots, f_k \in$ KB.
  If matching rule $\dfrac{f_1, \ \ldots \ , f_k}{g}$ exists:
    Add g to KB.

Given a set of inference rules (e.g., modus ponens), we can just keep on trying to apply rules. Those rules generate new formulas which get added to the knowledge base, and those formulas might then be premises of other rules, which in turn generate more formulas, etc.

KB derives / proves α (KB $\vdash \alpha$) iff α eventually gets added to KB (by blindly applying rules)

* The rule says that if the premises are in the KB, then you can add the conclusion to the KB.

Example: Modus ponens inference

Starting point:
  KB = {Rain, Rain → Wet, Wet → Slippery}
Apply modus ponens to Rain and Rain → Wet:
  KB = {Rain, Rain → Wet, Wet → Slippery, Wet}
Apply modus ponens to Wet and Wet → Slippery:
  KB = {Rain, Rain → Wet, Wet → Slippery, Wet, Slippery}
Converged.

Can't derive some formulas: ¬Wet, Rain → Slippery 由于推理规则 Modus Ponens 的限制

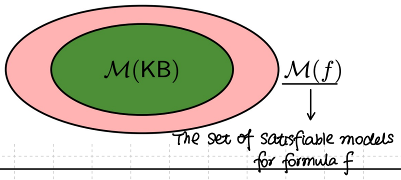
4. Desiderata for inference rules

Syntax:

Inference rules **derive** formulas: KB $\vdash f$

Semantics

Interpretation defines **entailed/true** formulas: KB $\models f$:



semantics gives us an objective notion of truth.

- 推理规则提供了一种机制，通过反复应用(repeated application)这些规则，可以生成一系列 formulas i.e. 所有由知识库KB通过句法推导得出的公式 $\{f \mid \text{KB} \vdash f\}$ 怎样判断一组推理规则正在进行有意义的操作?

  e.g. $[\cdots, P(g(x)), f(x), z)]$ $[\sim P(y, f(w), a), \cdots]$
  unifiable with
  $\theta_1 = \{x/b,\ y/g(b),\ z/a,\ w/b\}$ yields $P(g(b)), f(b), a)$
  but also with
  $\theta_2 = \{x/f(z),\ y/g(f(z)),\ z/a,\ w/f(z)\}$ yields $P(g(f(z)), f(f(z)), a)$

  } Sometimes [ ] is not derivable because of too specific substitutions.

- Most General Unifier (MGU) 最一般合一   MGU指能够使两个字面量一致的最一般性替换.
  θ is a MGU of the literals l_1 and l_2 iff
  - θ unifies l_1 and l_2 θ能够应用于l_1和l_2，使得l_1和l_2在应用替换后相等.
  - 表明θ"最一般" for every unifier θ', there is a substitution θ^* s.t. $\theta' = \theta\theta^*$
  对于任何其它能合一的θ': 都可以通过在θ上应用某种替换θ*来获得. i.e. 任何其它的unifier都是通过对θ的进一步具体化得到的.

If I is a literal, θ is a substitution, then $I\theta$ is the result of the substitution
If C is a clause θ is a substitution, then $C\theta$ is the result of the substitution
e.g. $\theta = \{x/a,\ y/g(x,b,z)\}$ $P(x, z, f(x,y))\theta = P(a, z, f(a, y))$

a Ground literal := a literal without variables
A literal I is an instance of I' if there is a θ s.t. $I = I'\theta$

Computing the MGU for a set of literals l_i: 即找到一个替换集合，使得所有谓词 l_i 在应用这些替换后变得相同.

1. Start with $\theta = \{\}$. 表明还没有进行任何替换
2. If all $l_i\theta$ are identical, then success: θ is the MGU.
   Otherwise look for the disagreement set DS, 寻找"不一致集合" DS:
   e.g. $P(a, f(a, g(z), \ldots \quad P(a, f(a, u, \ldots$
   $DS = \{u, g(z)\}$. DS中的元素 := 在不同的谓词 l_i 中不一致的部分
3. Find a variable $v \in DS$ and a term $t \in DS$ which does not contain v (occur check). Otherwise abort: not unifiable. 在DS中寻找一个变量 v 和一个不包含该变量的项 t 来创建替换（否则会产生递归）
4. $\theta = \theta\{v/t\}$
5. Goto 2. 用更新后的 θ 重复步骤2.

$[\cdots, P(g(x)), f(x), z)] \quad [\sim P(y, f(w), a), \cdots]$ 的 MGU:

S2: $DS = \{g(x), y, x, w, z, a\}$

S3: $\theta = \{y/g(x)\}$

S4: 应用 θ 到两个句 $[\cdots, P(g(x)), f(x), z)] \quad [\sim P(g(x)), f(w), a) \quad \cdots]$

$\downarrow$

S2: $DS = \{x, w, z, a\}$

S3: $\theta = \{x/w\}$

S4: $[\cdots, P(g(w)), f(w), z)] \quad [\sim P(g(w)), f(w), a) \quad \cdots]$

$\downarrow$

S2: $DS = \{z, a\}$

S3: $\theta = \{z/a\}$

S4: $[\cdots, P(g(w)), f(w), a)] \quad [\sim P(g(w)), f(w), a) \quad \cdots]$

$\Rightarrow \theta = \{x/w, \ y/g(w), z/a\}$

$\theta_1 = \{x/b, y/g(b), z/a, w/b)\} = \theta\{w/b\}$

$\theta_2 = \{x/f(z), y/g(f(z)), z/a, w/f(z)\} = \theta\{w/f(z)\}$

- How does $\{f : \mathrm{KB} \models f\}$ relate to $\{f : \mathrm{KB} \vdash f\}$?

语义上蕴涵的公式 — $\{f : \mathrm{KB} \models f\}$

句法上推导得出的公式 — $\{f : \mathrm{KB} \vdash f\}$

The truth

First, we think about proofs in a purely syntactic way.

A proof

- starts with a set of premises,
- transforms the premises based on a set of inference rules (by pattern matching),
- and reaches a conclusion.

We write

$$\Sigma \vdash_{ND} \varphi \quad \text{or simply} \quad \Sigma \vdash \varphi$$

By applying Inference Rules

we're filling up the glass with water

Case1 : We never overflow the glass

Definition: soundness

A set of inference rules Rules is sound if:

$$\{f : \mathrm{KB} \vdash f\} \subseteq \{f : \mathrm{KB} \models f\}$$

nothing but the truth

Sound: not derive any false formulas

Every derivable sentence is a logical consequence of the KB

Case2 : We fill up the glass to the brim and possibly go over

Definition: completeness

A set of inference rules Rules is complete if:

$$\{f : \mathrm{KB} \vdash f\} \supseteq \{f : \mathrm{KB} \models f\}$$

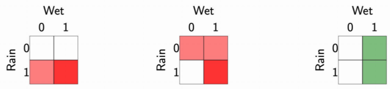
whole truth

Complete: derive all true formulas

Every logical consequence of the KB is derivable by the inference method.

Is $\dfrac{\text{Rain}, \quad \text{Rain} \to \text{Wet}}{\text{Wet}}$ (Modus ponens) sound?

$\mathcal{M}(\text{Rain}) \cap \mathcal{M}(\text{Rain} \to \text{Wet}) \overset{?}{\subseteq} \mathcal{M}(\text{Wet})$ ✓



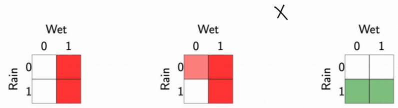
Sound!

We can derive Wet using modus ponens

- To check Entailment: map all the formulas into semantics-land (i.e. the set of satisfiable models)
- If we had other formulas in the KB, that would reduce both sides of $\subseteq$ by the same amount and won't affect the fact that the relation holds. Therefore, this rule is sound.

Is $\dfrac{\text{Wet}, \quad \text{Rain} \to \text{Wet}}{\text{Rain}}$ sound?

$\mathcal{M}(\text{Wet}) \cap \mathcal{M}(\text{Rain} \to \text{Wet}) \overset{?}{\subseteq} \mathcal{M}(\text{Rain})$ ✗



Unsound!

Recall completeness: inference rules derive all entailed formulas (f such that KB $\models f$)

Example: Modus ponens is incomplete

Setup:

KB $= \{\text{Rain}, \text{Rain} \vee \text{Snow} \to \text{Wet}\}$

$f = \text{Wet}$

Rules $= \{\frac{f, \quad f \to g}{g}\}$ (Modus ponens)

Semantically: KB $\models f$ (f is entailed).

Syntactically: KB $\not\vdash f$ (can't derive f).

Incomplete!

Completeness is trickier, and here is a simple example that shows that modus ponens alone is not complete, since it can't derive Wet, when semantically, Wet is true!

Fixing completeness

Option 1: Restrict the allowed set of formulas

propositional logic

$\downarrow$ make the water glass smaller

propositional logic with only Horn clauses

Option 2: Use more powerful inference rules

Modus ponens

$\downarrow$ pour more vigorously into the same glass

resolution

TOPIC Resolution 归结

DATE 归结原理是对命题逻辑和一阶逻辑中的句子的推理规则

1. Resolution Rule in Propositional Logic

- 归结规则接受 包含互补文字的两个子句(是文字的析取式 DNF)，生成 '带有除了互补文字之外的所有文字'的一个新子句
  Resolution Rule in Propositional Logic / is a single valid inference rule / that produces a new clause / implied by two clauses containing complementary literals.

- **Definition: resolution inference rule**

$$\frac{f_1 \vee \cdots \vee f_n \vee p, \quad \neg p \vee g_1 \vee \cdots \vee g_m}{f_1 \vee \cdots \vee f_n \vee g_1 \vee \cdots \vee g_m}$$ ⇝ entails

the left resolved literal　　the right resolved literal

$$\frac{C_1 \cup \{p\}, \quad \{\sim p\} \cup C_2}{C_1 \cup C_2}$$

the resolvent of the input clauses relative to p

归结生成的子句叫做"两个输入子句的归结 (resolvent)"

⭐ Special Case: [p] and [~p] resolve to [ ] (i.e. C_1 and C_2 are empty)

Hence $\{[p], [\sim p]\} \models$ False, or $\{[p], [\sim p]\}$ is unsatisfiable. 如果在应用归结规则之后推导出空子句，则全部的公式是不可满足的。

- Theorem: $S \rightarrow [\ ]$ iff $S \models$ False (i.e. S is unsatisfiable iff $S \rightarrow [\ ]$)
- Resolution is sound and complete for [ ]
- If $S \rightarrow c$, then $S \models c$ (reverse does not always hold)

⭐ Resolvents are implications of the input clauses.

Suppose $I \models (p \vee \alpha)$ and $I \models (\neg p \vee \beta)$.
　Case 1: Let $I \models p$. Then $I \models \beta$ and hence $I \models (\alpha \vee \beta)$.
　Case 2: Let $I \not\models p$ Then $I \models \alpha$ and hence $I \models (\alpha \vee \beta)$.
Therefore, in any case, $I \models (\alpha \vee \beta)$. Thus $\{(p \vee \alpha), (\neg p \vee \beta)\} \models (\alpha \vee \beta)$.

2. Resolution Rule in FOL

大前提 / Major Premise	$\forall x,\ P(x) \rightarrow Q(x)$	
小前提 / Minor Premise	$P(a)$	
结论 / Conclusion	Therefore, $Q(a)$	

- 归结规则把传统的 Inference Rule 的 直言三段论 (Syllogism) 被缩成了一个单一的规则
  要使用归结技术，　<i> 子句们必须转换为合取范式 (CNF)　<见两页之前>
  　　<ii> 从前两个子句推导出最后一个子句:
  - 找到包含相同谓词的子句: 这个谓词在一个子句中是否定的，在另一个子句中是肯定的，在两个谓词上进行合一
  - 若在被合一的谓词中，any unbound variables also occur in other predicates:
    replace them with their bound values (terms) there as well.
    i.e.: Clauses with variables:
    A literal with variables represents ALL its instances. We allow inference over all instances.
    Hence, given: $[P(x,a), \sim Q(x)]$ and $[\sim P(b,y), \sim R(b,f(y))]$
    Since $[P(b,a), \sim Q(b)]$ is an instance of $[P(x,a), \sim Q(x)]$
    $[\sim P(b,a), \sim R(b,f(a))]$ is an instance of $[\sim P(b,y), \sim R(b,f(y))]$
    We'd like to infer $[\sim Q(b), \sim R(b,f(a))]$
  - 丢弃被合一的谓词，然后合并两个子句中 余下的谓词 到一个新子句中，并用 ∨ 算子连接起来.

- Resolution Rule can be generalized to FOL to:

$$\frac{C_1 \cup \{I_1\}, \quad \{\sim I_2\} \cup C_2}{(C_1 \cup C_2)\theta}\ \theta$$

where θ is a MGU of I_1 and I_2 and C_1 and C_2 have no common variables

If two different clauses C_1 and C_2 contain same variable x, we can rename x to some other x' in one of C_1 or C_2.

e.g.

$\forall x,\ P(x) \rightarrow Q(x)$
$\forall x,\ Q(x) \rightarrow R(x)$
$\xrightarrow{(i)}$
$\neg P(x) \vee Q(x)$
$\neg Q(y) \vee R(y)$
变量 x 被重命名 to make it clear that variables in different clauses are different
$\rightarrow$
Unifying $Q(x)$ with $\neg Q(y)$ means that x and y become the same variable anyway. Substituting this into the remaining clauses and combining them gives the conclusion: $\neg P(x) \vee R(x)$

$\forall x,\ P(x) \rightarrow Q(x)$
$P(a)$
$\xrightarrow{(i)}$
$\neg P(x) \vee Q(x)$
$P(a)$
$\xrightarrow{(ii)}$
In P: { 'x' is an unbound variable; 'a' is a bound variable (term) }
Unifying the two produces the substitution x/a
丢弃合一的 P，并把这个代换应用到余下的谓词 Q(x)
生成结论 $Q(a)$

POP: A Partial-Order Planner

In this lecture, we look at the operation of one particular partial-order planner, called POP. POP is a regression planner; it uses problem decomposition; it searches plan space rather than state space; it build partially-ordered plans; and it operates by the principle of least-commitment. postpone commitment unless forced

In our description, we'll neglect some of the fine details of the algorithm (e.g. variable instantiation) in order to gain greater clarity.

1 POP plans

We have to say what a plan looks like in POP. We are dealing with partially-ordered steps so we must give ourselves the flexibility to have steps that are unordered with respect to each other. And, we are searching plan-space instead of state space, so we must have the ability to represent unfinished plans that get refined as planning proceeds.

A plan in POP (whether it be a finished one or an unfinished one) comprises:

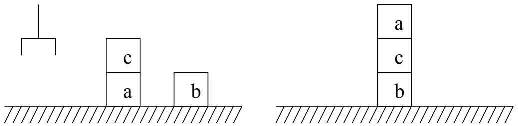
- A set of plan *steps*. Each of these is a STRIPS operator, but with the variables instantiated.
- A set of *ordering constraints*: $S_i \prec S_j$ means step S_i must occur sometime before S_j (not necessarily immediately before).
- A set of *causal links*: $S_i \xrightarrow{c} S_j$ means step S_i achieves precondition c of step S_j.

So, it comprises actions (steps) with constraints (for ordering and causality) on them.

The algorithm needs to start off with an *initial plan*. This is an unfinished plan, which we will refine until we reach a solution plan.

The initial plan comprises two dummy steps, called Start and Finish. Start is a step with no preconditions, only effects: the effects are the initial state of the world. Finish is a step with no effects, only preconditions: the preconditions are the goal.

By way of an example, consider this initial state and goal state:



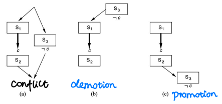
Initial state Goal state

These would be represented in POP as the following initial plan:

```
Plan(STEPS: {S1: Op( ACTION:   Start,
                     EFFECT:   clear(b) ∧ clear(c) ∧
                               on(c, a) ∧ ontable(a) ∧
                               ontable(b) ∧ armempty),
             S2: Op( ACTION:   Finish,
                     PRECOND:  on(c, b) ∧ on(a, c))},
     ORDERINGS: {S1 ≺ S2},
         LINKS: {})
```


This initial plan is refined using POP's *plan refinement operators*. As we apply them, they will take us from an unfinished plan to a less and less unfinished plan, and ultimately to a solution plan. There are four operators, falling into two groups:

- *Goal achievement operators*
  - *Step addition*: Add a new step S_i which has an effect c that can achieve an as yet unachieved precondition of an existing step S_j. Also add the following constraints: $S_i \prec S_j$ and $S_i \xrightarrow{c} S_j$ and Start $\prec S_i \prec$ Finish.
  - Use an effect c of an existing step S_i to achieve an as yet unachieved precondition of another existing step S_j. And add just two constraints: $S_i \prec S_j$ and $S_i \xrightarrow{c} S_j$.
- Causal links must be *protected* from *threats*, i.e. steps that delete (or negate or *clobber*) the protected condition. 破坏 If S threatens link $S_i \xrightarrow{c} S_j$:
  - **Promote**: add the constraint $S \prec S_i$; **or**
  - **Demote**: add the constraint $S_j \prec S$



The goal achievement operators ought to be obvious enough. They find preconditions of steps in the unfinished plan that are not yet achieved. The two goal achievement operators remedy this either by adding a new step whose effect achieves the precondition, or by exploiting one of the effects of a step that is already in the plan.

The promotion and demotion operators may be less clear. Why are these needed? POP uses problem-decomposition: faced with a conjunctive precondition, it uses goal achievement on each conjunct separately. But, as we know, this brings the risk that the steps we add when achieving one part of a precondition might interfere with the achievement of another precondition. And the idea of promotion and demotion is to add ordering constraints so that the step cannot interfere with the achievement of the precondition.

Finally, we have to be able to recognise when we have reached a *solution plan*: a finished plan.

A solution plan is one in which:

- every precondition of every step is achieved by the effect of some other step and all possible clobberers have been suitably demoted or promoted; and
- there are no contradictions in the ordering constraints, e.g. disallowed is $S_i \prec S_j$ and $S_j \prec S_i$; also disallowed is $S_i \prec S_j$, $S_j \prec S_k$ and $S_k \prec S_i$.

Note that solutions may still be partially-ordered. This retains flexibility for as long as possible. Only immediately prior to execution will the plan need *linearisation*, i.e. the imposition of arbitrary ordering constraints on steps that are not yet ordered. (In fact, if there's more than one agent, or if there's a single agent but it is capable of multitasking, then some linearisation can be avoided: steps can be carried out in parallel.)

2 The POP algorithm

In essence, the POP algorithm is the following:

1. Make the initial plan, i.e. the one that contains only the Start and Finish steps. 先画 Start & finish
2. Do until you have a solution plan
   - Take an unachieved precondition from the plan; achieve it 从 Action 中选: 未满足的 pre-con.
   - Resolve any threats using promotion or demotion

But what the above fails to show is that planning involves search. At certain points in the algorithm, the planner will be faced with choices (alternative ways of refining the current unfinished plan). POP must try one of them but have the option of returning to explore the others.

There are basically two main 'choice points' in the algorithm:

- In goal achievement, a condition c might be achievable by any one of a number of new steps and/or existing steps. For each way of achieving c, a new version of the plan must be created and placed on the agenda.

  Question. A condition c might be achievable by new steps or existing steps. When placing these alternatives on the agenda, why might we arrange for the latter to come off the agenda before the former?
- When resolving threats, POP must choose between demotion and promotion.

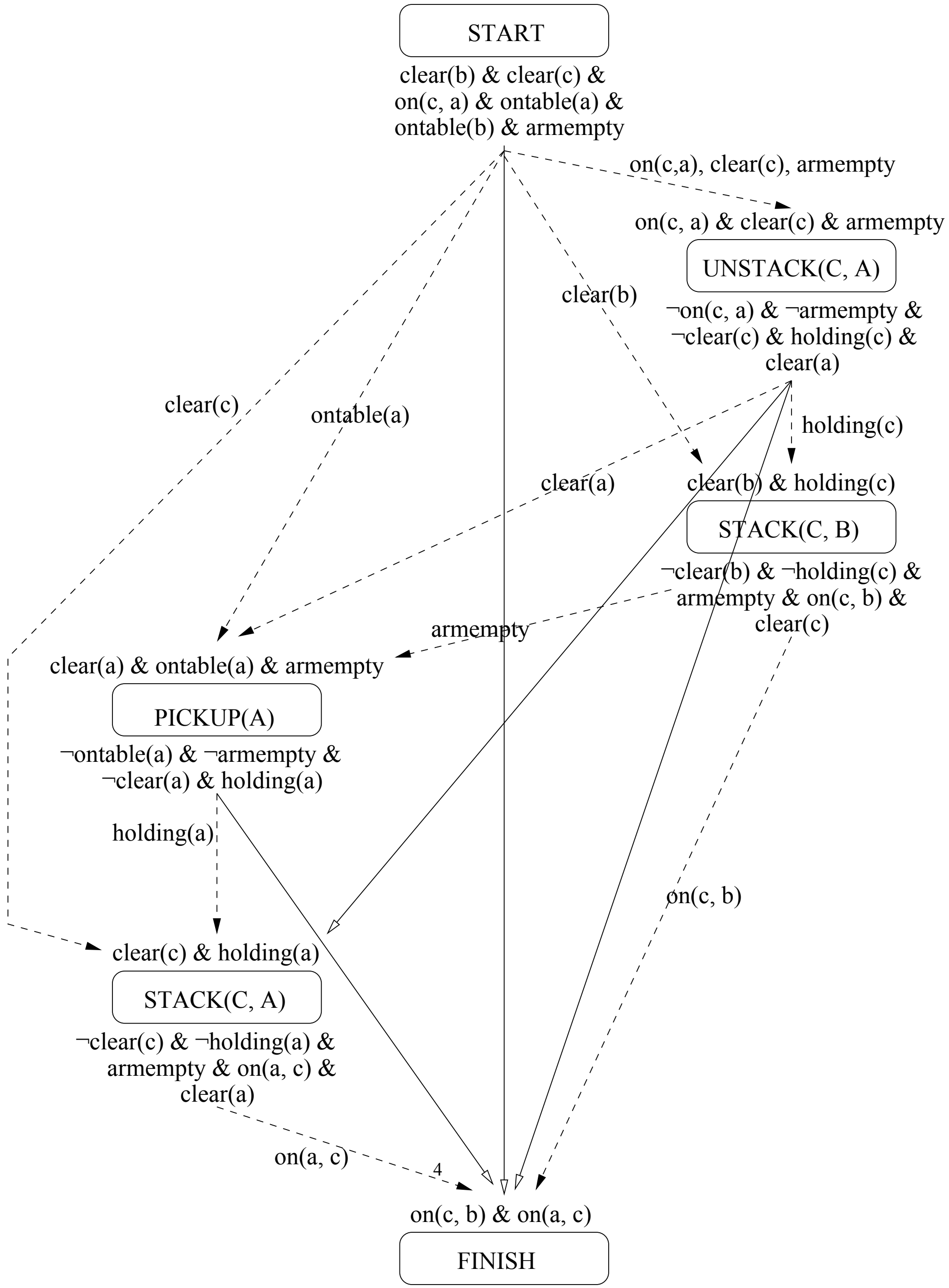
(Some people think that the choice of which precondition to achieve next also gives rise to search. But, in fact, all preconditions must eventually be achieved, and so these aren't alternatives. The choice can be made irrevocably.)

Provided your implementation of POP uses a complete and optimal search strategy, then POP itself is complete and optimal.

However, POP's branching factor can still be high and the unfinished plans that we store on the agenda can be quite large data structures, so we typically abandon completeness/optimality to keep time and space more manageable. Search strategies that are more like depth-first search might be preferable. And we might use heuristics to order alternatives or even to prune the agenda.

In the lecture, we will dry-run the POP algorithm.

Afterwards, convince yourself that POP is a regression planner, that it uses problem decomposition, that it searches plan space, that it build partially-ordered plans and that it operates by the principle of least commitment.



Exercise (Past exam question)

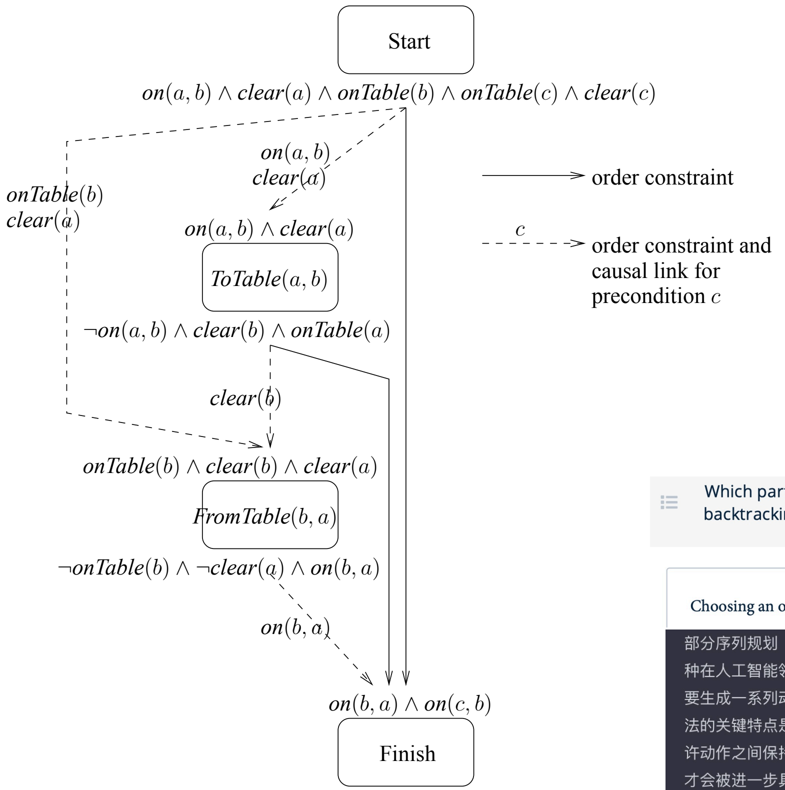
1. An A.I. planner operates in a simplified Blocks World. The only operators in its repertoire move a block x from the table to another block y:

   Op(ACTION: *FromTable*(x, y),
   PRECOND: *onTable*$(x) \wedge$ *clear*$(x) \wedge$ *clear*(y),
   EFFECT: $\neg$*onTable*$(x) \wedge \neg$*clear*$(y) \wedge on(x, y)$)

   and move a block x from block y to the table:

   Op(ACTION: *ToTable*(x, y),
   PRECOND: $on(x, y) \wedge$ *clear*(x),
   EFFECT: $\neg on(x, y) \wedge$ *clear*$(y) \wedge$ *onTable*(x))

   Here is an incomplete plan of the kind that could be built by the POP planner covered in lectures:



   (a) Give the *initial* world state and *goal* of this plan.

   (b) Copy this plan onto your answer sheet. (Copy just the boxes and arrows; there is no need to copy the preconditions & effects.)
   - Choose an unachieved precondition in the plan.
   - Add a *new step* to the plan to achieve your chosen precondition. Draw it onto your copy of the diagram. Include its preconditions & effects, all order constraints and all causal links.
   - If your new step threatened any existing causal links, then state which link(s) were threatened; state what extra ordering constraint(s) you added to protect the threatened link(s); state whether what you did was an example of promotion or demotion; and briefly explain why the extra ordering constraint(s) fix the plan.

   (c) Is the plan now complete? Explain your answer.

2. Write *STRIPS operators* that would enable a planner to build plans that it could give to photocopier repair robots. Use the following predicate symbols:

copier(x)	:	x is a photocopier
robot(x)	:	x is a robot
noToner(x)	:	x has no toner
hasToner(x)	:	x has toner
hasPaper(x, n)	:	x has n sheets of paper
at(x, y)	:	x is at y

   You can also use the predicates $<, \leq, >, \geq$ and $=$, the function symbols $+$ and $-$ and the constant symbols 0 and 1 if you wish, all with their usual meanings from arithmetic.

   You should write the following three operators:
   - *replaceToner*(x, y): To replace the toner, the copier (y) must be out of toner, a robot (x) must be at the copier and it must have some toner, all of which it puts into the copier.
   - *insertPaper*(x, y, n): To put n sheets of paper into the copier (y), a robot (x) must be at the copier and it must have at least n sheets of paper. (You should assume that the copier has no maximum amount of paper.)
   - *makeCopy*(x, y): To make a copy (using up one sheet of paper), a robot (x) must be at a copier (y) that has toner and that has at least one sheet of paper.

Which parts of the algorithm for partial-order planning (POP) may require backtracking? 2 Punkte

Choosing an operator and resolving threats may result in backtracking. (Selecting a subgoal does not.)

i.e. plan steps with unfulfilled preconditions

It does NOT matter in which order subgoals are chosen when looking for a solution.

部分序列规划（Partial-Order Planning, POP）算法是一种在人工智能领域用于自动规划的技术，特别是在处理需要生成一系列动作以达到特定目标状态的问题时。POP算法的关键特点是它不是线性地生成解决方案路径，而是允许动作之间保持某种部分顺序关系，这些关系只在必要时才会被进一步具体化。在这个过程中，可能需要回溯（backtracking）的部分主要包括：

1. **动作选择**：在规划过程中，算法需要选择合适的动作来解决当前的子目标。如果所选动作导致无法解决的冲突或无法进一步接近目标状态，算法可能需要回溯到选择动作的步骤，尝试其他的动作。
2. **因果关系链的建立**：为了达到目标状态，算法需要建立动作之间的因果关系链。如果发现某些动作之间的因果关系构建不当（比如，产生了逻辑上的矛盾或违反了先前的约束），可能需要回溯到更早的点，重新构建这些关系。