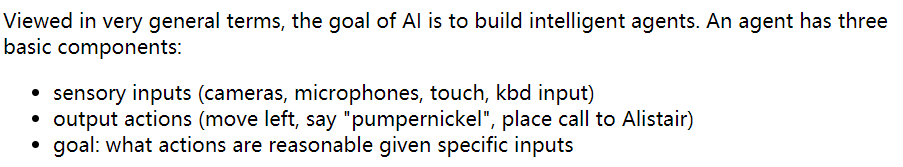
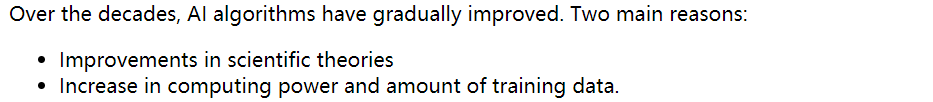
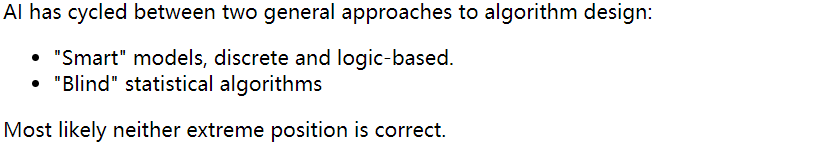
ECE 448 midterm1

**Historical and other trivia:**

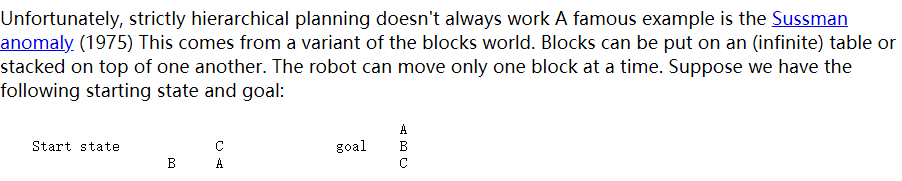






**Shakey and Blocks world：** (1966-72) the world's first mobile "automaton." A∗ search was developed around 1968 as part of the Shakey project at Stanford.

Towards the end of the project, it was using a PDP-10 with about 800K bytes of RAM. The programs occupied 1.35M of disk space. This kind of memory starvation was typical of the time. For example, the guidance computer for Apollo 11 (first moon landing) in 1969 had only 70K bytes of storage.



So strictly hierarchical planning doesn't work. We need a more flexible approach to planning. For example, expand what's required to meet **subgoals**, then try to order all the little tasks. So you can "interleave" sub-tasks from more than one main goal.

**Waltz line labelling** Constraint propagation was developed by David Waltz in the early 1970's for the Shakey project. The original task was line labelling (from Lana Lazebnik based on David Waltz's 1972 thesis).

**STRIPS planning：** STRIPS planning, or the Stanford Research Institute Problem Solver, was developed in 1971. It is a simple version of classical planning, which is used to manage the high level goals and constraints of a complex planning problem, like waypoints for a robot doing assembly/disassembly tasks.

**Boston Dynamics**:

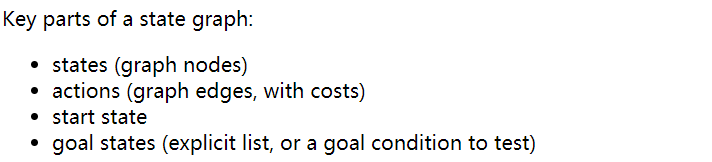
We often imagine intelligent agents that have a sophisticated physical presence, like a human or animal.

robot falling down 2017

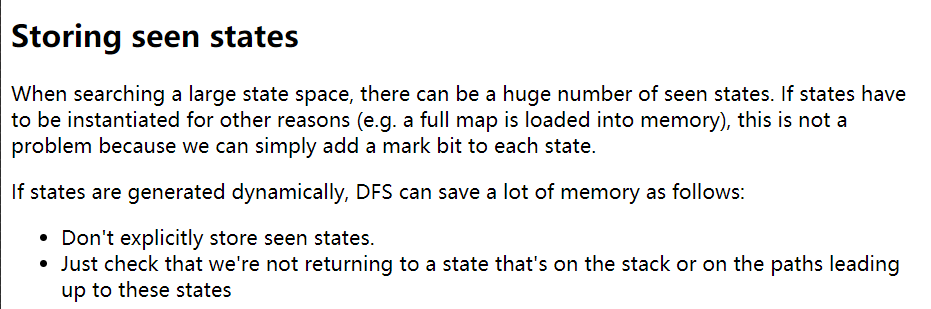
**Google self-driving bike** fake

**McCulloch and Pitts**: early neural nets, 1940s

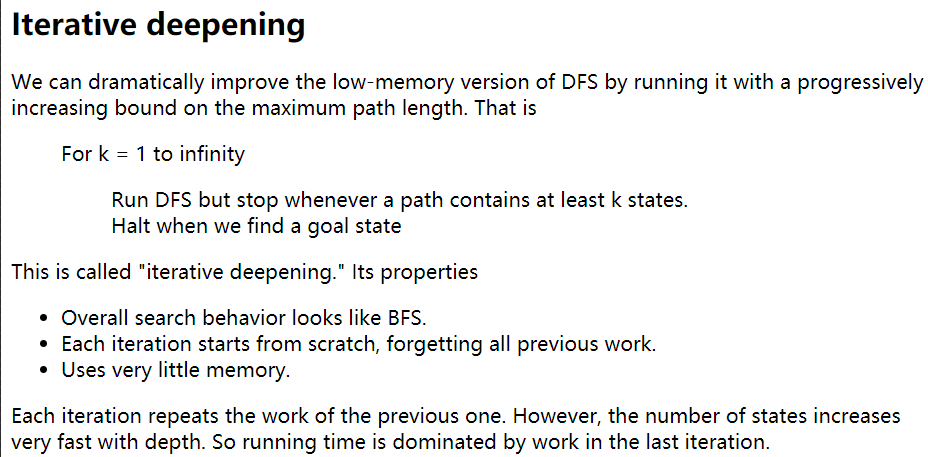
**Search:**



Our first few search algorithms will be uninformed. That is, they rely **only** on computed distances from the starting state.

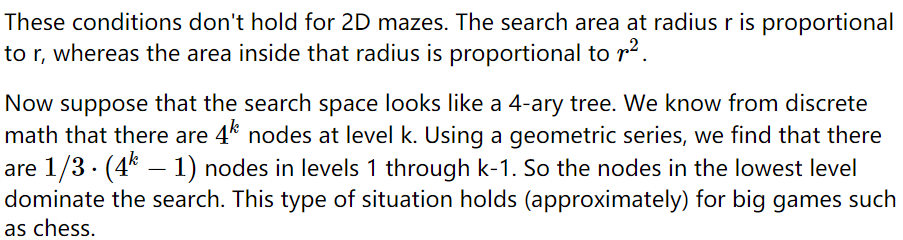


The cost is that this version of DFS does redundant work because it has a poor memory for what it has already examined. This tradeoff can be good or bad depending on the structure of the state space.



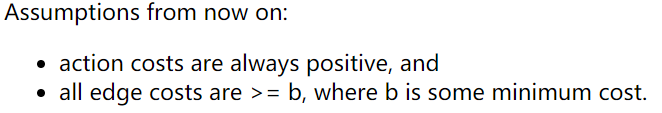
During search, the frontier contains states that have been seen but their outgoing edges have not yet been explored. BFS stores the frontier as a queue; DFS stores it as a stack. Iterative deepening is a hybrid that runs DFS with depth bound that gradually increases. Compared to BFS, iterative deepening does some redundant work but uses less memory.

When starting each iteration, DFS forgets its previous work and has to search from scratch. This idea is most useful when the new search area at level k is large compared to the area search in previous iterations, so that it's not a huge extra cost to re-do the earlier work.

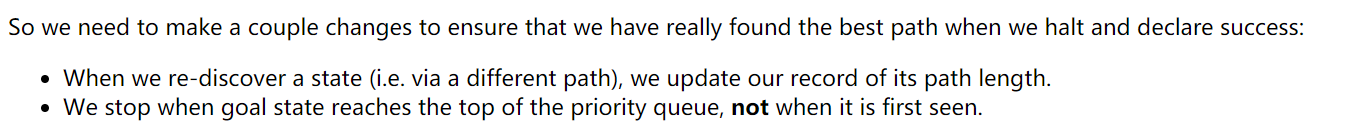


**Uniform cost search**

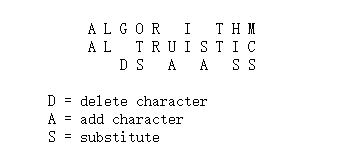
BFS is primarily designed to handle graphs in which all edges have the same cost. If costs don't vary too much, it will tend to find a lost-cost path. But it's only guaranteed to find an optimal path when all edges have the same cost. Uniform Cost Search (UCS) is a modification of BFS which handles edges with varying costs.

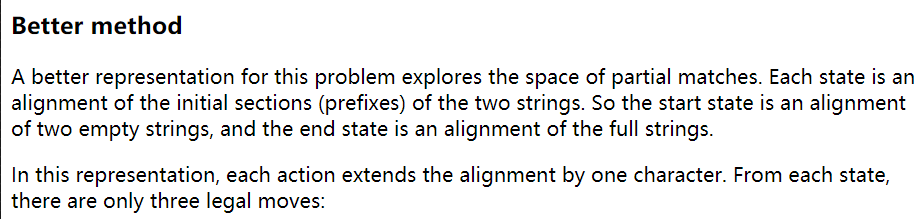


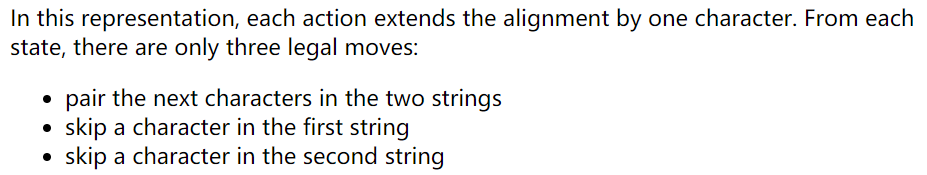
The main change between BFS and UCS is that the frontier is stored as priority queue. In each iteration of our loop, we explore neighbors of the best state (the shortest path length) seen so far.

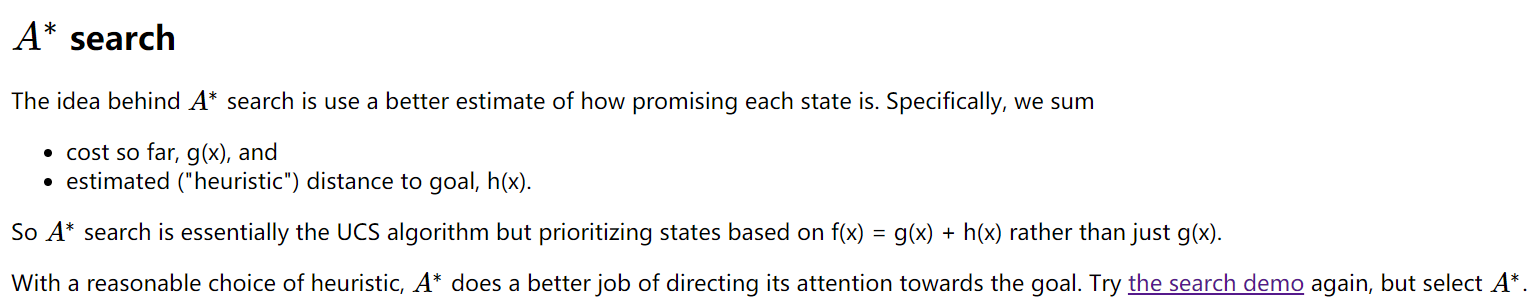


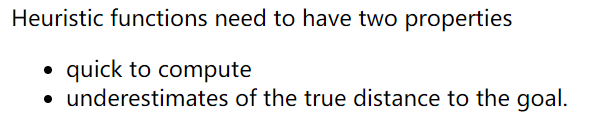
Edit distance

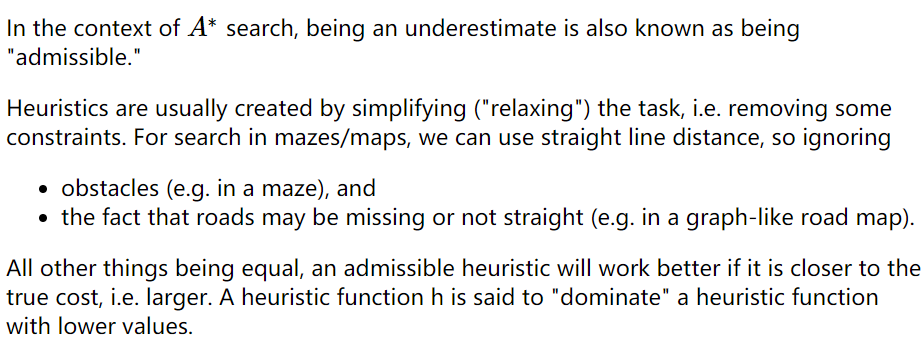




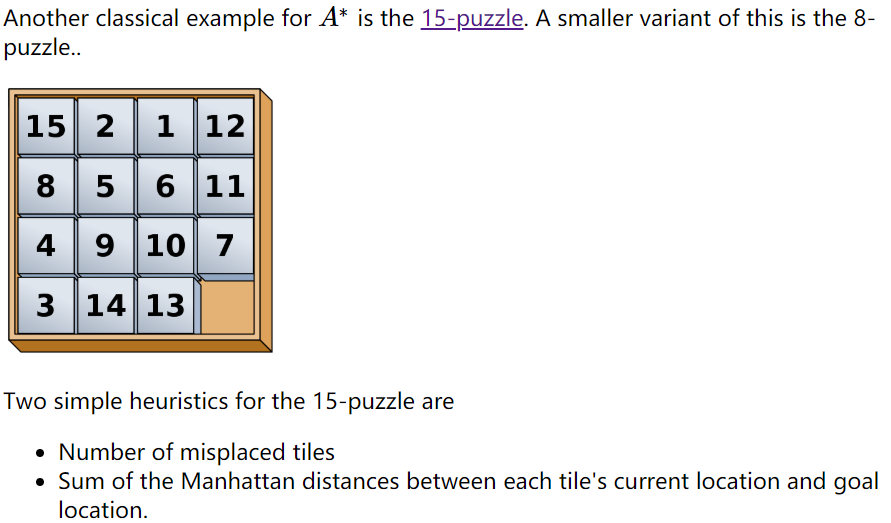


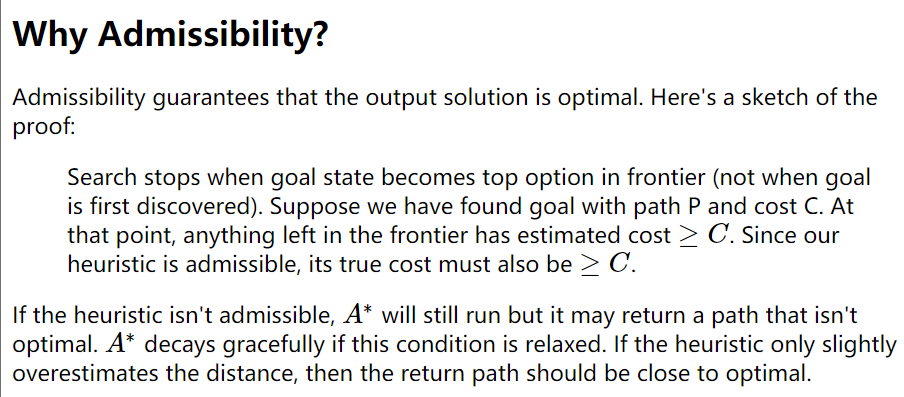


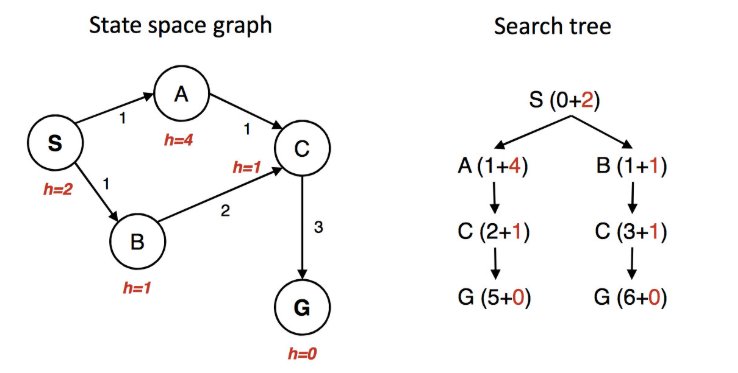


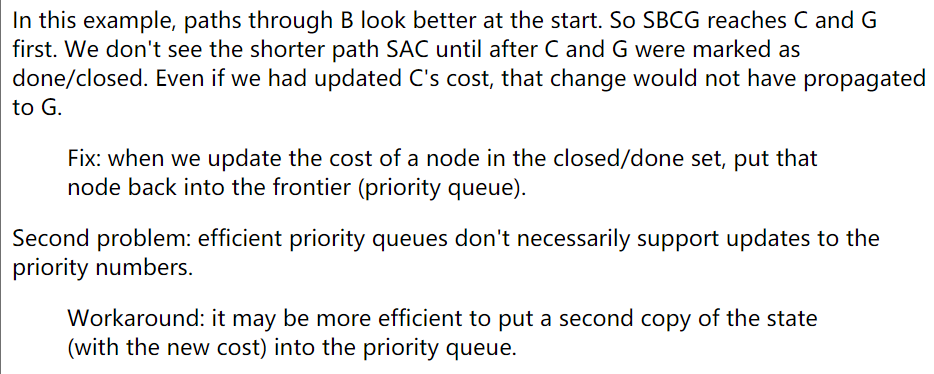


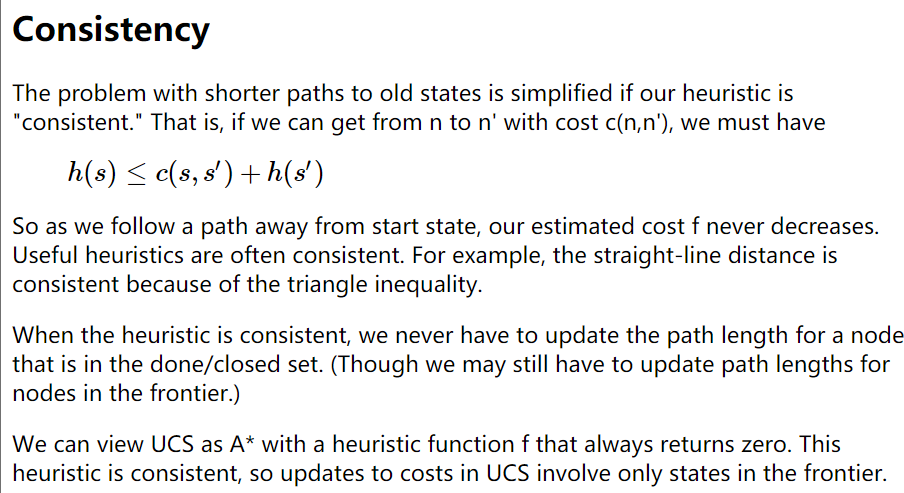
For example, suppose that we're searching a digitized maze and the robot can only move in left-right or up-down (not diagonally). Then **Manhattan** **distance** is a better heuristic than straight line distance. The Manhattan distance between two points is the difference in their x coordinates plus the difference in their y coordinates. If the robot is allowed to move diagonally, we can't use Manhattan distance because it can overestimate the distance to the goal.

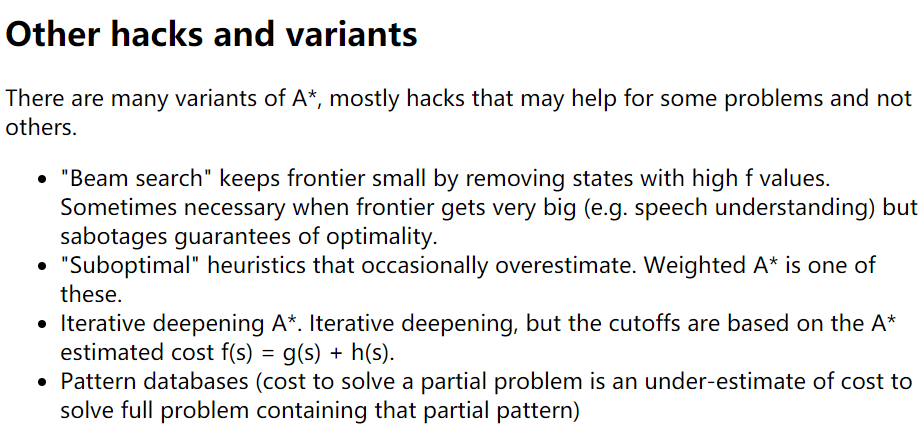




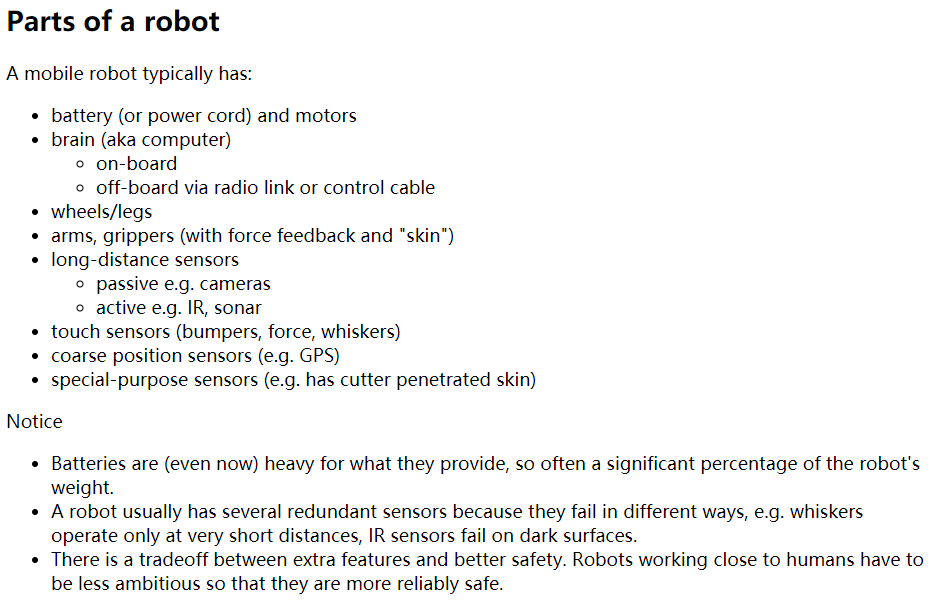


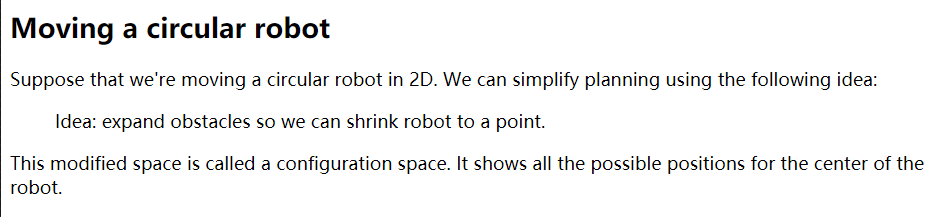


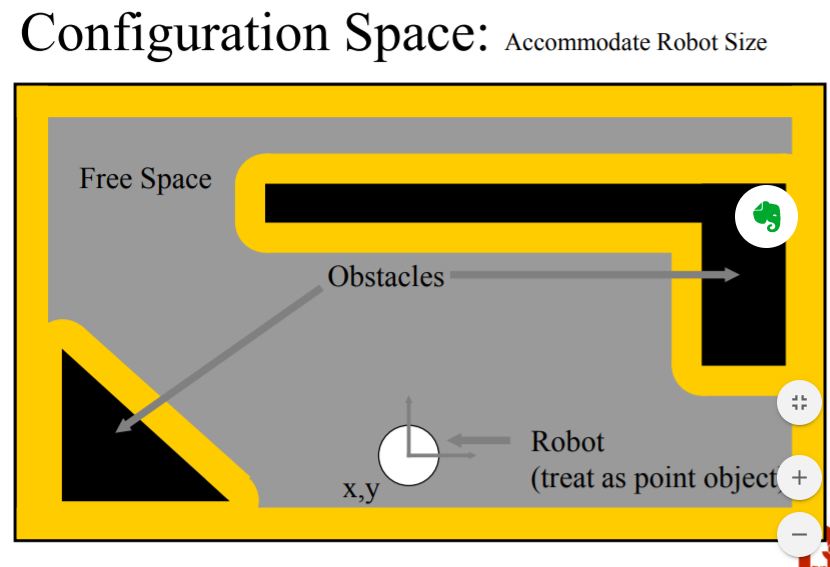


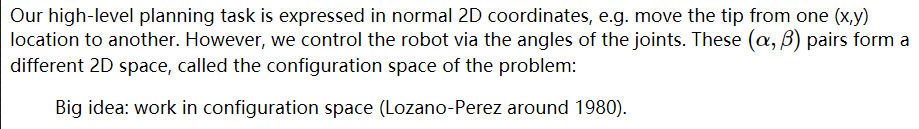


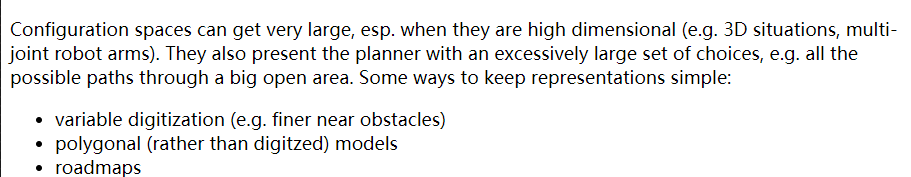
**Robot Planning**

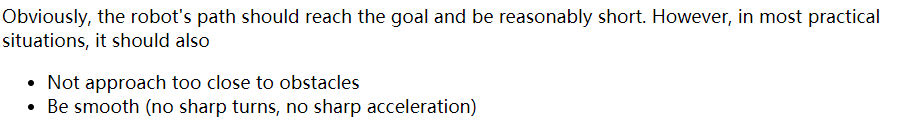




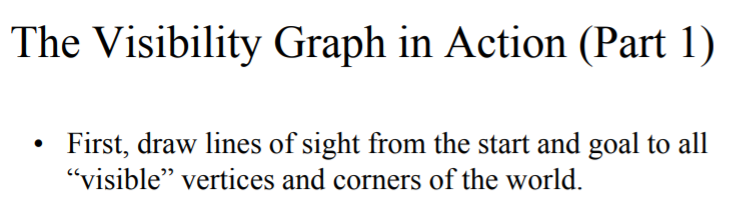


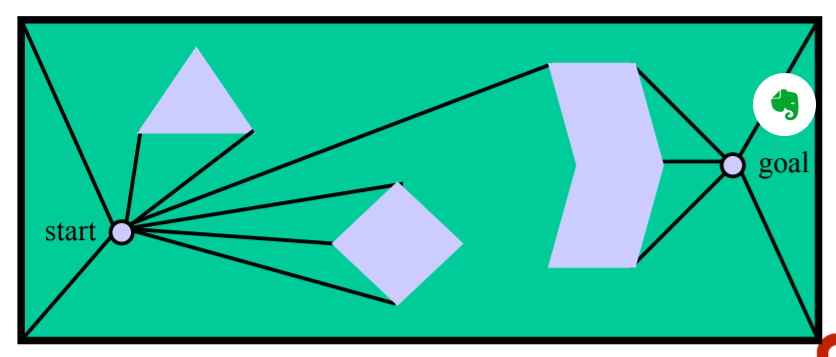


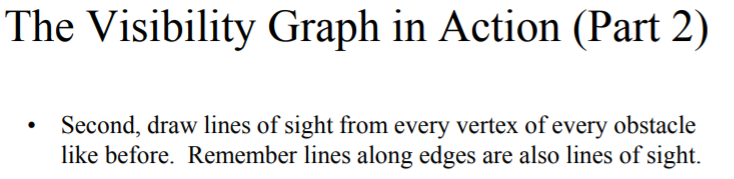


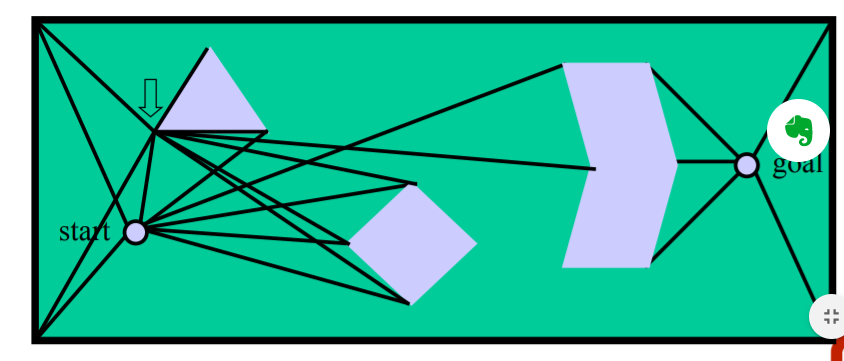


**Roadmaps:** visibility graph

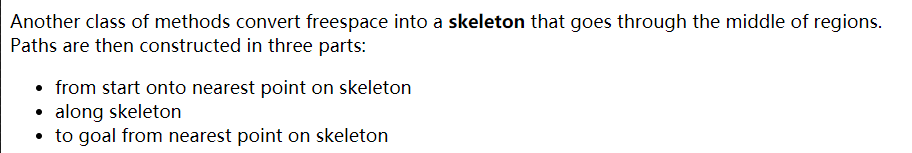


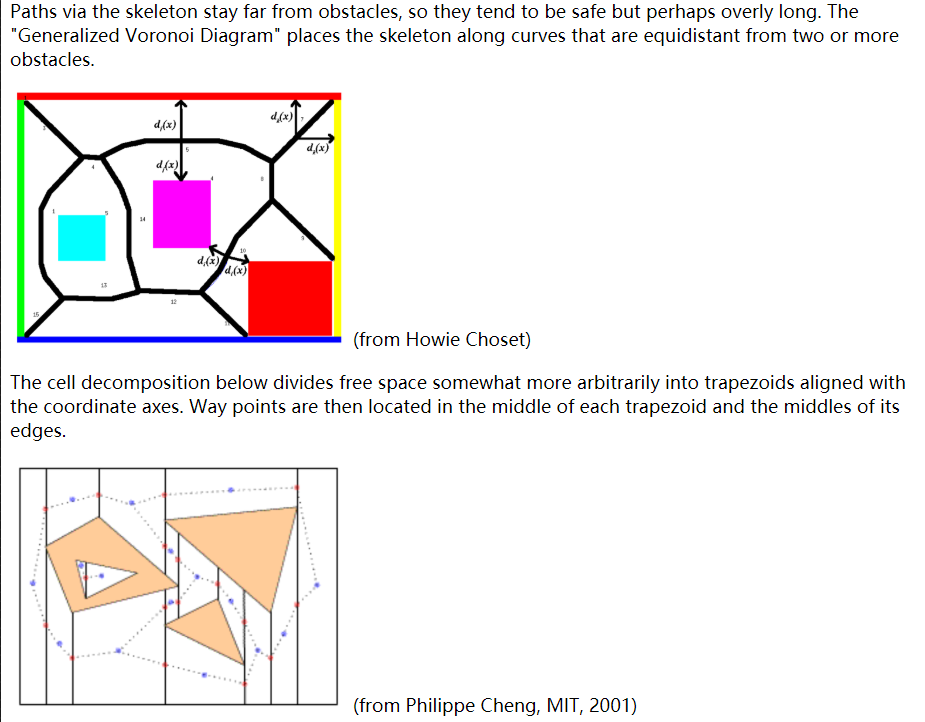






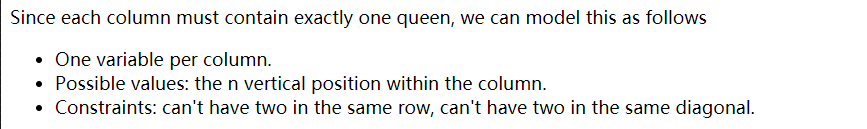
Repeat and repeat

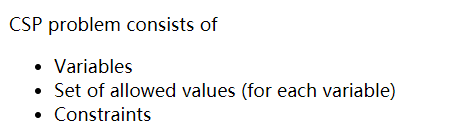




**Constraint Satisfaction Problems**

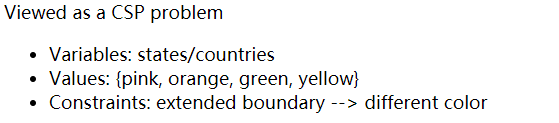
**N Queen Problem**:





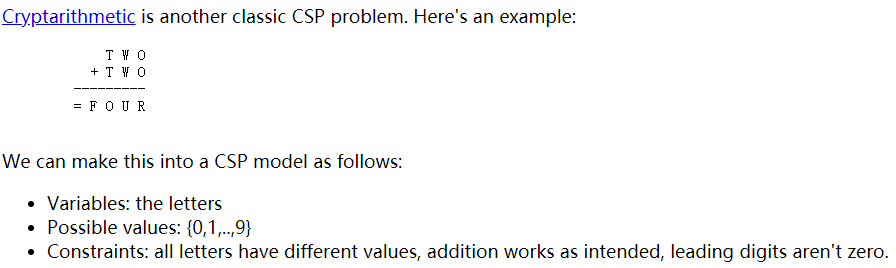
In basic search problems, we know the goal and want to find a path to it. In a constraint-satisfaction problem, we have only a test for whether a state is a goal. Our problem is to find a goal state. The path to the goal is only temporary scaffolding.

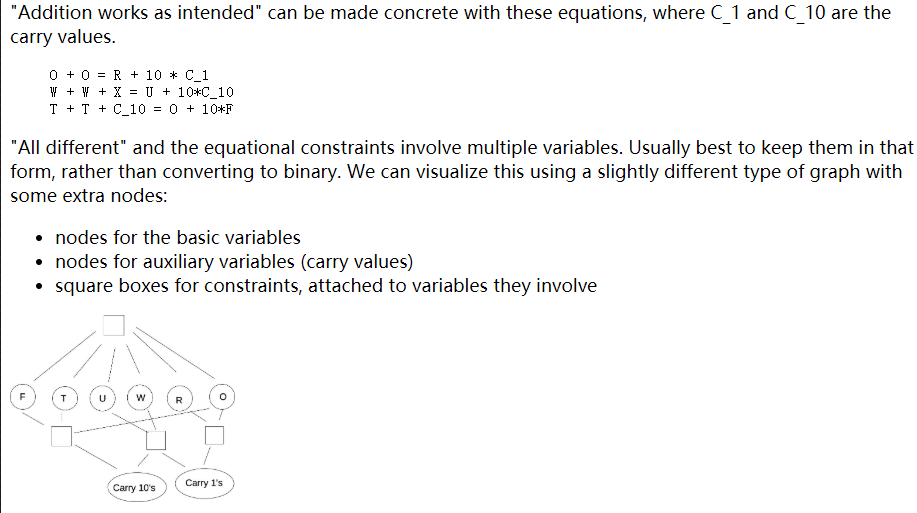
**Map Coloring**

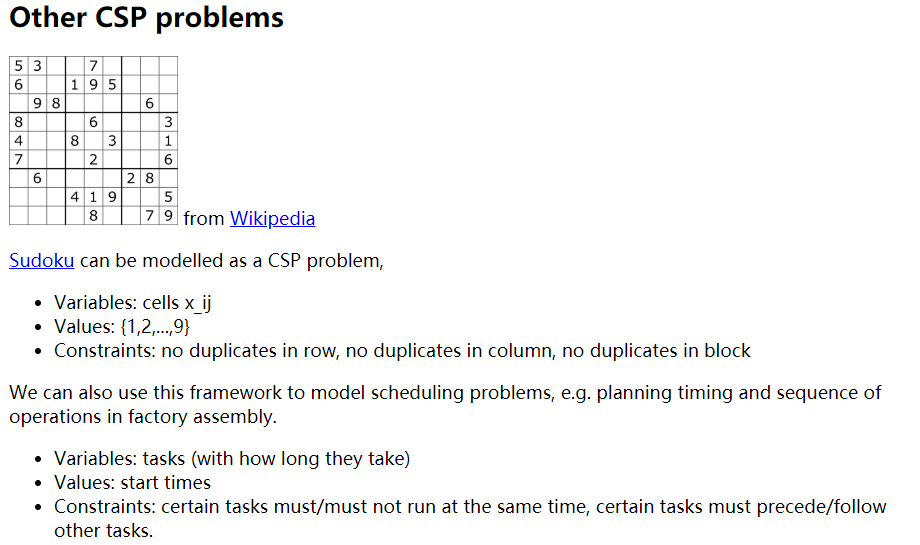


**Notice that graph coloring is NP complete.** We don't know for sure if NP problems require polynomial or exponential time, but we suspect they require exponential time. However, many practical applications can exploit domain-specific heuristics (e.g. linear scan for register allocation) or loopholes (e.g. ok to have small conflicts in final exams) to produce fast approximate algorithms.

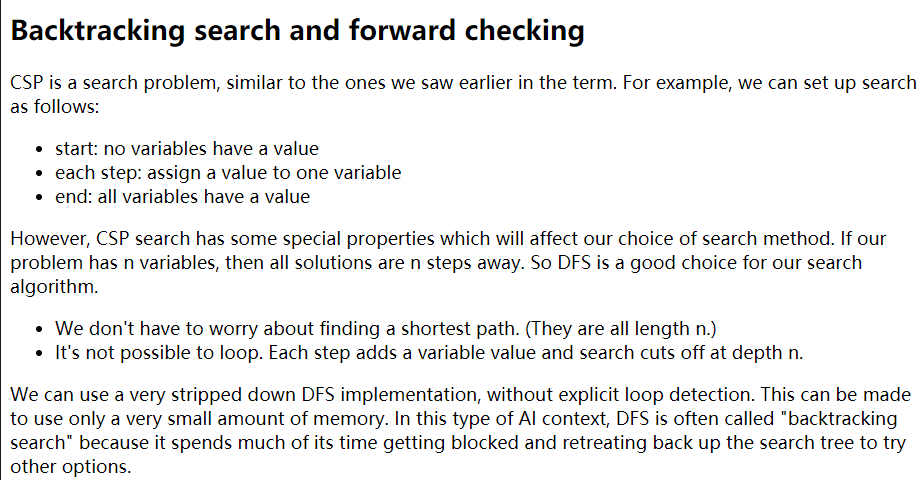
**Cryptarithmetic**







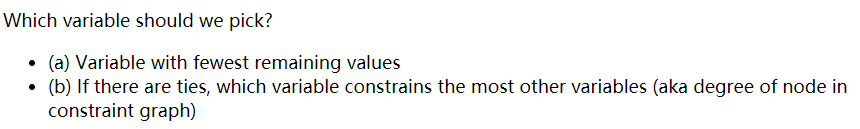
※※※※※※※※※※※※



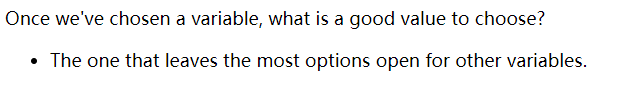
When to stop and backtrack?

Smart method (**forward checking**): During search, each variable keeps a list of its possible values. At each search step, remove values from these lists if they violate constraints, given the values we've already assigned to other variables. Back up if any variable has no possible values left.

Heuristics for variable assignments



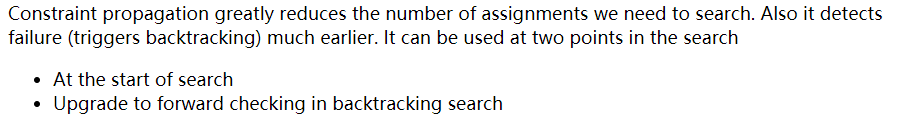
Choosing a value

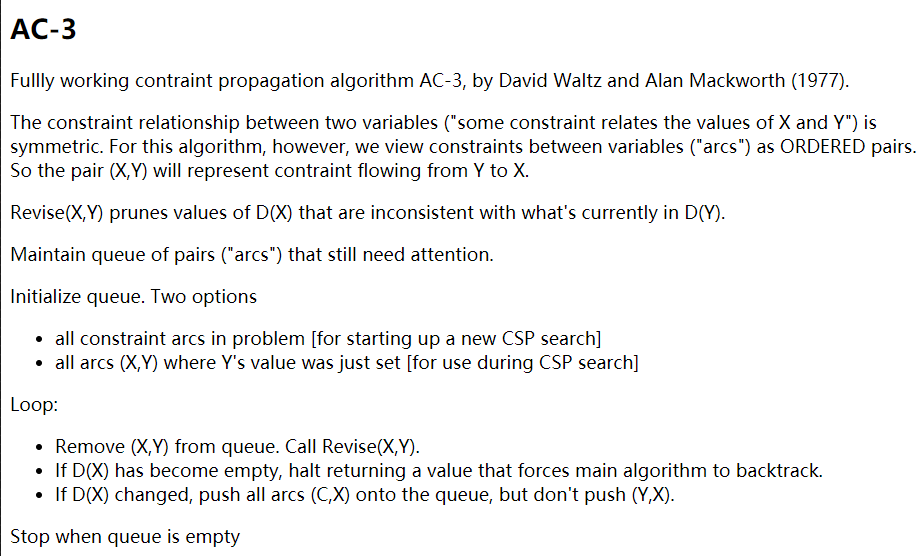


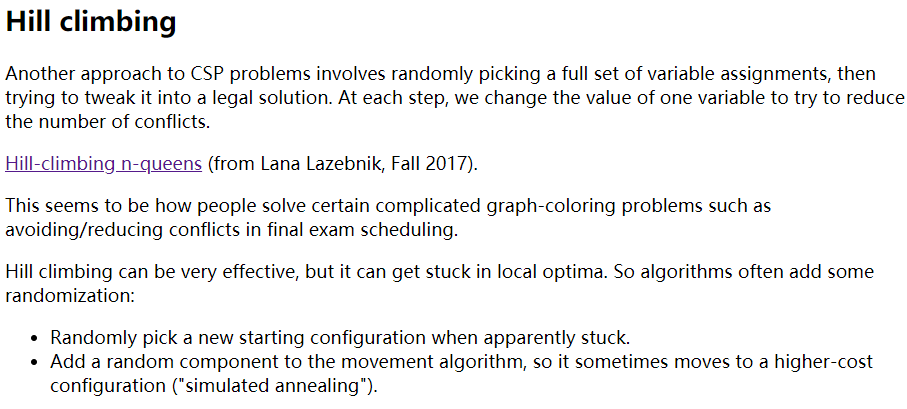
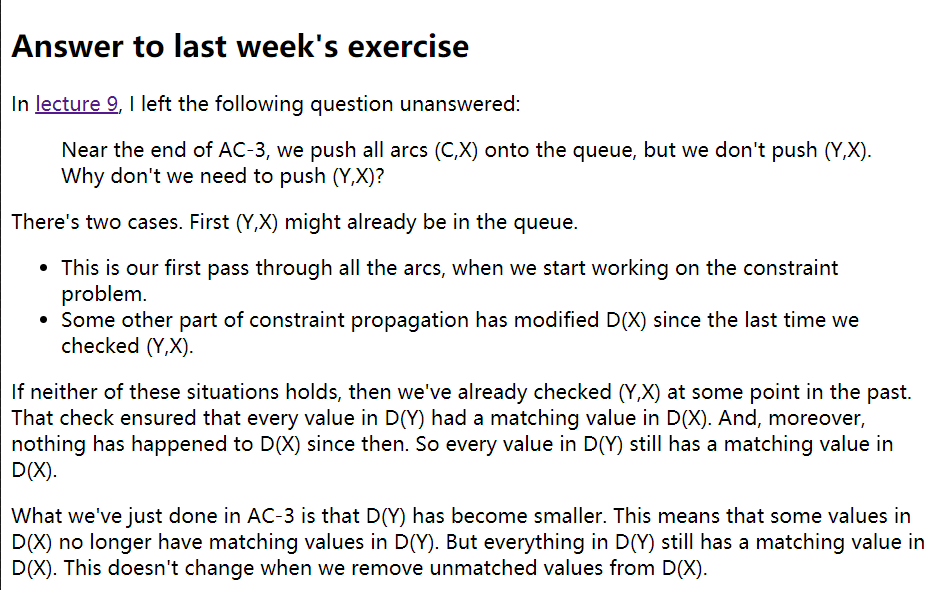
Constraint propagation

When we assign a value to variable X, forward checking only checks variables that share a constraint with X, i.e. are adjacent in the constraint graph. This is helpful, but we can do more to exploit the constraints. Constraint propagation works its way outwards from X's neighbors to their neighbors, continuing until it runs out of nodes that need to be updated.

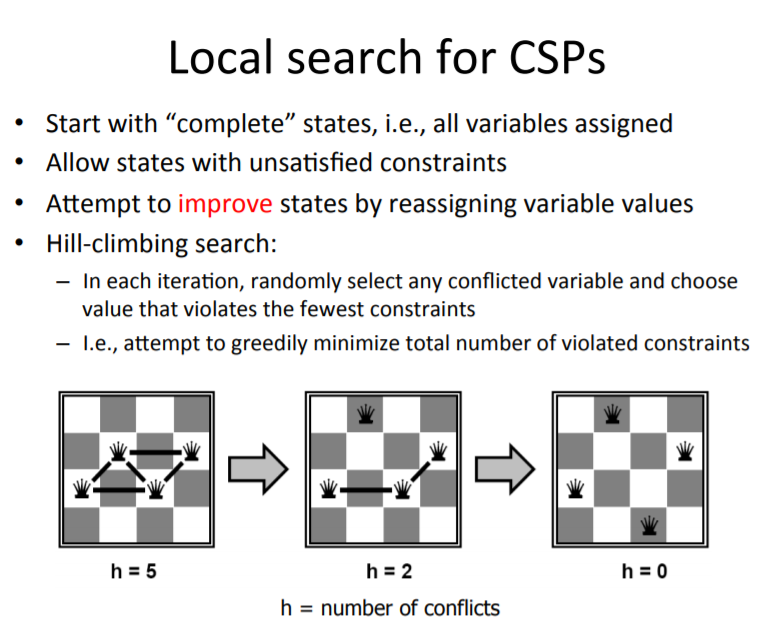
The exact sequence of updates depends on the order in which we happen to have stored our nodes. However, constraint propagation typically makes major reductions in the number of options to be explored, and allows us to figure out early that we need to backtrack.

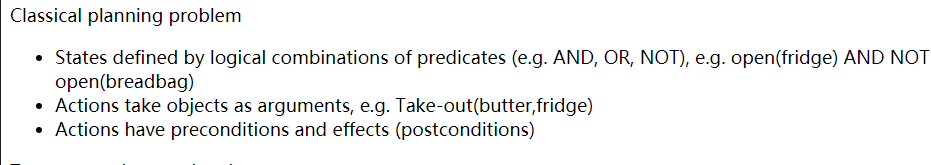


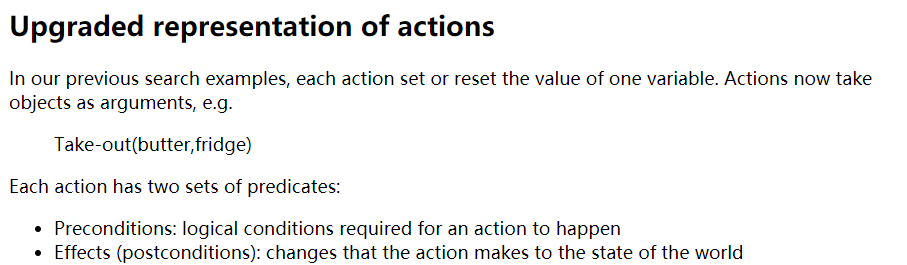




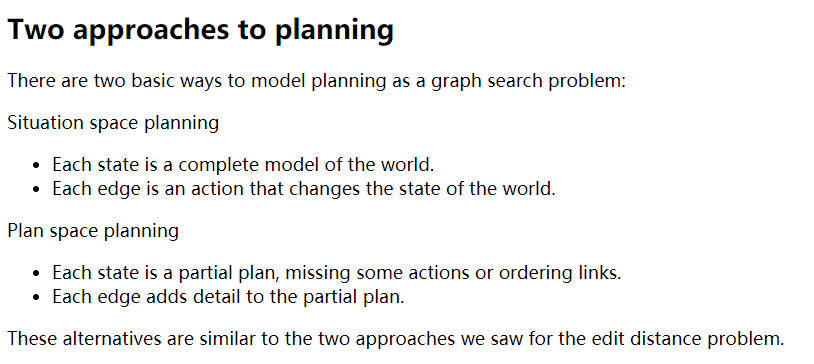
**Hill climbing can be very effective, but it can get stuck in local optima.**

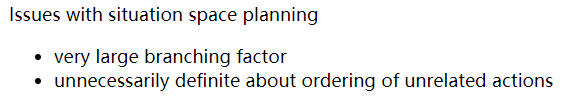


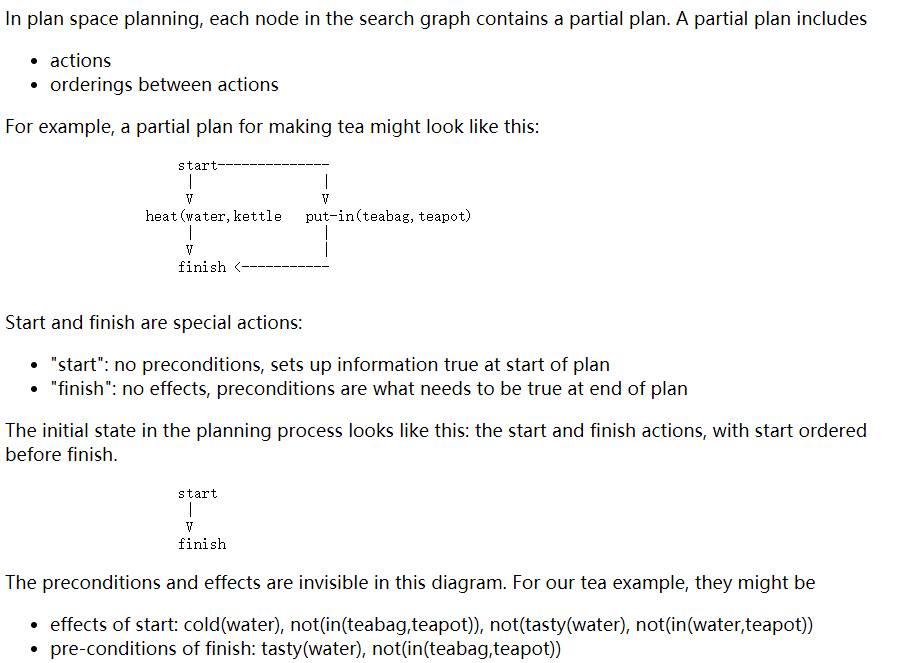


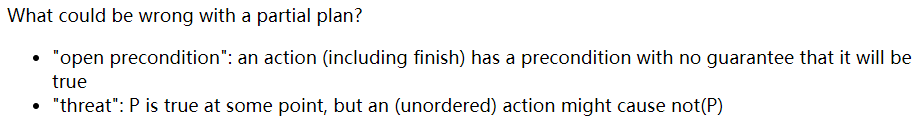


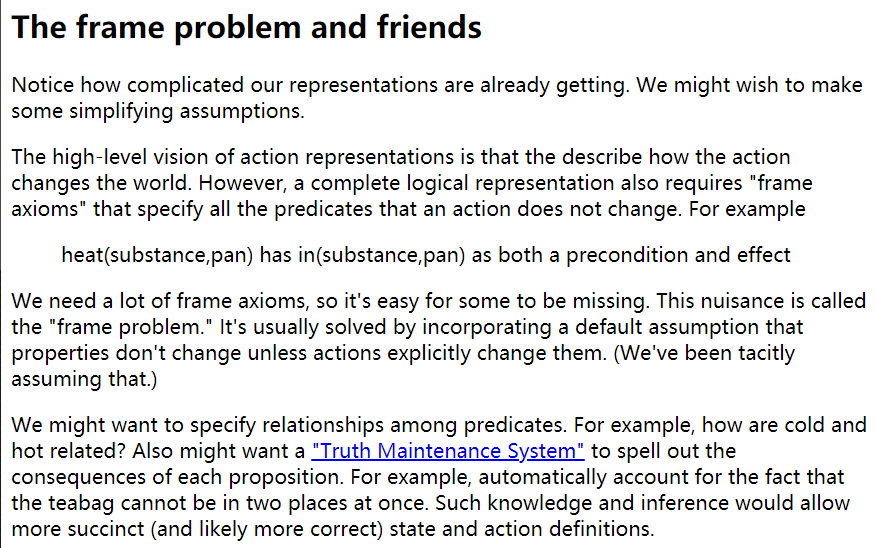
In general, classical planning can get very difficult. Consider the Towers of Hanoi puzzle. (Also see demo.) Solutions to this puzzle have length exponential in the number of disks. When modelling realistic situations, planning constraints and background knowledge (e.g. action pre- and post-conditions) can get very large. So it's essential to exploit all available domain constraints. For practical tasks (e.g. robot assembly, face recognition) folks often engineer the task/environment to make the AI problem doable.





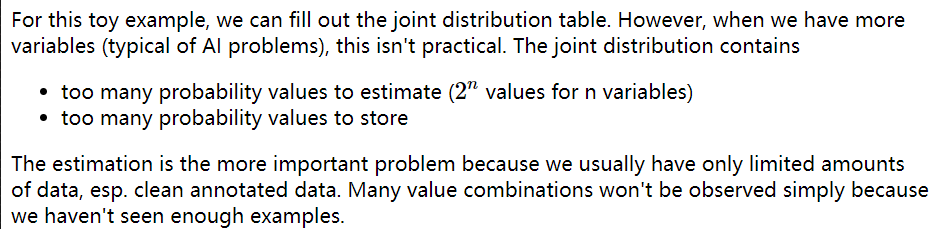


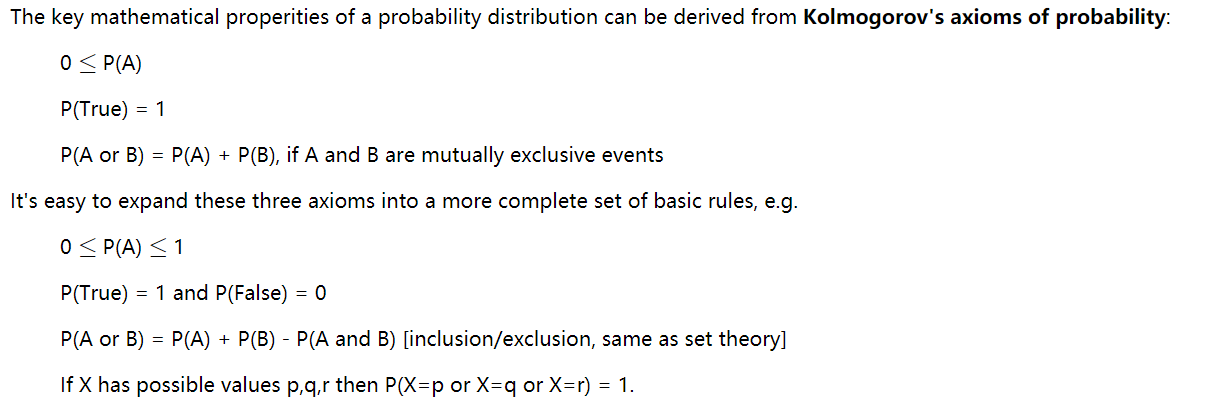


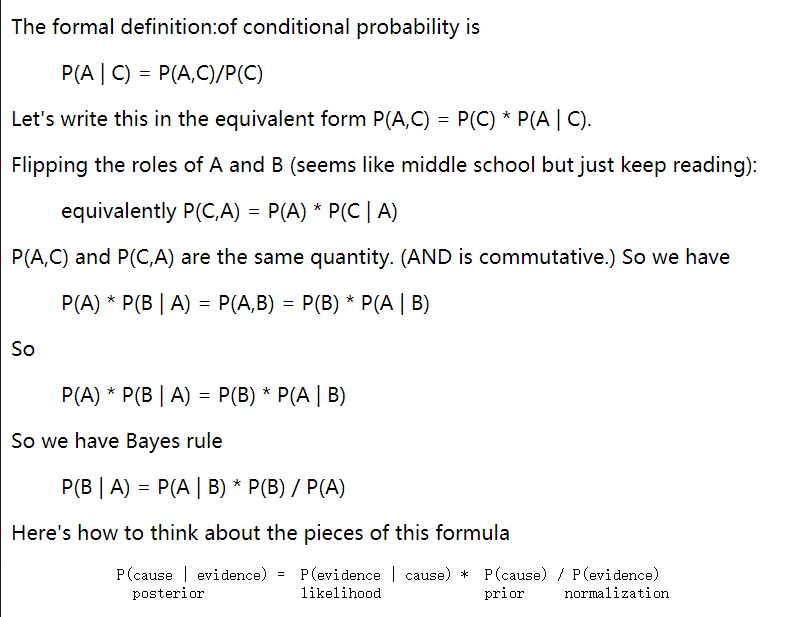


We need a lot of frame axioms, so it's easy for some to be missing. This nuisance is called the "frame problem." It's usually solved by incorporating a default assumption that properties don't change unless actions explicitly change them. (We've been tacitly assuming that.)

**Probability**



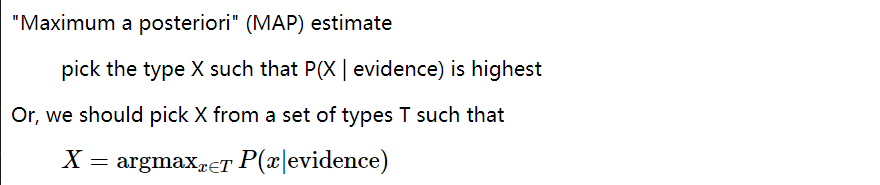


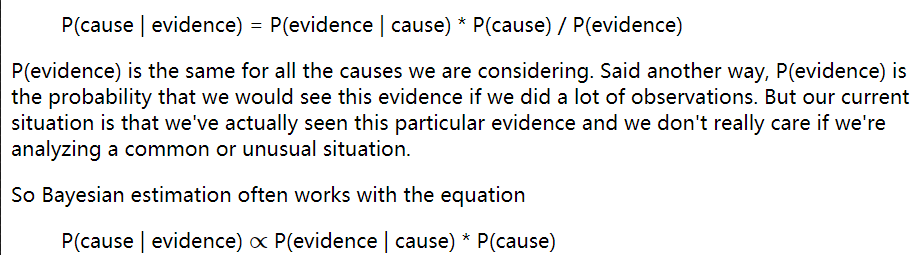


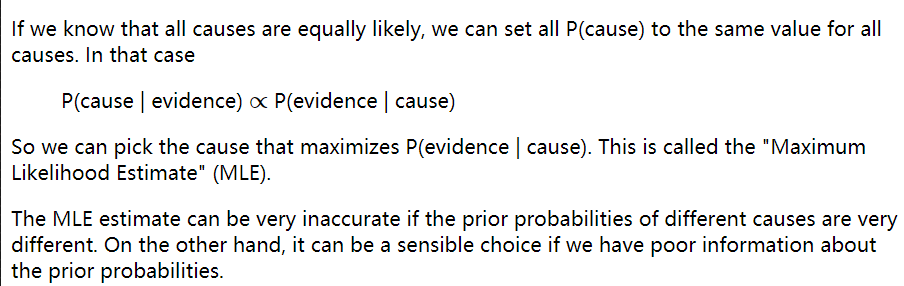
In non-toy examples, it will be easier to get estimates for P(evidence | cause) than direct estimates for P(cause | evidence).

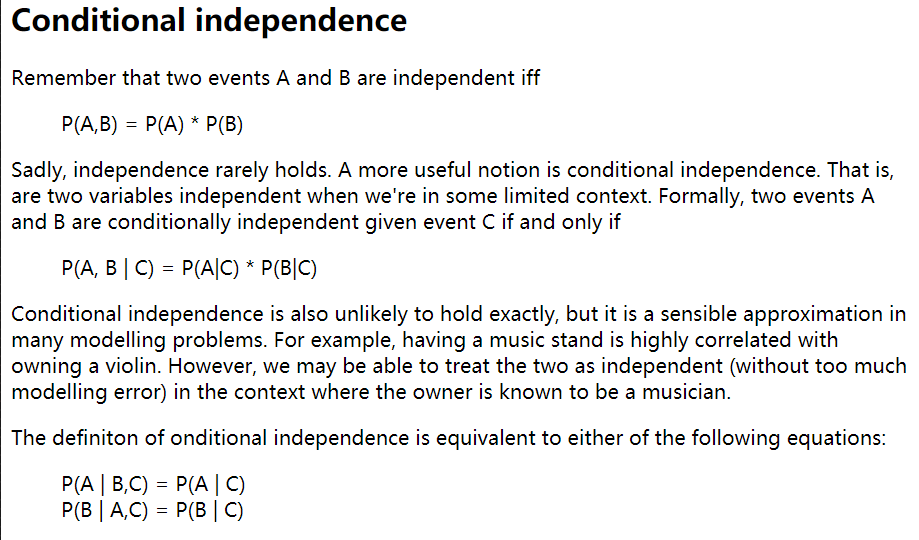
P(evidence | cause) tends to be stable, because it is due to some underlying mechanism.

P(cause | evidence) is less stable, because it depends on the set of possible causes and how common they are right now.









**Basic Naive Bayes model**

