Search

Benefits: • Many problems don't have specific algorithms for solving them • Useful in approximation (e.g., local search in optimization problems) • Some critical aspects of intelligent behaviour, e.g., planning, can be cast as search.

Limitations: Only shows how to solve the problem once we have it correctly formulated.

Hypothetical Reasoning: What state will the agent be in after taking certain actions, or after certain sequences of events

Formalizing a Problem as a Search Problem

1. **State Space**: A state is a representation of a configuration of the problem domain. The state space is the set of all states included in our model of the problem.
2. **Initial State**: The starting configuration.
3. **Goal State**: The configuration one wants to achieve.
4. **Actions (or State Space Transitions)**: Allowed changes to move from one state to another.

Optional Ingredients:

1. **Costs**: Representing the cost of moving from state to state
2. **Heuristics**: Help guide the search process.

e.g.:

- **State Space**: Pair of numbers (a, b), where a denotes number of liters in the 3-liter jug and b denotes number of liters in the 4-liter jug.
- **Actions**: Empty-3-liter, Fill-3-liter, Pour-3-into-4 Empty-4-liter, Fill-4-liter, Pour-4-into-3
- **Initial State**: (0, 4)
- **Goal State**: (2, 0), (2, 1), (2, 2), (2, 3), (2, 4)

Graphical Representation

Assuming a finite search space

- **Vertices** represent states in the search space.
- **Edges** represent transitions resulting from actions (or successor functions).
- A **search tree** reflects the behaviour of an algorithm as it walks through a search problem.
- Has two important attributes:

1. Solution depth, usually denoted by d.
2. Maximum branching factor, usually denoted by b.

- Note that the same state may appear many times in the tree.

States vs Nodes

- A **state** represents a possible configuration of the world.
- A **node** is a data structure constituting part of search tree. It includes:

- a state, - the parent node, - the action that has taken the parent node's state to the current state,

- the cost of the path from the initial node to the current node (if applicable).

- Intuitively speaking, each node corresponds with a path from the initial state to the node's state.
- Two different nodes are allowed to contain same world state.

Algorithms for Search

- Initial **Node**
- **Successor Function** S (x): returns the set of nodes that can be reached from node x via a single action.
- **Goal Test Function** G(x): returns true if node x satisfies the goal condition.
- Action Cost Function C(x, a, y): returns the cost of moving from node x to node y using action a.

(C (x, a, y) = ∞ if y is not reachable from x via a).

Output: A sequence of actions that transforms the initial node to a node satisfying the goal test. The sequence might be, optimal in cost for some algorithms, optimal in length for some algorithms, come with no optimality guarantees from other algorithms.

Algo: • Put nodes have not yet expanded in a list called the **Frontier** (or Open). • Initially, only the initial node is in the Frontier. • At each iteration, pull a node from the Frontier, apply S (x), and insert the children back into the Frontier.
• Repeat until pulling a goal node

```
TreeSearch(Frontier, Sucessors, Goal?)
  If Frontier is empty: return failure
  Curr = select state from Frontier
  If (Goal ?(Curr))  return Curr
  Frontier' = (Frontier – {Curr}) ∪ Successors(Curr)
  Return TreeSearch(Frontier', Successors, Goal)
```

remove the current examined one, add it successor, may add back this node in the future

Critical Properties of Search

- **Completeness**: Will the search always find a solution if a solution exists?

a search algorithm is complete if whenever there is a path from the initial state to the goal, the algorithm will find it.

- **Optimality**: Will the search always find the least cost solution? (when actions have costs)
- **Time complexity**: What is the maximum number of nodes that can be expanded or generated?
- **Space complexity**: What is the maximum number of nodes that have to be stored in memory

Selection Rule

All search techniques keep the Frontier as an ordered set

-> how do we order the nodes on the Frontier?

Uninformed Search Strategies - never look-ahead to the goal.

adopt a fixed rule for selecting the next node to be expand

AI search algorithms work with implicitly defined state spaces.

- Actions are compacted as successor state functions.
- Nodes must contain enough information to allow the successor state function to perform its computation.

BFS & DEFS does not take into account edge weights.

Breadth-First Search (BFS): explores the search tree level by level: • Place the children of the current node at the end of the Frontier. • Frontier is a queue. Always extract first element of the Frontier.

- **Completeness**? Yes! The length of the path (from the initial node to the node removed from the Frontier) is non-decreasing since we replace each expanded node with path length k with a node with path length of k + 1.

complete as long as the state space has a finite branching factor

- **Optimality**: Shortest length solution? Yes! All nodes with shorter paths are expanded prior to any node with longer path. We examine all paths of length < k before all paths of length k. Thus, if there is a solution with length k, we will find it before longer solutions.

Least cost solution? Not necessarily...Shortest solution not always cheapest solution if actions have varying costs.

- **Maximal Branching Factor b**: Maximum number of successors of any node.
- **Depth of the shallowest solution d**: Length of the path from root (at depth 0) to the shortest solution at level d.
- **Time Complexity**: $1 + b + b^2 + .. + b^d + b(b^d - 1) \in O(b^{d+1})$
- **Space Complexity**: $O(b^{d+1})$

In the worst case, only the last node of depth d satisfies the goal. So all nodes at depth d except the last one will be expanded by the search and each such expansion will add up to b new nodes to the Frontier. So we can have up to $b(b^{d+1})$ nodes on the Frontier by the time we stop by expanding a goal node

Typically BFS runs out of space before running out of time.

Assume 1. Expand nodes in layer d prior to discovering the goal 2.data space may not be able to explicitly represented.

Depth-First Search

- Place the children of the current node at the front of the Frontier.
- Frontier is a stack. Always extract first element of the Frontier.

- **Completeness**? No - Infinite paths cause incompleteness!

- Prune paths with cycles to get completeness, if state space is finite. • **Optimality**: No... (complete if finite depth)

- **Time Complexity**: $O(b^m)$ where m is the length of the longest path in the state space.

- Very bad if m is much larger than d, but if there are many solution paths it can be much faster than BFS.

- Using heuristics to determine which successor to explore first can help in getting lucky.

- **Space Complexity**: O(bm). Linear space complexity! A significant advantage of DFS. DFA only explores a single branch of the search tree at a time. Frontier only contains the current node v along with the m descendants of v. Each node can have at most b unexplored siblings and there are at most m nodes on the current branch.

Depth Limited Search (DLS): • Truncate the search by looking only at paths of length D or less.

- Perform DFS but only to a pre-specified depth limit D.

- No nodes with path of length greater than D is placed on the Frontier.

- Benefit: Infinite length paths are not a problem.
- Limitation: Only finds a solution if a solution of depth less than or equal to D exists.

Note: The root of a search tree is at depth 0. A path consisting only of the root is a path of length 0. Recall that the length of a path is equal to the number of actions (edges) in the path.

```
def DLS(start, frontier, successors, goal?, maxd):
    frontier.insert(<start>) #frontier must be a stack for DFS
    cutoff = false
    while not frontier.empty():
       p = frontier.extract() #remove node from frontier
       if (goal?(p.final()):
           return (p,cutoff) #p is solution
       if length(p) < maxd: #Only successors if length(d) < maxd
           for succ in sucessors(p.final()):
               frontier.insert(<p,succ>)
        else:
           cutoff= true. #some node was not expanded because of depth limit
    return (null, cutoff)
```

Iterative Deepening Search (IDS)

- Starting at depth limit d = 0, iteratively increase the depth limit and perform a depth limited search for each depth limit.
- Stop if a solution is found, or if the depth limited search failed without cutting of any nodes because of the depth limit.

- If no nodes were cut of, the search examined all nodes in the state space and found no solution, hence no solution exists.

```
def IDS(start, frontier, successors, goal?):
    maxd = 0
    while true:
       (p, cutoff) = DLS(start, frontier, successors, goal?, maxd)
       if p:
           return p
       elif not cutoff:        #no nodes at deeper levels exit
           return fail
       else:
           maxd = maxd + 1
```

- **Completeness**: Yes!
- **Optimality**: - Shortest length solution? Yes!

  - Least cost solution? Not necessarily...

* Can use a cost bound instead of depth bound.
* Only expand nodes of cost less than the cost bound.
* Keep track of the minimum cost unexpanded node in each iteration, increase the cost bound to that on the next iteration.
* Can be computationally expensive since need as many iterations of the search as there are distinct node costs.

- **Time Complexity**:

$$(d+1)b^0 + db + (d-1)b^2 + .. + b^d \in O(b^d)$$

- **Space Complexity**?: O(bd) Still linear!

IDS vs BFS

- Time complexity of IDS can be better than BFS since it does not expand nodes at the solution depth while BFS (in the worst case) must expand all the bottom layer nodes until it expands a goal node. - With a simple optimization BFS can achieve the same time complexity as IDS.
- Space complexity of BFS is much worse than IDS.

- In practice BFS can be much better depending on the problem: effective cycle checking can be employed with BFS.

- IDS cycle checking will make the space complx as bad as BFS.

Path Checking

- A path p_k is represented as a tuple of states 〈s0, s1, . . . , sk〉, where s0, s1, . . . , sk are the states in p_k in the same order as they appear in pk . • Suppose s_k is expanded to obtain a child successor state c.

The obtained path <s0, s1, . . . , sk , c> can be written as <p_k , c>.

In every path <pk, c>, where pk is the path <s0, s1, . . . , sk>, ensure that the final state c is not equal to any ancestors of c along this path. That is $c \notin \{s_0, s_1, \dots, s_k\}$

- Paths are checked in isolation!
- Advantage: Does not increase time and space complexity.
- Limitation: Does not prune all the redundant states. (e.g., redundant node in sibling)

Cycle Checking (aka Multiple Path Checking)

- Keep track of the all nodes previously expanded during the search using a list called the closed list.

[add to closed list, before expand]

- When we expand n_k to obtain successor c

- Ensure that c is not equal to any previously expanded node. If it is, we do not add c to the Frontier.

- Advantage: Very effective in pruning redundant states.
- Limitation: Expensive in term of space.

Space Complexity: $O(b^d)$ with optimization, $O(b^{d+1})$ without optimization (same as the space complexity of BFS).

For DFS, space complexity linear -> exponential, better use BFS

Cycle Checking and Optimal Cost

- Keep track of each state as well as the known minimum cost of a path to that state.
- If a more expensive path to a previously seen state is found, don't add the corresponding node to the Frontier.
- If a cheaper path to a previously seen state is found, add the corresponding node to the Frontier and

  * Remove other more expensive nodes to the same state from the Frontier or Lazily, ignore these more expensive nodes when/if they are removed for expansion

- If we reject a path <p, c> because we have previously seen state c via a different path p0, it could be that <p, c> is a cheaper path to c than p0

Uniform-Cost Search (UCS) - Finding optimal cost solution

- Always expand the least cost node on the Frontier.

  – priority queue (min heap, key=cost)

- Identical to BFS if all actions have the same cost.

1.Pop least cost node 2.Add successors

e.g. Frontier = {(d,3), (e, 9), (q, 16)} the cost is from to start to cur

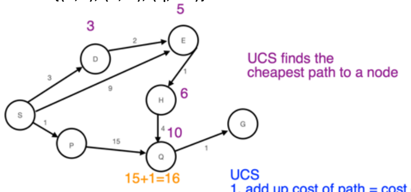


Figure 1: A search problem.

UCS
1. add up cost of path = cost of node
2. take cheapest total path

(Node, Path)	Frontier
	{(S,0)}
(S, S)	{(SP,1), (SD, 3), (SE, 9)}
(P, SP)	{(SD,3), (SE, 9), (SPQ,16)}
(D, SD)	{(SDE,5), (SE,9), (SPQ,16)}
(E, SDE)	{(SDEH,6), (SE,9), (SPQ,16)}
(H, SDEH)	{(SE,9), (SDEHQ,10), (SPQ,16)}
(E, SE)	{(SEH,10), (SDEHQ,10), (SPQ,16)}
(H, SEH)	{(SDEHQ,10), (SEHQ,14), (SPQ,16) }
(Q, SDEHQ)	{(SDEHQG,11), (SEHQ,14), (SPQ,16)}
(G, SDEHQG)	{(SEHQ,14), (SPQ,16)}

• **Completeness**: Yes, under the following condition
- If there is a non-zero constant lower-bound ϵ on the cost of each transition. That is, each transition has costs ⩾ ϵ > 0.
- Under this condition the cost of the nodes chosen to be expanded will be non-decreasing and eventually we will expand all nodes with cost equal to the cost of a solution path.

• **Optimality:**
- Finds minimum cost solution if each transition has cost ⩾ ϵ > 0.
- Explores nodes in the search space in non-decreasing order of cost. So must find minimum cost path to a goal before finding any higher costs paths leading to a goal.

Proof:
Lemma 1: Let c(n) denote the cost of a node n on the Frontier. If a node n2 is expanded after n1 by UCS, then c(n1) ⩽ c(n2).
Lemma 2: Let n be an arbitrary node expanded by UCS in a search space. All the nodes in the search space with cost strictly less than c(n) are expanded before n.
Lemma 3: Let p be the first path UCS find whose final state is a state s. Then p is a minimal cost path to s.
Given that each step cost exceeds some small positive constant ϵ, completeness may be assumed. Consequently:
•Whenever UCS expands a node n, the optimal path to that node has been found. If this was not the case, there would have to be another frontier node n ′on the optimal path from the start node to n. By definition, n ′would have a lower value of g than n, and thus would have been selected first.
• If step costs are nonnegative, paths never get shorter as nodes are added.
The above two points together imply that UCS expands nodes in the order of their optimal path cost.

• **Time and Space Complexity**: $O(b^{floor(\frac{C^*}{\epsilon})+1})$
- UCS has to expand all nodes with cost less than C* and potentially all nodes with cost equal to C*.
- In the worst case, there are : $O(b^{floor(\frac{C^*}{\epsilon})+1})$ nodes with cost less than or equal to C*.
• Note: In the worst case, there are $floor\left(\frac{C^*}{\epsilon}\right)$ actions in a path of cost C*, and each state has b successors.
C* denotes the optimal cost (i.e., minimum cost of paths from the initial state to a goal state).
We also assume each transition has cost ⩾ ϵ > 0.
[adding cycle checking to Uniform Cost Search can improve its efficiency in terms of running time and potentially reduce space complexity by avoiding redundant path explorations. It does not negatively affect the algorithm's completeness or its ability to find a cost-optimal solution.]
Maintenance of ordered frontier adds to space and time complexity … typically employs a priority queue (which takes logarithmic time to update). Additional caveat is that, if the state space can be explicitly represented, time and space bounds can be reduced. Complexity can also be reduced if we institute goal tests prior to placing successors in the queue

Heuristic Search – guess the cost to the goal through node n
• Develop a domain specific heuristic function h(n) such that h(p) guesses the cost of getting to a goal state from a node n
• h(n) is a function only of the state of n. If the states of n1 and n2 are the same, then h(n1) should be equal to h(n2)
- We might even use state, than node, as the argument of h.
• h must be defined so that h(n) = 0 for every goal node.

Greedy Best-First Search
• Use h(n) to rank the nodes on the Frontier.
• Always expand a node with lowest h-value.
- Greedily trying to achieve a low-cost solution.
- Ignores the cost of n, so it can be lead astray exploring paths that cost a lot but seem to be close to the goal.

• Greedy search can be very efficient in practice at finding solutions but that requires developing a good heuristic.
• Greedy search is **incomplete**. • The solution returned by a greedy search can be very far from optimal.

A* Search take into account the cost of the path & heuristic
• define an evaluation function f(n) = g(n) + h(n)
- g(n): the cost of the path to n; h(n): the heuristic estimate of the cost of achieving the goal from n
• always expand the node with lowest f-value on the frontier
• f(n) is an estimate of the cost of getting to the goal via n

With cycle checking

(Node, Path)	Frontier
	{(A: 0+8=8)}
(A, A)	{(AC: 1+7=8), (AB: 4+3=7)}
(B, AB)	{(AC: 1+7=8), (ABC: 6+7=13), (ABD: 10+0=10)}
(C, AC)	{(ACB: 3+3=6), (ACD: 10+0=10), (ABC: 6+7=13), (ABD: 10+0=10)}
(B, ACB)	{(ACBC: 5+7=12), (ACBD: 9+0=9), (ACD: 10+0=10), (ABC: 6+7=13), (ABD: 10+0=10)}

Cycle-checking
1. if we already have a path to that node in frontier: keep only the cheapest path
(only where path ends matters)

No Solution Case: A* as well as all of the uninformed search algorithms, have the same behaviour when there is no solution:
• If there are an infinite number of different states reachable from the initial state, then these algorithms never terminate.
• If there are a finite number of different reachable states (and nodes) and we do either path checking or cycle checking, they will eventually terminate and correctly declare that there is no solution (assuming costs are always ⩾ ϵ > 0).

Completeness
Theorem 1. A* will always find a solution if one exists as long as
1.the branching factor is finite.
2.every action has finite cost greater than or equal to ϵ;
3. h(n) is finite for every node n that can be extended to reach a goal node.
Proof:
• If a solution node n exists, then at all times either (a) n been expanded by A* or (b) an ancestor of n is on the Frontier.
• Suppose (b) holds and let the ancestor on the Frontier be n_i. Then n_i must have a finite f -value.
• As A* continues to run, the f -value of the nodes on the Frontier eventually increase. So, eventually either A* terminates because it found a solution OR n_i becomes the node on the Frontier with lowest f -value.
• If n_i is expanded, then either n_i = n and A* returns n as a solution OR n_i is replaced by its successors, one of which n_{i+1} is a closer ancestor of n.
• Applying the same argument to n_{i+1} we see that if A* continues to run without finding a solution it will eventually expand every ancestor of n, including n itself and so finds and returns a solution.

Admissibility -> optimality
Admissible heuristic
Let h*(n) be the cost of an optimal path from n to a goal node (∞ is there is no path). An admissible heuristic is a heuristic that satisfies the following condition for all nodes n in the search space: $\boldsymbol{h(n) \le h^*(n)}$
[$h^*(n)$ is the actual opt cost, assume exist and known]
To achieve **optimality**, we must put some conditions on h(n) and the search space.
• Each action in the search space must have cost ≥ ϵ > 0.
• h must be admissible.
Intuition:
• An admissible heuristic never over-estimates the cost to reach the goal, i.e., it is optimistic.
• h(n) ⩽ h*(n) implies that the search won't miss any promising paths.
If it really is cheap to get to a goal via n (i.e., both g(n) and h∗(n) are low), then f (n) is also low, and eventually n will be expanded.
Theorem 2. A* with an admissible heuristic always finds an optimal cost solution, if s solution exists and as long as
-the branching factor is finite
- every action has finite cost greater than or equal to ϵ > 0
Proposition 1. A* with an admissible heuristic never expands a node with f -value greater than the cost of an optimal solution.
Proof: Let C* be the cost of an optimal solution.
Let p : $< s_0, s_1, \ldots, s_k >$ be an optimal solution.
So cost(p) = cost($< s_0, s_1, \ldots, s_k >$)= C*.
• It can be shown for each node in the search space that is reachable from the initial node, at every iteration an ancestor of the node is on the frontier. (induction)
• let n be a node reachable from the initial state and $n_0, n_1, \ldots, n_i, \ldots n$ be ancestors of n. So at least one of $n_0, n_1, \ldots, n_i, \ldots n$ is always on the frontier.
• We show that with an admissible heuristic, for every prefix (ancestor) n_i of n we have f (n_i) ⩽ C*:
C* = cost($< s_0, s_1, \ldots, s_k >$)
= cost($< s_0, s_1, \ldots, s_i >$)+ cost($< s_i, \ldots, s_k >$)
= g(n_i) + h*(n_i) by (1)
⩾ g(n_i)+ h(n_i) = f(n_i) by (2)
(1) g(n_i) is equal to cost(n_i) = cost($< s_0, s_1, \ldots, s_i >$)
H*(n_i) is the cost of an optimal path from s_i to any goal state, which must be equal to cost($< s_i, \ldots, s_k >$) since ($< s_0, s_1, \ldots, s_k >$) is optimal.
(2) h*(n_i) ⩾ h(n_i) since h is admissible
• We know that A* always expands a node on the Frontier that has lowest f -value. So every node A* expands has f -value less than or equal to f (n_i), which is less than or equal to C*
Proof: Let C* be the cost of an optimal solution.
• If a solution exists then by **Theorem 1**, A* will terminate by expanding some solution node n.
• By **Proposition 1**, f (n) ⩽ C*.
• Since n is a solution node, we have h(n) = 0. So f(n) = g(n) = cost(n). We also have that C* ⩽ cost(n) = f (n) since no solution can have lower cost than the optimal.
• So cost(n) = C*. That is, A* returns an optimal solution.

Monotone heuristics (consistency)
A monotone (aka consistent) heuristic h is a heuristic that satisfies the triangle inequality: for all nodes n_1, n_2 and for all actions a we have that $\boldsymbol{h(n_1) \le C(n_1, a, n_2) + h(n_2)}$
where $C(n_1, a, n_2)$ denotes the cost of getting from the state of n_1 to the state of n_2 via action a
Theorem 3. Monotonicity implies admissibility. That is
$(\forall n_1, n_2, a) h(n_1) \le C(n_1, a, n_2) + h(n_2) \Rightarrow (\forall \boldsymbol{n})\, \boldsymbol{h(n)} \le \boldsymbol{h^*(n)}$

Proof:
If no path exists from the state of n to a goal, then $h^*(n) = \infty$ and $h(n) \le h^*(n)$.

Else, let n_k be an arbitrary path for which there exists a path from its state to a goal state s_g.
Let $p_{k,g} : \langle s_k, s_{k+1}, \ldots, s_g \rangle$ be an *optiaml* path from the state of n_k to s_g.
The cost of $p_{k,g}$ is $h^*(n_k)$.
We prove $h(n_k) \le h^*(n_k)$ by induction on the length of $p_{k,g}$.

- **Base Case:** $s_k = s_g$.
  By our conditions on h, $h(n_k)$ and $h^*(n_k)$ are equal to zero. So $h(n_k) \le h^*(n_k)$
- **Induction Hypothesis:** $h(n_{k+1}) \le h^*(n_{k+1})$.

$$\begin{aligned} h(n_k) &\le C(n_k, a, n_{k+1}) + h(n_{k+1}) && \text{by monotonicity of } h \\ &\le C(n_k, a, n_{k+1}) + h^*(n_{k+1}) && \text{by IH} \\ &= h^*(n_k) \end{aligned}$$

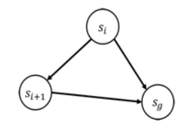
Solution: The proof can be done by induction on $k(s_i)$, which denotes the number of actions required to reach the goal from a node s_i to the goal node s_g.
Base case: ($k = 1$, i.e. the node s_i is one step away from s_g) Since the heuristic function h is consistent,

$$h(s_i) \le c(s_i, s_g) + h(s_g)$$

Since $h(s_g) = 0$,

$$h(s_i) \le c(s_i, s_g) = h^*(s_i)$$

Therefore, h is admissible.
Induction step:



Suppose that our assumption holds for every node that is $k-1$ actions away from s_g, and let us observe a node s_i that is k actions away from s_g; that is, the optimal path from s_i to s_g has $k > 1$ steps. We can write the optimal path from s_i to s_g as

$$s_i \to s_{i+1} \to \ldots \to s_{g-1} \to s_g$$

Since h is consistent, we have

$$h(s_i) \le c(s_i, s_{i+1}) + h(s_{i+1})$$

Now, note that since s_{i+1} is on a least-cost path from s_i to s_g, we must have that the path $s_{i+1} \to s_{i+2} \to \ldots \to s_{g-1} \to s_g$ is a least-cost path from s_{i+1} to s_g as well. By our induction hypothesis, we have

$$h(s_{i+1}) \le h^*(s_{i+1})$$

Admissible if
h(s) ≤
FutureCost(s)
h(s) ≤ h*(s)
'optimistic'

Combining the two inequalities, we have

$$h(s_i) \le c(s_i, s_{i+1}) + h^*(s_{i+1})$$

Note that $h^*(s_{i+1})$ is the cost of the optimal path from s_{i+1} to s_g; by our previous observation (that $s_{i+1} \to s_{i+2} \to \ldots \to s_{g-1} \to s_g$ is an least-cost path from s_{i+1} to s_g), we have that the cost of the optimal path from s_i to s_g, i.e. $h^*(s_i)$, equals $c(s_i, s_{i+1}) + h^*(s_i)$, which concludes the proof.

Consequences of Monotonicity:
• If a node n_2 is expanded after n_1 by A* with a monotonic heuristic, then f (n_1) ⩽ f (n_2).
• With a monotone heuristic, the first time A* expands a node n, n must be a minimum cost solution to n.state.
• **Completeness**: Yes, see in Theorem 1.
• **Optimality**: With admissible heuristic, Yes. See in Theorem 2.
• **Space and Time Complexity**:
- When h(n) = 0, for all n, h is monotone. A* becomes uniform-cost search!
- It can be shown that when h(n) > 0 (for some n) and an admissible, the number of nodes expanded can be no larger than uniform-cost search.

Hence the same bounds as uniform-cost apply: $O(b^{floor(\frac{C^*}{\epsilon})+1})$
Still exponential unless we have a very good h!
-In real-world problems, we sometimes run out of time and memory.

Prove optimality with cycle checking

Solution: The proof uses the following notation: The function $\hat{g}(n)$ denotes the current best known path cost from the initial state to node n. Note that the value of $\hat{g}(n)$ is updated during the execution of the algorithm. The function $g(n)$ denotes the minimum path cost from an initial state to state n.

g = sum c
along optimal path

The function $h(n)$, known as a heuristic, denotes the approximated path cost from state n to the (nearest) goal state.

The functions $f(n)$ and $\hat{f}(n)$ are evaluation functions that are adopted by the informed search algorithm in question. For example, in the case of the greedy best-first search algorithm, $f(n) = \hat{f}(n) = h(n)$; in the case of the A^* search algorithm, $f(n) = g(n) + h(n)$ and $\hat{f}(n) = \hat{g}(n) + h(n)$. $\hat{f}_{\text{pop}}(n)$ denotes the value of $\hat{f}(n)$ when it is popped from the frontier.

^: estimate to be updated
(heuristic is fixed)

Proof: Mathematically, we would want to prove that $\hat{f}_{\text{pop}}(s_g) = f(s_g)$, i.e. when the goal node s_g is popped from the frontier, we would have found the optimal path to it. Let

$$s_0, s_1, \ldots, s_{g-1}, s_g$$

be the path from the start node s_0 leading to the goal node s_g.

Base case: $\hat{f}_{\text{pop}}(s_0) = f(s_0) = h(s_0)$.

Induction step: Assume that for all $s_0, s_1, \ldots, s_k$, $\hat{f}_{\text{pop}}(s_i) = f(s_i)$. We know that

$$\begin{aligned}\hat{f}_{\text{pop}}(s_{k+1}) &= \hat{g}_{\text{pop}}(s_{k+1}) + h(s_{k+1}) \\ &\geq g(s_{k+1}) + h(s_{k+1}) \\ &= f(s_{k+1}) \qquad (1)\end{aligned}$$

consistent: h(s) ≤ c(s, next) + h(next)

In order to make sure that each s_{k+1} is only explored after when we pop s_k, the condition of $f(s_i) \leq f(s_{i+1})$ is required, leading to the need for the consistency of h. By popping s_k, we have:

g(next) ≤ g(s) + c(s, next)
g ≤ g^pop ≤ g^
g = g^pop if optimal

$$\begin{aligned}\hat{f}_{\text{pop}}(s_{k+1}) &= \min\{\hat{f}(s_{k+1}), \hat{g}_{\text{pop}}(s_k) + c(s_k, s_{k+1}) + h(s_{k+1})\} \\ &\leq \hat{g}_{\text{pop}}(s_k) + c(s_k, s_{k+1}) + h(s_{k+1}) \\ &= g(s_k) + c(s_k, s_{k+1}) + h(s_{k+1}) \qquad \text{from our IH} \\ &= g(s_{k+1}) + h(s_{k+1}) \\ &= f(s_{k+1}) \qquad (2)\end{aligned}$$

From equations 1 and 2, we obtain $\hat{f}_{\text{pop}}(s_{k+1}) = f(s_{k+1})$. Hence, by induction, whenever we pop a node from the frontier, the optimal path to the node would have been found.

IDA* Iterative Deepending A*

- Reduce memory requirements for A*
• Like iterative deepening, but now the cut-off is the f -value rather than the depth.
• At each iteration, the cut-off value is the smallest f -value of any node that exceeded the cut-off on the previous iteration.
• Avoids overhead associated with keeping a sorted queue of nodes, and the Frontier occupies only linear space.

How to build a heuristic?

Simplify a problem when building heuristics and let h(n) be the cost of reaching the goal in the easier problem.
Example: In the 8-Puzzle we can only move a tile from square A to B if A is adjacent (left, right, above, below) to B and B is blank.
We can relax some of these conditions and:
1. allow a move from A to B if A is adjacent to B (ignore whether or not position is blank); leads to the Manhattan distance heuristic
2. allow a move from A to B if B is blank (ignore adjacency);
3. allow all moves from A to B (ignore both conditions).
leads to the misplaced tiles heuristic

Admissible Heuristics:
• h(n) : number of misplaced tiles in n.state.
• h(n): total Manhattan distance between tile locations in n.state and goal locations

Recap:
Ignore weights / cost: BFS, DFS
Optimal, but uninformed: UCS
Informed, but ignores cost: Greedy Best-First
Informed with cost: A*
Use costs -> more optimal
Use heuristics -> faster

Constraint Satisfaction Problems

CSPs do NOT require finding a path (to a goal). They only need the configuration of the goal state.
• Represent states as values assigned to vectors of features.
- A set of k variables (known as **features**).
- **Each variable** has a domain of different values.
- A **state** is specified by an assignment of values to all variables.
- A **partial state** is specified by an assignment of a value to some of the variables.
• A **goal** is specified as conditions on the vector of feature values.
• **Solving a CSP**: find a set of values for the features (variables) so that the values satisfy the specified conditions (constraints).

Formalization of a CSP
• A set of **variables** V1, ..., Vn; e.g. tom, piano
• A (finite) **domain** of possible values Dom[v_i] for each variable Vi;
• A set of **constraints** C1, ..., Cm. e.g. {1,2,3}
• Each variable v_i can be assigned any value from its domain: Vi = d where d ∈ Dom[v_i]
• Each constraint C
- Has a set of variables it operates over, called its scope.
Example: The **scope** of C(V1, V2, V4) is {V1, V2, V4}
- Given an assignment to variables C is
 True if the assignment satisfies the constraint;
 False if the assignment falsifies the constraint.
• **Solution to a CSP**: An assignment of a value to all of the variables such that every constraint is satisfied.
• A CSP is unsatisfiable if no solution exists.
• **Unary Constraints** (over one variable) C(Y) : Y > 5
• **Binary Constraints** (over two variables) C(X, Y) : X + Y < 6
• **Higher-order constraints**: over 3 or more variables: ALL-Diff(V1, .., Vn): V1 ≠ V2,..., Vn ≠ Vn-1.

N-Queens
• **Variables**: N variables Q_i, one per row.
• **Domains**: Value of Q_i is the column the Queen in row i is placed. Possible values {1, .., N}. # of config = N^N
Constraints: • Cannot put two Queens in same column: for all i, j, if i = j, then Q_i= Q_j
• Diagonal constraints: abs(Q_i, Q_j) ≠ abs(i j)

CSP Backtracking Search
• Searching through the space of partial assignments, rather than paths.
• Decide on a suitable value for one variable at a time. Order in which we assign the variables does not matter.
• If a constraint is falsified during the process of partial assignment, immediately reject the current partial assignment.

CSP Search Tree:
• **Root**: Empty Assignment.
• **Children of a node**: all possible value assignments for a particular unassigned variable.
• The tree **stops descending** if an assignment violates a constraint.
• **Goal Node**: (1) The assignment is complete
(2) No constraints is violated.

We will apply a **recursive** implementation:
- If all variables are set, print the solution and terminate.
- Otherwise:
  - Pick an unassigned variable V and assign it a value.
  - Test the constraints corresponding with V and all other variables of them are assigned.
  - If a constraint is unsatisfied, return (backtrack).
  - Otherwise, go one level deeper by invoking a recursive call.

```
def BT(Level):
1.  if all Variables assigned
2.     PRINT Value of each Variable
3.     EXIT or RETURN                  # EXIT for only one solution
                                       # RETURN for more solutions
4.  V := PickUnassignedVariable()
5.  Assigned[V] := TRUE
6.  for d := each member of Domain(V)  # the domain values of V
7.     Value[V] := d
8.     ConstraintsOK := TRUE
9.     for each constraint C such that (i) V is a variable of C and
                                      (ii) all other variables of C are assigned:
10.        if C is not satisfied by the set of current assignments:
11.              ConstraintsOK := FALSE
12.    if ConstraintsOk == TRUE:
13.        BT(Level+1)
14.  Assigned[V] := FALSE     # UNDO as we have tried all of V's values
15.  RETURN
```

In CSPs, there might be variables that have no possible value, but BT doesn't detect this until it tries to assign them a value.

CSP & Inference

1. pop first var in queue
2. Update domains of other vars (fill in possible numbers for var into constraints, see which number other vars can have)
3. Add updated vars to queue (do not add var if var is already in the queue)
Once queue is empty: finished or choose

Idea: After assigning a value to a variable, infer the obvious restrictions imposed by the current assignments on values that unassigned variables can take and reduce their domains accordingly. "Obvious" means things we can test/detect efficiently.
• Inference can be applied during the search; potentially at every node of the search tree.
• An inference step needs some resources (in particular, time).
If the inference procedure is slow, this can slow the search down to the point where using it makes finding a solution take longer!

• Every time we assign a value to a variable V, we check all constraints over V and prune values from the **current domain** of the unassigned variables of the constraints.
- **What values are pruned?**
  Let C be a constraint that includes V in its scope and V_i be a variable (other than V) in the scope of C.
  All values in the current domain of V_i must have a supporting assignment. If a value doesn't have any supports, it must be **pruned**.
- A value d in the current domain of V_i has a **supporting assignment** if there exist at least one value assignment for other variables of C such that C **is satisfied** under V_i = d and that value assignment.

• Removing a value from a variable domain may remove a support for other domain values.
We should **repeat** the procedure until **all remaining values have a support**:
- Have a queue of variables that need to be checked.
- A variables is added (back) to the queue if its domain is changed.
- The procedure **stops** when the queue is empty.

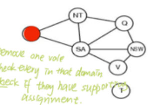
• After backtracking from the current assignment the values that were **pruned** (as a result of that assignment) must be **restored**.
Some **bookkeeping** needs to be done to remember which values were pruned by which assignment.

constraint : neighbor can't have same color

$C_1(SA,WA): SA \neq WA$, $C_2(NT,WA): NT \neq WA$, $C_3(SA,NT): SA \neq NT$
$C_4(SA,Q): SA \neq Q$, $C_5(SA,NSW): SA \neq NSW$, $C_6(SA,V): SA \neq V$
$C_7(NT,Q): NT \neq Q$, $C_8(Q,NSW): Q \neq NSW$, $C_9(NSW,V): NSW \neq V$

Value Assignments: $WA = R$ CurDom[WA] = {R}
Then SA and NT must be put on the queue.

Current Domains: R- No supporting assignment.   R   NT≠R
$CurDom[SA] = \{\cancel{R}, G, B\}$ $CurDom[NT] = \{\cancel{R}, G, B\}$
$CurDom[Q] = \{R, G, B\}$ $CurDom[NSW] = \{R, G, B\}$
$CurDom[V] = \{R, G, B\}$ $CurDom[T] = \{R, G, B\}$



remove one vale
check every in that domain
check if they have supporting assignment.

| ~~WA~~ | ~~SA~~ | ~~NT~~ | Q | NSW | V | SA |

we don't put WA again
SA   NT
NT已有，就不再加了
{R, G, B}   check每个value有无 supporting Assign:
eg 如果 Q=R, 它的neighbor NT, SA, NSW, V 有 support?
没有就prune R
add Q的neighbor 到 Queue
已有的就不再加了.
可以check有没有duplicate => no sign influence on RT.

CSC384H | University of Toronto   10

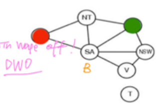
domain wipe out

$C_1(SA,WA): SA \neq WA$, $C_2(NT,WA): NT \neq WA$, $C_3(SA,NT): SA \neq NT$
$C_4(SA,Q): SA \neq Q$, $C_5(SA,NSW): SA \neq NSW$, $C_6(SA,V): SA \neq V$
$C_7(NT,Q): NT \neq Q$, $C_8(Q,NSW): Q \neq NSW$, $C_9(NSW,V): NSW \neq V$

Value Assignments: $WA = R, Q = G$
Then SA, NT and NSW must be put on the queue.

domain wipe out
DWO

Current Domains:
$CurDom[SA] = \{\cancel{G}, B\}$ $CurDom[NT] = \{\cancel{G}, \cancel{B}\}$ => Domain wipe off! DWO
$CurDom[Q] = \{R, G, B\}$ $CurDom[NSW] = \{R, G, B\}$
$CurDom[V] = \{R, G, B\}$ $CurDom[T] = \{R, G, B\}$



| ~~WA~~ | ~~Q~~ | ~~SA~~ | ~~NT~~ | NSW | SA |

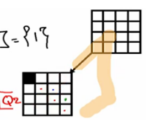
~~G~~   ~~G~~ ~~B~~
check if NT=B 有 support:
SA={A}. No support for SA.
NT=B pruned.

DWO -> try another value of Q

CurDom[Q]={1}

Value Assignments: $Q_1 = 1$
Then Q_2, Q_3, Q_4 must be put on the queue.

| ~~Q1~~ | ~~Q2~~ | ~~Q3~~ | ~~Q4~~ | ~~Q2~~ | ~~Q3~~ | Q4 | Q2 |



→DWO

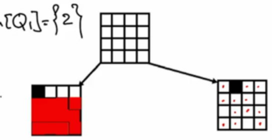
Current Domains:
CurrDom$[Q_2] = \{\cancel{1}, \cancel{2}, \cancel{3}, 4\}$ CurrDom$[Q_3] = \{\cancel{1}, \cancel{2}, \cancel{3}, \cancel{4}\}$ CurrDom$[Q_4] = \{\cancel{1}, 2, 3, \cancel{4}\}$

Q2=3 : No support in CurrDom[Q3]
Q3=4 : " " " CurrDom[Q2]
Q3=2 : " " " CurrDom[Q4]

- Remove a node from queue, check each of its value in its domain, does each of them have supporting assignment, based on related node's current domain
- if no supporting assignment, prune this value in the domain -> add affected nodes to queue
Another way:
-Pop a node from queue, check its related node's node (don't over check), if domain change, add the related node in queue.

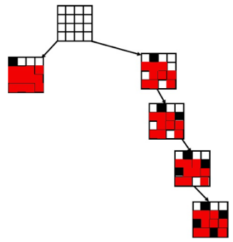
CurDom[Q1]={2}

Value Assignments: $Q_1 = 2$
Then $Q_2 = 1, Q_2 = 2, Q_2 = 3, Q_3 = 2,$
$Q_3 = 4, Q_4 = 2$ become arc inconsistent.



Current Domains:
CurrDom$[Q_2] = \{\cancel{1}, \cancel{2}, \cancel{3}, 4\}$ CurrDom$[Q_3] = \{1, \cancel{2}, \cancel{3}, \cancel{4}\}$ CurrDom$[Q_4] = \{\cancel{1}, \cancel{2}, 3, \cancel{4}\}$

Current Domains: $CurDom[Q_2] = \{4\}, CurDom[Q_3] = \{1\}, CurDom[Q_4] = \{3\}$.
Now search no longer has to branch since only one value left for each variable.
It just walks down to a solution assigning each variable in turn.



The **outcome** of an inference step is one of the followings:
1.each domain has a single value
2.at least one domain is empty -> backtrack
3.some domains have more than one value.
Need to solve this new CSP **simpler** problem:
Same constraints, domains have been reduced

Inference: The Algorithm

```
def BT_with_Inference(Level)
1.  if all Variables assigned
2.      PRINT Value of each Variable
3.      EXIT or RETURN                # EXIT for only one solution
                                      # RETURN for more solutions
4.  V := PickUnassignedVariable()
5.  Assigned[V] := TRUE
6.  for d := each member of CurDom(V)
7.      Value[V] := d     assign value to node V   start inference
8.      Prune all values other than d from CurDom[V]
9.  DWO if(Inference(V) != DWO)   Domain wipe out
10. backtrack  BT_with_Inference(Level+1) # all remaining domain values are ok
11.     RestoreAllValuesPrunedByInference()
12.  Assigned[V] := FALSE     # UNDO as we have tried all of V's values
13.  RETURN
```

```
def Inference(var)   push node that assigned value
1.  VarQueue.push(var)
2.  while VarQueue not empty
3.      W := VarQueue.extract()   extract one node from Queue
4.      for each constraints C where W∈scope(C)   Constraints include W
5.          for V := each member of scope(C) \ W   go over all variables in the scope of that constraint
6.              S := CurDom[V]   S = {1,2,3}
7.  每个 for d := each member of CurDom[V]
8.              Find an assignment A for all other variables in scope(C)
                such that C(A ∪ V=d) is True   supporting assignment
9.              if A not found
10.                 CurDom[V] = CurDom[V] - d   # remove d from the domain of V
11.                 if CurDom[V] == {}    # DWO for V
12.                     empty VarQueue
13.                     return DWO     # return immediately
14.     if CurDom[V] ≠ S   curr{1,2}   # if domain of V has changed
15.         VarQueue.push(V) -> 把V加到Q里
16. return TRUE   # loop exited without DWO
    keep track Curr Domain
    Restore if backtrack.
```

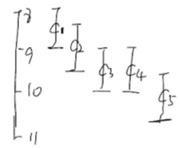
Def inference():
Vars = ['a', 'b', 'c']
Constraints =[]
Domain {a:[1,2,3],...}
Queue = []
For var in queue:
var = queue.pop(0)
For possible_value in domain(var):
Var = possible_value
For constraint in constrains:
For other_var in constraints
Updated = update_domain(other_var)
If updated and other_var not in queue:
Queue.add(other_var)
For var in vars:
If not domain(Var):
Return false

• Computation for inference function can be expensive, and may potentially slow down the search. • Alternative implementation attempts to limit the depth of the inference in order to reduce computational cost. • There is a trade off between the information that can be derived from **deeper levels of inference** and **run-time** efficiency.

Variable popped/inferred	Domain update	Queue
x_2	$x_1 = \{1,2\}$ $x_2 = \{1,2,3\}$ $x_3 = \{1,2\}$ $x_4 = \{1,2,3\}$	$\{x_1, x_3\}$
x_1	$x_1 = \{1,2\}$ $x_2 = \{1,2\}$ $x_3 = \{2\}$ $x_4 = \{1,2,3\}$	$\{x_3, x_2\}$
x_3	$x_1 = \{2\}$ $x_2 = \{1\}$ $x_3 = \{2\}$ $x_4 = \{\emptyset\}$	$\{x_2, x_1\}$

Dom(C_1) = {Bob}
C_2 {~~Bob~~, Jerry}
C_3 {Anna, ~~Jerry~~, Bob}
C_4 {Anna, ~~J~~, B}
C_5 {Jerry, Bob}

Constraints
$C_1 \neq C_2$ $C_2 \neq C_3$ $C_2 \neq C_4$ $C_3 \neq C_4$
$C_1 \neq C_3$ $C_3 \neq C_4$



Queue
pop C_1
$C_2 \neq$ Bob C_2
pop C_2
C_1
$C_3 \neq$ Jerry C_3 C_4
$C_4 \neq$ Jerry
pop C_3
C_2
C_4 C_4 C_5
$C_5 \neq$ ~~Bob~~
C_5 可以为 Bob
与 C_5 相关 2 个 constraint:
C_5: Bob { $C_3 \neq C_5$ => C_3 = Anna; $C_4 \neq C_5$ => C_4 = Anna } > 不需再想这个!!
If constraint 里 all variable 有无 supporting assi

∴ Cur Domain
C_1 {B}
C_2 {J}
C_3 {A, B}
C_4 {A, B}
C_5 {J ~~J~~ B}

1. Consider the following constraint satisfaction problem:

Variables: X, Y, Z
Domains: $D_X = D_Y = D_Z = \{0, 1, 2, 3, 4\}$
Constraints: $C_1 := (X = Y + 1)$
$C_2 := (Y = 2Z)$

x{1, 2, 3, 4}
y{0, 1, 2, 3, 4}
z{0, 1, 2, 3, 4}

Assuming that initially no variable has been assigned and var = Y, perform inference on these constraints.

learned	Inference	Queue
pop Y		Y
$X \neq 0$	C_1	X, Z
$Z \neq 3,4$	C_2	
pop X		
$Y \neq 4$	C_1	Z Y
~~$X \neq 2$~~		
pop Z		
$Y \neq \{1,3\}$	C_2	Y
pop Y		
$X \neq \{2,4\}$	C_1	X Z
$Z \neq \{2\}$	C_2	
		Z Y
		Y
		$\emptyset$

Y X Z Y X Z Y
对于 pop 出来的 node 可以 check 它的 domain 是否变化 (没上)
也可以 check 与它 relate 的 node 的 domain 是否变化 (见上)
不要写!
→ pop x → 与它 relate 的 node 不看 domain.

Forward Checking

Forward Checking: The Algorithm

```
def FC_Inference(var)        only look at first level
1.  VarQueue.push(var)        don't check influence of prune
2.  while VarQueue not empty   W=d
3.      W := VarQueue.extract()   W=d
4.      for each constraints C where W∈scope(C)
5.          for V := each member of scope(C) \ W
6.              S := CurDom[V]
7.              for d := each member of CurDom[V]
8.                  Find an assignment A for all other variables in scope(C)
                    such that C(A ∪ V=d) is True
9.                  if A not found
10.                     CurDom[V] = CurDom[V] - d   # remove d from the domain of V
11.                     if CurDom[V] == {}    # DWO for V
12.                         empty VarQueue
13.                         return DWO       # return immediately
14.     X # if CurDom[V] ≠ S
15.       #     VarQueue.push(V)
16. return TRUE   # loop exited without DWO
```

Forward Checking: Don't add variables to VarQueue at every iteration.
That is, *remove Lines 14 and 15* of $Inference(var)$

Other alternative

Backtracking with Inference: Other Alternatives

- Forward Checking can rule out many inconsistencies but it does not infer further ahead to ensure consistency for all the other variables.
- **Alternative:** Replace the $CurDom[V] \neq S$ condition (Line 14) with $CurDom[V] == 1$. (check this early on; only consider influence that contain only 1 remaining value) That is, find inference for variables that have *only ONE valid value* in the current domain.
  This goes one *step further* from Forward Checking, and allows for further inferences for the variables that have one valid value in the domain.
- Depending on the complexity of the problem, the condition can be set to any particular value, e.g $CurDom[V] \leq 2$, $CurDom[V] \leq 3$, and so on.

Backtracking with Inference: Other Alternatives

```
def Alternative_Inference(var)
1.  VarQueue.push(var)
2.  while VarQueue not empty
3.      W := VarQueue.extract()
4.      for each constraints C where W∈scope(C)
5.          for V := each member of scope(C) \ W
6.              S := CurDom[V]
7.              for d := each member of CurDom[V]
8.                  Find an assignment A for all other variables in scope(C)
                    such that C(A ∪ V=d) is True
9.                  if A not found
10.                     CurDom[V] = CurDom[V] - d   # remove d from the domain of
11.                     if CurDom[V] == {}    # DWO for V
12.                         empty VarQueue
13.                         return DWO       # return immediately
14.             if CurDom[V] == 1
15.                 VarQueue.push(V)
16. return TRUE   # loop exited without DWO
```

• In general, solving a CSP problem in the worst-case can take **exponential time**. general class of CSPs is **NP-complete**.
• But, typically, every NP-complete family contains large sub-classes of simpler problems.
• The purpose of developing **inference techniques and heuristics** is to solve those simpler sub-classes faster. For example:
- FC often is about 100 times faster than plain BT.
- FC with the MRV heuristic (Minimal Remaining Values) often 10000 times faster.
- On some problems the speed up can be much greater.
Converts problems that are not solvable to problems that are solvable.

Variable and Value Ordering Heuristics

• Heuristics can be used to determine
-> the **order** in which variables are **assigned**:
PickUnassignedVariable()
-> the **order** of **value**s tried for each variable.
• The choice of the next variable can vary from branch to branch.
Example: Under the assignment V1 = a we might choose to assign V4 next, while under V1 = b we might choose to assign V5 next.
• This **dynamically** chosen variable ordering has a tremendous impact on performance

Degree Heuristic: Select the **variable** that is involved in the largest number of constraints on other unassigned variables.
->drive more inference
-> The most constrained variable heuristic makes sense because it chooses a variable that is (of all other things being equal) likely to cause a failure, and it is more efficient to fail as early as possible
2 (thereby reducing the search space).

Minimum Remaining Values Heuristics (MRV):
• Always branch on a **variable** with the smallest remaining values (smallest CurDom). (use min heap)
->faster to identity inconsistency

Least Constraining Value Heuristic:
• Always pick a **value** in CurDom that rules out the **least domain values** of other neighboring variables in the constraint.
This allows for the maximum flexibility for subsequent variable assignments.
->higher probability to find solution
-> The least constraining value heuristic makes sense because it allows the most opportunities for future assignments to avoid conflict.

Example: Map Colouring

- $\{SA = red, NT = blue, Q = green, NWS = blue\}$ (using MRV and Degree Heuristic)

we didn't backtrack heuristic to pick variable.
use FC & MRV -> degree
=> one branch of search tree.
X Full inference. -> use F.C.
forward checking prune
un assigned value