1 Constraint Satisfaction Problem

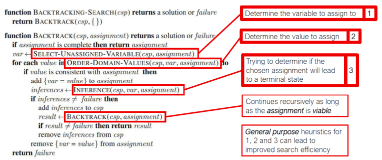
- Aims to improve on systematic search as they tend to be computationally expensive by reducing search space
- Idea is to use factored representation of states (variables $X = \{x_1, \cdots, x_n\}$ where each has domain $D_i = \{d_1, \cdots, d_m\}$)
- If a state satisfies all constraint then it is a goal state
- CSPs systematically search for goal states by pruning invalid subtrees as early as possible

1.1 CSP Formulation

- State representation
  - Variables: $X = x_1, ..., x_n$
  - Domains: $D = d_1, ..., d_k$
  - Initial state: All variables unassigned
  - Intermediate state: Partial assignment
- Goal Test: Each c_i in constraints $C = c_1, ..., c_m$ are satisfied (Consistency) and all variables are assigned valid values (Complete)
- Actions: Assignment of values to variables (cost not used)

1.2 Solving CSPs

- We don't care about search path
- Use DFS to save space
- Backtracking algorithm: at each depth l: $(|X| - l \cdot |d|)$ states, total number of leaf states: d^n (if order of variable not impt) else $n!m^n$, where $|X|$ = number of variable, d is the domain and $m = |d|$



1.3 Variable-Order Heuristics

1.3.1 Minimum-Remaining-Values

- Choose the variable with fewest legal values (most constraint variable)
- Idea is to place larger subtrees near the root so that we can eliminate larger subtrees earlier if we find any invalid states
- Performs better than static or random ordering

1.3.2 Degree Heuristic

- Tie-breaking mechanism of MRV
- Picks variables with most constraints relative to unassigned variables
- Idea is that by selecting variable that restricts the most number of other variables, we reduce branching factor

1.4 Value Order Heuristic

1.4.1 Least-Constraining-Value Heuristic

- Choose the value that rules out the fewest choices (most choices)
- Idea is that when picking values we want to avoid failures (empty domains) to get to a solution as fast as possible

1.5 Strategy in picking Variable vs Values

- With **variables**, we want to **fail fast** since it typically leads to fewer successful assignments to backtrack over
- With **values**, want to **fail last** since it allows us to have more options and thus have a higher probability of reaching a goal node via DFS

1.6 Inference Algorithms

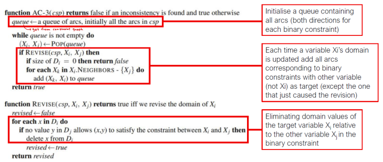
1.6.1 Forward Checking

- Track remaining legal values for unassigned variables and **terminate search when any variable has no legal values**
- Does not provide early detection for failures

1.6.2 Constraint Propagation

- Inference step to ensure local consistency of ALL variables
- After action taken, traverse constraint graph and ensure domain of each variable are both **node** (unary constraint) and **arc** (binary constraint) consistent
- Node-consistency (unary constraint) done as **pre-processing**
- To maintain Arc Consistency, remove any value from the target variable if it makes a constraint impossible to satisfy
- Arcs are directed (binary constraint is 2 arcs)
- Arc consistency is ran during backtracking or as a pre-processing step. Detects failure earlier than forward checking
- For AC, if we check arc(X_a, X_b) and propagate to check arc(X_b, X_a), we don't have to check arc(X_a, X_b) again!

1.7 AC-3 Algorithm



1.7.1 Time Complexity of AC-3

- CSPs have at most $2 \cdot {}^nC_2$ or $O(n^2)$ directed arcs (n variables)
- Each arc (X_i, X_j) can be inserted at most d times because X_i has at most d values to delete (given domain size d) - Checking consistency of an arc takes $O(d^2)$ time
- Total time complexity: $O(n^2 \times d \times d^2) = O(n^2 d^3)$

1.7.2 Maintaining Arc Consistency

- Run AC as pre-processing step (since it takes a long time to run) to reduce domain sizes and branching factor of search tree
- After assignment, use forwards checking as inference (does not ensure arc consistency)

2 Adversarial Search

2.1 Games and Search

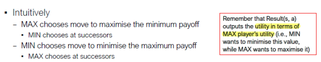
- Assumes a zero-sum game (winner gets paid, loser pays) where there are 2 players - MIN and MAX
- Simulating a play against a **utility** maximizing opponent
- **Winning Strategy**: p_1 WINS for any strategy p_2 takes
- **Non-losing strategy**: p_1 WINS or TIES for any strategy p_2 takes

2.1.1 Formulating Games

- **State** - as per normal search problem
- **TO-MOVE**(state) - returns which player's turn to move given current state
- **ACTION**(state) - Legal moves in state s
- **RESULT**(s, a) - Transition model, returns resultant state after taking action a at state s
- **IS-TERMINAL**(s) - returns whether game is over
- **UTILITY**(s, p) - Gives a numerical value to player p when game ends in a **terminal state** s
  - If assuming zero sum game: UTILITY(MAX, s) + UTILITY(MIN, s) = 0

2.2 Minimax Algorithm

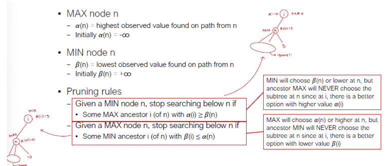
$$Minimax(s) = \begin{cases} Utility(s, \text{MAX}) & \text{if Is-Terminal}(s) \\ \max_{a \in Actions(s)} \text{Minimax}(\text{Result}(s,a)) & \text{if To-Move}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{Minimax}(\text{Result}(s,a)) & \text{if To-Move}(s) = \text{MIN} \end{cases}$$



- Complete if game tree is finite and optimal (given optimal gameplay)
- MAX chooses the move that maximises MAX player utility, MIN chooses move to minimises MAX player utility
- Time: $O(b^m)$, Space: O(bm) - follows DFS
- **Limitation**: In most cases game trees are massive (chess has 10^{123} nodes) and we cannot expand entire tree

2.3 $\alpha - \beta$ Pruning

- Idea is to not explore moves that would never be considered
- Maintain bounds on values seen thus far while searching
  - α bounds MAX's values (highest MAX seen so far)
  - β bounds MIN's values (lowest MIN seen so far)



- Ordering matters for α-β pruning!!
  - "Perfect ordering" will have a time complexity of $O(b^{m/2})$
  - Random ordering will have complexity of $O(b^{3/4m})$
- Faces issues with max depth of tree (traverses to terminal states)
- Solved using **heuristic minimax** - cutoff test (e.g. DLS or IDS) or using evaluation function to estimate expected utility of state

3 Knowledge Representation

3.1 Recap: Problem Solving Agents

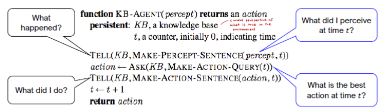
- Tries to find solution via Search
- No real model of what the agent knows
  - Each state contains knowledge on state of the whole environment - transition models, actions, implicit knowledge of environment (e.g. path finding has non -ve road lengths)
  - Atomic representations limiting - e.g. in minesweeper game, environment is partially observable and agent would not know where all mines are

3.2 Knowledge-Based/Logical Agents

- Represent agent domain knowledge using logical formulas
- Idea: Make inference on existing information - use old knowledge to infer new knowledge
- States similar to CSPs - assignments of values to variables
- Agent contains knowledge base and inference engine
- Cannot plan entire path to goal since environment is only **partially observable**

3.2.1 Knowledge Base

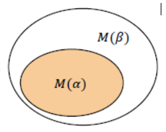
- Set of sentences in a formal language (expressive and parsable)
- Pre-populate with domain knowledge (rules, general knowledge)
- **Declarative** approach to problem solving
  - Tell it what it needs to know - update percepts/state/action
  - Ask itself what to do = make inferences based on KB on what actions to take



- Agent must be able to represent states/actions, incorporate new percepts, update internal representation of environment and deduce hidden environment properties and corresponding actions

3.3 Entailment

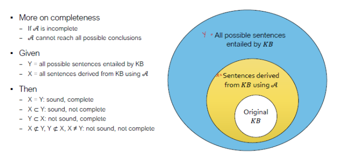
- v models α if α is true under v (if v makes α true, it models α)
  - v is one set of value assignments (applied to sentences α)
  - v corresponds to one instance of the environment (known part of a state)
- Let $M(\alpha)$ be the set of all models for α
- Entailment ($\models$) means that one thing follows from the other - e.g. $\alpha \models \beta \equiv M(\alpha) \subseteq M(\beta)$ (can also be understood as $\alpha \to \beta$)



3.4 Inferences

3.4.1 Soundness and Completeness

- KB $\vdash_{\mathcal{A}} \alpha$ - sentence α is derived/inferred from KB by inference algorithm $\mathcal{A}$
- Soundness - $\mathcal{A}$ is sound if KB $\vdash_{\mathcal{A}} \alpha$ implies KB $\models \alpha$, i.e. $\mathcal{A}$ will not infer nonsense
- Completeness - $\mathcal{A}$ is complete if KB $\models \alpha$ implies KB $\vdash_{\mathcal{A}} \alpha$ i.e. $\mathcal{A}$ can infer any sentence that KB entails



3.4.2 Truth Table Enumeration

- Given a bunch of clauses and α_1, KB $\models \alpha_1 \leftrightarrow$ whenever KB is true, α_1 is also true (if there is a case where KB is true and α_1 is false, then KB $\not\models \alpha_1$)
- $O(2^n)$ time complexity, O(n) space complexity (DFS)
- Guaranteed completeness and soundness

3.5 Proof Methods

- Model checking (Special case of CSPs where domains are T/F): Proofed using truth table enumeration or resolution
- Applying inference rules (i.e. theorem proving), generate new sentence from old and proof using sequential application of inference rules - rules help to deduce valid actions which improves efficiency by ignoring irrelevant proposition

3.6 Validity & Satisfiability

- Sentence α is valid if it is true for **ALL** possible truth value assignments (i.e. Tautologies)
- Validity is connected to entailment via deduction theorem - (KB$\models \alpha$)$\Leftrightarrow$((KB$\Rightarrow \alpha$) is valid) (i.e. entailment = implication)
- Sentence is satisfiable if it is true for SOME truth value assignment and unsatisfiable if true for NO truth value assignment (i.e. contradictions)
- Satisfiability shown by showing that (KB$\models \alpha$)$\Leftrightarrow$((KB$\wedge \neg\alpha$) is unsatifiable)

3.7 Inference Algorithm

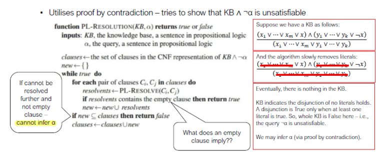
- Using inference to grow the knowledge base is similar to a search problem
  - **States**: Versions of KB
  - **Actions**: application of inference rules
  - **Transition**: update KB with inferred sentence
  - **Goal**: KB contains sentence to prove/disprove
- Some inference rules
  - And-Elimination: $a \wedge b \models a$; $a \wedge b \models b$
  - Modus Ponens: $a \wedge (a \Rightarrow b) \models b$
  - Logical Equivalences: $(a \vee b) \models \neg(\neg a \wedge \neg b)$
- Inference is related to Truth Table Enumeration since it **goes through all cases** including false ones and **all inferences are modeled by knowledge base**

3.8 Conjunctive Normal Form

- CNF = Conjunction of disjunctive sentences e.g. $(x_1 \vee \neg x_2) \wedge (x_2 \vee x_3 \vee \neg x_4)$
- Rules for converting to CNF
  - $\alpha \Leftrightarrow \beta \equiv (\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$
  - $\alpha \Rightarrow \beta \equiv \neg\alpha \vee \beta$
  - $\neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$
  - $\neg(\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta$
  - $\neg(\neg\alpha) \equiv \alpha$
  - $(\alpha \vee (\beta \wedge \gamma)) \equiv (\alpha \vee \beta) \wedge (\alpha \vee \gamma)$
  - $\alpha \oplus \beta = (\neg\alpha \vee \neg\beta) \wedge (\alpha \vee \beta)$

3.9 Resolution Algorithm

- Method of simplifying KB to prove entailment of query α
- Resolution under propositional logic is sound and complete



- Steps involved:
  - Make clause list - copy KB in CNF and add in $\neg\alpha$
  - Repeatedly resolve 2 clause from clause list and add resolvent to clause list
  - Keep doing it till empty clause found or no more resolution - empty clause means that we can infer α and if no more resolution available and still not empty clause then α does not hold
  - Soundness is due to the fact that each resolvent is implied by generating clauses and if $\emptyset$ is found, then (KB$\wedge\neg\alpha$) is unsatisfiable which mean (KB$\wedge\alpha$) must be true

4 Baeysian Network

4.1 Dealing with Uncertainty

- Possible sources of uncertainty:
  - Partial observability
  - Noisy Sensor
  - Uncertainty in action outcomes
  - Complexity in modeling and predicting traffic
- Logical agent either risk falsehood by saying that an uncertain action WILL get me to a goal or reach a weaker conclusion by saying that it will reach goal given certain constraints

4.2 Probability Recap

4.2.1 Joint Probability

- Given two random variables X and Y
  - The joint probability of an atomic event $(x, y) \in D_X \times D_Y$ is $p_{X,Y}(x, y) = \Pr[X = x \wedge Y = y]$
- In particular $p_X(x) = \sum_{y \in D_Y} p_{X,Y}(x, y)$
- Example

Income (in SGD) / AGE	15-24	25-34	35-44	45-54	55-64	65+
< S\$2500	0.062	0.051	0.037	0.019	0.015	0.039
S\$2500 – S\$5000	0.078	0.068	0.061	0.057	0.031	0.053
> S\$5000	0.015	0.051	0.094	0.119	0.111	0.039

$\Pr[Age = (25-34)] = 0.051 + 0.068 + 0.051 = 0.17$

4.2.2 Conditional Probability & Bayes Theorem

- $\Pr[A|B] = \frac{\Pr[A \wedge B]}{\Pr[B]}$
- $\Pr[A \wedge B] = \Pr[B|A] \cdot \Pr[A]$
- Bayes Rule: $\Pr[A|B] = (\Pr[B|A] \cdot \Pr[A]) / \Pr[B]$
- If $\Pr[A \wedge B] = \Pr[A] \cdot \Pr[B]$, then A and B are independent
- $\Pr[A|B] = \Pr[A]$ if A and B are independent
- $\Pr[A|B] = 1 - \Pr[A'|B]$

4.2.3 Chain Rule

- With more than two events, we have

$$\Pr[R_1 \wedge \cdots \wedge R_k] = \Pr[(R_1 \wedge \cdots \wedge R_{k-1}) \wedge R_k]$$
$$= \Pr[R_k \mid R_{k-1} \wedge \cdots \wedge R_1] . \Pr[R_{k-1} \wedge \cdots \wedge R_1]$$

$\boxed{\Pr[A \mid B] = \frac{\Pr[A \wedge B]}{\Pr[B]}}$

- And by induction, we have

$$\Pr[R_1 \wedge R_2 \wedge \cdots \wedge R_k] = \prod_{j=1,\ldots,k} \Pr[R_j \mid R_1 \wedge \cdots R_{j-1}]$$

- Example:

$$\Pr[A \wedge B \wedge C \wedge D] = \Pr[D|C \wedge B \wedge A].\Pr[C \wedge B \wedge A]$$
$$= \Pr[D|C \wedge B \wedge A].\Pr[C|B \wedge A].\Pr[B \wedge A]$$
$$= \Pr[D|C \wedge B \wedge A].\Pr[C|B \wedge A].\Pr[B \wedge A]$$
$$= \Pr[D|C \wedge B \wedge A].\Pr[C|B \wedge A].\Pr[B|A].\Pr[A]$$

4.3 Effect of Independence on Joint Probability Table

- Suppose we have n variable each with domain of size d, if the variables are **not independent**, we will have $d \times d \times \cdots \times d = d^n$ sized table
- If the variables are independent, the joint distribution table will be $d + d + \cdots + d = dn$ instead

4.4 Conditional Independence in Baeysian Network

- Want to have as many independence as possible to determine and store less information and also to decrease the number of enumeration needed
- Relies heavily on **conditional independence**, e.g. a person takes 2 ART test, the 2 tests assuming the person has Covid will now be independent - i.e. A, B are independent given knowledge of underlying cause, $\Pr[A \wedge B|S] = \Pr[A|S] \cdot \Pr[B|S]$
- Full joint distribution table with n boolean variable will have $2^n - 1$ entries
- With conditional independence a full join distribution table using chain rule becomes: $\Pr[T_1 \wedge \cdots \wedge T_{n-1} \wedge S] = \Pr[T_1|S] \cdot \Pr[T_2|S] \cdots \Pr[T_{n-1}|S] \cdot \Pr[S]$, which means we only need to store 2(n-1) + 1 entries
- Pr[Cause|Effect] = Pr[Cause] / Pr[Effect]· Pr[Effect|Cause] = $\alpha \cdot$ Pr[Cause] $\cdot \prod_{i=1,\ldots,k} \Pr[Effect_i \mid \text{Cause}]$

4.5 Normalization

- Example
  - What is the most likely value for X?
  - Need to determine $\Pr[X \mid T_1 = T_2 = 1, T_3 = 0] = \frac{\Pr[X]}{\Pr[T_1 = T_2 = 1, T_3 = 0]} . \Pr[T_1 = T_2 = 1, T_3 = 0 \mid X]$
  - Notice that $\frac{1}{\Pr[T_1 = T_2 = 1, T_3 = 0]}$ is constant over each X in $\Pr[X \mid T_1 = T_2 = 1, T_3 = 0]$
  - So only compute $\Pr[X] . \Pr[T_1 = T_2 = 1, T_3 = 0 \mid X]$ for all X
  - As defined earlier, we let $\alpha = \frac{1}{\Pr[T_1 = T_2 = 1, T_3 = 0]}$

$\Pr[Test(X) = positive \mid X = healthy] = 0.1$
$\Pr[Test(X) = positive \mid X = carrier] = 0.7$
$\Pr[Test(X) = positive \mid X = sick] = 0.9$

$\Pr[X = healthy] = 0.7$
$\Pr[X = carrier] = 0.2$
$\Pr[X = sick] = 0.1$

$\Pr[X]$ $\Pr[T_1 = 1 \mid X]$ $\Pr[T_2 = 1 \mid X]$ $\Pr[T_3 = 0 \mid X]$

$$\Pr[X = healthy \mid TestResults] = \alpha \times 0.7 \times 0.1 \times 0.1 \times 0.9 = 0.0063\alpha$$
$$\Pr[X = carrier \mid TestResults] = \alpha \times 0.2 \times 0.7 \times 0.7 \times 0.3 = \boxed{0.0294\alpha}$$
$$\Pr[X = sick \mid TestResults] = \alpha \times 0.1 \times 0.9 \times 0.9 \times 0.1 = 0.0081\alpha$$

4.6 Conditional Probability Tables

Conditional Probability Tables

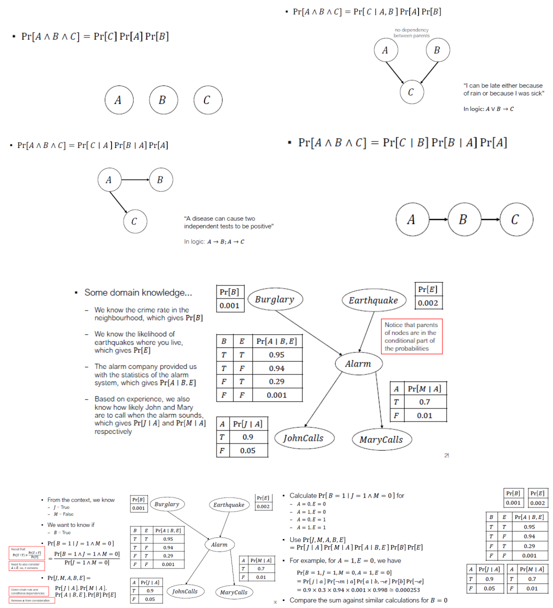
- Given the chain rule and conditional independence assumption

$$\Pr[Cause \mid Effect] = \frac{\Pr[Cause]}{\Pr[Effect]} . \Pr[Effect \mid Cause]$$
$$= \alpha . \Pr[Cause] . \prod_{i=1,\ldots,k} \Pr[Effect_i \mid Cause]$$

- We only need the Conditional Probability Table (CPT) with
  - Each $\Pr[Effect_i \mid Cause]$
  - $\Pr[Cause]$
  - i.e., $k + 1$ entries (assuming k effects)

4.7 Baeysian Network

- Represents joint distributions via a graph
  - Vertices are variables and edges from X to Y → X directly influences Y (i.e. X causes Y)
  - Each node in a bayesian network has a conditional distribution for the node, given its parents.
  - Max number of edges in Baeysian network = $n(n-1)/2$, complete graph



4.8 Decision-Theoretic Agents

- Rationality in the face of uncertainty
  - Probability Theory – accounting for uncertainty
  - Utility Theory – accounting for value (dependent on agent)
  - Decision Theory = Probability Theory + Utility Theory
- Agent is rational iff it chooses actions that maximise utility
- Maximum Expected Utility (MEU) Principle - Pick action with highest utility weighted over probable outcomes

function DT-AGENT(*percept*) **returns** an *action*
  persistent: *belief_state*, probabilistic beliefs about the current state of the world
    action, the agent's action
  update *belief_state* based on *action* and *percept*
  calculate outcome probabilities for actions,
    given action descriptions and current *belief_state*
  select *action* with highest expected utility
    given probabilities of outcomes and utility information
  return *action*

Agent determines an action with each percept (not planning)

Maintains BELIEF STATE – i.e., what it currently believes about the problem environment

Accounting for the uncertainty – e.g., via a Bayesian Network – to determine the likelihood of intermediate states given actions

For each action, a_i, that can be taken at the current state s', determine $\sum_{s \in T} utility(s) \times probability(s)$ where T denotes the set of states that one may transition to from state s' when action a_i is taken

Choose the action that yields the highest value

5 Goal Search

- Path to goal is irrelevant, the goal state itself is the solution.
- Advantages: (1) use very little $O(b)$/constant memory, (2) can find reasonable solns in large/infinite continuous state spaces
- Useful for **pure optimization problems**: objective is to find the best state according to an **objective function**. e.g. Vertex cover, TSP, Boolean Satisfiability Problem (SAT), Timetabling

5.1 Hill-climbing Algorithms

5.1.1 Problem Formulation

- Start with **complete** state i.e. no partial state (removes build up stage and start checking on 1st iteration)
- Each state is a possible solution

5.1.2 Steepest Ascent - Greedy

- Start with random initial state, in each iteration find successor that improves on current state
- Requires **actions** and **transition** to determine successors
- Requires some heuristic to give value to each state e.g. $f(n) = -h(n)$ and find maxima
- **Can be stuck at local maxima** and return non-goal state
- Problems arises with **shoulders/plateau, local maxima or ridge**

```
current = initial_state
while true:
    neighbour = highest_valued_successor(current)
    if value(neighbour) ≤ value(current): return current
    current = neighbour
```

5.1.3 Stochastic Hill Climbing

- Instead of choosing highest-valued-successor in each step, **choose randomly among states with better values** instead
- Idea is to make the choosing of next state less deterministic to give to algo more chance of finding global maxima
- Takes longer but could lead to better solutions

5.1.4 First-choice Hill Climbing

- Handles high branching factor by randomly generating successors until one with better value is found
- Possible to achieve $O(1)$ space with this

5.1.5 Sideways Move

- Replace the ≤ sign in steepest ascent with <
- Allows algo to traverse shoulders/plateaus

5.1.6 Random-restart

- Adds outer loop that randomly pick new starting state and keep attempting restarts until solution is found (up to a threshold)
- Also allows for sideways move

```
current = NULL
while current is NULL or not isGoal(current):
    current = random_initial_state()
    while true:
        neighbour = highest_valued_successor(current)
        if value(neighbour) < value(current):
            break
        current = neighbour
return current
```

5.2 Analysis of Hill Climb

- Hill climbing (via steepest-ascent) with random restarts
  - Solution: p_1 = 14% (expected solution in 4 steps; expected failure in 3 steps)
  - Expected computation = 1×(steps for success) + ((1 - p_1) / p_1)×(steps for failure)
    = 1×(4) + (0.86/0.14) ×(3)
    = 22.428571428571427 steps

(1 - p_1) / p_1) determines the expected number of failed attempts

- Adding sideways moves
  - Solution: p_2 = 94% (expected solution in 21 steps; expected failure in 64 steps)
  - Expected computation = 1×(steps for success) + ((1 - p_2) / p_2)×(steps for failure)
    = 1×(21) + (0.06/0.94) ×(64)
    = 25.085106382978722 steps
- 8-Queens possible states = 8^8 = 16777216

Extremely efficient for such a large space

5.3 Local Beam Search

- Stores k states instead of 1
- Algo begins with k random restarts which generates successors for all k states
- Next iteration will repeat the above step with best k among ALL generated successors found (unless goal is found)
- Better than k parallel random restarts, Since best k among ALL successors taken (not best from each set of successors, k times)
- Also has a stochastic variant to increase probability of escaping from local maxima

6 Uninformed Search Summary

Uninformed Search Strategies

Property	BFS	UCS	DFS	DLS	IDS
Complete	Yes*	Yes**	No***	No	Yes*
Optimal	No*	Yes	No	No	No*
Time	$O(b^d)$	$O(b^{1+\lfloor \frac{C^*}{\epsilon} \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^d)$	$O(b^{1+\lfloor \frac{C^*}{\epsilon} \rfloor})$	$O(bm)$	$O(bl)$	$O(bd)$

*: BFS, IDS – complete if b/state space is finite or if there is a solution, optimal if step costs are identical

**: UCS is complete if b is finite and action cost $> \epsilon > 0$

***: DFS is complete only on finite depth & branching factor graphs

C^* is the optimal cost

7 Graph Search Versions

```
frontier = {Node(initial_state, NULL)}
reached = {initial_state: Node(initial_state)}
while frontier not empty:
    current = frontier.pop()
    if isGoal(current.state): return current.getPath()
    for a in actions(current.state):
        successor = Node(T(current.state, a), current)
        if successor.state not in reached:
            frontier.push(successor)
            reached.insert(successor.state: successor)
return failure
```

```
frontier = {Node(initial_state, NULL)}
reached = {initial_state: Node(initial_state)}
while frontier not empty:
    current = frontier.pop()
    if isGoal(current.state): return current.getPath()
    for a in actions(current.state):
        successor = Node(T(current.state, a), current)
        if successor.state not in reached or
          successor.getCost() < reached[successor.state].getCost():
            frontier.push(successor)
            reached.insert(successor.state: successor)
return failure
```

Graph search V1 ensures nodes are not revisited which could omit optimal paths

Graph search V2 solves that by allowing revisits provided cost is lower

Graph search V3 inserts into visited when popped

8 Heuristics

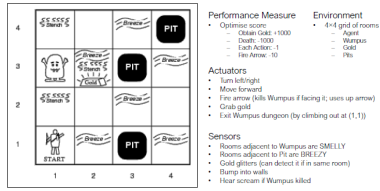
- **Admissible Heuristic** never overestimates the cost to reach the goal: $\forall n, h(n) \le h^*(n)$ where $h^*(n)$ = true cost
  - Consequence: all paths with actual costs less than P must be searched
- **Consistent Heuristic**: $h(n) \le c(n, n') + h(n')$, will make $f(n)$ monotonically increasing along a path
- Consistency $\implies$ Admissibility
- $h(n)$ is **admissible**:
  - Optimal under **Tree Search** and **Graph Search V2**
  - Non-optimal under **Graph Search V1**
- $h(n)$ is **consistent**:
  - Optimal under **Tree Search, Graph Search V2 and V3** (insert into visited when popped)
  - Non-optimal under **Graph Search V1**
- **Complete** if b & m finite OR has a solution and all action cost $> \epsilon > 0$, **Optimal**, **Time** $O^{h^*(s_0) - h(s_0)}$ where $h^*(s_0)$ is the actual cost of getting from root to goal, **Space** $O(b^m)$
- **Dominant heuristic**: if $\forall n, h_2(n) \ge h_1(n)$ then h_2 **dominates** h_1
- More dominant heuristics incur lower search cost

9 Environment Properties

- **Fully observable** vs **Partially observable**: Partially observable agent does not have access to all information (e.g. fully observable maze VS slowly expanding maze based on actions taken). Requires dealing with **uncertainty** i.e. backtracking algos
- **Deterministic** vs **Stochastic**: if the next state of the env is completely determined by the current state and the action executed **VS** otherwise. (A fully observable environment that has randomness with action is stochastic) (e.g. Sudoku VS Poker)
- **Episodic** vs **Sequential**: actions only impact current state **VS** action impact future decisions
- **Discrete** vs **Continuous**: in terms of state of env, time, percepts and actions (and to discretize continuous environments)
- **Single agent** vs **Multi-agent**: whether there are any other agent in the environment whose actions directly influence the performance of this agent, multi-agent further divided into **competitive** and **cooperative**
- **Static** vs **Dynamic**: if the environment is unchanged while an agent is deliberating **VS** otherwise

10 Miscellaneous

10.1 Wumpus World Example



10.2 Logical Equivalences

De Morgan's Laws	$\neg(p \vee q) \equiv \neg p \wedge \neg q$	$\neg(p \wedge q) \equiv \neg p \vee \neg q$
Idempotent laws	$p \vee p \equiv p$	$p \wedge p \equiv p$
Associative laws	$(p \vee q) \vee r \equiv p \vee (q \vee r)$	$(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$
Commutative laws	$p \vee q \equiv q \vee p$	$p \wedge q \equiv q \wedge p$
Distributive laws	$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$	$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$
Identity laws	$p \vee False \equiv p$	$p \wedge True \equiv p$
Domination laws	$p \wedge False \equiv False$	$p \vee True \equiv True$
Double negation law	$\neg\neg p \equiv p$	
Complement laws	$p \wedge \neg p \equiv False \wedge \neg True \equiv False$	$p \vee \neg p \equiv True \vee \neg False \equiv True$
Absorption laws	$p \vee (p \wedge q) \equiv p$	$p \wedge (p \vee q) \equiv p$
Conditional identities	$p \Rightarrow q \equiv \neg p \vee q$	$p \Leftrightarrow q \equiv (p \Rightarrow q) \wedge (q \Rightarrow p)$