# Search

### Components of Search Problems

- State space: A state is a representation of a configuration of the problem domain.

  The state space is the sets of all states included in our model of the problem
- Initial State: The starting configuration
- Action: Allowed change from one state to another
- Goal State: The configuration that want to achieve
- Cost(Optional): A cost associated with each action
- Heuristic Function(Optional): A heuristic function to guide the search.

Complex situations like flipping coins may require techniques for reasoning under uncertainty. We could assign a probability to give outcomes

### Node structure

A node is a data structure that includes:

- A state
- A path from the initial state to the node's state (including actions)
- Cost of the path from the initial node to the current node

### Tree Search

**Frontier[List]:** nodes we haven't explored but wish to consider

Initially, the frontier contains the initial state. At each search iteration, we pop a node from the frontier, apply the successor, and insert the children into the frontier.

General algorithm for searching:

**Search(Initial node, Successor Func, Goal Test)**

Frontier = {Initial node}
**while** Frontier is not empty: **do**
  **Select and Remove** Node $curr$ from Frontier
  **if** Goal Test($curr$) **then**
    return
  **end if**
  Add Curr's successors to Frontier.
**end while**
return no solution

### Search Kinds and Key Properties

Search Kinds:

- DFS: remove the newest node added (frontier = stack)
- BFS: remove the oldest node added (frontier = queue)
- IDS: Set Depth limit $l$ and perform DFS until limit $l$, if reached, increase limit by 1 and do DFS again.
- UCS: remove node with the smallest total path cost (frontier = priority queue)
- GBFS: Remove node with smallest heuristic value
- A*: Remove node with smallest path cost + heuristic value

Key properties:

- Completeness: Will the search always find a solution (if exists)
- Optimality: Will the search always find the least cost solution?
- Time Complexity: How many nodes need to expand?
- Space Complexity: How many nodes have to be stored in memory?

### BFS Property

- $b$ as the maximum number of successors of any state
- $d$ as the number of actions in the shortest solution
- $m$ as max depth of the search tree (length of the longest path)
- **Worst Case:** Solution at the least node in depth $d$
- **Time and Space Complexity:** $1 + b + b^2 + \cdots + (b^d - 1) \in O(b^{d+1})$
- **Completeness:** Yes, BFS is complete as long as the state space has finite $b$
- **Optimality:** Not cost-optimal, but its length-optimal

### DFS Property

- **Worst Case:** Could process the whole tree
- **Time Complexity:** If $m$ is finite, $O(b^m)$
- **Space Complexity:** $O(bm)$ since we explore a single path at a time and at most $m$ nodes on the current path and at most $b$ siblings for each node.
- **Completeness:**NO, $m$ could be infinite
- **Optimality:**NO, it finds the "left-most"(or rightmost) solution, regardless of depth or cost.

### IDS Property

- **Completeness:** Yes, IDS is guaranteed to terminate at level $d$. Explores the tree level by level $\approx BFS$
- **Optimality:** NO, not optimizing costs but finds shortest-length solution $\approx BFS$
- **Space complexity:** $O(bd)$: Explores one path of length at most $d$, remembers at most $b$ siblings for each state
- Time Complexity: $(d+1)b^0 + db + (d-1)b^2 + \cdots + b^d = O(b^d)$

### UCS Property

- $C^*$: Cost of Optimal Solution
- $\epsilon$ : Minimum Cost of each action
- Effective Depth $\frac{C^*}{\epsilon}$
- **Time / Space Complexity:** $O(b^{\frac{C^*}{\epsilon}+1})$
- **Completeness:** YES by assuming $b$ is finite, the best solution has a finite cost and each action has cost $\geq \epsilon > 0$
- **Optimality:** Cost Optimal, assuming each action has cost $\geq \epsilon > 0$

### Node Pruning

**Problem:** There can be multiple tree nodes with the same state; we only need to find one (or the best) path to a state.

**Solution:** By keeping all/some of the *visited states* in a set. Expanding a node only if its state is **not in the set**.

### Path Checking

Remember the visited states on **a single branch**

- Advantage: Doesn't increase time and space complexity
- Cons: Doesn't prune all the redundant states

### Cycle Checking

Remember **all** visited states. using a list called the closed list.

- Advantage: Very effective in pruning redundant states.
- Limitations: Expensive in terms of space. Space complexity with BFS is $O(b^d)$, with DFS, it could be worse.

# Heuristic

A good heuristic function $h(n)$ has the properties below:

- Non-negative
- $h(n) = 0$ if $n$ is a goal node
- Can compute $h(n)$ efficiently and without search

### GBFS Property

- **Worst Case:** Like a badly-guided DFS.
- **Completeness:** Not complete
- **Optimality:** Not optimal
- **Time and Space Complexity:** Exponential

### A* Search Property

A* Search uses $f-$value, $f(n) = g(n) + h(n)$ where $g$ is the path cost, $h$ is the heuristic cost.

- **Optimality and Completeness for non-cycle checking**: A* with a **non-negative admissible heuristic** always finds an optimal cost solution, if a solution exists as long as:
  - The branching factor $b$ is finite
  - every action has finite cost $\geq \epsilon > 0$
- With Cycle checking, it requires **Consistent** heuristic function instead of admissible.

**Admissibility**
Let $h^*(n)$ be the cost of an optimal path from $n$ to a goal node ($\infty$ if there is no path). An **admissible** heuristic that satisfies the following condition for all nodes $n$ in the search space:

$$h(n) \leq h^*(n)$$

**Consistency**
A consistent (aka Monotone) heuristic $h$ is a heuristic that satisfies the triangle inequality: for all nodes $n_1, n_2$ and for all action $a$

$$h(n_1) \leq C(n_1, a, n_2) + h(n_2)$$

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 $a$
**Note:** Consistency **implies** Admissibility

**Iterative Deepening $A^*$**
**Goal:** reducing memory requirements for $A^*$
Like IDS, but now the cutoff value is the f-value.
At each iteration, the cutoff value is the smallest f-value of any node that exceeded the cutoff on the previous iteration.
For example, at $k$ iteration, do not expand on with $f(n) > k|1$.
This makes it, under certain conditions, complete, guaranteed to find optimal and more costly than $A^*$ in general.

**UCS vs $A^*$ Contours**
Uniform-cost expands equally in **all directions**
$A^*$ expands mainly toward the goal but hedges its bets to ensure optimality.

## Past notes for True False

- **IDS cycle checking** will make space complexity as bad as BFS
- DFS with path checking, DFS no longer gets stuck in a cycle, but still may **not complete**
- UCS with Cycle checking will still be optimal

## Constraint Satisfiction Problems

A CSP consists of:

- A set of variables $V_1, \cdots V_n$
- A domain of possible values $Dom[V_i]$ for each variable $V_i$
- A set of constraints $C_1, \cdots, C_m$

**BT: Backtracking Search**

1. Start with Empty Assignment.
2. Children of a node: All possible value assignments for a particular unassigned variable.
3. The tree stops descending if an assignment violates a constraint.

Goal Node: 1. Assignment complete, and no constraints violated.
**Worst runtime:** $O(d^N)$ where $d$ is the max size of a variable domain and $N$ is the number of variables.

**FC: Forward Checking**
Intuition: At every step of the backtracking search, when instantiating a variable $V$, do the following for all constraints $C$ that have **only one unassigned** variable $X$ remaining:

1. Check all the values of $X$
2. Prune those values that violate $C$

The purpose of developing constraint propagation techniques such as FC is to solve those simpler sub-classes faster.
**Run time for FC:** FC is often 100 times faster than BT, but it can also do **worse**.

**Value Ordering Heuristics**
***Degree Heuristic:***

- Select the variable that is involved in the **largest number** of constraints on other unassigned variables

***MRV*: Minimum Remaining Values Heuristics:**

- Always branch on a **variable** with the **smallest remaining values**
- FC with MRV often 10000 times faster than BT

***Least Constraining Value Heuristic:***

- Always pick a **value** in CurrDom that rules out the **least domain values** of other neighboring variables in the constraint.
- This allows for the maximum flexibility for subsequent variable assignments.

**GAC: Generalized Arc Consistency**
Value $d$ of variable $V_i$ **is not consistent** wrt a constraint: that is, there is no assignment to other variables that satisfy the constraint when $V_i = d$

- $d$ is said to be **arc inconsistent**
- We can **remove** $d$ from the domain of $V_i$ as this value cannot lead to a solution

A **Support** for a value assignment $V = d$ in a constraint $C$ is an assignment $A$ to all of the other variables in $scope(C)$ s.t. $A \cup \{V = d\}$ satisfies $C$
A constraint $C$ is **GAC** iff for **every** variable $V_i$ in its scope, **every** value $d_i \in CurrDom(V_i)$ has a support in $C$
A CSP is GAC iff all of its constraints are GAC, accomplished by removing from the domains of the variable's all arc-inconsistent values.
Notices that removing a value from a variable domain may trigger further inconsistency, we have to repeat the procedure until everything is consistent.
**Note:** GAC enforce **does not** find a solution! To find a solution, we must use do search while enforcing GAC.
For example: $X, Y \in \{1, 2\}$ with constraints $X = Y, X \neq Y$. This is GAC but there is no solution.

**GAC Complexity**
Worst-case complexity of arc consistency procedure on a problem with $N$ variables, $c$ binary constraints, and $d$ be the max size of a variable domain:

- Maximum number of times we prune the domain of variable $V$: $O(d)$
- How many constraints will be put on the queue when pruning domain of a variable $V$: $O(degree(V))$

  Note: degree of a variable is the number of constraints defined over the variable
- sum of degrees of all variables: $2 \times C$
- Overall, $O(cd)$ constraints will be put on the queue
- Checking consistency of each constraint $O(d^2)$
- **Overall Complexity** $O(cd^3)$

## MinMax Search

Assumption: The other player always plays its best move
Decision: Play a move that minimizes the payoff that the other player could gain.
The minmax value assigned to each node represents MAX's payoff for each game state, assuming both players always play optimally.
**Compute minmax value**

- Starting from the terminal node's parents, compute minmax values for non-terminal states
  - MAX always picks the highest valued child.
  - MIN always picks the lowest valued child.

**DFMiniMax(s, Player)**

```
// Return Utility of state s given that Player is MIN or MAX
if s is TERMINAL then
    return U(s)   // Return terminal state's utility
end if
ChildList ← s.Successors(Player)
if Player == MIN then
    return min{DFMiniMax(c, MAX) | c ∈ ChildList}
else
    return max{DFMiniMax(c, MIN) | c ∈ ChildList}
end if
```

Let $b$ represent the number of legal moves at each state, $d$ is the total number of turns for both players **Time Complexity:** $O(b^d)$

**Alpha-Beta Pruning**
Alpha-Beta Pruning is an algorithm for identifying and pruning branches that do not affect the calculation of correct minmax decisions.
At a **Max** node $S$

- $\alpha$**(changes)** : The **highest** value of $S$'s branches examined so far. as branches of $S$ are examined
- $\beta$**(fixed)** The lowest value found so far by $S$'s parent $P$, from previously-explored siblings of $S$.

At a **Min** node $S$

- $\alpha$**(fixed)** : The **highest** value found so far by $S$'s parent from previously-explored siblings of $S$
- $\beta$**(Change)** : The lowest value of $S$'s branches examined so far

**def max-value(state,** $\alpha, \beta$**)**

initialize $v = -\infty$
**for** each successor of state **do**
  $v = \max(v, value(successor, \alpha, \beta))$
  $\alpha = max(\alpha, v)$
  **if** $\alpha \geq \beta$ **then**
    return $v$
  **end if**
**end for**
return $v$

**def min-value(state,** $\alpha, \beta$**)**

initialize $v = +\infty$
**for** each successor of state **do**
  $v = \max(v, value(successor, \alpha, \beta))$
  $\beta = min(\beta, v)$
  **if** $\beta \leq \alpha$ **then**
    return $v$
  **end if**
**end for**
return $v$

**Move ordering**
Ideally, we want to visit the best branch first. For MAX/MIN nodes, the best pruning occurs if the optimal move for MAX/MIN is explored first.

**Effectiveness of Alpha-beta search**
With perfect ordering: $O(b^{d/2})$

# Uncertainty

**VE Example Breakdown**
- Express numerator and denominator of Bayes' Rule
- Expand the joint distribution using the network's structure
- Sum out all irrelevant variables
- At each step, define intermediate functions ($f1, f2, \cdots$) over smaller variable sets
- Result: two values (for $D = true, D = false$) that are then normalized

**Factor Operations**
- Product (Multiply Factors)

  Combine two factors over shared variables.

  If $f(X, Y)$ and $g(Y, Z)$, then $h(X, Y, Z) = f(X, Y) \times g(Y, Z)$

  Like the outer join, but the newly added row is the product of the original two rows
- Sum Out (Marginalize)

  Eliminate a variable by summing over its values.

  If $f(X, Y)$, then : $h(Y) = \sum_{x \in dom[X]} f(x, Y)$
- Restrict (Condition on Evidence)

  Fix a variable to a known value and drop others.

  If $f(X, Y)$ and $X = x$, then: $h(Y) = f(X = x, Y)$

**Variable Elimination**
Given query var $Q$, evidence vars $E$(set of variables observed to have values $e$), remaining vars $Z = V \setminus \{Q, E\}$. Let $F$ be the set of original CPTs

1. Replace each factor $f \in F$ that mentions a variable in $E$ with its restriction $f_{E=e}$ (this might yield a factor over no variables, a constant)
2. For each $Z_j$ – in the order given – Eliminate $Z_j \in Z$ as follows
   (a) Compute new factor $g_j = \sum_{Z_j} f_1 * f_2 * \cdots * f_k$, where the $f_i$ are t Ohe factors in $F$ that include $Z_j$
   (b) Remove the factors $f_i$ that mention $Z_j$ from $F$ and add new factor $g_j$ to $F$
3. The remaining factors refer only to the query variable $Q$. Take their product and normalize to produce $Pr(Q|e)$

**Complexity of Variable Elimination**
- Complexity of VE is exponential in the size of the largest factor generated during the VE, including the input CPTs
- Different elimination orderings can lead to different factor sizes
- Heuristics can be used for picking more efficient orderings

**Min Fill Heuristic**
- A fairly effective heuristic is always eliminate next the variable that creates the smallest size factor
- Size refers to the number of remaining variables in the scope of the factor

**Blocking in D-Separation**

Two variables $X, Y$ are conditionally independent given evidence $E$ if $E$ d-separates $X$ and $Y$ by blocks every undirected path in the BN between $X$ and $Y$ Let $P$ be an undirected path from $X$ to $Y$ in a BN.
Let $E$(evidence) be a set of variables.
We asy $E$ blocks path $P$ iff **there is some** node $Z$ on the path $P$ such that
- $Z \in E$ and $X \rightarrow Z \rightarrow Y$
- $Z \in E$ and $X \leftarrow Z \rightarrow Y$
- $Z \in E$ and $X \rightarrow Z \leftarrow Y$ and neither $Z$ nor any of its descendants($Z$'s Child and grandchild) are in $E$

**Elimination Width**
Elimination width of $\pi$ is the maximum size (number of variables) of any hyper-edge in any of the hypergraph
In other words, the width is the number of maximum parameter defined for those factor functions.

**Polytree**
A polytrees is a singly connected Bayes Net: in particular, there is only one path between any two nodes.

# Knowledge Representation

**Fundamentals of Knowledge Representation & Reasoning**
- **KR** = representing facts and rules explicitly (symbolically)
- **Reasoning** = deriving new conclusions from known facts
- **Knowledge Base(KB)** = a set of known facts (sentences).
  - Explicit knowledge: directed stored in the KB.
  - Implicit knowledge: derived from reasoning
- **Inference Engine:** uses logic to compute logical consequences of the KB

**Logical Representations for KR&R**
- **Logical Entailment:** Sentences $P_1, P_2, \cdots, P_n$ entail sentence $P$ iff the truth of $P$ is implicit in the truth of $P_1, P_2, \cdots, P_n$.

  That is, in any world that $P_1, P_2, \cdots, P_n$ are true, $P$ is also true
- **Logical Inference:** the process of calculating entailments
- **Logic:** study of entailment relations, truth conditions, and rules of inference.

**Propositional Logic**
- **Propositional Variable:** A variable which takes only True or False as values

  Example: $V = \{p, r\}$
- **Truth Assignment:** A function $\tau$ from the propositional variables into the set of truth values $\{T, F\}$

  $\tau_1 : V \rightarrow \{T, F\}, \tau_1(p) = T, \tau_1(q) = F$
- **Satisfy:** $\tau$ satisfies a set $\Phi$ of formulas iff $\tau$ satisfies all formula in $\Phi$
- **Satisfiable:** A set $\Phi$ of formulas is satisfiable iff some truth assignment $\tau$ satisfy $\Phi$.
- **Logical Consequence:** $\Phi \models A$: A formula $A$ is a logical consequence of $\Phi$ iff for every truth assignment $\tau$, if $\tau$ satisfies $\Phi$, then $\tau$ satisfies $A$

**First Order Logic**
Core components

- Variables (V): stand for objects in the domain (e.g. $x, y, z$)
- Function Symbols (F): map tuples of objects to objects (e.g. $mother\_of(x)$)
- Predicate Symbols (P): describe properties or relations (e.g. $Loves(x, y)$)
- **First-order vocabulary:** A set of $\mathcal{L}$ of function and predicate symbols.
- **constant symbols:** 0-ary functions symbols.

Term:

- Every variable is a term.
- If $f$ is an $n-$ary function symbol in $\mathcal{L}$ and $t_1, \cdots, t_n$ are $\mathcal{L}-$terms, then $f(t_1, \cdots, t_n)$ is a $\mathcal{L}-$term

**Structure**
Let $\mathcal{L}$ be a first-order vocabulary. An $\mathcal{L}-$structure $\mathcal{M}$ consist of the following:

1. A nonempty set $\mathcal{M}$ called the universe(domain) of discourse.
2. For each $n-$ary function symbol $f \in \mathcal{L}$, an associated function $f^M : M^n \to M$

   Note: If $n = 0$, then $f$ is a constant symbol and $f^M$ is simply an element of $M$. $f^M$ is called the extension of the function symbol $f$ in $M$
3. For each $n - ary$ predicate symbol $P \in L$, an associated relation $P^M \subseteq M^n$. $P^M$ is called the extension of the predicate symbol $P$ in $M$

**Object assignment**

- Let $M$ be a structure and $X$ be a set of variables. An object assignment $\sigma$ for $M$ is a mapping from variables in $X$ to the universe of $M$
- For an $\mathcal{L}-$Formula $C$, $M \models C[\sigma]$($M$ satisfies $C$ under $\sigma$, or $M$ is a model of $C$ under $\sigma$)
- Note $\sigma(m/x)$ is an object assignment function exactly like $\sigma$, but maps the variable $x$ to the individual $m \in M$. That is:
  - For $y \neq x : \sigma(m/x)(y) = \sigma(y)$
  - For $x : \sigma(m/x)(x) = m$

**Models**

- An occurrence of $x$ in $A$ is **bounded** iff it is in a sub-formula of $A$ of the form $\forall xB$ or $\exists xB$. Otherwise the occurrence is **free**

  In a structure $M$, formulas with free variables might be true for some object assignments to the free variables and false for others.

  For example, $P(x, y) \wedge P(y, x), M = \{a, b\} P^M = \{(a, a)\}$ where it is a logical formula with two free variables $x, y$. We know that the formulas are true for object assignment that $x = a, y = a$.
- A formula is **closed** if it contains no free occurrence of a variable.

  A closed formula is called a sentence
- If $\sigma$ and $\sigma'$ agree on the free variables of $A$, then $M \models A[\sigma]$ iff $M \models A[\sigma']$
- Corollary: If $A$ is a sentence, then for any object assignments $\sigma$ and $\sigma'$, $M \models A[\sigma]$ iff $M \models A[\sigma']$

  This means if $A$ is a sentence, $\sigma$ is irrelevant and we omit mention of $\sigma$ and simply write $M \models A$

**Proof Procedures**

- A proof procedure is **sound** if whenever it produces a sentence $A$ by manipulating sentences in a KB, then A is a logical consequence of KB (i.e. $KB \models A$).

  That is, all conclusions arrived via the proof procedure are correct; They are logical consequences.
- A proof procedure is **complete** if it can produce all logical consequences of KB. That is, if $KB \models A$, then the procedure can produce $A$

**Resolution by Refudation**

- A **literal** is an atomic formula or the negation of an atomic formula
- A **clause** is a disjunction of literals;
- A **clause theory** is a set of clauses. It can also be considered as conjunction of clauses.

We denote a **contradiction** by an **empty clause: ()**

**Conversion to Clausal Form**
A Clause is a disjunction(OR) of literals, where a literal is either a positive atomic form or its negation.

- Eliminate Implications: $A \to B \iff \neg A \vee B$
- Move Negations Inwards (and simplify $\neg\neg$)
- Standardize Variables: Rename variables so that each quantified variable is unique
- Skolemization: Remove existential quantifiers by introducing new function symbols.

  For example, $\exists y(elephant(y) \wedge friendly(y))$, to remove the existential, we invent a "name" for this individual $a$. Note that this name must be a **new** constant symbol.

  Use a new function symbol that mentions every universally quantified variable that scopes the existential $\forall x(loves(x, g(x)))$ where $g$ is the new function symbol
- Convert to Prenex Form: Bring all quantifiers to the front. We use the following equivalences, where $x$ does not occur free in Q

  $\forall xP \wedge Q \leftrightarrow Q \wedge \forall xP \leftrightarrow \forall x(P \wedge Q)$

  $\forall xP \vee Q \leftrightarrow Q \vee \forall xP \leftrightarrow \forall x(P \vee Q)$
- Distribute Conjunctions over Disjunctions: $A \vee (B \wedge C) \leftrightarrow (A \vee B) \wedge (A \vee C)$
- Flatten nested $\wedge$ and $\vee$: $A \vee (B \vee C) \to (A \vee B \vee C)$
- Convert to Clauses: **Remove universal quantifiers** and break apart conjunctions

Once reduced to clausal form, all remaining variables are universally quantified.

**Decidability of FOL**
If the clauses are unsatisfiable, then return YES, otherwise NO.

- **Decidable:** An algorithm always gives a YES/NO answer in finite time for every possible input.
- **Semi-decidable:** An algorithm gives a YES answer if unsatisfiable, but may loop forever if satisfiable.

Core Results in FOL: FOL Unsatisfiability = semi-decidable
Problem: Want to produce entailments of KB as needed for immediate action. But full theorem proving may be too difficult for KR&R
Solution:

- Satisfiability: Some first-order cases can be handled by converting them to a propositional form
- Calculating Entailment (Unsatisfiability):
  - Giving control to the user
  - Using decidable fragments of FOL (which are less expressive)