CS 161: Fundamentals of Artificial Intelligence

Stanley Wei

Prof. Van den Broeck | Winter 2024

Contents

1 *Was Ist A.I.?*

What is considered "***intelligent***"/"***A.I.***" is context-dependent:

- *Ex.* Chess bots, GPS routing were considered A.I. 20 years ago, but are rarely described as being "A.I." today.

	Acting	**Thinking**
Humanly	*Chatbots*	*Computational Neuroscience*
Rationally	*Lots of modern A.I.*	

Table 1: 4 A.I. archetypes

$\exists$ many possible & historical definitions for "A.I.": Turing test, Winograd schemas, etc., but still no hard line/boundary; A.I. is, in some sense, a moving target.

Definition (***A.I. (Historical)***)**.** ***A.I.*** is the study of **intelligent, rational agents**.

Definition. A ***rational agent*** is something that acts to achieve the best [expected] outcome based on some some objective.

- More formally: A ***rational agent*** is something that, for every possible ***percept*** sequence (perception of ***the world***), should select an ***action*** that is expected to maximize its ***performance measure*** given the percept sequence and the agent's own internal ***knowledge***.

Issue: Defining a performance measure, is not always clear/easy to conceive; ex:

1. *Is there a clear objective?*: What is the performance measure of Facebook?
2. *How do we define & quantify success?*: ChatGPT was built on human grading as a performance measure; in a sense, it was trained to fool humans/lie satisfactorily?
3. *Could there be unforeseen effects?* See: paperclip maximizer

2 Search

Search: The problem of finding a sequence of actions to a goal state.

Search can be viewed as a function (called a ***search engine***) that:

1. Takes two inputs: a ***search problem formulation*** and a ***search strategy***
    - The behavior of a search is strongly sensitive to both inputs.
2. Gives one output: a ***solution***

Applications of Search:

- Game solving
- Mathematical problems & proofs
- Probabilistic reasoning
- Satisfiability Problem
- Route-finding & Traveling Salesman

2.1 Problem Formulation

Problems are usually described in the context of two types of objects:

1. ***Problem states***, indicating potential configurations of the *world*. The ***state space*** is the set of all possible states.
2. ***Actions***, transforming one problem state into another. The ***action space*** is the set of all potential actions.

Definition. A ***search problem formulation*** or ***specification*** consists of:

1. An ***initial state***
2. A ***goal***: A goal may be defined either as a specific state to achieve (a ***goal state***), or as a test that determines whether a given state meets goal criteria (a ***goal test/predicate***).
3. ***Actions***: the set of actions that may be performed
4. A ***transition model/successor***: a set of rules governing how actions affect states.
5. (*Optional*) A ***cost***: a function used for determining the "optimality" of a given solution.

Definition. A ***problem solution*** is a sequence of actions to get from the initial state to a state that fulfills the goal. An [the] ***optimal solution*** is a solution with lowest cost.

(∗) *Ex*: A cost may be defined in terms of the number of actions taken; then an optimal solution is a solution requiring the least number of actions to achieve the goal state.

Given a problem, there may be more than one way to describe/interpret its actions.

(∗) *Ex (Sliding Tile):*

1. "An action is moving a number either up, left, right, or down"
   $\implies$ Actions as number-direction pairs [e.g. $(4, \rightarrow)$] [# actions: $8 \cdot 4 = \mathbf{32}$]
2. "An action is moving a number into the direction of the blank space"
   $\implies$ Actions as numbers [e.g. (4)] [# actions: **8**]
3. "An action is moving the blank space either up, left, right, or down"
   $\implies$ Actions as directions [e.g. $(\rightarrow)$] [# actions: **4**]

2.2 Search Trees

Definition. A ***search tree*** is a tree, rooted at the initial state, where each node represents a problem state and has as children all possible states resulting from performing an action on the parent state.

More specifically: a node in the tree has one child for every possible action.

Search Trees:

- In the case of *repeated states* (i.e. cases where a state is descendant of itself), repeated states can simply be treated as dead ends.
  - (∗) *Reasoning*: If a solution can be found that visits a repeated state, a shorter solution can be found with no repeated states.
    - Real-world search algorithms will often weigh the gains given by remembering states vs. memory capacity required to do so.
- A problem with infinite states can potentially have infinite paths in its search tree; however, any solution can only be a finite path.
- A *search tree* is, notably, distinct from a ***search space***, i.e. the set of all states connected by actions.
  - A search space is a mathematical concept/space, a set of states; a search tree is an algorithmic data structure.

Given a problem, its difficulty may be gauged via a number of metrics:

- ***Complexity***: the size of the state space
- ***Branching factor***: the number of actions
- ***Solution depth***: the number of steps needed to reach the goal

The way states are represented inside a problem may affect the number of distinct possible actions, i.e. the branching factor of the problem.

Simplifying states and actions results in a smaller search tree, results in a faster search; can be done by eliminating "no-information" actions:

(∗) *Ex: Missionaries & Cannibals:*

1. Represent all 6 people as distinct + actions distinguish direction of boat travel
   $\implies (6 \cdot 6) \cdot 2 =$ 72 possible actions (bad)

2. Represent people as missionary/cannibal + actions distinguish dir. of boat travel
   $\implies (2 \cdot 3) \cdot 2 =$ 12 possible actions (better)
3. Represent people as missionary/cannibal, no direction of boat travel
   $\implies (2 \cdot 3) =$ 6 possible actions (best)

Some problems (e.g. object placement/assignment problems; *n-Queens*, *cryptarithmetic*) can be formulated using one of two ways:

1. ***Incremental state formulation***: Begin with no objects placed, then add one object per step until all n objects have been placed.
   $\implies$ solution depth: exactly n.
2. ***Complete state formulation***: Begin with all n objects placed randomly, then re-place one object properly per step until all n objects have been placed correctly.
   $\implies$ solution depth: $\leq n$. (better)

($*$) **Additional Types of Search**:

- Search to minimize a cost function
- Search where the initial state is (i) unsure or (ii) unknown
- Search where the effects of actions are (i) unknown, (ii) non-deterministic, or (iii) random
- ***Contingent planning***: a form of search where plans are continuously adjusted as additional observations of the world are recorded.
- ***Conformant planning***: a form of search without an ability to observe the state of the world (i.e. without knowledge of initial/current state)
  - Problem state represented via ***belief states***: a set of all possible world states
    - Initial belief state given as the set of all possible initial states
    - An action is performed on all states within a belief state simultaneously

3 Uninformed Search

An ***uninformed search*** is a search problem where no knowledge about the world is known beyond the initial state and the set of possible actions. An uninformed search is performed by exploring a search tree

Search Tree Terminology:

1. ***Generation***: Given a node S and an action A, we ***generate*** a child S' of S by computing the state resulting from performing A on S_0 and assigning that value to S'.
2. ***Expansion***: Given a state S in the tree, we ***expand*** it by generating a new child state S' for every possible action performable on S.
3. ***Frontier***: The ***fringe/frontier*** of a search tree is the set of all nodes in the tree that have been generated but not yet expanded.

Given a search tree, we can traverse it via the ***tree search algorithm***:

Tree Search Algorithm

while $frontier$ is not empty:

    pop *node* from $frontier$

    $expand(node)$

    for *child* in $expand(node)$:

        if $child = goal$, return

        $frontier$ += *child*

(∗) **Alternative**: ***Graph search*** is similar to tree search, but uses an additional data structure to keep track of which nodes have already been seen during the search process.

- **Advantage**: No duplicate nodes; each node is expanded at most once
- **Disadvantage**: Requires an additional cost for maintaining the data structure

3.1 Uninformed Search Algorithms

The essence of search is the problem of determining which state to expand.

A search algorithm is classified in terms of 4 characteristics:

1. ***Completeness***: If $\exists$ a solution, will the algorithm always find it?
2. ***Optimality***: Will the solution found by the algorithm be the best (i.e. minimum cost) of all possible solutions?
3. ***Time Complexity***: What is the most amount of time (i.e. number of steps) taken by the algorithm?
4. ***Space Complexity***: What is the most amount of additional space used by the algorithm?

Two paradigms for node expansion: ***BFS*** & ***DFS***.

Breadth-First Search: *Expand the state closest to the initial state*
Depth-First Search: *Expand the state furthest from the initial state*

To analyze these algorithms: assume they are being run on some algorithm with (i) branching factor b, (ii) solution depth d, and (iii) maximum depth m.

Properties of BFS:

1. **Is complete**: Every state will eventually be seen by the algorithm
2. **Is optimal**: Taking cost to be # of actions to the goal, BFS will always look at the layers closest to goal first.
3. **Time complexity**: $O(b^d)$
   (a) Exponent depends on when each node S is tested for goal:
      (i) **After expanding** S: $O(b^{d+1})$ [bad]
      (ii) **After generating** S: $O(b^d)$ [good]
4. **Space complexity**: $O(b^d)$
   (a) Either $O(b^{d+1})$ [bad] or $O(b^d)$ [good], similar to time complexity

Properties of DFS:

1. **Is not complete**: If a search tree is infinite (i.e. of infinite depth), DFS may keep going down the wrong path and never terminate
   - Is complete on finite state spaces
2. **Is not optimal**: The solution found by DFS is not guaranteed to be optimal
3. **Time complexity**: $O(b^m)$
4. **Space complexity**: $O(mb)$

3.2 Extensions of DFS

When evaluating search algorithms, space complexity is more important than time complexity:

- Time is [generally] *unbounded*, i.e. we assume we have infinite time.
- Space is usually *limited/bounded* (by hardware constraints, e.g.)

DFS gives a much better space complexity than BFS [O(mb) vs O(b^d)], but:

(i) DFS is not guaranteed to terminate: if a state space is infinite, it may continue forever down a "bad" branch.

(ii) DFS is not complete

(iii) DFS is not optimal

(iv) DFS has worse time complexity: O(b^m) vs O(b^d)

3.2.1 Depth-Limited Search

Improvement 1: To make DFS terminate, we can set a maximum depth limit l that limits how far DFS is allowed to explore. (***Depth-Limited Search***)

Properties of DLS:

1. **Is not complete**: DLS finds a solution only if $l \geq d$
   - This can be useful if a solution is known to exist within a certain depth
   - ($*$) *Ex.* (*Shortest Path*):
     - Given a graph with n nodes, any longest path between two nodes in the graph will have at most $n - 1$ edges
     - Let k be the longest path between any two nodes; then the shortest path between those nodes uses at most k edges.
2. **Is not optimal**: Same behavior as DFS
3. **Time complexity**: O(b^l)
4. **Space complexity**: O($b \cdot l$)

3.2.2 Iterative Deepening Search

Improvement 2: To make DFS complete and optimal: we can set an initial depth limit and keep incrementing it until a solution is found. (***Iterative Deepening Search***)

Properties of IDS:

1. **Is complete**: If we keep incrementing our depth limit l, then we will eventually reach $l = d \implies$ our algorithm will find a solution
2. **Is optimal**: When our depth limit reaches $l = d$, we will find the optimal solution; before then, we can never return a suboptimal solution
3. **Time complexity**: $O(b \cdot d)$
4. **Space complexity**: $O(b^d)$
   - We see that our algorithm will generate the nodes in the first layer d times, the nodes in the second layer $d-1$ times, ..., and the nodes in the d^{th} layer 1 time

$$\implies (d)b^1 + (d-1)b^2 + \ldots + (1)b^d = b^d(1 + 2b^{-1} + \ldots + db^{1-d})$$
$$\leq b^d \sum_{n=0}^{\infty} nb^{n-1} = b^d \left(1 - \frac{1}{b}\right)^{-2} \sim b^d$$

   Consequence: Despite rerunning DLS d times, our time complexity is no worse than our best algorithm [BFS].

	Breadth-First	**Depth-First**	**Depth-Limited**	**Iterative Deepening**
Complete	Yes	No	Only if $d \leq l$	Yes
Optimal	Yes	No	No	Yes
Time	$O(b^d)$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^d)$	$O(b \cdot m)$	$O(b \cdot l)$	$O(b \cdot d)$

(Note: We can also *combine* different search algorithms, e.g. BFS & DFS, to create a new algorithm.)

3.3 Bidirectional Search (BDS)

Observation. In the case of our search algorithms, the algorithms will start at states near the initial state and progressively increase the size of the ***search area*** (i.e. time taken/number of nodes examined) until the goal is found.

Bidirectional Search: Rather than just searching from the initial state; if we know the goal state beforehand, we can run **two searches in parallel** until a *shared state* is found:

1. One search starting from the initial state
2. One search "in reverse", starting from the goal state.

In terms of search area:

Before: One search area with "radius" d

$\implies$ **Time Complexity**: $O(b^d)$

After: Two search areas with radius $d/2$

$\implies$ **Time Complexity**: $O(b^{\frac{d}{2}}) = O\left(\left[\sqrt{b}\right]^d\right)$

Drawbacks of BDS:

1. BDS only works if we already know the goal state
2. For at least one search area, we need to remember every node in its frontier in order to check for shared states

   $\implies$ **Space Complexity**: $O(b^{\frac{d}{2}})$

3.4 Uniform-Cost Search

We may need to run search in scenarios where cost is not always 1 per action (though we do assume that costs are non-negative).

(∗) *Ex*: Distances between graph nodes in a weighted graph

Previous algorithms fail (ex: vanilla BFS will no longer be optimal); use ***uniform-cost ssearch/Djikstra's algorithm***.

Uniform-Cost-Search:

Defining evaluation function $f(n)$ to be the distance $g(n)$ of a node from the start:

Djikstra's Algorithm

1. While the goal has not been expanded, keep expanding the cheapest/least-cost node in the frontier
   - For each node in the tree, keep track of/store the least-cost path to that node from the initial state
   - When the goal is expanded, take its least-cost path as the solution

Optimality: When the goal is expanded, all other unexpanded nodes in the tree will be of higher cost [to reach] than the current solution; then any path through those nodes will be worse than the current solution.

Complexity: Define new parameters: cost of optimal solution c^*, cost of cheapest action ϵ

$\implies$ maximum solution depth: $\frac{c^*}{\epsilon}$

$\implies$ **Time, Space Complexity**: $O\left(b^{\frac{c^*}{\epsilon}+1}\right)$ [similar to BFS with goal test on expansion]

4 Informed Search & Heuristics

4.1 Informed Search

In the case of ***informed search***, we may be able to use problem-specific domain knowledge to help find solutions faster; we can represent our knowledge as a ***heuristic function*** $h(n)$.

A heuristic typically represents an *educated guess* about which states are best/most likely to lead to a solution/an optimal solution.

- *Trivial heuristic*: $h(n) = 0 \; \forall \, n$ is equivalent to doing a blind search

$(*)$ *Ex*: A heuristic may represent a distance from a goal state (with $h(n_{goal}) = 0$)

Greedy/best-first search: Taking cost function $g(n) = h(n)$, always expand the frontier node with lowest cost.

$(+)$ Generally very fast

$(-)$ Not optimal; is complete only on finite state spaces

$(-)$ Worst-case time/space complexity is the same as DFS (generally better in practice)

4.1.1 A* Search

Insight: We can make a new algorithm by combining greedy search and uniform-cost search.

- Greedy search by itself is too greedy (i.e. no memory)
- Uniform-cost is too conservative (no knowledge of goal)

Idea: Given a node n, we can estimate the cost $f(n)$ of a solution through n in terms of:

1. Distance from initial state $g(n)$
2. Estimated distance from goal $h(n)$

$$\implies \underline{\textbf{Estimated cost}}: \; f(n) = g(n) + h(n)$$

A **Search***: Taking $f(n)$ as above, run uniform-cost search.

- Every time a node is added to frontier, delete all copies with worse cost

Definition (***Admissible Heuristic***)**.** A heuristic is ***admissible*** if it never overestimates the true cost of reaching the goal.

- Must be overly optimistic - assume best-case scenario

For A* search, want heuristics to be (1) non-negative and (2) **admissible**.

Properties of A*:

1. **Is optimal** if heuristic is admissible
2. **Is complete**: Same argument as uniform-cost
3. **Time/Space Complexity**: Same as uniform-cost

Proof of Optimality

Proof. Let x be the last non-goal node in an optimal solution; suffices to show that the goal will never be expanded before x (algorithm will never terminate before expanding x).

Comparing costs for each node:

1. Goal: $f(goal) = g(goal) + h(goal) = g(goal)$
2. x: $f(x) = g(x) + h(x)$

Assuming the goal was generated by a non-optimal solution, then $g(goal) > c^*$.

Assuming our heuristic $h(n)$ is admissible, then $f(x) = g(x) + h(x) \leq c^*$.

Then $f(x) < f(goal)$, therefore x will always be expanded before the goal. □

Can also define ***consistent heuristics***: a heuristic h such that $h(n)$ is less than or equal to the cost of any neighbor $h(n')$ plus the cost of reaching that neighbor

$\implies$ ***graph search A**** becomes optimal if the heuristic is consistent.

- Consistent heuristics are a subset of the admissible heuristics

Iterative Deepening A*: IDS, but coupled with a heuristic

- **Issue**: if we bound our solution depth by the distance of a path, it might occur that a single increment only gives one more possible path $\implies$ requires lots of incrementing

  Consequence: IDA* works best on graphs where incrementing the limit causes many more nodes to be added to the graph

4.2 Heuristics

One metric for the quality of a heuristic is the ***effective branching factor*** b^*, used to describe the # of actions removed from consideration under the heuristic.

- A smaller effective branching factor means the algorithm takes a more direct path to goal
- The total # nodes generated up to a level d is given by $N = 1 + b^* + \ldots + (b^*)^d$

Ideally, we want our heuristic to always be as high as possible, while still remaining admissible.

- Intuition: A^* will explore only nodes with cost less than the goal, and will explore all such nodes; to reduce the number of nodes explored, we want our heuristic to push as many nodes above that threshold as possible

4.2.1 Comparing Heuristics

A heuristic h_1 ***dominates*** another heuristic h_2 if $h_1 \geq h_2$ for all nodes n.

- Ideally, want heuristics that are high (dominate many others), but are also also computationally cheap
- A *perfect heuristic* is a heuristic that, given a node n, always gives the exact cost from that node to the goal.
  - A perfect heuristic will explore only the nodes on the solution path; dominates all other heuristics
  - Ex: Running A^* as a heuristic

When choosing between two heuristics h_1, h_2 that do not dominate each other, choice of heuristic depends on priority:

- Fast: Take whichever one computes faster
- Efficient: Take $h_3 = \max\{h_1, h_2\}$ as the heuristic
  - Automatically dominates both h_1, h_2

(∗) Sliding Tile Heuristics

Three heuristics for sliding-tile:

0. h_0: $h_0(n) = 0 \; \forall \; n$ (trivial heuristic)
1. h_1: # of tiles in incorrect position

- *Is admissible*: each move moves at most one tile → at least that many moves needed to put every tile in right position

2. h_2: sum of lengths of shortest paths from all tiles to their goal positions
    - *Is admissible*: Via similar argument

Observation. $h_1 \leq h_2$ for all nodes n. (h_2 dominates h_1).

4.2.2 Finding Heuristics

Methods for finding heuristics:

- ***Domain knowledge***: For some problems, there may be some property of the problem or problem solutions that can help the search algorithm find a solution faster
- ***Relaxation***: Come up with game that is easier to solve, and solve it
    - If a game has a number of rules, can loosen or delete some of them to obtain an easier game
    - resulting heuristics are admissible - any solution for the original game still solves the simplified one, just not vice versa
- ***Subproblems***: Change the goal state to make the goal easier, and solve
    - admissibility - solutions to real problem are subset of solutions to relaxed problem/-subproblem

(∗) Ex: Sliding-Tile (*Relaxation, Subproblems*)

First, look at the initial rules for sliding tile:

1. One piece can move per turn
2. A piece A can move to a square B if:
    (a) A is adjacent to B
    (b) B is blank

Making easier games:

1. Delete 2(a), 2(b): # moves to solve is # tiles in wrong location (same as h_1)
2. Delete 2(b): h_2

Alternatively, define a ***subproblem*** with a revised goal: only certain tiles (e.g. 1, 2, 3) need to be in the right position.

(∗) Ex: Traveling Salesman (*Relaxation, Domain Knowledge*)

Insight: A solution to the traveling salesman problem is a spanning tree, plus one edge

$\implies$ A minimum spanning tree minus its smallest edge is smaller than an optimal solution to traveling salesman. (In particular: finding an MST is a *relaxation* of traveling salesman.)

Heuristic: Merge all traversed edges, find cost of MST on resulting graph.

4.2.3 (∗) Patterned Databases

Observation. A subproblem-based heuristic is running a search inside of a search; doing this naively may accidentally repeat searches (incurring additional cost).

Improvement: Rather than resolving a subproblem at each step, pre-compute a ***patterned database*** containing solutions to all possible subproblems.

- Generating a pattern database: generate all possible subproblems, solve them, compute solution costs, and store in some data structure.
  - Heuristic becomes a simple lookup once patterned database is built
  - Combining patterned databases: Take a max
- Is slow to pre-compute (same time as solving from scratch), but leads to large speedups once database is built
  - Can solve large problems where regular heuristics fail

Alternative approach: From goal state, go backwards - compute the cost of reaching all possible initial states (from goal state) and use this cost as the heuristic.

(∗) To combine two patterned databases, just use their max as the heuristic.

For certain problems, can take a partition of the original problem (into subproblems) to create ***disjoint patterned databases***; want to be able to "add up" heuristics from each database to create a new heuristic.

(∗) Issue: adding normally might overcount actions (is not admissible); can solve by changing how actions are counted within subproblems

(∗) Ex: For a sliding tile subproblem (only placing 1, 2, 3 in correct locations, e.g.), only count the actions that move a 1, 2, or 3

5 Constraint Satisfaction

Constraint satisfaction problems [**CSPs**] are search problems that can be written in a formal language (i.e. in a form similar to mathematics)

Constraints specify what constitutes a solution to a CSP; analogous to goal tests

- Many real-world problems can be modeled as CSPs

CSPs typically written in terms of assigning values to some number of variables; written out as ***factorized state representations*** [breaking up states into all individual variables]

- *Ex*: For a colouring problem, can describe color of each section as a variable
  - Can specify color constraints explicitly (e.g. for two adjacent sections A and B, write out all possible legal & illegal states)

Types of Constraints:

1. Unary: Only involves one variable
2. Binary: Involves two variables; better than unary constraints for solvers
3. Higher-order: Involves more than two variables
   - Global: Involves all variables
   - CSP solvers typically prefer many smaller constraints over a single larger constraint; smaller constraints may be faster to compute and fail more quickly, e.g.

(∗) Can also talk about *preferences/soft constraints*

(∗) In the case of constraints that are mathematical expressions (e.g. $X^2 + Y = 4$), can distinguish between *linear* vs *non-linear* constraints

- Linear constraints over $\mathbb{R}$ can be solved efficiently in polynomial-time [*linear programming*]
  - Linear programming only works for solutions in $\mathbb{R}$; solving becomes NP-complete if solutions need to be integers
  - Works for both soft & hard constraints
- Diophantine equations (e.g. $x^3 + y^3 + z^3 = 42$) proven unsolvable/undecidable, i.e. there cannot exist any algorithm to solve them

Ex: Cryptarithmetic

- Variables are the letters [to be assigned values]
- Domain for variables are integers $\in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- Constraints:
  1. *All variables are different values* - written as many $\neq$ constraints
  2. *Overall expression fulfills equality* - written explicitly
     - *Issue*: Would prefer to write as smaller constraints for individual digits rather than a single global expression, but cannot write as constraints for individual digit places without strange operators (e.g. modulos to control for carrying)

       Solution: Introduce "artificial" variables representing carried values

Can visualize CSPs using ***constraint graphs*** - graphs where each node is a variable, edges between nodes represent constraints across variables

Type of graph depends on order of constraints:

1. Binary constraints (simple case) - edges have two endpoints [*binary constraint graph*]
2. Higher-order constraints - use *hypergraphs/bipartite graphs/higher-order constraint graphs*
   - Introduce new type of "node" representing a constraint; for variables connected by a constraint, draw an edge from the variable nodes to the constraint "node"

5.1 CSP Search

Framing CSP search as a search problem:

1. *Initial state*: Initial state consists of empty assignments to all variables
2. *Actions*: Each action consists of assigning a value v to a variable X
   - *Successor*: Successor function adds $X = v$ to the state
3. *Goal*: Goal test checks that all variables are assigned, all constraints are satisfied

Naive CSP search uses uninformed search (DFS), no heuristics

- Is complete and optimal - search tree is finite, and all solutions have depth equal to number of variables

- *Complexity*: Given n variables on a domain of size d:
  - Naive DFS: Branching factor $n \cdot d \implies \mathrm{O}(n! \cdot d^n)$ [very bad]
    - Observation: Computes $n! \cdot d^n$ nodes, but there are only d^n possible solutions
  - Order the assignments of variables and use backtracking search $\Rightarrow \underline{\mathrm{O}(d^n)}$

5.1.1 Optimizing CSP Search

Can optimize CSP search in three respects:

1. Order of picking variables
2. Order of picking values
3. Detecting failure early

Variable Order: For variable order, use variable selection heuristics: want to pick the variables that are most connected/most specified by constraints (to detect failure/violate constraints earlier - fail-first/fail-faster)

1. ***Most constrained variable/minimum remaining values***: Pick the variable with the fewest legal values remaining
2. ***Degree heuristic***: Pick the variable with the most constraints on its remaining values
   - Easy to determine from constraint graph; can be used as a tie-breaker after most constrained variable

Value Order: For value order, want to first test the values that are most likely to lead to success $\Rightarrow$ use the ***least constraining value heuristic***: Pick the variable value that rules out the fewest values in remaining variables

Detecting Failure: To detect failure early, want to keep track of remaining legal values for variables; can stop a search prematurely if any variable has no more legal values

3 ways to check legal values:

1. *Only check constraints*: After assigning a value to a variable, only check to make sure the assignment did not break a constraint [simplest]
2. ***Forward checking***: Upon assigning a value, update the legal values of adjacent variables
3. ***Arc consistency***: Upon assigning a value to on a variable, update the legal values for all other variables

- Updating legal values: for every pair of variables (X, Y) connected by a constraint, for all legal values x for X, verify that there exists a legal value y for Y such that (x, y) satisfies the constraint
  - Any changes to legal values of a variable are then propagated through all of its connected variables
- *Time Complexity*: d^2 (for checking two variables on an edge) $\cdot\, d$ (run at most d times on every edge until all values removed) $\cdot\, n^2$ (total edges) becomes $\mathrm{O}(n^2 d^3)$
  - Arc consistency generally used as a preprocessor before starting CSP search; forward checking used during the search itself

6 Local Search & Optimization

A search problem becomes ***local search*** if either:

1. The path to goal is irrelevant, we just want to find a goal state
   - *Ex*: n-queens, crypt-arithmetic
2. The goal is to find maximize/minimize some value, rather than a specific state

Consequence: Local search algorithms only need to know the current state; do not need to store paths in memory, e.g.

- Benefit: Constant memory requirement (a single node/state)

Local Search Terminology

- An ***objective function/value*** is a value we want to minimize/maximize.
  - *Ex*: given a mathematical function, what point maximizes its value?
    - use gradient descent to locate
- Given an objective function, we distinguish between ***local extrema*** (best within a given subset of the state space) vs ***global extrema*** (best across the entire state space)
- A state space may be either ***continuous*** (e.g. $\mathbb{R}$ for a real-valued function) or ***discrete*** (e.g. placement of n queens in n-queens)

Recall: In mathematics, we can find the local extrema of a function by following the gradient at each point (***gradient descent***).

6.1 Hill-Climbing Search

Hill-Climbing Search/Greedy Local Search: At a node n_0, always take the neighboring state n that maximizes the objective function; stop when $f(n_0) > f(n) \; \forall \; n$.

- Analogy: Taking the objective value of a node to be the *elevation* of that node, hill-climbing search operates by always going in whichever direction is "uphill".

($*$) *Ex (n-Queens)*: Defining the objective function on a state to be the number of queen pairs that are attacking each other:

1. Begin by randomly initializating the placement of queens
2. For every queen, compute the objective function for all possible states from moving that queen
3. Perform a move that results in a state minimizing the objective function
4. Recompute & continue until goal
   [*Almost always succeeds on random instances of n-Queens*]

(∗) *Ex (Traveling Salesman)*: objective function as length of tour

- define action: delete two edges, reconnect four affected cities with any new edges

Issue: A single instance of greedy local search may get stuck on local extrema, rather than reaching a global extremum.

Improvement 1: Perform multiple greedy searches and pick the best result

*Improvement 2 (**Beam Search**)*:

1. Start with $k > 1$ starting points/states ("agents")
2. For each agent, compute all of its next possible states
3. Rather than picking one next state for each agent, pick the k best states out of all found states and continue searching from those states

(∗) *Ex*: Used in LLMs - keep finding next-best token/word via beam search

6.1.1 Simulated Annealing

Simulated annealing introduces randomness to step of picking next states:

1. Begin on some state
2. For each state, pick a random next state:
   - If the next state is better than the current state, move to the next state [commit]
   - If the next state is worse than the current state by ΔE, then commit to the next state with probability $e^{-\frac{\Delta E}{T}}$; otherwise, stay on current state

Intuition: Simulated annealing is okay with moving to states that are worse than our current state, if they are not worse by too much

- T is the temperature parameter - indicates how heavily "worseness" should be weighed

- Can start with T high (since the start is likely not near an extremum), then decrease T over time
- If $T = 0$, algorithm is equivalent to hill-climbing

- Overall structure is a random walk, but trends toward higher points ove time
- Etymology: E represents energy

6.2 Genetic Algorithms

Genetic algorithms attempt to "simulate" the process of natural selection:

1. Start with an "initial population" of states
2. For each state, assign a score to that state via a *fitness function*
3. For each state, compute a probability that they will "participate in procreation" (take score of state divided by total score, e.g.)
4. Selection: Pair the procreating states up as parents
5. Crossover: For each pair of parent states, generate a child state(s) that contains "features" of both parents
6. Mutation: For each child state, perform some random alteration
7. Repeat process with set of child states; over time, population becomes better

Genetic algorithms are inspired by real-life processes of genetic inheritance, evolution

- *Ex.* (*n-Queens*): Given two parent states, create a child state with the same columns 1-3 as first parent, same columns 4-8 as second parent
  - Two child states per pair [two ways to assign first, second parent]
  - Mutation - pick a random column [within each child] and move the corresponding queen to a different position
- Genetic algorithms are popular due to novelty, but are also often slow and wasteful due to testing many bad sequences
  - Are rarely best solutions to problems, but can offer reasonable and simple solutions to new, not-yet-solved problems
  - (∗) *Ex*: Genetic algorithms currently used for neural architecture search - task of learning the ideal structure for an ML model (# layers, e.g.)

7 Adversarial Search & Games

Computer science long used to study games (ex: chess)

- Games are very abstract, can be very complicated, but still possess very well-defined rules

Subclasses of Games:

- Perfect vs imperfect information games - how much of the game state is immediately visible to each player?
  - In the case of imperfect information, may need to use actions to obtain additional information (ex: poker)
- Deterministic vs chance games - is there a degree of randomness?
- Zero-sum vs non-zero-sum - is one player's gain always the other player's loss?

Some complications in adversarial search problems are:

- Presence of an active opponent:
  - Opponent actions add uncertainty
  - Due to opponent presence, a sequence of actions [that assumes what the opponent's actions are] is not enough as a solution; need a different kind of solution
    - Easier: A solution is just a single action [i.e. just returns the next action]
    - Harder: A solution is a ***strategy*** - specifies an action for every possible scenario
- Often involve high branching factor, large search tree depth

  (∗) *Ex*: Chess has $b = 35$, $d \approx 100$
- May sometimes have time limits on decisions

Typically judge/compare game outcomes via a ***utility function*** (*ex*: win=1/loss=-1/draw=0)

(∗) *Terminology*

- Move - pair of actions (one action for each player)
- Half-move/ply - one action from one player

7.1 Minimax Algorithm

The ***minimax algorithm*** uses a search tree and assigns a ***utility*** for each node.

Simple case: two players (one trying to minimize the utility function; one trying to maximize utility function) alternating turns:

1. Generate a search tree rooted at the initial state
2. Compute the utility of nodes as follows:
   - If the node is a leaf, then find its utility via utility function
   - If the node is not a leaf:
     - If the next action from that node is by the minimizing player, assign its utility as the minimum of the utilities of its children
     - If the next action from that node is by the maximizing player, assign its utility as the maximum of the utilities of its children
3. Build the tree:

   (i) Strategy 1: Build bottom-up (starting from the leaves), store entire search tree
   - Time: b^m, space: b^m

   (ii) Strategy 2: Keep running depth-first search on branches, starting from the root
   - Time: b^m, space: bm

Minimax Algorithm:

- Consequence of minimax: If all players play optimally, the game will always end in the same way (either a specific player wins, or all players draw)
  - *Ex*: Tic-tac-toe is solved - always ends in a draw
- Behavior from minimax depends only on ordering of preferences (i.e. relative values of utility function on various end-states), not numerical value of rewards
- Minimax algorithm assumes optimal behavior by both players when optimizing utility
  - If a player is playing non-optimally, the other player may be able to obtain a better outcome [utility] than given by minimax

In the case of multiple players alternating moves, similar principle: compute the utility of each node based on the objective of whichever player is acting next

(∗) May run into Nash equilibria (differences between optimizing common, individual goods)

7.1.1 Optimizing Minimax

During minimax, can prematurely determine that certain subtrees will never be taken and stop evaluating them early (α-β pruning):

- *Ex*: If the maximizing player is on a node n:
  - If the utility of the first child is x, mark utility of n as $\geq x$
  - Evaluate all remaining children:
    - If, at any point, a descendant has value ≤ 5, we can stop calculating that subtree and return
- Use α to denote the best choice for max player, β best choice for the min player

Degree of optimization (i.e. effective branching factor) depends on the order states are visited

- Want most constraining alpha/beta (highest alpha, lowest beta) to reduce search tree as much as possible
  - Difficult to do exactly, but can approximate using heuristics to order outcomes
    (∗) *Ex*: In chess, prioritize outcomes that capture a piece
- Time complexities (α-β pruning):
  - Random visit order: $b^m \to b^{\frac{3}{4}}$, typically
  - Optimal visit order: $b^m \to b^{\frac{m}{2}} = \left(\sqrt{b}\right)^m$

For complex games, it is typically only feasible (and is still sufficient) to only search to a certain depth; search depth required to win depends on opponent:

(∗) *Ex*: Chess may take from 4 moves (bad players) up to 12 moves (grandmasters)

Issue (***Horizon Effect***): Even if a game state has an inevitable outcome, a player might be able to "postpone" that outcome until it goes beyond the search depth of an algorithm.

To solve, need to let algorithms know to search deeper:

- Quiescence search: If a state has a child state that is significantly different from itself (e.g. a queen is captured), continue searching rather than evaluating the current state
- Singular extension: If a move is obviously good, search deeper in that direction
- Forward pruning: if a move is obviously bad, ignore it

Minimax Algorithm:

- For further optimization, can use iterative deepening within minimax to save space
- *Issue*: Defining a utility function can be difficult, since slight variations in game state may result in very different outcomes
  - Modern approach: use trained neural networks as utility functions for evaluation
- Additional optimizations/modifications:
  - Opening tables - rather than recomputing the first few moves, use the same openings as those used by existing experts
  - Can build specialized hardware for specific games (e.g. IBM)

In the case of games with chance/randomness involved, add a chance node for each move; children are all possible resulting states (with associated probabilities)

- Assign the utility of a chance node to be the expected value of its children's utility (weighted sum of children's utility, weighted by probability)
  - Consequence: Due to taking an expected value, the magnitude/scale of rewards begins to matter (no longer just relative order)
- *Note*: Adding chance nodes increases tree size dramatically; usually results in shallower trees [less moves simulated]

7.2 (∗) Monte-Carlo Tree Search

Current state of the art: ***Monte-Carlo tree search*** (reinforcement learning)

- Used to solve Go (in conjunction with a neural network, as evaluation function)
- Proven to converge to minimax after infinite # attempts

Principle: Traverse search tree randomly (randomly play games) and look for the moves that most often lead to success

- For each node, keep track of:
  (i) A counter n, indicating how many times the node was visited/passed through
  (ii) A value v, containing the average outcome value/utility of all games that passed through that node

Monte-Carlo Tree Search (Algorithm):

1. Create a search tree (initially empty) in memory
2. *Selection*: Randomly perform actions [select edges in the search tree] until a leaf node (i.e. a node with a not-yet-tested action) is reached
3. *Playout*: Simulate (play out) the rest of the game by playing random moves
4. *Expansion*: Based on results of the game, create a new child under the leaf with scores assigned based on the result of game
5. *Backprop*: Backpropagate the result to update the scores of ancestor nodes in the tree

During selection phase, decide which nodes to select based on on certain statistics

- Two possible criteria/approaches for picking a child:
  - Criterion 1: Pick the child with the highest value [most promising]
  - Criterion 2: Pick the least-visited child [least information]
- Want to find an approach that balances both criteria (reinforcement learning: *exploitation-exploration tradeoff*)

  Solution: Rank using upper confidence bound [UCB] score $UCB = v_i + C\sqrt{\frac{\ln N}{n_i}}$
  - C a constant; higher C prioritizes exploration more heavily
  - Originally used in multi-armed bandit

8 Propositional Logic

A ***reasoning engine*** takes in a knowledge base and questions; outputs conclusions/deductions obtained via ***deductive inference***:

- The ***knowledge base*** (KB) contains all knowledge known about the world
  - Logic operates only on information in knowledge base; needs knowledge base to be comprehensive & complete
  - Stored as declarative sentences in knowledge-representation language
- ***Observations/queries*** specify what information what needs to be known
- ***Conclusions*** [*deductions*] are the results of reasoning

Logic is a way to overcome ignorance about the world via reasoning

- Conclusions are guaranteed to be correct

(∗) Two models for reasoning/inference: causal model (X causes Y) vs evidential model (if X, conclude from X that Y)

- Causal models generally easier, more applicable to probabilistic models

Applications of Logic:

- *Verification*: Given specification α and implementation β, can use logic to show that β entails/fulfills α [implementation matches specification]
  - Can be used for hardware/software verification
- Generating mathematical proofs (*ex*: boolean Pythagorean triples)
- Natural language processing [symbolic/logical knowledge]

(∗) Dominant AI paradigm from 1958-1988; later overtaken by probabilistic models

8.1 Syntax & Semantics

Syntax composed of two types of objects:

1. ***Atoms***
2. ***Logical connectives***: unary, and/or, implies, if, if and only if

Syntax given via logical formulas/***sentences***; sentences either considered atomic [composed of a single ***literal***] or compound [contain multiple ***literals***]

- Literal - an atom, or its negation

Syntax are just symbols; are mapped to real-world objects via *semantics*:

- Given a sentence α, world ω: either sentence α is true in world ω ($\underline{\omega \models \alpha}$, ω ***satisfies*** α) or α is false in world ω ($\underline{\omega \not\models \alpha}$)
  - Meaning of α: only possible worlds are worlds satisfying α (models of α: $M(\alpha) = \{\omega : \omega \models \alpha\}$)
- Can express the same models using many different sentences

To determine whether a world satisfies a set of sentences, can check each sentence individually

- Can be done in linear time (relative to number of sentences)
- Alternative: Build a ***truth table*** listing all possible worlds, determine values of compound sentences from values of literals
  - A world is simply an assignment of a value True/False to each atom

8.2 Relationships Between Sentences

Interpret knowledge base KB as a set of sentences; is true if $\omega \models KB$

Definitions:

- A sentence α is ***valid*** if all worlds satisfy α ($\forall\, \omega : \omega \models \alpha$; $M(\alpha) = W$)
  - Represent subset of universe W with α satisfied, as circle with vertical lines
- A sentence α is ***satisfiable*** [**SAT**] if at least one world satisfies α ($\exists\ \omega : \omega \models \alpha; M(\alpha) \neq \emptyset$)
- A sentence α is ***unsatisfiable*** [**UNSAT**] if it holds in no world ($\forall\ \omega : \omega \not\models \alpha$; $M(\alpha) = \emptyset$)
  - Also called: inconsistent, not consistent, not satisfiable
- α ***entails*** β ($\alpha \models \beta$) if: whenever α holds, β must also hold ($\forall\, \omega$: if $\omega \models \alpha$ then $\omega \models \beta$)
  - Alternatively: $M(\alpha) \subseteq M(\beta)$, $\alpha \implies \beta$
  - If α true, β false; but can be that β true, but α false
- α and β are ***equivalent*** if $\omega \models \alpha$ iff $\omega \models \beta$, $M(\alpha) = M(\beta)$
- α and β ***mutually exclusive*** (*mutex*) if $\forall\, \omega : \omega \not\models \alpha \vee \omega \not\models \beta$; $M(\alpha) \cap M(\beta) = \emptyset$

Relationships between Logical Notions:

1. *Validity to SAT*: α is valid iff $\neg\alpha$ is UNSAT
2. *Entailment to SAT*: $KB \models \alpha$ iff $KB \cap \neg\alpha$ UNSAT (KB: knowledge base)
3. *Entailment to Validity*: $KB \models \alpha$ iff ($KB \implies \alpha$ is valid)
4. *Equivalence to SAT*: $\alpha = \beta$ iff $[(\alpha \cap \neg\beta) \cup (\neg\alpha \cap \beta)]$ UNSAT

($*$) ***Monotonicity of Logic***: If $KB \models \alpha$ and we learn a new fact β, then it is still the case that $(KB \cap \beta) \models \alpha$.

- *Consequence*: If we "know" something, it is not possible to revise this knowledge/retract a claim [vs non-monotonic probabilistic logic]
- Is accurate mathematically, but not necessarily true to real-life reasoning

8.3 Normal Forms

Terminology:

- A ***literal*** is a variable X or a variable negation $\neg X$
  - A literal is called *positive* if it is not a negation
- A ***clause*** is a *disjunction of literals*, i.e. a union/OR of literals ($A \cup B \cup \neg C$, e.g.)
  - A ***Horn clause*** is a clause that contains at most 1 positive literal
  - A ***defininte clause*** is a clause that contains exactly 1 positive literal
- A ***term*** is a *conjunction of literals*, i.e. an intersection/AND of literals ($A \cap \neg B \cap C$, e.g.)

Observations:

- Removing a literal from a clause returns a subset of the original clause
  - The *empty clause* (all literals removed) is equivalent to *False*
- Removing a literal from a term returns a superset of the original clause
  - The *empty term* (all literals removed) is equivalent to *True*

Normal Forms: restrictions of permissible syntax

- ***Conjunctive normal form*** (**CNF**): conjuction of disjunction of literals
  - *Disjunction*: $A \cup \neg B$
    - Removing terms from clauses: removing a term (e.g. $A \cup B \cup \neg C$ to $A \cup B$) takes a subset of original clause
  - *Conjunction*: $(A \cup \neg B) \cap (B \cup \neg C \cup \neg D)$, *a la* SAT
  - Any propositional logic sentence can be expressed in CNF
- ***Disjunctive normal form*** (**DNF**): disjunction of conjunction of literals
  - Conjunction of literals - term/conjunctive clause
- ***Horn form***: conjunction of Horn clauses (clauses with at most 1 positive literal)
  - Is a subset of CNF - imposes additional restriction on clauses
  - *Interpretation*: $(B \cup \neg C \cup \neg D)$ implies $(C \cap D \implies B)$

    - $\neg B \cup \neg C$ says $C \cap E \implies$ False (integrity constraint - specifies something not allowed/cannot happen)
    - $A \cup \neg B$ says $B \implies A$ (is a definite clause: has exactly 1 positive literal)

Complexity of Normal Form solvers:

- Check if **CNF SAT**: *NP*-complete
- Check if **CNF Valid**: Linear in # clauses
  - Check all clauses sequentially; if any clause is not valid (i.e. can create a world where clause is false), fail
  - CNF clause is valid iff contains a variable $\neg\alpha \vee \alpha$
- Check if **DNF SAT**: Linear in # clauses
  - Check all clauses sequentially; if any clause is satisfiable, return true
  - Note: negation of CNF is DNF; negation of DNF is CNF
  - Equivalent of checking DNF SAT is checking negation [CNF] valid
- Check if **DNF Valid**: *NP*-complete
  - Since negation of DNF Valid is CNF SAT

Reducing to CNF (Steps)

1. Eliminate $\Longleftrightarrow$: convert to $\Longleftarrow, \Longrightarrow$
2. Eliminate $\Longleftarrow, \Longrightarrow$: convert $\alpha \implies \beta$ to $\neg\alpha \vee \beta$ [disjunctions]
3. Move $\neg$ inside: if $\neg$ on a compound expression, move inside
   - Ex: $\neg(\alpha \vee \beta)$ to $\neg\alpha \wedge \neg\beta$
4. Distribute $\vee$ over $\wedge$ (only want $\wedge$ on outside, $\vee$ oon inside)
   - Ex: $\alpha \vee (\beta \wedge \gamma)$ to $(\alpha \vee \beta) \wedge (\alpha \vee \gamma)$
   - Worst case: CNF size increases exponentially
     - Because of this, cannot convert "is DNF valid" to "is equivalent CNF valid" and solve in linear time

9 Logical Inference

Logical inference is the process of checking if a knowledge base KB entails a sentence α.

Three strategies for logical inference:

1. ***Enumerating models***: Checking if $M(KB) \subseteq M(\alpha)$
2. ***Deduction***: Using known logical patterns to try and *deduce* α from the KB
3. ***Refutation***: Testing if assuming $KB \wedge \neg\alpha \implies UNSAT$

9.1 Enumerating Models

Enumerating Models: To check if $M(KB) \subseteq M(\alpha)$, build a truth table & compare values of KB, α in each row [world].

- If set of worlds satisfying KB is subset of set of worlds satisfying α (every world with $\omega \models KB$ has $\omega \models \alpha$), then $KB \models \alpha$

$(*)$ Drawback: Computationally expensive (exponential)

9.2 Deduction

Deduction: We can *deduce* α from the KB $[KB \vdash \alpha]$ using ***deduction rules***.

- Most efficient if the length of the proof is small

Deduction rules: Logical statements that if certain patterns $\text{pattern}_1, \ldots, \text{pattern}_n$ exist and hold, then a pattern_{n+1} holds ($\frac{p_1,\ldots,p_n}{p_{n+1}}$):

- **Modus ponens**: If $\alpha \implies \beta$ and α holds, then β holds $[\frac{\alpha, \alpha \implies \beta}{\beta}]$
- **Double negation**: If $\neg\neg\alpha$ holds, then α holds (and vice versa)
- **Or-introduction**: If α and β, then $\alpha \vee \beta$ holds
- **Bi-directional implication**: A bidirectional implication $\alpha \iff \beta$ is equivalent to two unidirectional implications $\alpha \implies \beta$, $\beta \implies \alpha$
- **Contrapositive rule**: An implication $\alpha \implies \beta$ is logically equivalent to the reverse implication $\neg\beta \implies \neg\alpha$.
- ***DeMorgan's Laws*** relate to the negation of conjunctions/disjunctions:

- Negation of disjunction: $\neg(\alpha \vee \beta) \implies \neg\alpha \wedge \neg\beta$
- Negation of conjunction: $\neg(\alpha \wedge \beta) \implies \neg\alpha \vee \neg\beta$

If we can use deduction rules to go from KB to α, have shown that α is entailed by the KB

(∗) Drawback: Have many rules, need to pick the right one [in humans: by intuition; machines: brute force]

For deduction to work, define various properties of deductive rules:

- Need ***soundness***: If $KB \vdash_R \alpha$, then $KB \models \alpha$.
  - (∗) Ex. (Not sound): $A, A \implies B$ says $\neg B$
- Want ***completeness***: If $KB \models \alpha$, then $KB \vdash_R \alpha$
  - (∗) Ex. (Sound, not complete): modus ponens $A, A \implies B$ says B [ex: if $KB = A \cap B$ and $\alpha = B$, modus ponens does not give]
  - (∗) Ex. (Complete, not sound): $\frac{}{\alpha}$

9.3 Refutation

Say a rule is ***refutation-complete*** if: $KB \models False \implies KB \vdash False$.

- Weaker version of completeness
- If rule sound and refutation-complete, the $KB \models$ False iff $KB \vdash_R$ False [via entailment-to-SAT]
  - Can check if $KB \vdash_R$ to check if $K \models$ True

$\Rightarrow$ ***Proof by refutation***: To determine if $KB \models \alpha$, test if $KB \wedge \neg\alpha \implies$ False/UNSAT.

- How to test: from KB, generate $KB' = KB \wedge \neg\alpha$; convert to CNF, test if valid

New rule: ***resolution*** - $\frac{\alpha \vee e, \neg e \vee \gamma}{\alpha \vee \gamma}$

- Sound, not complete: $KB = A \vee B, \alpha = A \vee B \vee C$
- Is refutation-complete if KB expressed in CNF
  - Resolution works on clauses/disjunctions

Resolution algorithm: Take $KB \wedge \neg\alpha$, keep applying Resolution, see if $KB \wedge \neg\alpha \vdash$ False (if so, implies $KB \wedge \neg\alpha \models$ False, therefore $KB \models \alpha$)

- If KB runs out of rules (we cannot prove False; $KB \wedge \neg\alpha \nvdash$ False), then per refutation-completeness, $KB \wedge \neg\alpha \not\models$ False; then $KB \not\models \alpha$

9.4 DPLL

Alternative approach: ***SAT solving/DPLL Search*** uses the constraint satisfaction algorithm (as seen previously) to perform refutation

1. Take individual literals/variables $A, B, C, \ldots$ as variables in CSP (possible values: True, False)
2. Take the set of CNF clauses as a set of CSP constraints
3. Use CSP to check if there is a set of assignments that satisfies $KB \wedge \neg\alpha$ [refutation]

(∗) Basic DPLL algorithm (SAT(F)) - Given a set of clauses F:

- If $F = \{\text{False}\}$, return 0
- If $F = \{\text{True}\}$, return 1
- Else, return $SAT(F_{x=0}) \vee SAT(F_{x=1})$

Want to do constraint propagation (similar to forward checking/arc consistency)

⇒ ***Unit resolution/unit propagation***: If F has a sentence α [single variable] and a sentence $\neg\alpha \vee \beta$, can infer the sentence β

- Either removes a clause, or removes literal from a clause [simplifies clause]

∗ Specialized form of arc consistency/forward checking

(∗) *DPLL Optimizations*:

- If a conflict clause (a simple clause entailed by the original KB) is found, can stop early and restart the search, incorporating that clause into the KB
- *Backjumping*: If a failure was caused by a variable assignment that was made 10 choices ago, backtrack 10 levels in search tree
- If the CNF is found to have separate components, can solve each component separately

- *Pure literal*: If a variable X only appears as positive literal (i.e. only as X, never as $\neg X$), can immediately set X to be True
  - If there is a solution, there exists a solution with $X =$ True; setting X to true from the start makes problem easier (eliminates clauses with X)
- Can use heuristics to determine variable, value assignments
- Can convert the CSP into a graph to find symmetries
  - Helps to avoid evaluating symmetric solutions
- For a very long clause, can designate two variables inside of it as "special" and only update the clause value when one of two specials is assigned
  - Avoids computational cost of re-evaluating clause for every assignment

Can also use local search for faster SAT-solving (rather than full backtracking search)

- *Issue*: If local search fails to find satisfying assignment, then either none exists, or one simply hasn't been found
  - Can only prove $KB \not\models \alpha$, if a SAT assignment is found; could never prove $KB \models \alpha$ without somehow knowing that all states have been seen

10 Probabilistic Reasoning

Most of modern reasoning is based on probability/probabilistic models, not pure logic

- Key breakthrough was creation of Bayesian networks (Judea Pearl, UCLA)

Motivation (*Issues with Logic*):

1. For a given event or observation, there may be many possible things that it implies or that may cause it; e.g. may have $\alpha \implies \beta \vee \gamma \vee \delta \vee \ldots$
   - Writing as pure logic can be intractable in both a theoretical/practical sense, be it due to laziness or ignorance
2. Monotonicity of logic is problematic for applications; to solve problems, must either:
   (a) Model everything exhaustively (potentially intractable/impossible)
   (b) Make various assumptions/approximations, potentially causing issues if these assumptions are later contradicted
      - With monotonicity, cannot revise existing knowledge with further observations

Can make an epistemological change: rather than solely believing in a set of possible worlds/-models, can instead believe in a *probability* for each world/configuration

- Probability corresponds to a notion of "***belief***" in a given world
  - Replaces binary possible/impossible with a probability of a world being true
- Does not change how the world *is* [ontology], only how world is *described* [epistemology]

Additional motivator for probability: decision making

- With logic, may not be able to prove conclusively that a goal is achievable
- With probability, can combine notions of utility (relative importance of goals/values of outcomes), probability (likelihoods of worlds) to define expected utilities

Within logic: can have propisitions/variables that are Boolean, categorical (e.g. sunny, rainy, cloudy; can alternatively be represented with Bools), continuous (infinitely many Boolean variables)

10.1 Probability Overview

A belief consists of/encodes the probabilities of each world; to determine the probability of a sentence [event] α, use Kolmogorov's axioms of probability:

1. Probabilities are non-negative
2. The probability of any true/certain event [i.e. holds in all possible worlds] is 1
3. If two sentences α and β are mutually exclusive, $\Pr(\alpha \vee \beta)=\Pr(\alpha)+\Pr(\beta)$

Everything (e.g. probability of a sentence) follows from axiomization:

- Each sentence is a disjunction of models ($\alpha = \omega_1 \vee \omega_3 \vee \ldots$), where each ω_i is world representing conjunctions of literals ($x_1 \wedge \neg x_2 \vee \ldots$)

  Since worlds are mutex, obtain:

  $$\Pr(\alpha) = \Pr(\omega_1) + \Pr(\omega_3) + \ldots = \sum_{\omega \models \alpha} \Pr(\omega) = \sum_{\omega \in M(\alpha)} \Pr(\omega)$$

  – Called a *marginal probability/single margin* - represents the sum of a slice of the table encoding complete joint distribution

($*$) *Notation*: $\Pr(\alpha \wedge \beta)$ written as $\Pr(\alpha, \beta)$

Probability Properties:

1. $\Pr(\alpha) + \Pr(\neg\alpha) = 1$
2. *Inclusion-Exclusion*: $\Pr(\alpha \vee \beta) = \Pr(\alpha) + \Pr(\beta) - \Pr(\alpha \wedge \beta)$

($*$) *Betting Semantics*: can interpret probability in terms of making bets, gambling

- If an agent has beliefs that are mathematically invalid (e.g. $\Pr(\alpha) + \Pr(\neg\alpha) > 1$), may be able to set up bets against that agent all with positive expected value [Dutch book]

10.1.1 Conditional Probability

Can define **conditional probability**: $\Pr(\alpha \mid \beta) = \frac{\Pr(\alpha \wedge \beta)}{\Pr(\beta)}$ [probability of α given β]

- Represents a way to "*undo*" or *update* prior beliefs: can condition existing beliefs on new observations to revise them in light of new information (i.e. no monotonicity of belief)

Additional Rules:

1. *Product Rule*: $\Pr(X \wedge y) = \Pr(X) \cdot \Pr(Y \mid X)$
2. *Chain Rule*: $\Pr(X_1, X_2, \ldots, X_n) = \Pr(X_1) \cdot \Pr(X_2 \mid X_1) \cdot \ldots \cdot \Pr(X_n \mid X_{n-1}, \ldots, X_1)$
   (∗) Alt. notation: $\Pr(X_1, X_2, \ldots, X_n) = \prod_i \Pr(X_i \mid X_{i-1}, \ldots, X_1)$
   [Describes an *auto-regressive* distribution - each X_i depends on preceding variables]
3. *Partition Rule*: $\Pr(Y) = \sum_Z \Pr(Y, Z) = \sum_Z \Pr(Y \mid Z)\Pr(Z)$

(∗) If computing conditional probabilities $\Pr(X \mid Y)$, $\Pr(\neg X \mid Y)$, can skip computing $\Pr(Y)$ - instead, say that results are $\Pr(X \mid Y) = \alpha \Pr(X \wedge Y)$ up to a normalizing constant α

⇒ After computing both $\Pr(X \wedge Y), \Pr(\neg X \wedge Y)$, obtain $\alpha = \Pr(X \wedge Y) + \Pr(\neg X \wedge Y)$

- Can represent $\Pr(X \mid Y)$, $\Pr(\neg X \mid Y)$ via a vector $\alpha \begin{pmatrix} \Pr(X \wedge Y) \\ \Pr(\neg X \wedge Y) \end{pmatrix}$

(∗) Partition Rule: $\Pr(X \mid e) = \alpha \Pr(X, e) = \alpha \sum_Y \Pr(X, Y, e)$

Define ***independence***: say that an event X is ***[absolutely] independent*** from events $Y_1, \ldots, Y_n$ if $\Pr(X, Y_1, \ldots, Y_n) = \Pr(X) \cdot \Pr(Y_1, \ldots, Y_n)$

- Alt: $\Pr(X \mid Y_1, \ldots, Y_n) = \Pr(X)$ and vice versa

From Product Rule, obtain ***Bayes' Rule***:

$$\Pr(\alpha \mid \beta) = \frac{\Pr(\beta \mid \alpha) \cdot \Pr(\alpha)}{\Pr(\beta)}$$

Bayes' Rule:

- Represents conditional probability, in reverse - allows for reversing direction of causality
  - Can use to find *posterior probabilities* by conditioning the probability of an event based on new observations
- Works for conditioning on multiple variables: $\Pr(\alpha \mid \beta, e) = \frac{\Pr(\beta \mid \alpha, e)\Pr(\alpha \mid e)}{\Pr(\beta \mid e)}$

Application (*Spam Filtering*): Given words $W_1, \ldots, W_n$ in an email, want to determine if the email is spam

⇒ Reframe as a probability: $\Pr(Spam \mid W_1, \ldots, W_n) = \alpha \Pr(W_1, \ldots, W_n \mid Spam)\Pr(Spam)$

To solve, use ***Naive Bayes*** - making a simplifying assumption: $\Pr(W_1, \ldots, W_n \mid Spam) = \prod_i \Pr(W_i \mid Spam)$ [i.e. words are independent of each other, given spam]

- Spam filtering used as an early application of ML for spam filtering in 90s

10.2 Bayesian Networks

Bayesian networks represent individual variables as nodes in a digraph.

- If a variable X is dependent on Y, represent dependence relationship graphically as a directed edge from Y [parent] to X [child] in the network
  - Assumption: No cycles
- For each child variable, write out its distributions conditioned on all possible sets of values of its parents

Within a Bayesian network with variables $A, B, \ldots, N$: can write $\Pr(A, B, \ldots, N)$ as product of the conditional probabilities of each variable conditioned only on its parents

- Represents an application of the Chain Rule + additional assumptions about the dependence/conditional dependence relationships within he network
- Similar principle to transformer models

Given a single set of variables [with independence and dependence relationships], can draw more than one potential Bayesian network

- Edges are drawn based on conditional independence; resulting Bayesian network depends on "order" in which nodes are made
- Networks with less edges are generally better (involve fewer parameters); want to find the orders resulting in less edges
  - Is generally best to use order following direction of causality

10.2.1 Topological Semantics of Bayesian Networks

For any node X, associate three groups of nodes:

1. ***Parents*** - variables on which X is directly dependent
2. ***Children*** - variables directly dependent on X
3. ***Non-descendants*** - variables that are neither parents nor children of X

A Bayesian network encodes certain pieces of information, called its ***topological semantics***. Namely, for any node X in a Bayesian network:

1. ***Markovian assumption***: Given the parents of X, then X is [conditionally] independent of all of its non-descendants.

2. ***Markov blanket***: Given (i) the parents of X, (ii) the children of X, and (iii) all parents of children of X, then X is [conditionally] independent of all other nodes in the network.

A ***Markov chain*** is a Bayesian network with the structure of a straight line (i.e. all variables are at most dependent on, and depended on by, one variable in each case).

- Can also find Markov chains as subgraphs of a larger Bayesian network
  - *Hidden Markov models* are Bayesian networks containing a Markov chain that is not directly knowable (can only be inferred from children of its variables).

(∗) *Ex.* Markov chains commonly used in computational biology to model evolution of DNA sequences (and find matches between sequences)

10.2.2 Probabilities in Bayesian Networks

Computing probabilities: Given knowledge/observations e, can compute probability of some event X as $\Pr(X \mid e) = \alpha \Pr(X, e) = \sum_y \Pr(X, e, y)$ (with sum taken over all possible values for all variables not specified in X, e)

- *Ex*: Variables A, B, C, D [all True/False]; observation $A = a$, want probability $B = b$
  $\implies \Pr(B = b \mid A = a) = \sum_{c \in \{T,F\}} \sum_{d \in \{T,F\}} \Pr(a, b, c, d)$

Optimization #1 - Factor out constant terms:

$$\alpha \sum_X \sum_Y \Pr(A)\Pr(B)\Pr(X)\Pr(Y) = \alpha \Pr(A)\Pr(B) \sum_X \Pr(X) \sum_Y \Pr(Y)$$

Observation: If Y is independent of X (in example above), then we only need to calculate the innermost sum once

Optimization #2 - Use dynamic programming to compute the sums in reverse [from innermost to outermost sum - *variable elimination*]

Ex - Variable Elimination:

Given an outer loop over e and an inner loop over a containing a term $\Pr(m \mid a)$, can create a table [***factor***] $\rho_m(a)$ storing the value of $\Pr(m \mid a = k)$ for all legal values k of a

- If the probability term depends on multiple variables [b, c, d, e.g.], can create a multidimensional table over all variables in quesiton [$\rho_m(b, c, d)$, e.g.]

If multiple conditional probabilities $\Pr(j \mid a)$, $\Pr(m \mid a)$: store a table $\rho_3(a)$ with values $\rho_3(a) = \rho_1(a) \cdot \rho_2(a)$ [ρ_1, ρ_2 from j, m]

Factor multiplication - To "multiply" two tables $\rho_1(a, b), \rho_2(b, c), \rho_3(b, c)$, can create a new table $\rho_4(a, b, c)$ with values $\rho_4(a, b, c) = \rho_1(a, b) \cdot \rho_2(b, c) \cdot \rho_3(b, c)$

- Keep building larger & larger tables until innermost sum has only a single table/factor

Summation: Upon reaching a sum, to sum a factor $\rho_4(a, b, c)$ over a, create new table $\rho_5(b, c)$ with values $\rho_5(b, c) = \sum_a \rho_4(a, b, c)$

- Entire process repeats/recurses to eventually obtain a single probability; avoids having to sum over any variable more than once

Time Complexity [*Table size*]: Exponential in table width

- Similar to constraint satisfaction
- The closer network is to a tree (i.e. less dense), the better
  - Tree-like networks: Markov chain/hidden Markov models

Variable Elimination:

- What kinds of probabilities are being considered (conditional probabilities, marginal probabilities, etc.) is not important; can treat everything as just "some numbers"
- Using dynamic programming to compute the sum of products is common in computer science; also seen in databases, e.g.
- *Challenge*: Finding the ideal order to sum the variables within probability expression is difficult - NP-hard
  - Can use various heuristics to guess/estimate

11 Machine Learning

Can divide inference/reasoning into two types: ***deduction*** and ***induction***

- ***Deduction*** (*General* $\rightarrow$ *specific*): Given general information [KB], asks specific questions: does $KB \models \alpha$ [$KB \models d_5$]? given ρ, what is $\rho(x)$?
  - Can be proven via precise mathematical rules
  - Ex: Propositional logic
- ***Induction*** (*Specific* $\rightarrow$ *general*): Given data points d_i, tries to create general $\hat{\rho}$ [generalization]
  - ***Machine learning*** is a form of inductive inference; attempts to infer a knowledge base from data.
  - Less philosophically straightforward than deduction

Principle (***Machine Learning***): Nature [the world] is an *unknown probability distribution* $p_r(x)$ over many variables x [$x_1, x_2, \ldots$]; given data points d_i [$d_1, d_2, \ldots$] sampled from the world, each with values for variables x_i, want to construct approximation $\hat{p}_r(x)$ of $p_r(x)$.

11.1 Machine Learning Overview

Types of Machine Learning:

- ***Unsupervised learning***: given points d_i from $\rho(x)$ (x variables in the world), try to recreate $\hat{\rho}(x)$ without any other specific objective
  - (∗) *Ex*: ChatGPT tries to model human communication, generative models try to recreate nature; Bayesian networks, RNNs, transformers
- ***Supervised learning***: given points $x_1y_1, x_2y_2, \ldots$ from $\rho(xy)$ (x variables in the world; y some target or label), try to infer $\rho(\hat{y \mid x})$
  - *Classification*: y is discrete (e.g. true/false, distinct categories)
    - *Ex*: Naive Bayes, logistic regression
  - *Regression*: y is continuous
    - *Ex*: linear regression
  - (∗) Additional categories:
    - *Reinforcement learning*: given an agent that receives rewards based on its actions, want to learn behavior in the environment

    - *Inverse reinforcement learning*: While observing people behave optimally in an unknown situation, try to infer what the goals/rules of the situation are.

Evaluating/comparing two inferred distributions $\hat{p}_{r_1}(x), \hat{p}_{r_2}(x)$:

- One method is to use *likelihoods*: compare the probability that we would obtain our observations $d_1, \ldots, d_n$ if $p_r(x) = \hat{p}_{r_1}(x)$ vs. if $p_r(x) = \hat{p}_{r_2}(x)$
  - Mathematically: compute $\arg\max_i \left[\hat{p}_{r_i}(d_1, \ldots, d_n) = \prod_j \hat{p}_{r_i}(d_j)\right]$
  - Implicit assumption: data points are i.i.d. (*independent and identically distributed*)
- Binary classification (supervised learning): compare the rates of true/false positives and true/false negatives to create a score quantifying how well classification was done
  - How to determine score depends on relative importance/priority of true/false positives, negatives for the specific application
  - For a simple metric, can use accuracy $= \frac{TP+TN}{total=TP+TN+FP+FN}$

Issue (***Overfitting***): A given set of data points may accomplish good scores in training, but fail to generalize well to other datasets

Solutions:

1. More data
2. Limit model complexity (via *regularization*, e.g.)
3. Evaluate the model on validation/test data separate from the training data

$(*)$ [*Statistical*] *learning theory* tries to prove that algorithm output will be similar to nature

Want a model to be complicated enough to represent nature, but not so complicated as to overfit - need to find a tradeoff

- *Intuition*: Given a ***hypothesis space*** H [set of all learnable models] and data about a desired function $f(x)$, want to find $h(x) \approx f(x)$
  - Hypothesis space is a subset of all possible functions
  - *Bias-variance tradeoff*: If $|H|$ is large [large hypothesis space - many possible functions] may require a lot of data to find h
    If $|H|$ is small, it may be difficult to find a function matching f

(∗) Contemporary machine learning: for unknown reasons, neural networks seem to avoid/minimize bias-variance tradeoff

- Validation error drops, but begins rising again at certain scale

Ex: Want to train a naive Bayesian network classifier with input variables $X_1, \ldots, X_n$, output label Y [True/False] based on data

1. Start with a parent node Y, children $X_1, \ldots, X_n$
   - Using naive Bayes assumption: $\Pr(Y, X_1, \ldots, X_n) = \Pr(Y) \prod_i \Pr(X_i \,|\, Y)$
2. Assigning variables:
   (a) Assuming $\Pr(Y)$ is unknown, can estimate based on examples in data
   (b) Similarly, estimate $\Pr(X_i \,|\, Y = k)$ based on data points where $Y = k$

(∗) *Issue*: If an event $\{X_i = k\}$ is not represented in data points, the network may set both $\Pr(X_i = k \,|\, Y = True), \Pr(X_i = k \,|\, Y = False) = 0$; sends entire product $\Pr(Y, X_1, \ldots, X_n)$ to 0

Similarly: if the data contains one instance of $X_i = k$ and no instances of $X_i = \neg k$, classifier may always say $Y = false$ whenever $X_i = k$

$\Rightarrow$ *Solution*: Make the model "simple" enough to never say $\Pr(Y, X_1, \ldots, X_n) = 0$ for any values $Y, X_1, \ldots, X_n$ [regularization]

- *Method* (*Pseudocounts*): Add a "fake" instance of $X_i = k$ with $Y = True$ and another fake instance of $X_i = k$ with $Y = False$
  $\implies$ Rather than inferring $\Pr(X_i = k \,|\, Y = True) = \frac{0}{n}$, will instead infer $\Pr(X_i = k \,|\, Y = True) = \frac{0+1}{n+2} \neq 0$ [e.g.]

11.1.1 Linear & Logistic Regression

Linear Regression: Given data points $\{(x_i, y_i\}$, want to find a linear function $h_w(x) = w_0 + w_1 x_1 + \ldots$ fitting the data [want to find coefficients $w_0, w_1, \ldots$]

- Estimates a numerical value y based on inputs $x_1, \ldots$
- Fit quality measured via ***loss function*** [ex: $L(w) = \sum_i (h_w(x_i) - y_i)^2$]; can progressively minimize loss via gradient descent
- To avoid overfitting, use ***regularization*** to make weights "prefer" to be near 0
  - Done via adding an additional loss term

Problem (***Classification***): Rather than a numerical value/estimate, output a label

- Perceptrons - early linear classifiers; output binary 0/1 classifications
  - Issue: Difficult to optimize via gradient descent
- Logistic regression - outputs continuous probabilities

Logistic Regression pushes linear prediction [from regression] through a ***sigmoid activation function*** to obtain a probability

- Sigmoid activation: using linear predictor $g_w(x)$, output $h_w(x) = \frac{1}{1+\exp(-g_w(x))}$
  - Turns numbers into probabilities, representing confidences in predicted labels
- Similar to linear regression: use gradient descent for training, regularization for overfitting
  - Loss function - use cross-entropy loss, e.g.

(∗) Nested logistic regression → neural networks/deep learning (informally)

- Reductively: each neuron is an individual logistic regression classifier that sends its outputs to other logistic regressions [other neurons]
  - Each neuron, given a number of weights and inputs, computes input function & pushes through activation f'n (sigmoid, e.g.) to create output prediction
  - Output prediction used as input for next layer of logistic regressions; neural networks use many, many logistic regressions

11.2 Decision Trees

Decision trees represent a "true function" (i.e. a function outputting true/false) as a tree with True/False leaves; data parameters determine the path taken from the root to some leaf

- Given a set of data points, attempts to construct decision tree that best matches data
- Size of hypothesis space (number of possible learnable trees): $\underline{2^{2^n}}$, given n binary variables
  - 2^n possible sets of assignments [rows] $\times$ 2 ways to assign a value T/F to each row
  - Can reduce space size with simpler variables
    - Ex: variables are conjunctions → 3^n possible variables (each variable is either in a clause as positive, in as negative, or not in)

- Generally use greedy to find a good decision tree

Can construct a tree via ***top-down induction***:

1. Pick a variable to split on [i.e. be the root of the decision tree]
2. On that variable:
    - If there are both + and − examples, split into + and − branches and recurse [i.e. learn a new decision tree] on both branches
        (∗) If there are no more attributes/features left to split on, take a majority vote
    - Eventually: if all data is either only + or only −, predict +/− accordingly [leaf]
        (∗) If there are no examples left, return a default value (e.g. overall majority class)

(∗) ***Tree pruning***: If few there are only a few data points left, simply take majority vote

- Justification: attempting to construct a tree on only a few points would likely overfit
- Post-pruning: once tree is constructed, delete nodes with uncertain value

Decisions trees are very flexible models; generally do not need as much data as neural networks

- Are well-suited to non-image/text data (tabular data, e.g.)
- ***Random forests*** consist of an ensemble of many individual decision trees based on slightly different subsets of data

Q: How to choose which variable to split on?

- Generally: variables with more conclusive splits (i.e. separate positive/negative points more strongly) are better, allow for making predictions earlier

Usefulness of information measured using ***Shannon entropy***:

$$H(< p_1, \ldots, p_n >) = \sum_i -p_i \log p_i$$

($\log_2$, dealing with bits 0/1)

Shannon Entropy:

- Represents how much new information is gained when a question is answered, based on the prior belief of probability of each of possible answers

- Higher entropy (measured in bits) corresponds to more uncertainty between answers.

  Ex. For a given true/false question [two options: p_1, p_2]:

  - Entropy $H(< p_1, p_2 >)$ *maximized* at 1 bit (both answers have probability 0.5)
    - Represents a state of maximal uncertainty
  - Entropy $H(< p_1, p_2 >)$ *minimized* at 0 bit (one answer has probability 1)

  $(*)$ Multiple-choice: Measure bits by modeling as a sequence of true/false questions
  - *Ex*: Four choices, maximally uncertain $\rightarrow$ two 50/50 questions, 1+1=2 bits

For decision trees, want splits resulting in subtrees with **minimum entropy** [least uncertainty]

- Compute as the expected entropy of splitting on any particular variable: weighted average of entropy of each child, based on probability of visiting that child
- Compare splits by looking at the difference in entropy before/after a split [representing information gained by making the split]

Intuition: Given a true/false question with frequency f^* of positive answers (p positive answers, n negative answers): have inital entropy $h^* = H(< \frac{p}{f^*(p+n)}, \frac{n}{(1-f^*)(p+n)} >)$

- Given graph of entropy H (0 to 1) vs probability p_1 (0 to 1), h^* is the point (f^*, h^*) located on the curve H
- Given a child node with p_i positive, n_i negative: compute h_i via similar formula, corresponding to point f_i, h_i on the curve
- Expected entropy: given children with h_1, h_2 and probability q_1 of taking h_1: f^* is $1 - q_1$ away from f_1, q_1 away from f_2.

  Drawing a line between (f_1, h_1) and (f_2, h_2), the *expected entropy* after splitting into two branches h_1, h_2 is point on that line corresponding to $x = f^*$

  - Information gain is y-distance between (f^*, h^*) and aforementioned point
  - In some cases, may have $h_1 > h^*$ or $h_2 > h^*$; however, expected entropy will always be $\leq h^*$ [splitting will always result in an information gain]

12 First-Order Logic

Why logic? Used in many instances

- Commonly: have a specification α and implementation β, want to show that β satisfies α
- Reasoning/learning

$\Rightarrow$ Why **first-order logic**?

(+) More expressive/succinct than propositional logic

- Relational databases as applications of first-order logic
  - SQL queries as checking sentences within world represented by database

(+) Allows for more efficient reasoning

(-) Can result in more complicated reasoning

12.1 First-Order Logic

First-Order Logic

First-order/predicate logic describes the world in terms of:

1. **Objects**: Discrete objects that exist in the world
2. **Relations** between & **properties** of objects
3. **Functions**: maps some number of objects to another object

First-order/predicate logic:

- A relation can be between many objects
- Can view a property as a relation between an object and itself

Can make different choices in terms of determining how to represent world mathematically

- Ex: is every natural number its own object?
  - Can have objects beyond those specified in the world

12.1.1 Syntax of First-Order Logic

Within first-order logic, have:

- *Constants* - strings of characters (e.g. King John, UCLA)
  - Represent names of objects
- *Predicates* - names for relations/properties (e.g. BrotherOf, ¿)
- *Functions* - map an object/objects to another object
- *Connectives* - $\vee, \wedge, \neg, \implies$, etc.
- *Equality* - $=$
- *Logical variables* - x, y, z
- *Quantifiers* - $\forall, \exists$, e.g.
  - Allow for making general statements about many objects at once

Semantics of first-order logic divided into *terms*, *predicates*, and *sentences*:

- ***Terms*** are pieces of syntax that refer to objects in the real world:
  - Constants and logical variables refer to/name objects
  - Function outputs (given some input) also refer to objects
- ***Predicates*** refer to relations between objects
- ***Sentences*** are True/False statements within a world, similar to propositional logic
  - *Atomic* sentences are predicates applied to terms (e.g. BrotherOf(KingJohn, Richard); are either True or False
    - Syntax: Predicate(Term1, Term2, ...)
      - Terms may be either constants or logical variables
    - Represent the simplest form of sentence
  - Can use connectives and quantified logical variables build more complex sentences from atomic ones
    - *Ex*: Sentence1 $\wedge$ Sentence2 $\implies$ Sentence3

12.2 Interpretation of First-Order Logic

Process of describing worlds in first-order logic divided into two phases: (i) ***pre-interpretation*** and (ii) ***interpretation***

1. Pre-interpretation starts by fixing ***domain of discourse*** - set of objects that [are assumed to] exist
2. From domain of discourse, construct ***interpretation*** of:
   (a) Constants: are mapped to real-world objects
   (b) Functions: are made as mappings of [existing] real-world objects to other [existing] real-world objects
      - The input of functions are real-world objects, not the constants mapped to them

      Ex: A function may take the person corresponding to "KingJohn", but not the string literal "KingJohn'
   (c) Predicates: map a number of real-world objects to either True or False
      - A True/False assignment corresponds to whether that relation is True between/-for that given set of objects

      (∗) Equality is a special predicate - already has predefined value
   (d) Statements like ∀ apply to all objects in DoD

Quantifiers make first-order logic powerful

(∗) Using $\implies$: $\exists\ z : X(z) \implies Y(z)$ is true either if there is z with X and Y true, or z' with X and Y both false

12.3 (∗) Reasoning in First-Order Logic

Similar to propositional logic, can perform reasoning with first-order logic

Rounding: To convert first-order logic to propositional logic, can replace ($\forall\ x\ [Y(x)]$ or $\exists\ x\ [Y(x)]$) statements with conjunctions/disjunctions of $Y(x)$ statement across all x

- Can use SAT solvers to perform inference on converted propsotional logic

Issue: The first-order logic sentences may generate an infinite propositional logic base

- *Theorem* (Herbrand): If a sentence is entailed by a first-order KB, then it is entailed by a finite subset of the corresponding propositional KB.

- Can apply search techniques (e.g. iterative deepening) to finite subsets of converted propositional KB

Issue/Theorem (Church/Turing): Entailment in first-order logic is only semi-decidable, i.e. first-order logic is not guaranteed to allow for inference

- Even if a sentence is entailed, may still end up looping forever

(∗) Related to Godel's Incompleteness Theorems: there will always exist something that is true (entailed by the knowledge base) but cannot be proved

(∗) Godel's Completeness Theorem: Everything can be proved, but may require using higher-order logics to do so