COMP 424 Crib Sheet
by Julian Lore Side 1 of 2

1 Uninformed Search

Search Problem State space S: all possible configs of domain, **initial state** $s_0 \in S$ (start state), **goal states** $G \subset S$ (end states), **Operators** A (actions avail), **Path**, **Path cost**, c, **Solution** of search problem: path from s_0 to $s_g \in G$, **Optimal solution**: path with min \$
Eight puzzle: States (conf of puzzle), goals (target conf), ops (swap blank with adj), path cost (# moves)
Represent state space search as graph, vertices are states, edges are operators. Build **search tree** to find goal state. Search tree nodes not same as graph nodes
Data struct for search tree: **Node** (state id, parent state + op, cost of path, depth. To expand node, apply all legal ops and gen new nodes.

Generic search alg Init search tree with s_0. Loop: If no nodes can be expanded, fail. Else choose node to expand, if node has goal, return path, else expand node by applying each op and getting new states, adding to tree.

Uninformed (blind) search If state isn't goal, you don't know how close to goal it might be.

Uninformed Search algs Key props: **Completeness** (guarantee soln if it exists), **Optimality** (how good is soln), **space complexity**, **time complexity**.

Search complexity Branching factor b, **solution depth** d
All uninf search time complex $O(b^d)$, very general and very expensive, no knowledge.
Breadth-first search (complete w/ finite b, guaranteed shortest path if unit cost = optimal, but not if weighted graph). $O(b^d)$ space. **depth-first search** ($O(bm)$ space, easy to do recursively, more efficient than BFS if many goal paths, **not optimal**, may not complete (cycles), don't use DFS for big d), **uniform-cost search** (BFS but with **general** (weighted graph) step costs, use a pqueue, opt & comp), **depth-limited search** (DFS but stop at goal or max depth, always terms, but not complete), **iterative deepening search**(depth-lim search but increasing depth, expands nodes mult times, complete, linear mem req like DFS, but more time, optimal if unit cost, preferred for large state spaces)
Revisiting states: maintain **closed list** to store expanded nodes, good for probs with repeated states. $O(|S|)$ time and space. Sometimes re-expanding states could be better (compare old and new path cost, also sometimes domain may be too large to store all states).
What method to use? To find opt: BFS, IDS for unit cost, uniform-cost search if general cost. Large state space: DFS max length is known, IDS otherwise. Limited mem: DFS/IDS. Quickly find best soln with budget: Depth-limited if unit cost, UCS if gen cost.

2 Informed Search

Use **heuristics** to guide search. Uninformed expand nodes based on dist from start node. Informed expands based on **distance to goal**. If we don't know exact distance, we use intuition, **heuristic**. Heuristics come from prior knowledge of prob, exact soln to relaxed vers of prob, learning from exp
Heuristic for path planning: straight-line distance between two places.
Eight puzzle: number of misplaced tiles, total Manhattan dist

Algs **Best-First Search**(greedy, expand most promising node first, close to BFS, if heuristic is 0, then same as BFS, opp of UCS(cost-so-far) vs cost-to-go, $O(b^d)$ time/space, good heuristic can make $O(bd)$, **not always complete**(loops), but complete if finite space if we check repeated, not optimal), **Heuristic Search**(Problem: best-first too greedy, doesn't account for cost so far. Soln: **Heuristic search**, greedy wrt to $f = g + h$, where g is cost so far, h is heuristic. Use pq, add to q w/ p: $f = g + h$, end when goal popped from q. Note we continue expanding nodes after finding goal if $\exists$ unexpanded w/ lower cost than current path to goal. Not optimal, unless we put conditions on heuristics), **A* Search** (Heuristic search with AH. Complete, $f(s) = g(s) + h(s) \leq g(s) + c(s, s') + h(s') = f(s')$ by consistency, so no node can be re-expanded. If soln, c must b bounded, so $A*$ will find. Optimal, prove by contradiction, showing impossible that subopt goal is expanded before opt. Still worst case $O(b^d)$, but $O(bd)$ with perfect heuristic because only expand nodes on opt path. With given h, no other search alg can expand less nodes), **Iterative Deepening A***(DFS, but use f to determine order to explore children instead of depth. Same props as $A*$, but less mem. If we remember expansion of old nodes $\rightarrow$ SMA*)

Admissible heuristics $h^*(n)$ is shortest p from n to any goal. h is **admissible heuristic** if $h(n) \leq h^*(n) \forall n$. They are optimistic. Trivial ah: $h(n) = 0 \forall n$, get UCS. Obviously $h(g) = 0 \forall g \in G$ for AH. Usually relaxed vers of prob gives AH.

Consistency AH h is called **consistent/monotone*** if for every state s and succ s', $h(s) \leq c(s, s') + h(s')$, i.e. h gets more precise as we get closer to goal. Vers of triangle ineq. Can fix inconsistent heuristics by: $f(s') = g(s') + h(s') \rightarrow f(s') = \max\{g(s') + h(s'), f(s)\}$

Dominance $h_2(n) \geq h_1(n) \forall n$ and both AH, then h_2 dominates h_1, i.e. more informative.

Decomposition Break complex prob into smaller parts. Decomp and putting soln together may give up optimality. Use decomp for probs we can't solve w/o. Subsolns can be cached & reused. Need to be careful that when we chose subgoal, overall prob still has soln.
Macro-action is sequence of actions from orig problem (e.g. make T for Rubik's)
Abstraction ignores info to speed up comp, make compact representation, map several real states to one abstract state

3 Optimization

Typically large continuous/combinatorial state space. Can't search all possible soln. Non-uniform cost.

Traveling salesman prob Vertices+dist between pairs. Get shortest path to visit each vert once. Tour = path that satisfies goal.
Optimization prob described by **states** and **evaluation function**, note that states are candidate solutions (can be partial or wrong) here, not description of world. Func corresponds to path cost.

Optimization Search **Constructive methods**, start from scratch and build up. **Iterative improvement**, start with soln, improve. Both involve local search.

Generic local search Start at init X_0, repeat until satisfied: Gen neighbors of X_i; eval them. Select one of the neighbors X_{i+1} to become current config.

Discrete Hill-Climbing Start with X_0, val $E(X_0)$. Gen neighbors of X and $E(X_i)$. Get $\max_i E(X_i)$. If this max is less than $E_{\max x}$ of init, then return. Else update X to be new X_i and E to be new $E_{\max}$. This is a variant of best-first search, easy to prog, no memory of past req, can handle large probs. Small neighborhood = less neighbors, possibly worse soln, large neighborhood = more to eval, possibly less local optima, better soln. Problem: hill climbing can easily get stuck in plateau or local opt, to fix, use random re-starts or pick any move that leads to improvement (**randomize hill climbing**).

Simulated annealing Like hill climbing, but allows bad moves to escape local opt. Decrease size+freq of bad moves over time.
Alg: Start with X_0 and $E(X_0)$. Loop until satisfied: choose random neighbor, if val is greater than E_{max}, replace current max. If greater than current val we holding, replace current X. Else, with prob p, still replace cur val. Return max at end.
What to use for p? Constant, val that decays to 0, val that depends on how bad move is. We usually use Boltzmann distribution $\mathbf{p = e^{-(E - E_i)/T}}$, bad $E_i \rightarrow$small p. T here is called **temperature**, usually start high then decrease to 0 over time. Can decrease T by mult by constant $0 < \alpha < 1$ at every iter. If T is high $\rightarrow$ alg is in exploratory phase, if low, exploitation phase.
If T decreases "slowly enough", optimal, but may take ∞ moves. SA better than HC when lots of local opt. HC preferred if func is smooth, not many local opt, most local opt are similar.

Parallel search Run mult separate searches (HC or SA) in parallel, keep best soln

Local beam search Like parallel search, but share info across searches. Start k searches in parallel, but keep k (**beam width**) top solns at each step.

Genetic algs **Individual**=candidate soln. Each indiv has **fitness** (quality of soln). **Population** = set of indivs. Pops change over **generations** by applying **operations**(mutation(inject random change with mutation rate = prob of mutation occur)/crossover(combine parts of indivs to make new indiv, use **crossover mask** to specify which parts taken from 1 indiv as bin string, rest taken from other)/selection) to indivs. Higher fitness = more likely to survive & reproduce. Usually represent indivs by **binary string**.
Alg: (params: fitness,threshold,p,r,m) init P with p rand indivs. Eval, get Fitness(h) $\forall h \in P$. While $\max_h Fitness(h) < threshold$: Select $(1-r)p$ members of P to put in P_s. Crossover $\frac{rp}{2}$ pairs of indivs. For each pair, produce two offspring and incl in P_s. Mutate 1 rand bit in mp rand indivs of P_s. Update $P \leftarrow P_s$. Eval fitness $\forall h \in P$.

Selection Survival of fittest: **Fitness proportionate**, might lead to crowding (mult copies of same soln), **tournament selection**, pick 2 rand indivs, with prob p select fitter one. **Rank selection**, sort all by fitness, prob of selection proportional to rank. **Softmax (Boltzman) selection** $P(i) = \frac{e^{Fitness(i)/T}}{\sum_{j=1}^{P} e^{Fitness(j)/T}}$.
Elitism, best soln can die during evol, so we preserve best soln encountered. Genetic algs more expensive than HC & SA.

Pros cons of gen alg Pro: Intuitive due to analogy, can be effective if tuned properly. Bad: Perform dependent on encoding of problem. Many params to tweak. Low mutation rate = overcrowding. Too high = too random.

4 Constraint Satisfaction Problems

Use constraints to narrow search space. Def: **variables** V_i that can take vals from domain D_i. **Constraints** specifying allowed combinations of values for variables. Constraints can be represented as a function or list of allowable vals. **CSP solution** is assignment of vals to vars st all constraints true. Usually want to find any soln or find that no soln exists.

Approaches **Constructive** approach, state = vals assigned so far. Use forward search to fill soln. Gen purpose, works for all CSPs. **Random approach**, start with broken complete assn of vals to vars. Fix broken constrs by re-assign vars. Use optimization.

Problem def **State** (vals assigned so far, can be partial/inconsistent). **Initial state** (all vars unassgn). **Operators** (assign val to unass var). **Goal test** = all vars assigned, no constraint false, complete and consistent assignment. Problem is deterministic. Note that depth is limited to # of vars, can use DFS or depth-limited search.
Uninformed search for map coloring: choose unassigned var, assign a val. This is complete and optimal, but complexity is worst possible, $n!d^n$, (n vars, d vals). Branching factor very high. Var assgnment order irrelevant, many paths equiv.

Constraint graph Nodes are vars, arcs show constraints. Can use graph struct to accelerate search. Use **inference** to reduce search space. pre process graph to remove inconsistencies. Var is **arc-consistent** if all val in domain satisfies vars binary constraints. Network is **generalized arc-consistent** if all vals in domain of all vars are all arc-cons. Keep applying arc-consistency until no changes to get generalized arc-cons.

Ex Map coloring: vars = countries, domains = r,g,b, constraints = adj countries cannot be same color, $C_1 \neq C_2 \ldots$
4 queens: 1 queen/col, vars = $Q_1, \ldots$, row of each queen. Domain = $1, 2, 3, 4$. Constraints: $Q_i \neq Q_j$ (not same row), $|Q_i - Q_j| \neq |i - j|$ (not same diag)

Backtracking search DFS but fix order of var assn $b = |D_i|$. If no assignment for specific var, backtrack to prev var and try diff val. Basic uninformed alg.

Forward checking Keep track of legal vals for unassigned vars. When you assign vars, look at unassigned vars connected via constraint and delete from their domain any inconsistent vals to new assgn.

Heuristics for CSP For selecting vars: **minimum-remaining vals** = choose var that is most constrained more info if one branch is not satisfiable (1/2 of branches bad vs 1/100 branches bad). **Degree heuristic** = choose var that imposes most constraints on remaining vars, can use to break ties from min-remain vals heuristic. To select a val: **least-constraining val**: assign a val that rules out fewest vals for other vars (less chance of conflict in future).
Worst-case d^n. Tree structured constraint graph gives $O(nd^2)$. Nearly-tree structured: $O(d^c(n-c)d^2)$ using **cutset conditioning**, find vars st removing them turns graph into tree. Instantiate them all possible ways, c is size of cutset.

Local search **Iterative improvement alg** Start with broken but complete assnment of vals & vars. Allow var assgns that don't satisfy some constraints. Randomly select conflicted vars. Ops reassign var vals. **min-conflict heuristic** chooses val that violates fewest constraints. This is hill climbing.

5 Uncertainty

Actions may be **non-deterministic**. Problems can be **(fully) observable**, **partially observable** or **non-observable**.

Searching under uncert Cannot determine future states in advance (i.e. depend on die roll). Soln is not path, but **contingency plan/strategy**.
Vacumm ex. Two rooms, vacuum in one of the rooms, rooms can be dirty or clean. When non-observable, need plan. What states possible after doing an action? Reason over **beliefs** (sets of states). Total # possible beliefs = power set of all states w/o empty. Less reachable beliefs though.

Conformant planning Find plan that leads to goal despite state uncertainty. Good heur, use actions that reduce uncertainty, reduce belief to 1 state, then do standard search.
Non-deterministic case: vacuum may sometimes deposit dirt instead of cleaning, sweep man sometimes clean adjacent. Make **AND-OR** search tree: **OR nodes** (agent chooses between actions), **AND nodes** (choice induced by env choice of outcome, non-det). Want subtree st all leaves are goal leaves. Soln is subtree that specifies one act at each OR node, inclds every outcome at each AND node, has goal node at ea leaf.
Slippery vacuum, moving sometimes fails. Apply **cyclic** soln, keep trying until it works. Can be ok soln if caused by random event but not if caused by unobserved event, like broken vacuum, can't move.

Partial observability Can only sense things locally, i.e. if current room is dirty. Account for possible observations that tell us about next state. Search over belief states. Problems number of reachable beliefs can be v large (use sampling or pruning). Number of states in each belief can b v large (use compact state rep, plan for each state sep)

6 Game Playing

We have **perfect** vs **imperfect**(hidden info) info, and **deterministic** vs **stochastic** (chance) games.

Game playing as search 2-player, perfect, determ games. State: state of board/player turn, opsLlegal moves, goal: states st W/L/D, cost: basic $(+1, 0, -1 \rightarrow$W/D/L), complex (points won, money, ...).

Want strat(way of picking moves) to max utility(prob of win/min cost)

We assume adversary is trying to minimize & playing optimally, Bad assumption.
Define **max player** (wants to max util) & **min player** (to min util).

Minimax search Expand complete search tree until terminal states have been reached, compute util. Go back up from leaves towards cur state. At **min nodes**, backup worst val of children, at **max nodes**, backup best val, where min/max nodes correspond to min/max players. Complete (if tree finite), optimal if advers playing opt, $O(b^m)$ time, $O(bm)$ space if DFS. Issues v expensive even with pruning. Requires reasonable eval func. Assumes both players playing opt wrt same eval func. What if non-determinism in game or don't know game well enough to make good eval func? → **random sims**.

Resource limitations Might be time restricted, can't search all nodes. Can use cutoff test (based on depth) & eval func ($v(s)$ represents "goodness" of board state, chance of winning at that pos. If features of board can be evaluated indep, use weighted linear func) for nodes @ cutoff. **Real-time search**. Eval func for chess can be # white queens − # black queens + # white pawns − # black pawns .... Move chosen should be same if we apply monotonic trans to eval func. Minimax cutoff: stop at some max depth, use eval func.

α-β pruning If path looks worse than what we have, discard. If best move at node cannot change, don't search further. Minimax but keeps track of best leaf val for player (α) and opponent (β), gives bounds $[\alpha,\beta] = [-\infty,\infty]$ at start. Update α at max and β at min. Pass vals up to parents as min/max, parents copy their bounds to children. Pruning can greatly incr eff. Pruning does not affect final result, best moves are same as mimimax, assuming opponent is optimal and eval func is good. With bad ordering, $O(b^m)$, nothing pruned. Perfect ordering: $O(b^{\frac{m}{2}})$. Usually $O(b^{\frac{3m}{4}})$. Cons, big b means depth is still too limited. Optimal only if opp is optimal. If using heuristics, opponent needs to use same heuristic.

Forward pruning (for domains with large b) only explore n best moves for our state. May lead to sub-optimal soln. Can b v eff.

Strats Make compact state rep. Using IDS for real-time. Use α-β pruning w/ eval func. Searching deeper usually more important than having good eval func. Consider diff strats for begin, mid, end. Use rand to break ties. Consider non opt opp.

Random Simulations Sim games by rand selecting moves for both players. At end, check if won or lost, keep track of initial move. After lots of sims, pick move w/ highest win rate. Using rand to gen sample for estimation called **Monte Carlo method** (also seen during sim anneal). Spend more search effort at **promising moves**. Can use minimax style search for few moves at top.

Monte Carlo Tree Search Search tree + Monte Carlo sims. Select promising node in search tree using **tree policy** (mapping from states to acts). Sample possible continuations from leaves using rand **default policy** for both players (usually @ end of gm). Val of move = avg of evals from sampled lines. Pick move with best avg/expected val.

Alg: Init search tree with curr state of gm. Repeat until no more comp budget: **Descent**: Choose + expand node in curr tree, use minimax or which one seems more promising. **Rollout**: when @ leaf, use MC sim to end of game/affordable d. **Update**: update stats for all nodes visited during descent by backpropagating. **Growth**: First state in rollout added to tree and stats initialized. Advantages vs α-β Not as pessimistic, converges to minimax soln in limit. Perf increases w/ # lines of play. Unaffected by b. Easy to parallelize. Disadvantages May miss opt play, policy is very important.

Tree Policy How to select next move in search tree? Balance **exploitation** (node that seems promising acc to estimates & sims) & **exploration** (node hasn't received many sims, so want **more info**). Def: $Q(s,a)$: val of taking act a from state s. Win rate of node based on sims so far. $n(s,a)$ # tries taken act a from state s. $n(s)$ # times visited s.

Upper Confidence Trees $Q^{\oplus}(s,a) = Q(s,a)+c\sqrt{\frac{\log n(s)}{n(s,a)}}$, c is scaling constant. 1[st] term is upper bound on val of taking a in s. 1[st] term after eq is exploitation, last term is exploration (gets smaller the more you select it). To decide which action to take, calc upper bd of all children, select min/max.

Rapid Action-Value Estimate Assume val of move is same no matter when played. Introduces bias, but reduces variability in MC estimates. Since only the move itself matters, state space is simplified, requires less sims (don't need many sims for each indiv pos), but might oversimplify board → bad estimates. Trade-off between **model complexity** and **representational power**.

7 Logic

Need a notion of knowledge, how to represent and reason.

Knowledge representation Perception what is my state? **Cognition** what action should I take? State recognition requires some form of representation. Choosing right action implies some sort of **inference**.

Declarative problem solving Agent has **knowledge base** (facts in some standard lang, domain specific) and **inference engine** (with rules for deducing new facts & concl, domain independent)

Logics = **formal langs** for representing info st concl can be drawn. Defined by **syntax** which defines valid **sentences** and **semantics**, giving meaning to sentences.

Propositional logic Propositions = assertions about state of world/game/prob, can be true or false. Can combine with **logical connectives.Interpretation** specifies T/F for each prop sym. **Model** of set of clauses = interpretation st each clause is T. Sentence is **valid** if T in all interps (tautology); **satisfiable** if T in ≥ 1 interp ; **unsatisfiable** if F in all interps. Truth of sentence depends on interp.

KB **entails** $\alpha \iff \alpha$ is T in all worlds where KB is T. Check validity via **inference**: $KB \models \alpha \iff (KB \implies \alpha)$ is valid. Check satisfiability via inference: $KB \models \alpha \iff (KB \wedge \neg\alpha)$ unsat, proof by contra.

KB $\vdash_i \alpha \implies \alpha$ can be derived from KB by inf proced i. Want i to be **sound** (KB $\vdash_i \alpha \implies$ KB $\models \alpha$) and **complete** (KB $\models \alpha \implies$ KB $\vdash_i \alpha$)

Inference methods **Model checking**: use truth table, KB $\models \alpha$ if KB = T $\implies \alpha = T$. Sound & complete, but inefficient, needs 2^n models for n literals.

Appl of inf rules Sound gen of new sentences from old. **Proof** = seq of inf rule appl. Can use inf rules as ops in search alg. Complexity of verifying validity of sent w/ n lits = 2^n. If we only use Horn clauses we can get **poly time** infer.

Normal/Standardized forms **Conjunctive Normal Form (CNF)**: conjunctions ($\wedge$) of disjunctions ($\vee$) of literals $(A \vee \neg B) \wedge (B \vee \neg C \vee \neg D)$ **Disjunctive Normal Form (DNF)**: opp of CNF, $(A \wedge B) \vee (A \wedge \neg C)$ **Horn Form**: Conjunction of Horn clauses (clauses w/ ≤ 1 +ve lit) $(A \vee \neg B) \wedge (B \vee \neg C \neg D)$, often written as $B \implies A$, $(C \wedge D) \implies B$.

Inf rules **CNF resolution** $\frac{(\alpha\vee\beta),(\neg\beta\vee\gamma)}{(\alpha\vee\gamma)}$ **Horn Modus Ponens** $\frac{\alpha_1,\ldots,\alpha_n,(\alpha_1\wedge\ldots\wedge\alpha_n \implies \beta)}{\beta}$.**And-elim** $\frac{\alpha_1\wedge\ldots\wedge\alpha_n}{\alpha_1,\ldots,\alpha_n}$. **Impl elim** $\frac{\alpha \implies \beta}{\neg\alpha\vee\beta}$. **De Morgan's law** $\neg(\alpha \vee \beta) \iff (\neg\alpha) \wedge (\neg\beta), \neg(\alpha \wedge \beta) \iff (\neg\alpha) \vee (\neg\beta)$ Can use rules with forward or backward search.

Forward chaining When new sent p added to KB, look for all sent that share lits with p, perform resolution and add new sent to KB and cont. data-driven, eager method, new facts inferred ASAP.
extends KB, improves understanding of world, used when focus is finding model of world.

Backward chaining When query q asked of KB: if $q \in$KB → T. Else use resolution for q with other sent in KB and cont. goal-driven, lazy reasoning method, facts only inferred as needed.
Frugal in terms of comp, KB grows less, focus on proof (usually more eff), does nothing until asked questions, used in proofs by contra.

Prop log is good because very simple, few rules. But bad because cannot express in compact way. Want to describe world in more compact and eff way and want to quantify over objs.

First-order logic Adds new elements: **predicates** to describe objects/props/relations, **quantifiers** $\forall,\exists$, **functions** to give you obj related to another obj, domain elements to domain elements, like $RightOf$ (includes **constants**). Can handle **infinite domains** w/ quantifiers.

Types of sentences **Term** (const, var, func), **atomic sentences** (predicates, equality of terms), **complex sentences** (combine atomic sentences w/ connectives)

$\forall x \forall y = \forall y \forall x, \exists x \exists y = \exists y \exists x$, but $\forall x \exists y \neq \exists y \forall x$. $\forall x A(x) = \neg\exists\neg A(x), \exists x A(x) = \neg\forall x \neg A(x)$

Truth in FOL Sentences are true wrt a **model** = $M = (D, I)$, where D is domain of obj, I is interpretations s.t. const symbols → obj, pred sym → relations of objs, func sym → func relations of objs

Inf Algs for FOL **Propositionalize** FOL → prop log, too expensive (Except in most trivial cases). **Search** (forward/backward chaining using generalize MP), w/ inf rules: **MP** $\frac{\alpha,\alpha \implies \beta}{\beta}$, $\wedge$ **intro** $\frac{\alpha \ \beta}{\alpha\wedge\beta}$, **Universal Elim** $\frac{\forall x\alpha}{\alpha x/\tau}$. Ops are inf rules, states are sentences, goal is to check states to see if they contain query.

Problem b is huge, esp for UE. Try to find substitution that makes rule match known facts. **Unification** = pattern matching to find good candidate for UE. Sub σ unifies atom sent p and q if $p\sigma = q\sigma$

Generalized Modus Ponens $\frac{p_1\sigma,\ldots,p_n\sigma,(p_1\wedge\ldots\wedge p_n \implies q)}{q\sigma}$. If we use GMP w/ KB of **Horn** clauses gives single atomic sent or clause of form (conj of atom sent) $\implies$ (atom sent). All vars assumed universally quant.

GMP is complete for KBs of universally quantified Horn clauses, but incomplete for general FOL. Entailment in FOL is **semi-decidable**, can find proof if $KB \models \alpha$, but not always if $KB \not\models \alpha$ → **Halting Problem**, don't know if proof will term.

Resolution Can **resolve** 2 clauses if they have complementary literals, one lit unifies with neg of other. Same as prop resol, except with unifications. Sound and complete inf meth for FOL. **Proof by negation**, to prove $KB \models \alpha$, prove $(KB \wedge \neg\alpha)$ unsat. Do so by expressing KB and $\neg\alpha$ are expressed in univ quant CNF. Use resolution to combine 2 clauses into 1. Continue until empty clause (**contradiction**).

Convert KB to CNF $P \implies Q \equiv \neg P \vee Q$. Move $\neg$ inwards: $\neg\forall x P \equiv \exists x \neg P$. Standardize vars apart, i.e. $\forall x \exists x \to \forall x \exists y$. Move quantifiers to left. Eliminate existential quantifiers by **skolemization** ($\exists x Rich(x) \equiv Rich(G1)$, $G1$ is a new **Skolem constant**. When $\exists$ inside $\forall$: $\forall x f(x) \implies \exists y g(y) \wedge l(x,y) \equiv \forall x f(x) \implies g(H(x)) \wedge l(x, H(x))$), $H(x)$ is a **Skolem function**. Drop universal quants, distribute over $\vee \to (P \wedge Q) \vee R \equiv (P \vee R) \wedge (Q \vee R)$

Resolution Strats **Unit res** prefer to do res if one clause is literal → shorter sentences. **Set of support** identify (hopefully small) subset of KB that every res will take clause from to resolve with another sent, add to set of support , can make inf (incomp). **Input resolution** always combine sent from query or Kb with another sent, not complete in general, doesn't use new sentences.

Pros **KB-systems** Expressible/human readable, simple inf proc, easy to change, easy to explain, machine readable, parallel.

Cons Can b hard to express, undesirable interactions among rules, non-transparent behavior, hard to debug, slow, where does KB come from?

Planning Coll of act for some task. Is a search problem, but more structured. In search, states and actions are atomic, in planning, states and goals are logical sent, actions are preconditions+outcomes.

STRIPS (Stanford Research Institute Planning System) **Domain**: typed objs as props. **States** as first-order preds over obj(repr as conj ($\wedge$) of preds), **closed-world assumption** (not stated = false, only obj in world are defined). **Operators** = **preconditions** (when can use act, rep as conj), **effects** = what happens after (rep as conj). **Ex.** S: $In(robot, room) \wedge Closed(door)$, G: $In(robot, r) \wedge In(Charger, r)$, OP: $Go(x, y)$, precond: $At(robot, x) \wedge Path(x, y)$, postcond/effects: $At(robot, y) \wedge \neg At(robot, x)$. **Action schema** defines the 3 things for each op. Effects: **Add-list** list of props that become T after act. **Delete-list** list of props that become F after
Semantically we have: If precond = F, do nothing, act cannot be applied. Else if T, delete items on delete-list, add items on add-list, order of ops important! . Can rep STRIPS state transitions as a tree. Pros restricted, inf more efficient. All ops = deletion+Addition of props of KB. Cons Assumes small # props change per act (else ops hard to def and reasoning expens). Limited lang (everything = conj, not applc to all domains).
STRIPS is sound, not complete (no backtracking), not optimal (no guarantee on shortest plan), expensive

Planning approaches **State-space planning** use search w/ states and ops, **plan-space planning** work at level of plans (won't talk)

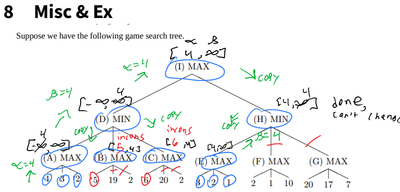
Progression (forward) planning det all ops applic from start. Ground ops, replace vars with constants. Choose op to apply, det new content of KB. Repeat until goal.

Regression (backwards) planning Pick acts that satisfy some goal props. Make new goal w/ preconds of these acts + unsolved goal props. Repeat until goal set satisfied by start.

SatPlan (satisfiability) Plan prob → gen all possible literals at all time slices. Solve humongous SAT prob. Optimal and complete, but NP-hard, PSpace if we allow plan duration to vary. **Heuristic-search planning** Don't use domain heuristics, use heuristics based on planning prob itself. Simple heur: ignore delete lists. **GraphPlan** make graph encoding constraints on possible plans. If $\exists$ valid plan, will be part of this graph, so search only this graph.

Planning Problems **Incomplete info** (unknown preds), disjunctive effects, things might cause more than we think. **Incorr info** cur state incorr, unanticipated outcomes → failure. **Qualification prob** Can never finish listing all req prec and cond outcomes of acts.
Solns? **Conditional (contingency) planning** plan w/ observation acts to get info (i.e. check tire, if intact ...), sub-plans made for each contingency. Expensive, plans for unlikely cases. **Monitoring/Replanning** Assume normal states+outcomes, check prog during exec, replay if needed.

8 Misc & Ex



Local beam search with $k = 1 \to$ hill climbing. LBS with 1 init state and return as many as possible $\to$ BFS. SA with $T = 0 \to$ HC, SA with $T = \infty \to$ rand walk.
Knapsack prob, var for each item, domain is $\{0,1\}$ (in bag or not), constraint is sum should be less than cap.
Map of diff cities, 2 friends in diff cities, each wait for other, only 1 new node at a time. Want to meet. States: pair of cities. Succ: neighbor nodes. Cost: max dist of both friends.