Artificial Intelligence Review Notes

Rui Qiu

May 31, 2018

1 Search

1.1 Intelligent Agents

Agents, percepts, agent function, agent program.

Definition 1.1. *A **rational agent** chooses whichever action maximizes the expected value of the performance measure given the percept sequence to date.*

But rational is not omniscient or clairvoyant, so might not be successful.

Definition 1.2. *4 elements of task environment (PEAS):*

- *Performance measure*
- *Environment*
- *Actuators*
- *Sensors*

Remark 1.3. *Properties of task environments*

- *fully vs partially observable: agent sensors can or cannot access all information*
- *deterministic vs stochastic: if next state determined by the current state and executed actions?*
- *known vs unknown: if the agent knows the environment's laws of physics*
- *episodic vs sequential: if the next decision independent of the previous ones?*
- *static vs dynamic: can the environment change whilst the agent is deliberating? or semi-dynamic: only the performance score changes*
- *discrete vs continuous: time, states, actions, percepts are represented in a discrete way?*
- *single vs multi-agent: single or multi compete or multi cooperate?*

Real world is partially observable, stochastic, sequential, dynamic, continuous, multi-agent.

Definition 1.4. *4 basic agents (increasing generality):*

- *simple reflex agents: decisions are made on the basis of the current state only*
- *reflex agents with state: the internal state keeps track of relevant unobservable aspects of the environment. (e.g., time passed, something might change)*
- *goal-based agents: the goal describes desirable situations; the agent combines goal and environment model to choose actions*
- *utility-based agents: the utility function internalizes the performance measure. Under uncertainty, the agent chooses actions that maximize the expected utility*

Remark 1.5. ***Learning agents:*** *action selection element, learning element (critic's feedback -¿ modify -¿ action selections), problem generator*

Summary:

- Agents interact with environments through actuators and sensors.
- The agent function describes what the agent does in all circumstances.
- Agent programs implement agent functions.
- The performance measure evaluates the environment sequence.
- A perfectly rational agent maximizes expected performance.
- PEAS descriptions define task environments.
- Environments are categorized along several dimensions: observable? deterministic? known? episodic? static? discrete? single-agent?
- Several basic agent architectures exist: reflex, reflex with state, goal-based, utility-based.
- All agents can improve their performance through learning.

1.2 Goal-Based Agents: Solving Problems by Searching

1.2.1 Problem formulation

Definition 1.6. *A* ***problem*** *is defined by four items:*

- ***initial state***
- ***successor function*** $S(x) =$ *set of action-state pairs*
- *goal test*
- *path cost (additive),* $c(x, a, y) \geq 0$ *is the step cost of going from state* x *to state* y *via action* a

A ***solution*** *is a sequence of actions leading from the initial state to a goal state.*

Abstraction should be easier than the original problem.

1.2.2 Tree search algorithm

Basic idea: offline, simulated exploration of state space by generating successors of already-explored nodes (expanding nodes).

A **state** is a physical configuration (no parents, no children, no depth or path cost). A **node** is a data structure constituting part of a search tree includes **state, parent, children, depth, path cost** $g(n)$.

Frontier implemented as priority queue of nodes ordered according to strategy.

1.2.3 Uninformed search strategies

a strategy $\rightarrow$ pick the order of node expansion

Uninformed strategies use only the information available in the definition of the problem.

Definition 1.7. *4 dimensions of strategy evaluation:*

- *completeness–if it is guaranteed to find a solution when exists?*
- *solution optimality–is it least-cost?*
- *time complexity–number of nodes generated*
- *space complexity–maximum number of nodes in memory*

Quantitatively, b = maximum branching factor, d = depth of the shallowest solution, m = maximum depth of the state space.

- Breadth-first search: expand shallowest unexpanded node
  - frontier = FIFO queue, new successors go at end.
  - Complete? Yes (if b and d are finite)
  - Time? $1 + b + b^2 + \cdots + b^d + (b^{d+1} - b) = O(b^{d+1})$, exp in d
  - Space? $O(b^{d+1})$
  - Optimal? Yes if cost=1 per step; not optimal in general
- Uniform-cost search (=BFS if cost=1 per step): expand the least-cost unexpanded node
  - frontier = queue ordered by path cost, lowest first
  - Complete? Yes if step cost $\geq \epsilon$
  - Time? number of nodes with $g \leq C^*, O(b^{1+\lceil C^*/\epsilon \rceil})$, where C^* is the cost of the optimal solution
  - Space? number of nodes with $g \leq C^*, O(b^{1+\lceil C^*/\epsilon \rceil})$
  - Optimal? Yes nodes expanded in increasing order of $g(n)$
- Depth-first search: expand deepest unexpanded node
  - frontier = LIFO queue, i.e. put successors at front
  - Complete? No, fails in infinite-depth, spaces with loops; but yes, complete in finite spaces

  - Time? $O(b^m)$ terrible if $m \gg d$ but if solutions are dense, may be faster than BFS.
  - Space? $O(bm)$, linear space (deepest node + ancestors + their siblings)
  - Optimal? No.

- Iterative deepening search = BFS + DFS advantages
  - completeness, returns shallowest solution, use linear amount of memory
  - Performs a series of depth limited depth-first searches
  - What is **Depth-limited search**? = depth-first search with depth limit l
  - cutoff: no solution within the depth limit, failure: unsolvable.
  - Complete? Yes (if b and d are finite)
  - Time? $(d+1)b^0 + db^1 + (d-1)b^2 + \cdots + b^d = O(b^d)$
  - Space? $O(bd)$
  - Optimal? Yes if step cost = 1

Use BFS when there exists short solutions. Use DFS when there exists many solutions.

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes*	Yes*	No	Yes, if $l \geq d$	Yes*
Time	b^{d+1}	$b^{1+\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d
Space	b^{d+1}	$b^{1+\lceil C^*/\epsilon \rceil}$	bm	bl	bd
Optimal?	Yes*	Yes	No	No	Yes*

Table 1: Summary of algorithms

1.2.4 Summary

- **Problem solving agents**: formulate a problem, search off-line for a solution, execute it
- **Problem formulation**: initial state, successor function, goal test, path cost
- **Example problems**: traveling around Romania, 8-puzzle, power supply restoration
- **Tree search algorithms**: build and explore a tree, strategy picks up the order of node expansion
- **Implementation**: select the first node on the frontier, test for goal, expand
- **(Uninformed) Strategies**: (breadth first, uniform cost, ...) characterized by their completeness, optimality, complexity
- **Iterative deepening**: complete, finds the shallowest solution, uses only linear space and no more time than uninformed strategies

1.3 Informed Search Agents

1.3.1 Evaluation functions (use heuristics)

$$f(n) = g(n) + h(n)$$

$g(n)$ = cost so far to reach n
$h(n)$ = estimated cost from n to the closest goal (**heuristic**). This is called the **heuristics**.
$f(n)$ = estimated total cost of path through n to goal
The lower $f(n)$, the more desirable n is.
frontier is a queue sorted in ascending/increasing value of $f(n)$

Special cases:
uniform cost search (uninformed): $f(n) = g(n)$
greedy search (informed): $f(n) = h(n)$
A* search (informed): $f(n) = g(n) + h(n)$

1.3.2 Greedy search

entirely heuristics = estimate of cost from n to the closest goal. (so it expands the node that **appears** to be closest to goal)

- Complete? No, can get stuck in loops. But complete in finite space with repeated-state checking.
- Time? $O(b^m)$
- Space? $O(b^m)$
- Optimal? No

1.3.3 A* search

Idea: avoid expanding paths that are already expensive.
Eval func: $f(n) = g(n) + h(n)$.
Admissible heuristic: $\forall n\ h(n) \leq h^*(n)$ where $h^*(n)$ is the **true** cost from n. (Also require $h(n) \geq 0$, so $h(G) = 0$ for any goal G.)
When $h(n)$ is admissible, $f(n)$ never overestimates the total cost of the shortest path through n to the goal.

Theorem 1.8. *If h is admissible, A^* search finds the optimal solution.*

A heuristic is **consistent** if

$$h(n) - h'(n) \leq c(n, a, n')$$

If h is consistent, then h is admissible, and $f(n)$ is nondecreasing along any path:

$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n, a, n') + h(n') \\ &\geq g(n) + h(n) \\ &= f(n) \end{aligned}$$

Consistency: A^* expands nodes in order of increasing f value. Gradually expands f-contours of nodes, where contour i has all nodes with $f = f_i$ where $f_i < f_{i+1}$.

- A^* expands all nodes with $f(n) < C^*$, some nodes with $f(n) = C^*$ but no nodes with $f(n) > C^*$
- Complete? Yes, unless infinitely many nodes with $f \leq C^*$
- Time? Exponential
- Space? Exponential
- Optimal? Yes, cannot expand f_{i+1} until f_i is finished

Definition 1.9. *2 admissible heuristics h_a, h_b, if $h_b(n) \geq h_a(n)$ for all n then h_a **dominates** h_b and is better for search.*

Definition 1.10. *Relaxed problem:*

1.3.4 Graph search

1. multiples paths to the same state may need to be explored and compared to find the optimal
2. keep the better node and update the depths and path-costs of the descendants of the worst one
3. trick to avoid updating descendants: re-open the explored node; its descendants will be updated when it is re-expanded

1.3.5 Summary

- **Heuristic functions**: estimate costs of shortest paths – good heuristics can dramatically reduce search cost
- **Greedy best-first** search expands lowest h – incomplete and not always optimal
- A^*: search expands lowest $g + h$ – complete and optimal
- **Admissible heuristics**: underestimate the optimal cost – they can be derived from exact solutions to relaxed problems
- **Graph search** can be exponentially more efficient than tree search – often needed to ensure termination and optimality – stores all expanded nodes and requires extra tests

1.4 Adversarial Search (Game Playing)

1.4.1 Games

competitive, multi-agent environments.

In AI, a game is often a **deterministic, turn-taking, two-player, zero-sum (utility values are opposite), fully observable game of perfect information**.

strategy	solution	useful when	frontier	algorithm and state space
DFS	arbitrary	many solutions exist	LIFO	tree-search for finite acyclic graphs, recursive algorithm for finite acyclic graphs, add cycle detection for finite graphs
BFS	shortest	shallow solutions exist	FIFO	tree search, graph search (simple) may improve performance
UC	optimal	good admissible, heuristic lacking	priority queue ordered by g	tree search for trees, graph-search (simple) for equal step costs, graph-search (optimal) for arbitrary step costs (no reopening needed)
A^*	optimal	good admissible, heuristics exist	priority queue ordered by f	tree search for trees and admissible heuristics, graph-search (optimal) for admissible heuristics, (no reopening needed for consistent heuristics)
greedy search	arbitrary but maybe good	good (inadmissible) heuristics exist	priority queue ordered by h	tree search for finite acyclic graphs, graph search (simple) for finite graphs

Table 2: algorithm and strategy

Definition 1.11. *A game consists of set of* ***players*** P*,* ***states*** S *and* ***moves*** M*. An* ***initial state*** $s_0 \in S$*,* ***the player to move****,* ***the set of legal moves****,* ***result****,* ***terminal test****,* ***utility function****.*

Definition 1.12. *Solution for a player is not a sequence of actions but a* ***strategy****. A* ***strategy*** *for a player: a function mapping the player's states to (legal) moves. A* ***winning strategy*** *always lead to a win from* s_0*.*

1.4.2 Perfect play - minimax decisions

Perfect play for deterministic, two-player, zero-sum, perfect-info games.

Idea: choose move to position with highest **minimax value = best achievable utility against best possible opponent**

- Complete? Yes, if tree is finite.
- Optimal? Yes, against an optimal opponent.
- Time complexity? $O(b^m)$
- Space complexity? $O(bm)$ (depth-first exploration), but we don't need to explore every path.

Example 1.13. *Minimax example:*
((((3 17) (2 12)) ((15) (25 0))) (((2 5) (3)) ((2 14))))
min: ((3 2) (15 0)) ((2 3) (2))
max: ((3 15) (3 2))
min: (3 2)
max: (3)

Example 1.14. $\alpha - \beta$ *example:*
((((3 17) (2 12)) ((15) (25 0))) (((2 5) (3)) ((2 14))))
β *is the minimum upper bound of possible solutions*
α *is the maximum lower bound of possible solutions*
N *is the current estimate of the value of the node*
for the initial node $\alpha = -\infty, \beta = \infty$*, which is equivalent to* $\geq -\infty, \leq \infty$
pass it down to depth
when a node has specified values, pass up the reverse $\alpha - \beta$ *to its parent, then pass down the parent value to its siblings.*
The value of a node also depends on if it is MIN or MAX

1.4.3 Perfect play - $\alpha - \beta$ pruning

α is the best value (to MAX, i.e. highest) found so far. If V is not better than α, MAX will avoid it, prune that branch. β is similarly for MIN.

Idea:

- α is the best (largest) value found by MAX on path to current node.
- β is the best (lowest) value found by MIN on path to current node
- Some values outside the interval $]\alpha, \beta[$ can be pruned.

Algorithm:

- Node passes its current values for α and β to its children in turn
- Child passes back up its value to Node
- Node updates its current value v (max or min with child's value)
- Node checks whether $v \leq \alpha$ (MIN) or $v \geq \beta$ (MAX)
- If so, child's siblings can be pruned and v returned to Parent
- Otherwise β (MIN) or α (MAX) is updated

Remark 1.15. *Not complete, does not affect final result. Perfect ordering improves effectiveness of pruning.*

1.4.4 Imperfect decisions in real time

Approach: **limit search depth and estimate expected utility**

Use CUTOFF test instead of TERMINAL test;

Use EVAL instead of UTILITY.

Exact values don't matter. Behaviour is preserved under any monotonic transformation of EVAL. Only order matters.

1.4.5 Stochastic games

Chance, random event, not adversarial, the value of chance positions is the expectation (average) overall possible outcomes of the value of the result.

$\alpha - \beta$ pruning is much less effective.

1.4.6 Summary

- A **Game** is defined by an initial state, a successor function, a terminal test, and a utility function
- The **minimax** algorithm select optimal actions for two-player zero-sum games of perfect information by a depth first exploration of the game-tree
- **Alpha-beta** pruning does not compromise optimality but increases efficiency by eliminating provably irrelevant subtrees
- It is not feasible to consider the whole game tree (even with alpha-beta), so we need to **cut the search off** at some point and apply an **evaluation function** that gives an estimate of the expected utility of a state
- Game trees and minimax can be extended to stochastic games by introducing **chance nodes** whose value is the expectation of that of their successors
- The value of alpha-beta pruning and lookahead is limited in stochastic games; the evaluation function needs to compensate

2 Knowledge Representation and Reasoning

2.1 First Order Logic Refresher

2.1.1 The idea of logic

Logic as a basis for KR

- Clear syntax: well-defined recursive structure, automation possible
- Clean semantics: correctness (and incorrectness) are definable, accuracy (ambiguities can be exposed and explained)
- General: works for all domains, pure logic is subject-neutral, definitions depend on form, not content
- Extensible: features of target domains; can add logic of time; can add agent attitudes; can add theories...

Definition 2.1. *Given a formulae set* Γ *of a formal KR language, and a specific formula A, logic determines if A is a* ***consequence*** *of* Γ*.*

- ***Semantic def:*** *A is* ***true*** *in every* ***possible*** *situation satisfying everything in* Γ*.*
- ***Syntactic def:*** *there is a* ***derivation*** *of A from* Γ
- *Either way, some basic properties:*
  - *if* $A \in \Gamma$*, A is a consequence of* Γ
  - *if* $\Gamma \in \Delta$ *then every consequence of* Γ *is a consequence of* Δ
  - *if* Γ *is a set of consequences of* Δ *then every consequence of* Γ *is a consequence of* Δ

Consequence is a matter of **necessity**, logic also defines non-consequence, **possibility**.

2.1.2 Propositional logic: connectives

Propositional logic is the most abstract level of logical language and reasoning.

- **Atomic** sentences p, q, r, etc.
  - treat them as atoms
  - logical operations (connectives) apply from outside
- **Connectives**
  - Apply to sentences (formulae) to make longer ones
  - some unary, some binary, etc.
- **Truth-functional** connectives: truth value of compound determined by values of parts, and, not, etc.

some basic connectives: negation, conjunction, disjunction, implication, equivalence, truth tables...

for n atoms there are 2^n value assignments...

Definition 2.2. ***Term*** *is a name or the result of applying a function symbol to terms, which picks out an individual or object.*

Definition 2.3. ***Predicate*** *applies to a given number of terms to form a sentence, represents a relation (set of n-tuples); sentence $P(t_1, \ldots, t_n)$ true if the objects $o_1, \ldots, o_n$ picked out by those terms are in the relation represented by P.*

Logic of these ground atoms is still just propositional.

2.1.3 First order logic: quantifiers

$\forall$ and $\exists$

2.1.4 Reasoning about systems

- very common to reason about **transitions** between **states** of a system
- logical useful for representing knowledge about **states** and **goals**
- can also represent about **transitions**: each trans has **preconditions, postconditions and frame conditions**
- **Ramification problem:** calculate (relevant) consequences of changes
- **Frame problem:** represent and calculate all frame conditions

2.2 Pure Logical Reasoning

The good and bad about SAT

- good for discrete problems
- bad for possible truth-value assignments growing exponentially, testing in the worst case
- SAT is the classic NP-complete problem

Better and even better news about SAT

- first step: simplify the structure of formulae by using $\vee, \wedge, \neg$ only
- second step: $\neg$ only applied to atoms, **Negation Normal Form (NNF)**
- third step: write it as a conjunction of disjunctions of literals (atoms and negated atoms), **Conjunctive Normal Form (CNF)** aka **Clause Form**

Definition 2.4. ***Resolution*** *is a logical inference rule which operates on clauses. (try to find contradiction $\perp$ out of clauses)*

DPLL: atom p in Γ is either true or false, therefore Γ is satisfiable iff either $\Gamma \cup \{p\}$ or $\Gamma \cup \{\neg p\}$ is satisfiable. Let $\Gamma' = \Gamma \backslash \{p\}$, and with $\neg p$ removed from all clauses in which it occurs. Then Γ' is satisfiable iff $\Gamma \cup \{p\}$ is satisfiable. Γ' is a smaller set now. Same for Γ'' (defined for $\neg p$ removed).

Unit propagation: A **pure literal** is one whose complement does not appear anywhere, so they can be made true without bad consequences. Therefore any clause containing a pure literal may be deleted. A **unit clause** is a clause with only one literal, has to be set true. Therefore its complement can be deleted from all clauses.

DPLL amounts to splitting + unit propagation.

Improving DPLL

- clause learning: the search backtracks when it runs into a contradiction, add complements of (a subset of) the chosen literals as a new clause, never backtrack twice for the same reason
- Choosing good atoms for branching: one that occurs most often in shortest clauses, or one that occurs most often in currently satisfied clauses
- intelligent backtracking: obviously jump back to a variable in the latest nogood, may pay to jump back further
- restarts: jump right back to the root of the search tree and probe it, depends heavily on learned clauses to prevent repeated work

Prenex normal form and removing quants

- 1st step: get all quantifiers to the front, $\Longrightarrow$ and $\Longleftrightarrow$ to $\wedge, \vee, \neg$.
- 2nd step: move quantifiers outside negation, $\neg \forall x A$ becomes $\exists x \neg A$; $\neg \exists x A$ becomes $\forall x \neg A$
- 3rd step: move quantifiers binding x outside another one
- 4th step: existential quantifiers removed by **Skolemisation**: variable replaced by a new name or function, e.g. $\exists x \forall y \exists z R(x, y, z)$ becomes $\forall y \exists z R(a, y, z)$ becomes $\forall y R(a, y, f(y))$
- 5th step: all quantifiers are now universal, they can be removed.

Remark 2.5. *Skolemised formula not equivalent to the original, but they are satisfiable iff the original is.*

2.2.1 Summary

- Problems from many domains can be coded as SAT: discrete, finite, not too much arithmetic
- Intelligent solution methods dominate brute force
- Reduction to clause form: apply logical equivalence, DeMorgan's laws, distribution
- Simple inference rules operate on clauses
  - Resolution (not much used for pure SAT problems)

  - DPLL and its variants generally preferred
  - SAT solvers now useful for real industrial problems
- Normal forms also for first order logic: prenex, skolem, clause form
- Resolution is more useful at the first order level – Resolution-like methods are the state of the art

2.3 Constraint Satisfaction Problems

2.3.1 Introduction

general problem: find an arrangement agreeing with a set of constraints.

2.3.2 Constraint Networks

Definition 2.6. *A **constraint network** is a triple $< V, D, C >$. V is a finite set of variables $v_1, \ldots, v_n$, D is a set of [finite] sets $D_{v_1}, \ldots, D_{v_n}$, C is a set of binary relations $\{C_{u,v} | u, v \in V, u \neq v\}$, $C_{u,v} \subseteq D_u \times D_v$.*

- A constraint $C_{u,v}$ is the allowed pairs of assignments to u and v
- Sometimes require domains to be finite (FD problem), sometimes allow domains to be infinite (e.g. integers, reals)
- Extension to non-binary constraints is simple
- SAT is the special case where all domains have just 2 cases
- Linear programming is a special case where domains are the real numbers and all constraints are linear inequalities

2.3.3 CSPs: the Logical View

problem have logical descriptions... **interpretation of logic**: assign a relation to each predicate, and a function to each function symbol... make formulae true or false... **solutions** to the CSP are exactly **models** of the theory.

2.3.4 Assignments, Consistency, Solutions

Definition 2.7. *Let $< V, D, C >$ be a constraint network, let a be a partial assignment.*

*a is **inconsistent** if there are variables u, v in V and a constraint $C_{u,v}$ in C such that $a(u)$ and $a(v)$ are defined, and $(a(u), a(v)) \notin C_{u,v}$.*

*In that case, a **violates** $C_{u,v}$.*

Consistent is local, inconsistent a already violates a constraint.

Definition 2.8. *Let $\gamma =< V, D, C >$ be a constraint network.*

*a is a **solution** to γ if it is a total consistent assignment for γ.*

*If a solution to γ exists, then γ is **solvable**. Otherwise, it is **unsolvable** or **over-constrained**.*

*A partial assignment a can be **extended to a solution** if there is a solution which agrees with a wherever a is defined.*

Not every constraint partial assignment can be extended to a solution though.

2.3.5 Backtracking

- **Search:** systematic enumeration of partial assignments, if a complete assignment is found, that's a solution, if the search space is exhausted, there are no (more) solutions
- **Backtracking:** pruning of inconsistent partial assignment (and all their extensions)
- there is a tradeoff, reduction in number of search nodes vs runtime needed for inference

pure backtracking

- informal version: recursively instantiate variables one by one, backing up out of a search branch if the partial assignment is inconsistent
- better than exhaustive search: avoids enumerating inconsistent (partial) assignments by detecting them early
- **advantages:** very simple to implement, very fast, complete
- **disadvantages:** does no reasoning except detecting actual inconsistency, cannot look further ahead than the current state

2.3.6 Constraint Modelling

Before any constraint solving, the CSP must be **defined**, i.e. V, D and C (high-level description).

Logical model is purely declarative: no algorithm.

2.3.7 Inference

two constraint network γ and γ' are equivalent iff they have the same solutions.

$\gamma' = (V, D', C')$ is tighter than $\gamma = (V, D, C)$ iff $\forall v \in V, D'_v \subseteq D_v$, $\forall C_{u,v} \in C, C'_{u,v} \subseteq C_{u,v}$, **strictly tighter** if $\gamma \neq \gamma'$

inference in CSP solving: deducing additional constraints that follow from the already known constraints

inference reduces the number of consistent partial assignments

How to use inference

- as offline pre-processing
  - **once before the search starts**
  - little runtime overhead
- during search
  - **at every recursive call of backtracking**
  - when backing up out of a search branch, retract any inferred constraints that were local to that branch because they depend on a
  - strong pruning power, may have large runtime overhead

2.3.8 Forward Checking

Forward Checking

- inference: for all variables v and u where $a(v) = d$ is defined and $a(u)$ is undefined, set D_u to $\{d' : d' \in D_u, (d', d) \in C_{u,v}\}$
- That is, remove from domains any value not consistent with those that have been assigned
- sound, does not rule out any solutions
- implemented incrementally for efficiency
- simple to implement and low computational cost

2.3.9 Variable and value ordering

Variable Ordering

- Common strategy: most constrained variable (aka first-fail) choose a variable with the smallest (consistent) domain, minimize $|\{d \in D_v : a \cup \{(v, d)\} \text{ consistent}\}|$
- minimizes branching factor (at the current node)
- extreme case: select variables with unique possible values first, value is forced by the existing assignment, obviously should be done in all cases
- common strategy: least constraining value, choose a value that won't conflict much with others, minimize $|\{d' \in D_u : a(u) \text{ undefined}, C_{u,v} \in C, (d, d') \notin C_{u,v}\}|$
- minimizes useless backtracking below current node
- other strategies:
  - most constraining variable: involved in as many constraints are possible, maximize $|\{u \in V : a(u) \text{ undefined}, C_{u,v} \in C|$
  - history-dependent strategies like involved in a lot of (recent) conflicts
  - random selection
- if no solutions, or if we want all, value ordering doesn't matter

2.3.10 Arc consistency

Arc Consistency

Definition 2.9. *Variable v is **arc consistent** with respect to another variable u iff for every $d \in D_v$ there is at least one $d' \in D_u$ such that $d, d') \in C_{u,v}$. A CSP $\gamma = (V, D, C)$ is said to be arc consistent (AC) iff every variable in V is arc consistent with every other.*

Remark 2.10.
- *At every iteration, all arcs not in the set M are consistent. M contains the arcs that still need to be checked, M often implemented as a queue*

- *on termination, the network is AC.*
- *unlike forward checking, makes inferences from unassigned variables*
- *slower (per node) than forward checking, but prunes more*

2.3.11 Summary

- **Constraint networks** consist of **variables** associated with (usually finite) **domains** and **constraints** which are [binary] relations specifying allowed pairs (or tuples) of values
- A **partial assignment** maps some variables to values; a **total assignment** does so for all variables. A partial assignment is **consistent** if it does not violate any constraint. A consistent total assignment is a **solution**.
- The **constraint satisfaction problem** (CSP) consists in finding a solution for a constraint network. Applications are everywhere.
- **Backtracking** instantiates variables one by one, cutting branches when inconsistent partial assignments occur.
- **Variable ordering** in backtracking can dramatically reduce the size of the search tree. **Value orderings** don't, but they may lead to solutions earlier.
- **Inference** tightens γ without losing equivalence, during backtracking. This reduces the amount of search needed. The benefit in reduced tree size must be traded off against the time cost of the reasoning.
- **Forward checking** removes values conflicting with an assignment already made.
- **Arc consistency** extends this to all variables, whether assigned or not. It is stronger than forward checking and unit propagation, but costs more to compute.

2.4 Solving Constraint Satisfaction Problems

2.4.1 Tightening CSPs by learning from mistakes

don't backtrack twice for the same reason

learned constrained may be added to the network or kept separately

store of nogoods require exponential space

it is common to forget them with strategy, e.g. let the longest ones lapse after a while or just keep the tail and discard them when backtracking leaves the region where it applies

2.4.2 Problem structure: constraint graphs

graph contains info about the structure of the problem

normal view: variables as vertices

dual view: constraints as vertices

bipartite view: variables and constraints as vertices

2.4.3 Symmetry

symmetry is good, but if a problem has lots of symmetries, could waste huge amounts of time; good part is, if one sub-space explored, prune them all!

2.4.4 CSPs and optimisation

two common ways of defining better or worse solutions: via an **objective function**, as a quantity to be minimized (or maximized) or via **soft constraints**, as can be violated but as little as possible.

The sum of soft constraint violations behaves as an objective function

2.4.5 DFBB

Depth First Branch and Bound (DFBB) Use a lower bound L on the objective and an upper bound B

Backtrack whenever, $L \geq B$

Revise B whenever a solution is found

First solution is at the bottom of the leftmost (complete) branch, fast and dirty

always trying to improve. so DFBB produces a sequence of (strictly) improving solutions

can interrupt at any time

2.4.6 Summary

- Constraint (nogood) learning from wipeouts usually improves efficiency, space (memory) is a limitation for nogood learning, so forgetting is also important
- constraint graphs give information about problem structure, certain constraint graphs (e.g. trees) indicate that problems are easy
- value symmetry and variable symmetry are frequently present in CSPs – pruning symmetric sub-spaces is a big winner where there is extensive symmetry
- optimization (minimizing a cost or objective function) is usual for CSPs
- Depth First Branch and Bound is commonly used in FD solvers – conveniently provides intermediate solutions of increasing goodness

2.5 Constraint Satisfaction and Local Search

Absolute optimality is too hard. Use **Local Search** instead.

2.5.1 Local Search in general

general idea: search in the space of **total** assignments

an alternative to backtracking: include some randomness

hill-climbing: **random-restart hill climbing** overcomes local maxima, **random sideways moves** escape from shoulders, but loop on flat maxima

simulated annealing: escape local maxima by allowing some bad moves, but gradually decrease their badness and frequency

Population-based methods

- **Local beam search**: maintain a population of k states, add some neighbours at each step, delete the worst ones to keep the population stable
- **Genetic (evolutionary) algorithms** crossover between pairs of states in the population (compare genetics), offspring have some features of each parent, cull according to a fitness measure...

2.5.2 Large Neighbourhood Search in particular

- Search in the space of **solutions** rather than **states**
- Given a current solution: destroy part of it by forgetting the values of some variables, see the problem of assigning values to those variables as a CSP, solve (optimally) using a complete search method such as DFBB
- Alternates **destroy** and **repair** phases: repair phase searches a neighbourhood of the current solution, neighbourhood size remains tolerable, even if the problem is huge
- The old solution is still in the neighbourhood, so there is always a next solution available
- may search for the optimal solution in the neighbourhood, or for an improvement on the previous solution, or just for any solution in the neighbourhood
- performance is sensitive to what to destroy
- tradeoff between neighbourhood size and speed
- local optima still a problem

2.5.3 Summary

- local search explores the space of total assignments
- usually step to a neighbour, looking for improvement – but sometimes jump further
- no guarantees (incomplete, sub-optimal,...) but in many cases scales up better than systematic search
- many varieties: hill-climbing, random walks, simulated annealing, population methods
- large neighbourhood search the best of both worlds?

3 Planning

3.1 Intro

Planning looks for a sequence of actions achieving a goal state (as before)

Planning uses search BUT in conjunction with adequate KR

Planning uses representation of states and actions allowing us to exploit the structure of the problem and lead to general heuristics for planning

Planners are general problem solvers that take as input a description of the problem in a high-level language

3.1.1 Planning

Definition 3.1. *Planning is the reasoning side of acting. It is an explicit deliberation process that choose and organizes actions, on the basis of their expected outcomes, in order to achieve some objectives as best as possible.*

Planner = solver over a class of planning models

3.1.2 Classical planning problems and plans

assumptions:

- **finite:** states, actions, observations are finite
- **static, single agent:** no event outside of the planner's control
- **deterministic:** unique initial state, unique resulting state
- **fully observable:** sensors provide all relevant aspects of the current state
- **off-line planning:** planning decoupled from execution
- **implicit time:** no durations, instantaneous actions
- **sequential**: solution is a sequence of actions
- **reachability goals:** acceptable sequences end in a goal state
- **cost function:** length or path-cost of the sequence

Classical planning model

- a finite set of states S
- a finite set of actions A
- a transition function $\gamma : S \times A \to S$
- an initial state s_0
- a set S_G of goal states
- a (step) cost function: $c : A \to R^+$
- If action a is not applicable in state s then $\gamma(s, a)$ is undefined

Classical plans

- linear plan (or sequence): totally ordered set $< a_1, \ldots, a_n >, a_i \in A$ such that $\gamma(\ldots\gamma(\gamma(s_0, a_1), a_2), \ldots, a_n) \in S_G$. Produced by state-space planning approaches.
- non-linear plan: partially ordered set $< \{a_1, \ldots, a_n\}, <>, a_i \in A$ such that each linearization is a valid linear plan. More flexible for execution. Produced by plan-space planning approaches.
- parallel plan: sequence of parallel action sets $< \{a_{1,1}, \ldots, a_{1,l(1)}\}, \ldots, \{a_{1,n}, \ldots, a_{1,l(n)}\} >$, $a_{i,j} \in A$. Actions in each set must not *interfere*: performing them in any order or in parallel must lead to the same result. Produced by SAT-based and graph-based planning approaches.

3.1.3 STRIPS and ADL representations

A classical planning model $(S, A, \gamma, s_0, S_g, c)$ can be solved directly by search. But that is too much computation.

We reply on adequate **representation** of the planning problem, enabling **concise** problem descriptions, exploiting the **structure** of the problem, scaling to large problems whilst maintaining **domain-independence**

The STRIPS representation

- states: fragment of first-order logic to represent states
  - **logical language** (predicates, connectives, variables, quantifiers, finite object set, no function)
  - a **property** of state (a set $S' \subseteq S$) is represented by a formula
  - the **goal** is often represented by a set of ground atoms for simplicity
  - a **state** $s \in S$ is represented by a set of ground atoms under the **closed world assumption**
- actions: use operators with logical pre-post conditions to represent actions
  - operator o has a name and parameters
  - precondition $PRE(o)$ is a set of positive literals that must be true for the action to be applicable
  - effect (postcondition) $EFF(o)$ is a set of literals that changes in the resulting state, is split into two sets of positive literals, add list $EFF^+(o)$ and delete list $EFF^-(o)$
  - an action $a \in A$ is represented by an instance of an operator
- transition: use the STRIPS rule to represent the transition function γ
  - assumption: atoms not affected by the action keep their value
  - $\gamma(s, a) = (s \backslash EFF^-(a)) \cup EFF^+(a)$ if $PRE(a) \subseteq s$, undefined otherwise (action not executable)

3.1.4 PDDL: Planning Domain Definition Language

- STRIPS
  - only positive literals in states, closed world assumption, unmentioned literals are false.
  - Effect $\{P, \neg Q\}$ means add P deletes Q
  - no support for equality and types
  - only positive literals in prec. and goals
  - effects are sets (conjunctions)
- ADL
  - Positive and negative literals in states, open world assumption, unmentioned literals are unknown
  - Effect $\{P, \neg Q\}$ means add P and ¬ Q, delete Q and ¬ P
  - Equality predicate (x=y) built in, variables may have types
  - prec. and goals are arbitrary formulae
  - conditional and universal quantified effets

complexity of propositional STRIPS planning

propositional STRIPS planning: all predicates and operators have been instanciated (grounded). Recall that for STRIPS, preconditions are positive.

n propositions can result in 2^n states: in the worst case, the shortest plan is exponentially long ($2^n - 1$ actions)

PLANSAT, PLANMIN, both are NP-complete if the plan length is polynomially bounded

n predicates with k arguments and m objects can give up to nm^k atomic propositions, these can give 2^{mn^k} states. of course, the worst case need $2^{nm^k} - 1$ actions

3.1.5 Summary

Planning is reasoning side of acting. Planning = Search +KR

Classical planning is an off-line process which assumes a single agent and a static environment, determinism, full observability, reachability goals, and ignores quantitative time

The STRIPS representation uses a logical language to represent properties of states; actions are represented by their preconditions and add/delete effects

STRIPS enables algorithms to **exploit the structure** of the problem. ADL is a useful extension of STRIPS. PDDL supports both and many extensions

Propositional STRIPS planning is PSPACE-complete

Planning algorithms differ by the search space they explore and the type of classical plan they produce: **sequential, partially ordered, or parallel plan**

3.2 Planning-Graph and Satisfiability Techniques (NOT EXAMABLE

3.2.1 Planning graph techniques

motivation: state-space search produces inflexible plans (linear plans), wastes time examining many different orderings of the same set of actions. Not ordering actions that can take place in parallel can speed up planning. state-space does not exploit the structure of states (except for heuristics)

The planning graph

- alternating layers of propositions and actions $P_0, A_1, P_1, \ldots, A_i, P_i, \ldots, A_k, P_k$
- A_{i+1} contains the actions might be able to occur at time step $i+1$, preconditions must belong to P_i
- P_{i+1} contains the propositions that are **positive** effects of actions in A_{i+1}

Mutual exclusion

pairs of actions which cannot happen in parallel and pairs of propositions which cannot be simultaneously true. these are mutex.

The action parallelism notion underlying the mutex relation is **independence**. Two actions are independent when executing them in any order is possible and yields the same result.

For independence, avoid:

- **interference:** one action deletes a precondition of the other (one of the two orderings is not possible)
- **inconsistence:** one action deletes a positive effect of the other (the two orderings yield different results)
- two actions at the same level of the graph are mutex if they
  - interfere
  - are inconsistent
  - have competing needs: they have mutually exclusive preconditions
- two propositions at the same level are mutex if they
  - have inconsistent support: all ways of achieving both are pairwise mutex

properties of the graph

propositions and actions monotonically increase across levels.

propositions and actions mutexes monotonically decrease across levels.

Necessary condition for plan existence: if the goal propositions are present and mutex-free at some level P_k then a k step parallel plan achieving the goal **might** exist

Heuristics for planning:

- single proposition p: cost of achieving p is the index of the first level in which p appears

- set of propositions: max (or sum) of the individual costs, or index of the first level at which they all appear mutex-free

planning: graphplan algorithm, build the graph up until the necessary condition is reached; try extracting a plan from the graph, if this fails, extend the graph over one more level, repeat until success or termination condition

Plan extraction Backward search, from the goal layer to the initial state layer.

- select an open precondition at the current layer, choose an action producing it, the action **must not be mutex** with any of the parallel actions already chosen for that layer
- when there is no more open precondition at that layer, work on achieving, at the previous layer.

3.2.2 SAT planning techniques

principles of SAT planning

- planning is PSPACE-complete and SAT is NP-complete
- SAT can only solve the **bounded** plan existence problem, does there exist a plan with k or less steps?
- solving the original planning problem requires solving a **sequence** of SAT problems, e.g. increment k until a plan is found

encoding

- variables: p@t, for t in [0, k], a@t for t in [0, k-1]
- clauses: initial and goal, preconditions and effects, explanatory frame axioms, mutual exclusion axioms

well-known encoding improvments

- fluent mutexes, use the plangraph to get them
- search bound knowledge (domain-specific)
- parallel plans
- operator splitting and axiom factoring

3.2.3 Summary

- **Graph-based planning** produces parallel plans. A planning graph is a relaxation of the state space, which gives us a necessary condition for the existence of a parallel plan of a given length. If one really exists, it can be extracted by backward search through the graph.
- **SAT planning** uses a SAT solver to solve the bounded (parallel) plan generation problem. Logical planning formalisms must deal with the frame problem.

- **Plan-space planning** produces partially-ordered plans. This approach does not commit to orderings or bindings unless necessary. It searches the space of partial plans, refining the plan at each step to remove flaws.
- **Current planning research** extends these methods to handle time, uncertainty, multiple agents, hybrid (discrete-continuous) systems, and many other aspects of real world applications.

3.3 State-Space Planning

3.3.1 State-space planning

planning procedure are often search procedures

differ by search space they consider

search space explored in many ways: forward, backward; a variety of strategies (search); a variety of heuristics...

3.3.2 Progression planning (forward search)

Forward-search used in conjunction with any search strategy...

FS is **sound**: any returned plan is a solution

Fs is **complete**: provided the underlying search strategy is complete, it will always return a solution

3.3.3 Heuristics

Domain-independent heuristics

Remark 3.2.
- *An **admissible** heuristic is **optimistic**: it gives a lower bound on the true cost of a solution to the problem*
- *an admissible heuristic can be obtained by **relaxing** a problem P into a simpler problem P': the cost of any optimal solution to P' is lower bound on the cost of the optimal solution to P*

some planning heuristics:

- delete relaxation heuristics: P be a planning problem, P^+ be a relaxed problem obtained by ignoring the negative effects of every action, P^+ is the **delete-relaxation** of P, the solution is a **relaxed plan**
- The cost h^+ is an optimal solution to P^+ is a lower bound on the cost of an optimal solution for P, hence an admissible heuristic. but it is still NP-hard, need further relaxation gives admissible h^{max} heuristics
- finding an arbitrary solution for P^+ (PLANSAT) is polynomial, derives inadmissible h^{FF} heuristics
- **Further relax the problem** by ignoring interactions between subgoals
- admissible h^{max} heuristic: cost to reach a set is the max of costs
- inadmissible h^{sum} heuristic: cost to reach a set is the sum of costs, assumes subgoal independence

- admissible generalization h^m heuristics: max of coasts to reach each subset of size m
- h^+ too hard, h^{max} not very informative, h^{sum} can greatly overestimate h^*, new heuristic h^{FF}, which is inadmissible, compromises between h^{max} and h^{sum}
- **abstraction heuristics**: simplify by ignoring parts of it: drop preconditions from actions, consider only a subset of predicates/propositions, count objects with a given property, ignoring the identity of objects, ignore so much that the abstract problem is small enough to be solved by uninformed search
- The **abstraction heuristic** h^ϕ is admissible and consistent
- proposition l is a **landmark** for problem P iff all plans for P make l true.
- **Landmark heuristics**: counts the number of yet unachieved landmarks, generalization of the number of unachieved goals heuristic used in the LAMA planner. inadmissible

3.3.4 Regression planning (backward search)

backward search start at the goal and compute inverse state transitions, aka **regression** leading to a new **goal**

An action a is **relevant** for goal g if it makes at least one of g's propositions true, and it does not make any of g's proposition false

backward search can be used with any search strategies...

Sound and complete

3.3.5 Lifting

reduce the branching factor if we only **partially instanciate** the operators

lifted-backward-search is sound and complete, can be used with any search strategies

3.3.6 Summary

- **State-space planning** produces totally-ordered plans by a forward or backward search in the state space. This requires **domain-independent heuristics** or domain-specific control rules to be efficient
- The **Delete-relaxation** of a problem ignores delete lists. Once a fact becomes true in a delete-relaxed problem, it remains true. The **optimal delete-relaxed plan** cost h^+ is an admissible heuristic but is NP-hard to compute
- The **critical path heuristic** h^{max} is admissible and computable in polynomial time but additionally ignores anything but the hardest subgoal of each goal set. h^{sum} is an inadmissible variant which sums the costs of subgoals.

- The inadmissible h^{FF} heuristic returns the number of actions in a relaxed plan, extracted from the **relaxed reachability graph**. **Abstractions** and **landmarks** lead to other powerful heuristics
- Backward search requires the ability to **regress** goals and leads to a **lifted** algorithm which does not need to fully instanciate operators

3.4 Plan-Space Planning

3.4.1 Motivation

no notion of states, just partial plans

adopts a **least-commitment strategy**: don't commit to orderings, instantiations etc unless necessary

produces a partially ordered plan: represents all sequences of actions compatible with the partial ordering

benefits: speed-ups (in principle), flexible execution, easier replanning

3.4.2 Partial plans

multiset O of **operators** $\{0_1, \ldots, o_n\}$

set $<$ of **ordering constraints** $o_i < o_j$

set B of **binding constraints** $x = y, x \neq y, x \in D, x \notin D$

set L of **causal links** $o_i \rightarrow^p o_j$ stating that effect p of o_i establishes precondition p of o_j, with $o_i < o_j$ and binding constraints in B for parameters of o_i and o_j appearing in p

nodes are partial plans

successors are determined by plan refinement operations, each operation add elements to $O, <, B, L$ to resolve a flaw in the plan.

no flaw: 1. no **open precondition**: all preconditions of all operators in O are established by causal links in L. 2. no **threat**: for every causal link, every o_k with $EFF^-(o_k)$ unifable with p such that $o_k < o_i$ or $o_j < o_k$

$<$ and B are consistent

3.4.3 Flaws

3.4.4 Plan-space planning algorithm

3.4.5 Example

3.4.6 Summary

- **Graph-based planning** produces a polynomial-size graph that gives us a necessary condition for the existence of a parallel plan of a given length. If one really exists, it can be extracted by backward search through the graph.
- **SAT planning** uses a SAT solver to solve the bounded (parallel) plan generation problem. This is efficient when optimal parallel plans are short. Logical planning formalisms must deal with the frame problem.
- **State-space planning** produces totally-ordered plans by a forward or backward search in the state space. This requires **domain-independent heuristics** or domain-specific control rules to be efficient

- **Plan-space planning** produces partially-ordered plans: this approach does not commit to orderings or bindings unless necessary. It searches the space of partial plans, refining the plans at each step to remove flaws.
- **current planning research** extends these methods to handle time, uncertainty, multiple agents, discrete/continuous systems, and real world applications

alpha-beta pruning
arc consistency