

## Intelligent Agents

Rationality: achieving maximum utility by some pre-defined metric or set of goals/intentions.  
depends on the usefulness of the choice and not on the process that led to that choice  
e.g., rational process for playing tic-tac-toe by tabulating game states  
has no decision process, but acts rationally

An agent has sensors which take in percepts and actuators which effect actions  
a percept is a set of perceptual inputs at a fixed point in time  
a percept sequence is composed of percepts

An agent function is  $f : P^* \rightarrow A$  where  $P^*$  is the percept sequence and  $A$  the set of actions  
the agent program is the specific architecture which implements this function  
not all agent functions can be implemented by some agent program  
e.g. halting problems, NP-hard problems, “too-large” problems (e.g. chess)

Performance measure: an objective criterion for the success of behavior of an agent

Rational agent: maximizes expected performance measure by its actions  
given the prior knowledge and percept sequence available to it  
e.g. vacuum world: 2 squares, could be dirty or clean  
action function: suck if dirty, move if clean  
under the measure of most clean squares/time period, this is rational  
if we seek to minimize movements as well, this is irrational

Autonomy: the ability to function beyond the prior knowledge of the designer

The task environment: performance measure, environment, actuators, and sensors  
partially vs. fully observable (perceive all aspects relevant to choice of action)  
stochastic vs. deterministic (next state a function only of current state, action)  
strategic: deterministic but for the actions of other agents  
episodic vs. sequential (current decision could affect future decisions)  
static vs. dynamic (environment can change while agent deliberates)  
semidynamic: environment doesn't change with time but performance score does  
semidynamic e.g. chess with a clock  
discrete vs. continuous (can apply to state, time, percepts/actions)  
single vs. multiagent: e.g. competitive multiagent, cooperative multiagent

Agent structure

the agent program only takes in the current percept  
the agent function maps from the entire percept history

A simple reflex agent uses only the current percept in its decision process  
responds via condition-action rules  
requires full observability to work effectively

A state-based reflex agent maintains internal state using the percept history

needs what is called a model of the environment  
knowledge about how it evolves, how actions affect it  
also called a model-based reflex agent

A goal-based agent seeks to achieve states which are considered favorable

Unlike reflex agents, makes use of foresight in its decision process

A utility-based agent evaluates future states using a utility function

Seeks to maximize utility of future states

A learning agent can be broken down roughly into four components

A learning element is responsible for improving the agent

A critic evaluates current performance and informs the learning element  
(uses a fixed performance standard)

A performance element selects actions

A problem generator suggests exploratory actions

Two of these are reflexive, the next two are planning-based.

Planning agents predict consequences of actions using a transition model.

Agents can vary in the degree to which they deliberate

one extreme: carefully construct a complex plan

another: start with a simple plan and rapidly correct as complications arise

## Search Problems

We consider problem-solving agents, which set out to achieve some desirable state

an uninformed search algorithm has no idea of where to look for solutions

relies only upon the problem definition

An agent plan in a search problem will first formulate, then search, then execute

Problem formulation:

simplifies environment, abstracts away unnecessary information

helps organize the behavior of the agent

Components of a well-defined problem:

(1) Initial state

(2) Possible actions: i.e., a successor function  $f : \{\text{states}\} \rightarrow \{(\text{action}, \text{consequent state})\}$

These first two implicitly define a state space, and a state space graph

State space graph has states as nodes and actions as edges

(3) Goal test

(4) Path cost: (e.g. induced by edge costs of a state space graph)

A path is a sequence of actions.

A solution is a path from an initial state to a goal state.

An optimal solution has lowest path cost.

Environments we are considering are static, observable, discrete, and deterministic.

Generalized tree search

envision the search space as a tree (states can be revisited)

nodes have: state, parent-node, action leading to that node, path-cost, and depth

the root of the search tree corresponds to the initial state of the problem

uses a fringe, initialized to contain only the root node  
the fringe contains nodes which have been generated but not yet expanded

At each node in the process:

perform goal test on the node in question, if succeeds, return corresponding solution  
expand node by i.e. applying successor function to generate new nodes  
if no such nodes exists, declare failure  
choose which node to analyze next according to some search strategy

Evaluation of a search strategy.

Completeness: whether or not it finds a solution, if a solution exists.

Optimality: if it finds a solution, whether or not such a solution is of lowest cost.

Complexity: space (nodes stored at any given time) or time (total nodes explored)

Factors often used to evaluate the complexity of a given algorithm.

Branching factor  $b$ : the maximum number of successors of any node.

Depth  $d$ : the minimum depth among goal nodes.

Maximum path length  $m$ : can be infinite.

Types of uninformed (aka blind) search strategies

contrast: informed search strategies take advantage of heuristics

cannot solve searches of exponential complexity for all but the smallest problems

Breadth-first search uses a queue (FIFO) as its fringe.

Satisfies completeness (if branching factor finite) but not optimality.

Identical space and time complexity (holds all nodes in state until goal is found)

Exponential complexity:  $\mathcal{O}(b^{d+1})$

In practice, spacial complexity too hard: very difficult to accrue enough memory

Uniform-cost search orders its fringe by path cost.

As long as each step cost  $> \epsilon > 0$ , satisfies completeness *and* optimality.

Again, same time and space complexity

Given  $\epsilon$  minimum action cost,  $C^*$  optimal solution cost, complexity is  $\mathcal{O}(b^{\lceil C^*/\epsilon \rceil})$

Depth-first search uses a stack (LIFO) as its fringe

Node storage is not exponential, space complexity is  $\mathcal{O}(bm)$ .

Backtracking search: nodes store set of successors, search expands only one at a time.

Backtracking search can reduce the space complexity to  $\mathcal{O}(m)$ .

Worst-case time complexity  $\mathcal{O}(b^m)$  where potentially  $m \gg d$  and even  $m = \infty$ .

Complete if  $m < \infty$ , but not optimal.

Depth-limited search

specifies a depth limit  $l$ , and performs a depth-first search up to that limit

if  $l < d$ , will be incomplete, and if  $d < l$ , can be non-optimal

has a time complexity of  $\mathcal{O}(b^l)$  and a space complexity of  $\mathcal{O}(bl)$ .

the diameter of the state space: min number of actions between any two states

is a great depth limit, but, we don't necessarily know the diameter, a priori

Depth-first search by iterative deepening

apply depth-limited search with  $l$  ranging over  $\mathbb{N}$

like DFS, has good memory (spatial complexity) at  $\mathcal{O}(bd)$

like BFS, if  $b$  is finite will be complete, and if path cost corresponds to depth, optimal

repeated state generation not costly: most nodes in the bottom level  
generated nodes:  $d(b) + (d-1)b^2 + \dots + (1)b^d$  (factor of  $d$  is node repetition)  
hence the time complexity is  $\mathcal{O}(b^d)$

better than BFS ( $\mathcal{O}(b^{d+1})$ ): IDS generates no nodes beyond the solution depth

When search space is large and solution depth unknown, depth-first IDS is generally best.

Bidirectional search run simultaneous searches from initial state and goals

Concludes when the searches meet.

Time and complexity  $\mathcal{O}(b^{d/2})$ , is reduced.

However, requires effective computation of predecessors: often non-trivial/impossible.

Avoiding repetition of states: Graph Search

adds a closed list to tree search: a list of all nodes which have been expanded

the term open list is sometimes used to refer to the fringe

current nodes which match a node on the closed list are discarded

possibly suboptimal, if search methods can reach nodes at a non-optimal cost first

thus uniform cost graph search is optimal but IDS graph search may not be

the closed list increases space requirements, possibly to unfeasibility

behavior of closed list is such that the memory used is proportional to the runtime

Partial Information Search

Sensorless Problems

know consequences of actions and the possible states

can coerce world into a particular state, with some cleverness

uses a belief state: a set of states currently regarded as possible

a solution is a path to a belief state in which all its members satisfy the goal test

this approach can be analogously applied to nondeterministic problems

Contingency Problems

agent can obtain new information from sensors after performing actions

solution: use an unfixed action sequence, with dependencies on percepts received

the action at each node will depend on all percepts received up to that point

agent can act before finding a guaranteed plan

approach referred to as interleaving search and execution

does not need to account for *all* contingencies, simply the ones that occur

## Informed Search and Exploration

Informed search uses problem-specific knowledge beyond the definition of the problem

Approach: select nodes for expansion based on an estimate of distance to goal

Use a priority queue ordered by some evaluation function  $f$

Called best-first search: since we order by nodes which seem best

A heuristic  $h : \{\text{nodes}\} \rightarrow \mathbb{R}^+$  estimates cost of cheapest path to a goal node

Greedy best-first search uses  $f = h$

susceptible to false starts, dead ends

same defects as DFS: not optimal, incomplete, worst-case time/space complexity  $\mathcal{O}(b^m)$   
 $m$  the maximum depth of search space

A\* search uses  $f = g + h$  where  $g$  is the path cost to a node  
hence  $f$  is the estimated cost of cheapest solution through  $n$   
for tree search, if  $h$  is admissible (never overestimates), A\* will be optimal  
for graph search, need to ensure the first generated path is optimal  
this shall occur if  $h$  is additionally consistent/monotone  
consistent  $h$  satisfies, for nodes  $n, n'$  and action  $n \xrightarrow{a} n'$ ,  $h(n) \leq c(a) + h(n')$   
satisfies a triangle inequality; i.e.,  $h$  must be a metric on the state space  
key consequence of a consistent  $h$ :  $f$  is always increasing along a path  
note: consistency implies admissibility  
A\* is optimally efficient for a given  $h$ :  $\nexists$  an optimal, more efficient algorithm  
however, nodes in goal contour search space still increase exponentially,  
unless  $|h - h^*| \leq \mathcal{O}(\log(h^*(n)))$  where  $h^*$  is the true cost  
generally  $|h - h^*| = \mathcal{O}(h^*(n))$  at best  
thus, it's often impractical to insist on finding an optimal solution  
A\* keeps all generated nodes in memory  $\rightarrow$  impractical for large-scale problems

Improvements on A\* with respect to space complexity

Can try iterative deepening A\* (IDA\*) using  $f = g + h$  rather than  $d$  as the cutoff  
but this incurs substantial overhead

Recursive best-first search

tracks the  $f$ -value of the current best alternative path  
winds back to this alternative path once  $f$  considered exceeds that stored  $f$   
optimal if  $h$  is admissible, with space complexity  $\mathcal{O}(bd)$   
time complexity can vary, depends on  $h$ , frequency of path changes  
can potentially explore a state multiple times (typical tree-search problem)  
stores only the value of  $f$  and  $\mathcal{O}(bd)$  nodes

MA\* (memory-bounded A\*) and SMA\* (simplified MA\*) make use of all memory

description of SMA\*: keep expanding until memory is full  
once memory is full, replace the worst (by  $f$ ) node in memory, breaking ties by age  
SMA\* is complete if there is enough memory to hold the shortest path to a goal  
practically, probably best for a graph state space and non-uniform path costs  
if too much switching, problems that A\* would solve become intractable for SMA\*  
time  $\leftrightarrow$  space tradeoff

Learning to search better

metalevel learning algorithm analyzes current method, seeks improvements

Admissible heuristic creation

often solutions to relaxed problems, where new actions are available

A heuristic  $h$  dominates  $h'$  if  $h \geq h'$  on the state space

higher admissible heuristics are stronger

trade-off between computation on heuristic efficacy and in searching  
taking the maximum over a set of admissible heuristics can be useful

## Local Search

In which the path does not matter, only the achievement of the goal state  
also useful to solve pure optimization according to an objective function  
state space landscape: manifold of the objective function on the state space  
seek global extrema: completeness and optimality defined as before

hill-climbing search (greedy local search):

- take any actions that improve the situation

- foiled by local extrema, ridges, plateaus on the manifold

- dealing with shoulders: allow sideways moves, but need to limit, because of plateaus

- random-restart hill climbing guaranteed to eventually find goal state

simulated annealing allows bad moves occasionally via thermodynamic principles

- randomly chooses an action, always goes ahead with improvements

- if a downgrade, accept it with probability dictated by a Boltzmann weighting

- start at high temperature, and slowly lower  $T$

- the algorithm is guaranteed to find a global optimum with probability  $\rightarrow 1$

local beam search runs  $k$  copies of a local search algorithm

- at each step, generate all the successors, and choose the  $k$  best successors

- to keep diversity among states high, can use stochastic beam search

- stochastic beam search introduces some randomness, probability  $\propto$  value

genetic algorithms: successors are generated as combinations of parent states

- $k$  randomly generated states, use evaluation function as a judge of fitness

- crossovers and mutations generate new states

- works well in some cases, but not others

## Uncertain Search

contingency plans to account for nondeterminism, stochasticity

- branch for each possible result following a given action, form a conditional plan

and-or search trees:

- or level on actions (only need 1 to work)

- and level on nondeterminism (need a plan to work for all branches)

- contingent solution cuts down on the tree by selecting actions

- all the leaves need to be goals

- can implement and-or search as corecursion with an and and an or function

cyclic solution

- all leaves are goal states

- every point in the plan has a path to every leaf

belief states to account for partial observability

- new transition model: predict based off of the action, update based off of the percepts

## Adversarial Search

Formally define game as a search problem with:

- initial state
- successor function from states to (move, state) pairs
- terminal test
- utility function or objective function (i.e. a score)
- these first two yield a game tree

an strategy is considered optimal relative to an infallible opponent

the minimax value of a node is its utility assuming mutual infallibility

algorithm computes best decision from current state, using recursion

$\alpha$ - $\beta$  pruning

$\alpha$  is the value of the current best for max

$\beta$  is the value of the current best for min

if choosing better node ordering for pruning possible,  $\mathcal{O}(b^d) \rightarrow \mathcal{O}(b^{d/2})$

random order for pruning leads to approximately  $\mathcal{O}(b^{3d/4})$

a transposition table is a hash table of previously seen positions

evaluation function: a heuristic for when depth is large

a weighted linear (or nonlinear) function of features

quiescent position: unlikely to exhibit wild swings in value in near future

it is best to expand until you are at quiescent states

horizon effect: minimax avoids a very good move for the opponent

even when it must eventually happen

better algorithms can see if there's a horizon effect present, accept it and move forward

forward pruning is also possible, but dangerous (could prune best paths)

impact of chance  $\rightarrow$  chance nodes

expectiminimax: use expected value over chance nodes

decisions invariant under positive linear transformations of the evaluation function

by contrast, any isotone transformation preserves the decisions of minimax

## Constraint Satisfaction

constraint satisfaction problems start to unpack the black box:

search and game-playing are highly abstracted atomic representations

constraint satisfaction problems, propositional logic are factored representations

even more complex are structured representations such as first-order logic

CSP breaks state down into variables  $X_i$ , each of which takes values from a domain  $D$

Goal test represented by a set of constraints upon allowed values

expressible implicitly  $Val(X) \neq Val(Y)$  or explicitly (enumerate all possibilities)

explicit expression e.g.  $(X, Y) \in \{(1, 2), (2, 3), \dots\}$

Binary CSP: each constraint relates at most two variables

any non-binary CSP can be converted to a binary CSP

can then be represented by a graph, vertices are variables, edges are constraints

Characterizing a finite CSP:  $n$  variables,  $d$  the maximum domain size

$\mathcal{O}(d^n)$  complete assignments exist

generally no better than exponential time, in the worst case

Search formulation for solving a CSP

- initial state:  $\{\}$ , action assigns a value to a variable

- continue until all variable assigned and all constraints satisfied

- prior methods (e.g. BFS, DFS) highly inefficient

backtracking search is a depth-first search with the following alterations:

- variable assignment is commutative: apply assignment to variables in a fixed order

- reducing branching factor from  $nd$  to  $d$

- only considers values that do not conflict with previous assignments

improvements to backtracking

- smallest domain first: variable ordering by minimum remaining values (MRV)

- break ties by degree heuristic: choose variable with most ties

- goal: fail as quickly as possible, to eliminate large sections of the tree

- least restrictive assignments: value ordering by least constraining value (LCV)

making inferences about the domains, using filtering

- forward checking eliminates values of adjacent variables that violate a constraint

better inferences: maintain arc consistency to detect failures earlier

- consistent arc  $(X, Y)$ :  $\forall x \in \text{dom}(X), \exists y \in \text{dom}(Y)$  satisfying constraints

- repeatedly check arc consistency to pare down the domains

can apply algorithm separately to connected components, independent

- breaking down greatly reduces complexity  $\mathcal{O}(d^n) \rightarrow \mathcal{O}(\frac{n}{c}d^c) = \mathcal{O}(n)$

Tree-Structured CSPs have graph representations which are trees

- Can be solved in  $\mathcal{O}(nd^2)$  time, no longer exponential

- choose any ordering, pick a root, make a linear chain

- apply arc consistency (working backwards)

- make assignments (moving forwards)

- worst case: check  $d$  values against each of the  $d$  values,  $n$  checks up,  $n$  checks down

simplify CSPs down to tree-structured CSPs if possible to reduce runtime

- set a value for some variables first if helps to reduce down to a tree-structure

- conditioning: instantiate a variable to affect the rest of the CSP favorably

- cutset conditioning: instantiate variables such that the remaining graph is a tree

- can then compute residual CSPs for each of the possible cutset value assignments

local search for CSPs: min-conflicts algorithm

- while not solved, randomly select any conflicted variable

- value selection: heuristic by minimum resulting conflicts

- extremely good for problems dense in solutions

- CSPs are very good at dealing with randomly-generated CSPs

- although exists a critical ratio  $R = \frac{\text{constraints}}{\text{variables}}$  at which becomes very very bad

## Logic

Built off of a knowledge base: a set of sentences in some formal language



Add sentences to the knowledge base

Apply a process of inference to determine actions

Inference engine independent of the knowledge on which it acts

Inference algorithm/reasoning allows for universality: can act on any knowledge

Reduces to a question of considering the knowledge base

Syntax: rules for allowable sentences

Semantics: possible worlds, truth relation between sentences and worlds

E.g. Propositional Logic: possible worlds are assignments of TF to variables

Semantics:  $\alpha \wedge \beta$  is true in a world iff  $\alpha$  is true and  $\beta$  is true

E.g. First-order logic

Syntax  $\forall x \exists y P(x, y) \wedge \neg Q(Joe, f(x)) \rightarrow f(x) = f(y)$

Possible world:

Objects  $o_1, o_2, o_3$ ;  $P$  holds for  $\langle o_1, o_2 \rangle$ ;  $Q$  holds for  $\langle o_3 \rangle$ ;  $f(o_1) = o_1$ ;  $Joe = o_3$ ; etc.

Semantics:  $\phi(\sigma)$  is true if  $\sigma = o_j$  and  $\phi$  holds for  $o_j$

Entailment:  $\alpha \models \beta$ ,  $\alpha$  entails  $\beta$  or  $\beta$  follows from  $\alpha$

the  $\alpha$ -worlds are a subset of the  $\beta$ -worlds [ $\text{models}(\alpha) \subset \text{models}(\beta)$ ]

the entailment of  $\beta$  makes it at most as strong as  $\alpha$ , and possibly weaker

A proof is a demonstration of entailment

Method 1: check in every possible world, that if  $\alpha$  is true then  $\beta$  is true too

Semi-decidable: if cannot be proven in this fashion, has no way of indicating such

Method 2: exhibit a sequence of applications of inference rules taking  $\alpha$  to  $\beta$

Sound inference algorithm: everything that it claims to prove is entailed

Complete inference algorithm: everything that is entailed can be proven

Propositional logic

Have a set of symbols, with distinguished symbols 'True' and 'False'

Sentences are generated by the set of symbols under  $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$

Semantics: symbols have truth values, can recurse over syntax to evaluate sentences

Forward chaining: given  $X_1 \wedge X_2 \wedge \dots \wedge X_n \rightarrow Y$  and  $X_1, X_2, \dots, X_n$ , infer  $Y$

knowledge base only contains definite clauses (of the above form)

therefore, cannot deal with disjunctions; no reasoning by cases

forward chaining algorithm: have table counting number of symbols in each premise

iterate over symbols, decrement count for each clause that has that symbol in premise

allows for  $\mathcal{O}(n)$  where  $n$  is the size of the knowledge base

sound (since Modus Ponens is sound) and complete for definite-clause KBs

Simple model checking for entailment: recursive enumeration of all worlds

go through all possible worlds (sets of TF assignments) to all symbols

for all worlds where KB is true, make sure that  $\alpha$  is true

shows when  $\alpha$  is entailed, woefully inefficient ( $\mathcal{O}(2^n)$  time, linear space)

A sentence is satisfiable if it is true in at least one world

SAT solvers take a sentence in conjunctive normal form and determines satisfiability

using a SAT (satisfiability) solver to check entailment

if  $\alpha \models \beta$ , then  $\alpha \rightarrow \beta$  in all worlds

hence  $\neg(\alpha \rightarrow \beta)$  is false in all worlds

hence  $\alpha \wedge \neg\beta$  is false in all worlds

if can show that  $\alpha \wedge \neg\beta$  is unsatisfiable then  $\alpha \models \beta$

analogous to an proof by contradiction

DPLL SAT solver: backtracking search over models with:

early termination: stop immediately if all clauses are satisfied, any clause is falsified

conj. normal form will be e.g.  $(A \vee B) \wedge (A \vee \neg C)$ ; this has 2 clauses

pure literals: symbol always has same sign in to-go clauses, just assign it the value

e.g.  $(A \vee B) \wedge (A \vee \neg C) \wedge (C \vee \neg B)$  then set  $A$  to be true

unit clauses: clause left with single literal, set symbol to satisfy clause

DPLL efficient enough to solve up to 100 variables

Tricks to improve efficiency:

order variables and values (just like CSPs)

divide into pieces if you can see that two sections don't depend on each other

cache unsolvable subcases as extra clauses to avoid redoing them

with these improvements, can solve problems with ten million variables

## Logical Agents

Knowledge-based agent:

percepts added to knowledge base, after converted to some logical sentences

figures out what its next action shall be, performs action

then it adds to knowledge base the fact that it has performed this action

Initial knowledge possessed by the agent

Sensor model: how the current percept is generated from the current state

transition model: how the next-state determined by action, current state

initial conditions: initial state

domain constraints: certain conditions that are generally satisfied

Set the knowledge, and then the SAT-solver does all the work