General Formula

1 Agents

Have:
- Abilities
- Goals/Preferences
- Prior Knowledge
- Stimuli
- Past Experiences
- Actions

Belief State: Internal belief about the world **Knowledge**: Information used to solve tasks **Representation**: Data structure to encode knowledge **Knowledge Base**: Representation of all knowledge possessed **Model**: How KB relates to world

Dimensions of Complexity

1. Modularity
   - Flat: No modularity in computation
   - Modular: Each component is separateand siloed
   - Hierarchical: Modular components are broken into a hierarchical manner of subproblems
2. Planning Horizon:
   - Non-Planning: World doesn't change as a result (Ex: Protein Folding)
   - Finite: Reason ahead fixed number of steps
   - Indefinite: Reason ahead finite number (but undetermined) of steps
   - Infinite: Reason forever (focus on process)
3. Representation:
   - States: State describes how world exists
   - Features: An attribute of the world
   - Individuals and relations: How features relate to one another (Eg: child.failing() relates to child.grade)
4. Computational limits:
   - Perfect rationality: Agent always picks best action (Eg: Tic-Tac-Toe)
   - Bounded rationality: Agent picks best action given limited computation (Eg: Chess)
5. Learning:
   - Given knowledge (Eg Road laws)
   - Learned knowledge (Eg How car steers in rain)
6. Uncertainty:
   - Fully observable: Agent knows full state of world from observations (Eg: Chess)
   - Partially observable: Many states can lead to same representation (Eg: Battleship)
   - Deterministic: Action has predictable effect
   - Stochastic: Uncertainty exists over effect of action to state
7. Preference:
   - Achievement Goal: Goal to reach (binary)
   - Maintenance Goal: State to maintain
   - Complex Preferences: Complex tradeoffs between criteria and ordinality (can't please everyone)
8. Num Agents:
   - Single agent
   - Adversarial
   - Multiagent
9. Interactivity:
   - Offline: Compute its set of actions before agent has to act, so no computations required
   - Online: Computation is done between observing and acting

2 Search

b is branching factor (max num children of any node)
m is max depth of search-tree
d is depth of shallowest goal node

Search Problem:
- Set of states
- Initial state
- Goal function
- Successor function
- (optional) cost

Frontier: Ends of paths from start node that have been explored

Graph Search Algorithm

```
frontier is just start node
while frontier isn't empty
    select and remove path ⟨n_0, ..., n_k⟩ from frontier
    if goal(n_k) then
        return ⟨n_0, ..., n_k⟩
    for each neighbor n of n_k do
        add ⟨n_0, ..., n_k, n⟩
```

Uninformed Search

Depth-first-search

Use frontier as stack, always select last element **Cycle Check**: Check if current node exists within path you are Checking **Space Complexity**: $O(bm)$ **Time Complexity**: $O(b^m)$ **Completeness**: No **Optimal**: No

Use:
- Restricted space
- Many solutions with long paths

Don't use:
- Infinite paths exist
- Optimal solutions are shallow
- Multiple paths to node

Breadth-first-search

Use frontier as queue, always select first element **Multiple-Path Pruning**: Check if current node has been visited by any previous path by maintaining explored set. **Space Complexity**: $O(b^d)$ **Time Complexity**: $O(b^d)$ **Completeness**: Yes **Optimal**: No (only finds shallowest goal node)

Use:
- Space isn't restricted
- Want shallowest arc

Don't use:
- All solutions are deep
- Problem is large and graph is dynamically generated

Iterative-Deepening

For every depth-limit, perform depth-first-search, this marries BFS and DFS by "doing BFS" but without space-concerns. However, we end up revisiting nodes.

Space Complexity: $O(bd)$ **Time Complexity**: $O(b^d)$, $b^d \sum_{n=1}^{d} n\left(\frac{1}{b}\right)^{n-1} = b^d\left(\frac{b}{1-b}\right)^2$ We visit level i $d-i$ times, and level i has b^i nodes, so that's the sum, then extending to infinity, and use geometric series. **Completeness**: Yes **Optimal**: No (only finds shallowest goal node)

Use:
- Space isn't restricted
- Want shallowest arc

Don't use:
- All solutions are deep
- Problem is large and graph is dynamically generated

Lowest-Cost-First-Search

Select a path on frontier with lowest cost. Frontier is priority queue ordered by path cost. *Technically uninformed/blind search because it's searching randomly* **Space Complexity**: $O(b^d)$ **Time Complexity**: $O(b^d)$ **Completeness and optimality**: Yes if branching factor is finite and cost of every edge is strictly positive. **Termination**: Only terminate when the goal node is first on the frontier, not if it's in the frontier.

Dijkstra's

Similar to LCFS but keep track of lowest cost to reach each node, if we find lower cost path, update that value and resort the priority queue. Ex: Suppose we have path in frontier $\langle P, Q, R, \rangle$ and the found path to Q is 10 and overall cost is 12, then we find a new path to Q of cost 9, then we should recompute path to get cost 11.

Heuristic Search

$h(n)$ is estimate of cost of shortest path from n to a goal node. Should only use readily obtainable information and be **much** easier than solving the problem.

Greedy Best-First Search

Select path whose end is closest to a goal based on heuristic. Frontier is a priority queue ordered by h. **Space Complexity**: $O(b^d)$ **Time Complexity**: $O(b^d)$ **Completeness and optimality**: Not guaranteed (could be stuck in a cycle or return sub-optimal path)

Heuristic Depth-First-Search

Do Depth-First-Search, but add paths to stack ordered according to h. Basically, do DFS, but sort the children by h to determine who to check. Same complexity and problems as DFS but used often.

A* search

Use both path cost and heuristic values. Frontier is sorted by $f(p) = \text{cost(p)} + h(p)$. Always selects node with lowest estimated distance.

Space Complexity: $O(b^d)$ **Time Complexity**: $O(b^d)$ **Completeness and optimality**: Only with admissable heuristic, finite branching factor, and bounded arc-costs (there is a minimum positive arc-cost). A* always expands the fewest nodes for all optimal algorithms and use the same heuristic. No algorithm with same info can do better. This is because if an algorithm does not expand all nodes with $f(n) < \text{cost}(s, g)$ they might not find the optimal solution.

Admissable Heuristic

Never overestimates the shortest path from n to goal node.

Procedure for construction:
1. Define relaxed problem by simplifying or removing constraints
2. Solve relaxed problem without search
3. Cost of optimal solution to relaxed problem is admissable heuristic for original problem.

Dominating Heuristic

Given two heuristics, $h_2(n)$ dominates $h_1(n)$ if $\forall n h_2(n) \geq h_1(n)$ and $\exists n h_2(n) > h_1(n)$ We prefer dominating heuristics because it reduces the nodes we have to expand (they're bigger, so we don't care)

Monotone Restriction

A* guarantees finding optimal goal, but not necessarily shortest path. In order to do that, we would want our estimate path $f(p)$ to indeed allow us to remove longer paths, but what if one path has shorter cost, but heuristic sums make the shorter path have larger $f(p)$? We can avoid that, by inducing monotonic restriction. $h(n') - h(n) \leq \text{cost}(n', n)$. This guarantees heuristic estimate is always less than actual cost and if we ever find a shorter estimate, that estimate will actually be shorter, so we can prune it.

Further, monotonic restriction with multi-path pruning always finds shortest path to goal, not just optimal goal itself.

Note that admissability guarantees heuristic is never bigger than shortest path to goal, monotonicity ensures heuristic is never bigger than shortest path to any other node.

Summary

Strategy	Frontier Selection	Halt?	Space	Time
Depth-first	Last node added	No	Linear	Exp
Breadth-first	First node added	Yes	Exp	Exp
Heuristic Depth-first	Local min $h(n)$	No	Linear	Exp
Best-first	Global min $h(n)$	No	Exp	Exp
Lowest-cost-first	min cost(n)	Yes	Exp	Exp
A*	min $f(n)$	Yes	Exp	Exp

Adversarial Search (Minimax)

For one node look to maximize the heuristic, for the other node look to minimize it (to simulate the adversarial search)

- Alpha-beta pruning can ignore portions of search tree without losing optimality. Useful in application but doesn't change asymptotics
- Can stop early by evaluating non-leafs via heuristics (doesn't guarantee optimal play)

Higher-level strategies

- **Bidirectional Search**: Search from backward and forward simultaneously taking $2b^{\frac{k}{2}}$ vs b^k and try to find where frontiers match
- **Island-driven Search**: Find set of islands between s and g as mini problems. With m islands, you get $mb^{\frac{k}{m}}$ vs b^k but it's harder to guarantee optimality.

3 Constraints

Constraint Satisfaction Problems

- A set of variables
- Domain for each variable
- Two kinds of problems:
  - Satisfiability problems: Assignment satisfying hard constraints
  - Optimizatoin: Find assignment optimizing evaluation function (soft constraints)
- Solution is assignment to variables satisfying all constraints
- Solution is model of constraints

CSPs as graphs

Search spaces can be very large, path isn't important, only goal, and no set starting nodes make this bad idea **Complete Assignment**: Nodes: Assignment of value to all variables Neighbors: Change one variable value **Partial Assignment**: Nodes: Assignment to first $k-1$ variables Neighbors: Assignment to k^{th} variable

Constraints

- Can be **N-ary** (over sets of N variables) (Ex: A + B = C involves is 3-ary for 3 vars)

Generate and Test

Exhaust every possible assignment of vars and test validity

Backtracking

Order all variables and evaluate constraints in order as soon as they are fixed. (Ex: $A = 1 \wedge B = 1$ is inconsistent with $A \neq B$ so go to last assigned variable and change its value)

Consistency

Represent constraints as network to determine how all variables are related. **Domain Constraint**: Unary constraint on values in domain written $\langle X, c(X) \rangle$ (Eg: $B, B \neq 3$) **Domain Consistent**: A node is domain consistent if no domain value violates any domain constraint, and a network is domain consistent if all nodes are domain consistent. **Arc**: Arc $\langle X, c(X, Y) \rangle$ is a constraint on X **Arc Consistent**: Arc $\langle X, c(X, Y) \rangle$ is arc consistent if for every valid x there is a valid y such that constraint is satisfied. **Path Consistent**: A set of variables is path consistent if all arcs and domains are consistent.

AC-3

Make Consistency network arc consistent
- To-Do Arcs Queue contains all inconsistent arcs
- Make all domains domain consistent
- Put all arcs in TDA
- Repeat until TDA is empty:
  - Select and remove an arc from TDA
  - Remove all values of domain of X that don't have value in domain of Y that satisfy constraint
  - If any were removed, add all arcs to TDA

Termination:
- If every domain is empty, no solution
- If every domain has a single value, solution
- If some domain has more than one value, split in two run AC-3 recursively on two halves
- Guaranteed to terminate
- Takes $O(cd^3)$ time, with n variables, c binary constraints, and max domain size is d because each arc $\langle X_k, X_i \rangle$ can be added to queue at most d times because we can delete at most d values from X_i. Checking consistency takes $O(d^2)$ time.

Variable Elimination

- Eliminate variables one-by-one passing constraints to neighbours.
- When single variable remains, if no values exist then network was inconsistent.
- Variables are eliminated according to elimination ordering.

Pseudocode:
- If only one variable, return intersection of unary constraints referencing it
- Select variable X
  - Join constraints affecting X, forming constraint R
  - Project R onto its variables other than X, calling this R2
  - Place new constraint between all variables that were connected to X
  - Remove X
  - Recursively solve simplified problem
  - Return R joined with recursive solution

Local Search

- Maintain assignment of value to each variable
- At each step, select neighbor of current assignment
- Stop when satisfying assignment found or return best assignment found
- Heuristic function to be minimized: Number of conflicts
- Goal is an assignment with zero conflicts

Greedy Descent

Select some variable (through some method) and then select the value that minimizes the number of conflicts. THe problem is that we could be stuck in a local minimum, without reaching the proper global minimum.

Stochastic Local Search

Do Greedy descent, but allow some steps to be random, and the potential to restart randomly, to minimize potential for being stuck in local minimum.

Problem: in high dimensions often consist of long, nearly flat "canyons" so it's hard to optimize using local search.

Simulated Annealing

Pick variable at random, if it improves, adopt it. If it doesn't improve, then accept it with a probability through the temperature parameter, which can get slowly reduced.

Tabu Lists

Variant of Greedy Satisfiability, where to prevent cycling and getting stuck in local optimum, we maintain a "tabu list" of the k last assignments, and don't allow assignment that has already existed.

Parallel Search

- Total assignment is called individual
- Maintain population of k individuals
- At each stage, update each individual in population
- Whenever individual is a solution, it can be reported
- Similar to k restarts, but uses k times minimum number of steps

Beam Search

- Like parallel search, with k individuals, but choose the k best out of all the neighbors. The value of k can limit space and induce parallelism

Stochastic Beam Search

- Like beam search, but probabilistically choose k individualls at next generation. Probability of selecting neighbor is proportional to heuristic: $e^{-\frac{h(n)}{T}}$. This maintains diversity among the individuals, because it's similar to simulated annealing.

Genetic Algorithms

- Like stochastic beam search, but pairs of individuals are combined to create offspring.
- For each generation, randomly choose pairs where fittest individuals are more likely selected
- For each pair, do cross-over (form two offspring as mutants of parents)
- Mutate some values
- Stop when solution is found

Comparing Algorithms

Since some algorithms are super fast some of the time and super slow other times, and others are mediocre all of the time, how do you compare? You use runtime distribution plots to see the proportion of runs that are solved within a specific runtime.

4 Inference and planning

Problem Solving

Procedural solving: Devise algorithm, program, execute **Declarative solving**: Identify required knowledge, encode knowledge in representation, use logical consequences to solve.

Logic

Syntax: What is an acceptable sentence **Semantics**: What do the sentences and symbols mean? **Proof procedure**: How to construct valid proofs? **Proof**: Sequence of sentences derivable using an inference recursively

Statements/Premises: $\{X\}$ is a set of statements or premises, made up of propositions. **Interpretation**: Set of truth assignments to propositions in $\{X\}$ **Model**: Interpretation that makes statements true **Inconsistent statements**: No model exists **Logical Consequence**: If for every model of $\{X\}$, A is true, then A is a logical consequence of $\{X\}$

Argument Validity is satisfied if any of the identical statements are true:
- Conclusions are a logical consequence of premises
- Conclusions are true in every model of premises
- No situation in which the premises are all true but the conclusions are false.
- Arguments → conclusions is a **tautology** (always true)

Proof

Knowledge Base: Set of axioms **Derivation**: KB $\vdash g$ can be found from KB using proof procedure **Theorem**: If KB $\vdash g$, then g is a theorem **Sound**: Proof procedure is sound if KB $\vdash$ g then KB $\models g$ (anything that can be proven must be true (sound reasoning)) **Complete**: Proof procedure is sound if KB $\models$ g then KB $\vdash g$ (anything that is true can be proven (complete proof system))

Complete Knowledge: Assume a closed world where **negation** implies failure since we can't prove it, if it's open there are true things we don't know, so if we can't prove something, we can't decide if it's true or false.

Bottom-up Proof (aka forward chaining)

Start from facts and use rules to generate all possible derivable propositions

To prove: $F \leftarrow A \wedge E$ $A \leftarrow B \wedge C$ $A \leftarrow D \wedge C$ $E \leftarrow C$ D C

Steps of proof: $\{D, C\} \rightarrow \{D, C, E\} \rightarrow \{D, C, E, A\} \rightarrow \{D, C, E, A, F\}$ Therefore, if g is an atom, KB $\vdash$ g, if g $\in C$ at the end of the procedure, where C is the consequence set.

Top-Down

Start from query and work backwards yes $\leftarrow F$ yes $\leftarrow A \wedge E$ yes $\leftarrow D \wedge C \wedge E$ yes $\leftarrow D \wedge C \wedge C$ yes $\leftarrow D \wedge C$ yes $\leftarrow D$ yes $\leftarrow$

Individuals and Relations

KB can contain **relations**: part_of(C, A) is true if C is a "part of" A KB can contain **quantification**: part_of(C, A) holds $\forall C, A$ Proofs are the same with extra bits for handling relations & quantification.

Planning

Decide sequence of actions to solve goal based on abilities, goal, state of the world Assumptions:
- Single agent
- Deterministic
- No exogenous events
- Fully-observable state
- Time progresses discretely from one state to another
- Goals are predicates of states to achieve or maintain (no complex goals)

Action: Partial function from state to state **Partial Function**: Some actions are not possible in some states, preconditions specify when action is valid, and effect determines next state

Feature-based representation of actions: For each action, there is a precondition (proposition) that specifies when action is valid. **Causal Rule**: When feature gets a new value **Frame Rule**: When feature keeps its value *Features are capitalized, but values aren't If X is a feature, X' is feature after an action*

Forward Planning: Search in state-space graph, where nodes are states, arcs are actions, and a plan is a path representing initial state to goal state. **Regression Planning**: Search backwards from goal, nodes correspond to subgoals and arcs to actions. Nodes are propositions (formula made of assignment of values to features), arcs are actions that can achieve one of the goals. Neighbors of node N associated with arc specify what must be true immediately before A so that N is true immediately after. Start node is goal to be achieved. Goal(N) is true if N is a proposition true of initial state.