Artificial Intelligence Modeling and Problem Solving

Hector Geffner
Universidad Simón Bolívar
Caracas, Venezuela
9/2000

Motivation

- Development of **General Problem Solvers** has been a main goal in Artificial Intelligence

Problem $\longrightarrow$ *Language* $\longrightarrow$ $\boxed{\begin{array}{c}\textit{Representation}\\ +\\ \textit{Algorithms}\end{array}}$ $\longrightarrow$ *Solution*

- Ideally one would describe problem at high-level, and computer would take care of the rest
- Example problems

8-puzzle	*rubik*	*mastermind*	*12 coins*
diagnosis	*scheduling*	*tsp*	*robot navig*
sorting	*minesweeper*	*crypto-arith*	*vehicle routing*
...	...		

Methodology

how to make sense of this variety of problems?

- Many approaches pursued; many useful ideas and techniques
- To a certain extent AI has become large **bag of tools and techniques**
- This is unfortunate for both teaching and research
- Goal of course is to provide **coherent framework** for AI modeling and problem solving
- Coverage is broad but necessarily incomplete

Coherent framework for AI Modeling and Problem Solving

Three distinguished components:

- **Representation languages** for describing problems conveniently
- **Mathematical models** for making sense of classes of problems
- **Algorithms** for solving these models

Mathematical Models: Example

State models are very common in AI and determined by

- finite and discrete state space S
- an initial state $s_0 \in S$
- a set $S_G \subseteq S$ of goal states
- actions $A(s) \subseteq A$ applicable in each state $s \in S$
- a transition function $f(a, s)$ for $s \in S$ and $a \in A(s)$
- action costs $c(a, s) > 0$

- A *solution* for this model is a sequence of applicable actions that maps the initial state s_0 into a goal state $s \in S_G$
- Problems like TSP, Sliding puzzles, and Rubik's cube can be naturally modeled and solved as **state models**
- Problems like Mastermind, Robot Navigation, Scheduling, and others do not

Some mathematical models we will consider

- State Models (SMs)
  - *simplest type of models*
- Non Deterministic and Probabilistic State Models
  - *more complex* **dynamics**
- Partially Observable Non-Det and Prob State Models
  - *more complex sensing (***feedback***)*
- Scheduling/Temporal Models
  - **time**, *action durations, concurrency, resources*

Scope of course

With the aid of this range of models, we will cover main ideas underlying current work in

- Problem Solving
- Planning
- Sequential Decision Making

Some relevant things we will **not** cover:

- Multi-agent systems; require different models/algorithms
- Problem solving techniques based on use of neural networks, genetic algorithms, . . . ; connection is weak and lot of 'art' involved
- Constraint Programming

Summary

Approach to AI Modeling and Problem Solving based on

- **languages** for describing problems conveniently
- **models** for making sense of problems
- **algorithms** for solving models

Underlying unity among models, languages, and algorithms to be considered. Scope will be broad with emphasis on

- coherent perspective
- ideas that appear to work best (so far)

Plan

- **Class 1:** State Models and Search Algorithms
- **Class 2:** Classical Planning (simplest action model)
- **Class 3:** Planning & Control (uncertainty/feedback)

State Models

- State models familiar in AI
- They are characterized by
  - finite and discrete state space S
  - an initial state $s_0 \in S$
  - a set $G \subseteq S$ of goal states
  - actions $A(s) \subseteq A$ applicable in each state $s \in S$
  - a transition function $f(s, a)$ for $s \in S$ and $a \in A(s)$
  - action costs $c(a, s) > 0$
- A *solution* is a sequence of applicable actions a_i, $i = 0, \ldots, n$, that maps the initial state s_0 into a goal state $s \in S_G$; i.e., $s_{n+1} \in S_G$ and for $i = 0, \ldots, n$

  $$s_{i+1} = f(a, s_i) \text{ and } a_i \in A(s_i)$$

- Optimal solutions minimize total cost $\sum_{i=0}^{i=n} c(a_i, s_i)$

Algorithms for solving State Models

Search algorithms explore/visit state space trying to find (optimal) path from s_0 to S_G

- **Blind search/Brute force algorithms**
  - Goal plays **passive** role in the search

    e.g., Depth First Search (DFS), Breadth-first search (BrFS), Uniform Cost (Dijkstra), Iterative Deepening (ID)

- **Informed/Heuristic Search Algorithms**
  - Search uses a function $h(s)$ that estimates 'distance' (cost) from state s to S_G to guide search

    e.g., A, IDA*, Hill Climbing, Best First Search (BFS), Branch & Bound*

General Search Scheme

```
Solve(Nodes)
  if Empty Nodes  -> Fail
  else
    Let Node = Select-Node Nodes
    Let Rest =  Nodes - Node

    if Node is Goal -> Return Solution
    else
      Let Children = Expand-Node Node
      Let New-Nodes = Add-Nodes Children Nodes
      Solve(New-Nodes)
```

- Different algorithms obtained by suitable instantation of
  - `Select-Node` *Nodes*
  - `Add-Nodes` *New-Nodes Old-Nodes*
- Nodes are data structures that contain state and bookkeeping info; initially `Nodes` $= \{root\}$
- Notation $g(n)$, $h(n)$, $f(n)$: accumulated cost, heuristic and evaluation function; e.g. in A*, $f(n) \stackrel{\text{def}}{=} g(n) + h(n)$

Some instances of general search scheme

Depth-First Search expands 'deepest' nodes n first

- `Select-Node` $Nodes$: Select First Node in $Nodes$
- `Add-Nodes` New Old: Puts New before Old
- Implementation: Nodes is a **Stack** (LIFO)

Breadth-First Search expands 'shallowest' nodes n first

- `Select-Node` $Nodes$: Selects First Node in $Nodes$
- `Add-Nodes` New Old: Puts New after Old
- Implementation: Nodes is a **Queue** (FIFO)

Additional instances of general search scheme

Best First Search expands best nodes n first; $\min f(n)$

- `Select-Node` $Nodes$: Returns n in Nodes with min $f(n)$
- `Add-Nodes` $New\ Old$: Performs ordered merge
- Implementation: Nodes is a **Heap**
- Special cases
  - **Uniform cost/Dijkstra**: $f(n) = g(n)$
  - **A***: $f(n) = g(n) + h(n)$
  - **WA*:** $f(n) = g(n) + Wh(n)$, $W \geq 1$

Hill Climbing expands best node n first and **discards others**

- `Select-Node` $Nodes$: Returns n in Nodes with min $h(n)$
- `Add-Nodes` $New\ Old$: Returns New; discards Old

Variations of general search scheme: Bounding

```
Solve(Nodes,Bound)

  if Empty Nodes  -> Report-Best-Solution-or-Fail
  else
     Let Node = Select-Node Nodes
     Let Rest =  Nodes - Node

    if f(n)  > Bound
           Solve(Rest)    ;;;    PRUNE NODE n

    else if Node is Goal -> Process-Solution Node Rest
         else
           Let Children = Expand-Node Node
           Let New-Nodes = Add-Nodes Children Nodes
           Solve(New-Nodes)

    ;; Select-Node & Add-Nodes as in DFS
```

Some instances of general bounded search scheme

Iterative Deepening (ID)

- Uses $f(n) = g(n)$
- Calls `Solve` with bounds 0, 1, .. til solution found
- `Process-Solution` returns Solution

Iterative Deepening A* (IDA*)

- Uses $f(n) = g(n) + h(n)$
- Calls `Solve` with bounds $f(n_0)$, $f(n_1)$, $\ldots$ where $n_0 = root$ and n_i is cheapest node pruned in iteration $i-1$
- `Process-Solution` returns Solution

Branch and Bound

- Uses $f(n) = g(n) + h(n)$
- Single call to `Solve` with high (Upper) `Bound`
- `Process-Solution:` updates `Bound` to Solution Cost minus 1 & calls `Solve(Rest,New-Bound)`

Properties of Algorithms

- **Completeness**: whether guaranteed to find solution
- **Optimality**: whether solution guaranteed optimal
- **Time Complexity**: how time increases with size
- **Space Complexity:** how space increases with size

	DFS	BrFS	ID	A*	HC	IDA*	B&B
Complete	No	Yes	Yes	Yes	No	Yes	Yes
Optimal	No	Yes*	Yes	Yes	No	Yes	Yes
Time	∞	b^d	b^d	b^d	∞	b^d	b^D
Space	$b \cdot d$	b^d	$b \cdot d$	b^d	b	$b \cdot d$	$b \cdot d$

– Parameters: d is solution depth; b is branching factor

– BrFS optimal when costs are uniform

– A*/IDA* optimal when h is **admissible**; $h \leq h^*$

A*: Additional Properties

- A* stores in memory **all nodes visited**
- Nodes either in **Open** (search frontier) or **Closed**
- When nodes expanded, children looked up in **Open** and **Closed** lists
- Duplicates prevented and no node expanded more than once

– A* is **optimal** in another sense: no other algorithm expands less nodes than A* with same heuristic function *(this doesn't mean that A* is always fastest)*

– A* expands 'less' nodes with **more informed heuristic**, h_2 more informed that h_1 if $0 < h_1 < h_2 \leq h^*$

Practical Issues: Search in Large Spaces

- Exponential-memory algorithms like A* **not feasible** for large problems
- **Time and memory** requirements can be lowered significantly by multiplying heuristic term $h(n)$ by a constant $W > 1$ (WA*)
- Solutions **no longer optimal** but at most W times from optimal
- For large problems, only feasible optimal algorithms are **linear-Memory** algorithms such as IDA* and B&B
- Linear-memory algorithms often use **too little memory** and may visit fragments of search space many times
- It's common to extend IDA* in practice with so-called **transposition tables**
- Optimal solutions have been reported to problems with **huge state spaces** such 24-puzzle, Rubik's cube, and Sokoban (Korf, Schaeffer); e.g. $|S| > 10^{25}$
- Key issues: heuristics, representation, symmetries, use of memory, branching rules, ...

Heuristics: where they come from?

- General idea: heuristic functions obtained as **optimal cost functions** of **relaxed problems**
- Examples:
  - *Manhattan distance in N-puzzle*
  - *Euclidean Distance in Routing Finding*
  - *Spanning Tree in Traveling Salesman Problem*
  - *Shortest Path in Job Shop Scheduling*
- Yet
  - how to get and solve suitable relaxations?
  - how to get heuristics automatically?

We'll get back to this in Planning . . .

Memory-Based Heuristics: Pattern Databases (Culberson & Schaeffer 95)

- A state is normally assignment of values to variables; e.g., in 15-puzzle, state assigns locations to tiles
- Size of state space is roughly the **number of variables** times the **size of their domains**
- In many problems, the state space S of a problem can be **projected** into smaller state space S' by focusing on values of **subset of variables**
- E.g., in 15-puzzle, one can **project** problem by focusing only on location of tiles $T0$, ..., $T3$ and $Blank$
- Projected problem has much smaller state space and can be solved exhaustively
- Cost function H^* of **projected problem** can be **computed for all projected states** and **stored in memory**
- This **pattern databases** can be used to define or refine **admissible heuristics**

Pattern Databases (cont.)

- Denote projection of state s as $p(s)$, and let $H^*(p(s))$ denote optimal cost function on projected space
- $h(s) \stackrel{\text{def}}{=} H^*(p(s))$ is an **admissible** heuristic
- More informed heuristics can be obtained by combining **several** pattern databases H_i^* over **different projections** p_i as

$$h(s) \stackrel{\text{def}}{=} \max_i \; H_i^*(p_i(s))$$

- Resulting heuristic is also admissible

Summary

- Coherent framework for AI modeling and problem solving
- Three main ingredients
  - **representation languages** for describing problems
  - **mathematical models** for making sense of problems
  - **algorithms** for solving models
- So far we've focused on **State Models** and **Algorithms** for solving them
- No **general problem solving** yet, but specialized solvers for problems that map into state models
- Next: **Planning**
  - incorporation of languages for representing problems
  - automatic extraction of heuristics