Course on Automated Planning: Planning and Heuristic Search

Hector Geffner
ICREA & Universitat Pompeu Fabra
Barcelona, Spain
Models, Languages, and Solvers

- A **planner** is a **solver over a class of models**; it takes a model description, and computes the corresponding controller

  \[ \text{Model} \implies \text{Planner} \implies \text{Controller} \]

- Many models, many solution forms: uncertainty, feedback, costs, . . .

- Models described in suitable **planning languages** (Strips, PDDL, PPDDL, . . .) where **states** represent interpretations over the language.
State Model for Classical Planning

- 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 $\sum_{i=0}^{i=n} c(a_i, s_i)$
Language for Classical Planning: Strips

• A problem in Strips is a tuple \( P = \langle F, O, I, G \rangle \):
  
  ▶ \( F \) stands for set of all atoms (boolean vars)
  ▶ \( O \) stands for set of all operators (actions)
  ▶ \( I \subseteq F \) stands for initial situation
  ▶ \( G \subseteq F \) stands for goal situation

• Operators \( o \in O \) represented by
  
  ▶ the Add list \( Add(o) \subseteq F \)
  ▶ the Delete list \( Del(o) \subseteq F \)
  ▶ the Precondition list \( Pre(o) \subseteq F \)
From Problem $P$ to State Model $S(P)$

A Strips problem $P = \langle F, O, I, G \rangle$ determines state model $S(P)$ where

- the states $s \in S$ are **collections of atoms** from $F$
- the initial state $s_0$ is $I$
- the goal states $s$ are such that $G \subseteq s$
- the actions $a$ in $A(s)$ are ops in $O$ s.t. $Prec(a) \subseteq s$
- the next state is $s' = s - Del(a) + Add(a)$
- action costs $c(a, s)$ are all 1

- (Optimal) **Solution** of $P$ is (optimal) **solution** of $S(P)$
- Thus $P$ can be solved by solving $S(P)$
Solving $P$ by solving $S(P)$

Search algorithms for planning exploit the correspondence between (classical) states model and directed graphs:

- The nodes of the graph represent the states $s$ in the model
- The edges $(s, s')$ capture corresponding transition in the model with same cost

In the planning as heuristic search formulation, the problem $P$ is solved by path-finding algorithms over the graph associated with model $S(P)$
Search Algorithms for Path Finding in Directed Graphs

- **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**
  - Goals play **active** role in the search through **heuristic function** $h(s)$ that estimates cost from $s$ to the goal
    - e.g., *A**, *IDA**, *Hill Climbing*, *Best First*, *DFS B&B*, *LRTA**, . . .
General Search Scheme

\[
\text{Solve}(\text{Nodes})
\]

\[
\begin{align*}
\text{if Empty Nodes} & \rightarrow \text{Fail} \\
\text{else Let Node} & \Rightarrow \text{Select-Node} \; \text{Nodes} \\
& \quad \text{Let Rest} = \text{Nodes} - \text{Node} \\
& \quad \text{if Node is Goal} \rightarrow \text{Return Solution} \\
& \quad \text{else Let Children} \Rightarrow \text{Expand-Node} \; \text{Node} \\
& \quad \quad \text{Let New-Nodes} = \text{Add-Nodes} \; \text{Children} \; \text{Rest} \\
& \quad \quad \text{Solve}(\text{New-Nodes})
\end{align*}
\]

- Different algorithms obtained by suitable instantiation of
  - Select-Node \textit{Nodes}
  - Add-Nodes \textit{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*, \( f(n) \overset{\text{def}}{=} g(n) + h(n) \)
Some instances of general search scheme

- **Depth-First Search** expands ‘deepest’ nodes $n$ first
  - Select-Node $\textit{Nodes}$: Select First Node in $\textit{Nodes}$
  - Add-Nodes $\textit{New Old}$: Puts $\textit{New}$ before $\textit{Old}$
  - Implementation: Nodes is a **Stack** (LIFO)

- **Breadth-First Search** expands ‘shallowest’ nodes $n$ first
  - Select-Node $\textit{Nodes}$: Selects First Node in $\textit{Nodes}$
  - Add-Nodes $\textit{New Old}$: Puts $\textit{New}$ after $\textit{Old}$
  - 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***: \( f(n) = g(n) + Wh(n), W \geq 1 \)
    - **Greedy Best First**: \( f(n) = h(n) \)

- **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: DFS Bounding

Solve(Nodes,Bound)

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

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

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

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 $n_0 = \text{root}$ 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

<table>
<thead>
<tr>
<th></th>
<th>DFS</th>
<th>BrFS</th>
<th>ID</th>
<th>A*</th>
<th>HC</th>
<th>IDA*</th>
<th>B&amp;B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Optimal</td>
<td>No</td>
<td>Yes*</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Time</td>
<td>$\infty$</td>
<td>$b^d$</td>
<td>$b^d$</td>
<td>$b^d$</td>
<td>$\infty$</td>
<td>$b^d$</td>
<td>$b^D$</td>
</tr>
<tr>
<td>Space</td>
<td>$b \cdot d$</td>
<td>$b^d$</td>
<td>$b \cdot d$</td>
<td>$b^d$</td>
<td>$b$</td>
<td>$b \cdot d$</td>
<td>$b \cdot d$</td>
</tr>
</tbody>
</table>

- 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 as 24-puzzle, Rubik’s cube, and Sokoban (Korf, Schaeffer); e.g. $|S| > 10^{25}$
Learning Real Time A* (LRTA*)

- LRTA* is a very interesting real-time search algorithm (Korf 90)
- It’s like a hill-climb or greedy search that updates the heuristic $V$ as it moves, starting with $V = h$.

1. **Evaluate** each action $a$ in $s$ as $Q(a, s) = c(a, s) + V(s')$
2. **Apply** action $a$ that minimizes $Q(a, s)$
3. **Update** $V(s)$ to $Q(a, s)$
4. **Exit** if $s'$ is goal, else go to 1 with $s := s'$

- Two remarkable properties
  - Each trial of LRTA gets eventually to the goal if space connected
  - Repeated trials eventually get to the goal optimally, if $h$ admissible!

- Generalizes well to stochastic actions (MDPs)
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 more into this as we get back to planning . . .