Automated Planning and Decision Making
Prof. Ronen Brafman

Search
Introduction

- In the past, most problems you encountered were such that the way to solve them was very clear and straightforward:
  - Solving quadratic equation.
  - Solving linear equation.
  - Matching input to a regular expression.
- Most of the problems in AI are of different kind.
  - NP-Hard problems.
  - Require searching in a large solution space.
Search Problem

- $S$ – a set of possible states. The search space.
- $s_i \in S$ the initial state
- Operators: functions from $S$ to $S$. Map $s$ to $o(s)$.
- Goal Conditions.
The Search Space

- A graph $G(V, E)$ where:
  - $V = \{v : v \in S\}$
  - $E = \{\langle v, u \rangle : \exists o \in O \text{ such that } o(v) = u\}$
The Task

▶ Find a path in the search space from $s_i$ to a state $s$ that satisfies the goal conditions.
  ▶ Sometimes we only care about $s$, and not the path.
  ▶ Sometimes we only care whether $s$ exists.
  ▶ Sometimes value is associated with states, and we want the best $s$.
  ▶ Sometimes cost is associated with operators, and we want the cheapest path.
Finding a solution to Rubik’s Cube.

$S_i$: Some state of the cube.

Operators: 90° rotation of any of the 9 plains.

Goal Condition: same color for each of the cube’s sides.

The search space will include:

- A node for any possible state of the cube.
- An edge between two nodes if you can reach from one to the other by a 90° rotation of some plate.
Another Example

States: Locations of tiles.
Operators: Exchange blank with right/left/down/up neighbor.
Goal Conditions: As in the picture.
Note: optimal solution of n-puzzle family is NP-hard
Example: \( n \) queens problem:
A constraint satisfaction problem

States: legal placements of \( k \leq n \) queens
Operators: add a queen
Goal condition: \( n \) queens placed with no conflicts
(no constraint violated)
Example: $n$ queens problem: An alternative formulation

States: placements of $n$ queens one per column
Operators: move one of the queens
Goal condition: $n$ queens placed with no conflicts
(no constraint violated)
Solving Search Problems

- Apply the operators on the states in order to expand (produce) new states, until we find one that satisfies the goal condition.
- At any moment we may have different options to proceed (multiple operators may apply)
- Q: In what order should we expand the States?
- Our choice affects:
  - Computation time
  - Space used
  - Whether we reach an optimal solution
  - Whether we are guaranteed to find a solution if one exists (completeness)
Search Tree

▶ You can think of the search as generating a tree:
  ▶ Root: The initial state.
  ▶ Children of node $v$ are all states reachable from $v$ by applying an operator.
▶ We can reach the same state via different paths.
  ▶ In such a case, we can ignore this duplicate state, and treat it as a leaf node.
    ▶ Requires that we remember all states we visited!
▶ Important parameters affecting performance:
  ▶ $b$ Average branching factor.
  ▶ $d$ Depth of the closest solution.
  ▶ $m$ Maximal depth of the tree.
▶ Fringe: set of current leaf (unexpanded) nodes
Search Methods

- Different search methods differ in the order in which they visit/expand the nodes.

- It is important to distinguish between:
  - The search space
  - The order by which we scan that space
  - These are orthogonal issues!

- Some algorithms are described abstractly by describing the space they generate w/o specifying how it is searched.
  - All you need to provide is: the initial state (root) and the operators. This defines the tree.
  - You can implement them in different ways by choosing different search algorithms

- Blind Search
  - No additional information about search states is used.

- Informed Search
  - Additional information used to improve search efficiency
Relation to Planning

- Simplest (and currently most popular) way to solve planning problem is to search from the initial state to a goal state
- Search states are states of the world
- Operators = actions
- There are other formulations!
  - For instance: search states = plans
- Search is a very general technique very important for many applications beyond planning
Blind Search
Blind Search

- Main algorithms:
  - **DFS** Depth First Search
    - Expand the deepest unexpanded node.
    - Fringe is a LIFO.
  - **BFS** Breath First Search
    - Expand the shallowest unexpanded node.
    - Fringe is a FIFO.
  - **IDS** Iterative Deepening Search
    - Combines the advantages of both methods.
    - Avoids the disadvantages of each method.

- There are many other variants (e.g., optimizing disk access, parallel search, etc.)
Breadth First Search
Properties of BFS

- Complete?
  Yes (if b is finite).

- Time?
  \[1 + b + b^2 + b^3 + \cdots + b(b^d - 1) = O(b^{d+1})\]

- Space?
  \(O(b^d)\) (keeps every node on the fringe).

- Optimal?
  Yes (if cost is 1 per step).

- Space is a BIG problem...
Breadth-first search without duplicate detection

Breadth-first search

\[
\begin{align*}
queue & := \textbf{new} \ \text{fifo-queue} \\
queue & .\text{push-back}(\text{make-root-node}(\text{init}())) \\
\textbf{while not} \ queue & .\text{empty}(): \\
\quad & \sigma = queue.\text{pop-front}() \\
\quad & \textbf{if} \ \text{is-goal}(\text{state}(\sigma)):\ \\
\quad & \quad \textbf{return} \ \text{extract-solution}(\sigma) \\
\quad & \textbf{for each} \ \langle o, s \rangle \in \text{succ}(\text{state}(\sigma)):\ \\
\quad & \quad \sigma' := \text{make-node}(\sigma, o, s) \\
\quad & \quad queue.\text{push-back}(\sigma') \\
\textbf{return} & \ \text{unsolvable}
\end{align*}
\]

- Possible improvement: duplicate detection (see next slide).
- Another possible improvement: test if \( \sigma' \) is a goal node; if so, terminate immediately. (We don’t do this because it obscures the similarity to some of the later algorithms.)
Breadth-first search with duplicate detection

```
queue := new fifo-queue
queue.push-back(make-root-node(init()))
closed := ∅
while not queue.empty():
    σ = queue.pop-front()
    if state(σ) ∉ closed:
        closed := closed ∪ {state(σ)}
        if is-goal(state(σ)):
            return extract-solution(σ)
        for each (o, s) ∈ succ(state(σ)):
            σ’ := make-node(σ, o, s)
            queue.push-back(σ’)
return unsolvable
```
Depth First Search
Properties of DFS

- **Complete?**
  - No if given infinite branches (e.g., when we have loops)
  - Easy to modify: check for repeat states along path
    - Complete in finite spaces!
    - May require large space to maintain list of visited states

- **Time?**
  - Worst case: $O(b^m)$.
  - Terrible if $m$ is much larger than $d$.
  - But if solutions occur often or in the left part of the tree, may be much faster than BFS.

- **Space?**
  - $O(m)$ Linear Space!

- **Optimal?**
  - No.
Depth Limited Search

- DFS with a depth limit $l$, i.e., nodes at depth $l$ have no successors.
Iterative Deepening Search

- Increase the depth limit of the DLS with each iteration:

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
```
Iterative Deepening Search
Complexity of IDS

- Number of nodes generated in a depth-limited search to depth \(d\) with branching factor \(b\):
  \[ N_{DLS} = b^0 + b^1 + b^2 + \cdots + b^{d-2} + b^{d-1} + b^d \]

- Number of nodes generated in an iterative deepening search to depth \(d\) with branching factor \(b\):
  \[ N_{IDS} = (d + 1)b^0 + db^1 + (d - 1)b^2 + \cdots + 3b^{d-2} + 2b^{d-1} + b^d \]

- For \(b = 10, \ d = 5\):
  - \(N_{DLS} = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111\)
  - \(N_{IDS} = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456\)

- Overhead = \((123,456 - 111,111)/111,111 = 11\%\)
Properties of IDS

- Complete?
  Yes.
- Time?
  \[(d + 1)b^0 + db^1 + (d - 1)b^2 + \cdots + b^d = O(b^d)\]
- Space?
  \[O(d)\]
- Optimal?
  Yes. If step cost is 1.
Informed Search
Informed/Heuristic Search

- **Heuristic Function** $h : S \rightarrow R$
  - For every state $s$, $h(s)$ is an estimate of the minimal distance/cost from $s$ to a solution.
  - Distance is only one way to set a price.
  - How to produce $h$? later on...

- **Cost Function** $g : S \rightarrow R$
  - For every state $s$, $g(s)$ is the minimal cost to $s$ from the initial state.
  - $f = g + h$, is an estimation of the cost from the initial state to a solution.
Properties of Heuristics

- $h$ is **perfect** if $h(s) =$ shortest distance to goal from $s$ (and $\infty$ if goal is unreachable from $s$)
- The perfect heuristic is denoted by $h^*$
- $h$ is **safe** if $h(s) = \infty$ implies $h^*(s) = \infty$
- $h$ is **goal-aware** if $h(s) = 0$ for every goal state $s$
- $h$ is **admissible** if $h(s) \leq h^*(s)$ for all $s$
- $h$ is **consistent** if $h(s) \leq h(s') + 1$ whenever $s'$ is a child of $s$
  - More generally, $h(s) \leq h(s') + c(s, s')$
  - Also called **monotone** because if $h$ is consistent, $f = g + h$ never decreases.
Best First Search

- Greedy on $h$ values.
- Fringe stored in a queue ordered by $h$ values.
- In every step, expand the "best" node so far, i.e., the one with the best $h$ value.
Straight-line distance to Bucharest

Arad: 366
Bucharest: 0
Craiova: 160
Dobrota: 242
Eforie: 161
Fagaras: 176
Giurgiu: 77
Hirsova: 151
Iasi: 226
Lugoj: 244
Mehadia: 241
Neamt: 234
Oradea: 380
Pitesti: 10
Rimnicu Vilcea: 193
Sibiu: 253
Timisoara: 329
Urziceni: 80
Vaslui: 199
Zerind: 374
Reaching Bucharest with BFS
Greedy best-first search

Greedy best-first search (with duplicate detection)

\[
\begin{align*}
\text{open} & := \text{new min-heap ordered by } (\sigma \mapsto h(\sigma)) \\
\text{open}.\text{insert}(\text{make-root-node}(\text{init}())) \\
\text{closed} & := \emptyset \\
\text{while not } \text{open}.\text{empty}(): \\
& \quad \sigma = \text{open}.\text{pop-min()} \\
& \quad \text{if } \text{state}(\sigma) \notin \text{closed}: \\
& \quad \quad \text{closed} := \text{closed} \cup \{\text{state}(\sigma)\} \\
& \quad \quad \text{if } \text{is-goal}(\text{state}(\sigma)): \\
& \quad \quad \quad \text{return } \text{extract-solution}(\sigma) \\
& \quad \quad \text{for each } (o, s) \in \text{succ}(\text{state}(\sigma)): \\
& \quad \quad \quad \sigma' := \text{make-node}(\sigma, o, s) \\
& \quad \quad \quad \text{if } h(\sigma') < \infty: \\
& \quad \quad \quad \quad \text{open}.\text{insert}(\sigma') \\
\text{return unsolvable}
\end{align*}
\]
Properties of Best First

- Complete?
  - No, can get into loops
  - Yes, with duplicate detection and safe $h$
- Time?
  - $O(b^m)$. But a good heuristic can give dramatic improvement.
- Space?
  - $O(b^m)$. Keeps all nodes in memory.
- Optimal
  - No.
A* Search

▸ Idea: Avoid expanding paths that are already expensive.
▸ Best First Search that is greedy on $f$ values.
▸ Fringe is stored in a heap ordered by $f$ values.
▸ Recall, $f(n) = g(n) + h(n)$, where:
  ▸ $g(n)$: cost so far to reach $n$.
  ▸ $h(n)$: estimated cost from $n$ to goal.
  ▸ $f(n)$: estimated total cost of path through $n$ to goal.
▸ Termination Condition: the top node in the open list is a solution.
  ▸ Notice that you do not terminate when you first generate a solution, only when it is extracted from the open list in order to expand it.
Bucharest via $A^*$
Bucharest via $A^*$
## Admissible Heuristics

- A heuristic \( h(n) \) is **admissible** if for every node \( n \), \( h(n) \leq h^*(n) \), where \( h^*(n) \) is the real cost to reach the goal state from \( n \).

- An admissible heuristic **never overestimates** the cost to reach the goal, i.e., it is **optimistic**

- **Theorem:**
  If \( h(n) \) is admissible, \( A^* \) using Tree-Search is optimal.
  - Tree search: nodes encountered more than once are not treated as leaf nodes. They are re-expanded.
  - Notice that in tree-search: node = path (because there is a single path in a tree from a node to the root)
Optimality of $A^*$ (Proof)

- Suppose some suboptimal goal $G_2$ has been generated and it is in the fringe (= open list).
- The open list must also contain some node $n$ that is on the shortest path to the optimal goal $G$ node.
- Claim: $h(n) < h(G_2)$ ($\rightarrow n$ is expanded before $G_2$).
Optimality of $A^*$ (Proof)

1. $f(G_2) = g(G_2)$ since $h(G_2) = 0$.
2. $f(G) = g(G)$ since $h(G) = 0$.
3. $f(G_2) > g(G)$ since $G_2$ is sub optimal.
4. $f(G_2) > f(G)$ by 1,2,3.

5. $h(n) \leq h^*(n)$ since $h$ is admissible.
6. $g(n) + h(n) \leq g(n) + h^*(n)$ by 5.
7. $f(n) \leq f(G)$ by definition of $f$.

8. $f(G_2) > f(n)$ by 4,7.

Hence $A^*$ will never select $G_2$ for expansion.
Consistent Heuristics

- A heuristic is consistent (monotonic) if for every node $n$, every successor $n'$ of $n$ generated by any action $a$, $h(n) \leq c(n, a, n') + h(n')$

- If $h$ is consistent, we have:
  
  $f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') + \geq g(n) + h(n) = f(n)$

  i.e., $f(n)$ is non-decreasing along any path.

- **Theorem**
  If $h(n)$ is consistent, $A^*$ using **Graph-Search** is optimal.
  
  - Graph-Search: nodes encountered a second time can be treated as leaf nodes.
Properties of $A^*$

- Complete?
  - Yes, unless there are infinitely many nodes with $f \geq f(G)$.

- Time?
  - Exponential.

- Optimal?
  - Yes.

- Space?
  - Keeps all nodes in memory.
  - A serious problem in many cases!
    - How can we overcome it?
Optimality of $A^*$

- $A^*$ expands nodes in order of increasing $f$ value.
- Gradually adds "f-contours" of nodes.
- Contour $i$ has all nodes with $f \leq f_i$, where $f_i$ increases with $i$.
- We can think of the contour in which a node appears as its "depth".
- Using this, we can emulate the idea of iterative deepening.
Iterative Deepening $A^*$ (IDA$^*$)

- Combine Iterative Deepening with $A^*$ in order to overcome the space problem.
- A simple idea:
  - Instead of depth limited search, use $f$-value limited search
    - Replace upper bound on depth with upper bound $u$ on $f$-values.
    - Use depth-first search within current $f$ contour.
  - If $f(n) > u$—ignore $n$, treat it as a leaf node.
  - If no solution is found given $u$, increase $u$.
- Another method – Weighted $A^*$ (soon)
Local search algorithms

- In many problems, the path to the goal is irrelevant; the goal state itself is all we care about.
- Example: $n$-queens problems
- More generally: constraint-satisfaction problems
- Local search algorithms keep a single "current" state, and try to improve it.
Hill-climbing search

- "Like climbing mount Everest in thick fog with amnesia..."
- Improve while you can, i.e. stop when reaching a maxima (minima)
- Many variants: tie-breaking rules, restarts

```plaintext
function HILL-CLIMBING(problem) returns a state that is a local maximum
inputs: problem, a problem
local variables: current, a node
               neighbor, a node

current ← MAKE-NODE(INITIAL-STATE[problem])
loop do
    neighbor ← a highest-valued successor of current
    if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
current ← neighbor
```

Automated Planning and Decision Making 2014
Hill-climbing search

- Problem: depending on initial state, can get stuck in local maxima.
Solution by Hill-climbing

- $h =$ number of pairs of queens that are attacking each other, either directly or indirectly.
- Moves: change position of a single queen
- On A, $h=17$.
- On B, $h=1$. local minimum.
Enforced hill-climbing

```
def improve(σ₀):
    queue := new fifo-queue
    queue.push-back(σ₀)
    closed := Ø
    while not queue.empty():
        σ = queue.pop-front()
        if state(σ) ∉ closed:
            closed := closed ∪ {state(σ)}
            if h(σ) < h(σ₀):
                return σ
            for each ⟨o, s⟩ ∈ succ(state(σ)):
                σ' := make-node(σ, o, s)
                queue.push-back(σ')
    fail

⇒ breadth-first search for more promising node than σ₀
```
Enforced hill-climbing (ctd.)

Enforced hill-climbing

\[
\sigma := \text{make-root-node}(\text{init}())
\]

while not is-goal(state(\sigma)):

\[
\sigma := \text{improve}(\sigma)
\]

return extract-solution(\sigma)

- one of the three most commonly used algorithms for satisficing planning
- can fail if procedure improve fails (when the goal is unreachable from \(\sigma_0\))
- complete for undirected search spaces (where the successor relation is symmetric) if \(h(\sigma) = 0\) for all goal nodes and only for goal nodes
Simulated annealing search

- Idea: escape local maxima by allowing some "bad" moves, but gradually decrease their frequency.

```plaintext
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
inputs: problem, a problem
        schedule, a mapping from time to “temperature”
local variables: current, a node
                next, a node
                T, a “temperature” controlling prob. of downward steps

current ← MAKE-NODE(INITIAL-STATE[problem])
for t ← 1 to ∞ do
    T ← schedule[t]
    if T = 0 then return current
    next ← a randomly selected successor of current
    ΔE ← VALUE[next] - VALUE[current]
    if ΔE > 0 then current ← next
    else current ← next only with probability \( e^{ΔE/T} \)
```
Properties of SAS

- One can prove: If $T$ decreases slowly enough, then SAS will find a global optimum with probability approaching 1.
- Widely used in VLSI layout, airline scheduling, etc.
Weighted $A^*$

- Makes $A^*$ more greedy
- $f(n) = g(n) + W \cdot h(n)$ for $W > 1$
- Gives more weight to the heuristic value, i.e. to getting closer to the goal
- Solution is at most factor $W$ optimal
- Usually reaches the goal faster and requires less memory, but $IDA^*$ uses less memory.
Weighted A* (with duplicate detection and reopening)

\[
\begin{align*}
\text{open} & := \textbf{new} \ \text{min-heap ordered by} \ (\sigma \mapsto g(\sigma) + W \cdot h(\sigma)) \\
\text{open}.\text{insert}(\text{make-root-node}(\text{init}())) \\
\text{closed} & := \emptyset \\
\text{distance} & := \emptyset \\
\textbf{while} \ \text{not} \ \text{open}.\text{empty}(): \\
& \quad \sigma = \text{open}.\text{pop-min}() \\
& \quad \textbf{if} \ state(\sigma) \notin \text{closed} \ \textbf{or} \ g(\sigma) < \text{distance}(state(\sigma)): \\
& \quad \quad \text{closed} := \text{closed} \cup \{\text{state}(\sigma)\} \\
& \quad \quad \text{distance}(\sigma) := g(\sigma) \\
& \quad \textbf{if} \ \text{is-goal}(\text{state}(\sigma)): \\
& \quad \quad \textbf{return} \ \text{extract-solution}(\sigma) \\
& \quad \textbf{for each} \ (o, s) \in \text{succ}(\text{state}(\sigma)): \\
& \quad \quad \sigma' := \text{make-node}(\sigma, o, s) \\
& \quad \quad \textbf{if} \ h(\sigma') < \infty: \\
& \quad \quad \quad \text{open}.\text{insert}(\sigma') \\
\textbf{return} \ \text{unsolvable}
\end{align*}
\]
Properties of weighted $A^*$

The weight $W \in \mathbb{R}^+_0$ is a parameter of the algorithm.
- for $W = 0$, behaves like breadth-first search
- for $W = 1$, behaves like $A^*$
- for $W \to \infty$, behaves like greedy best-first search

Properties:
- one of the three most commonly used algorithms for satisficing planning
- for $W > 1$, can prove similar properties to $A^*$, replacing optimal with bounded suboptimal: generated solutions are at most a factor $W$ as long as optimal ones
Generating a Heuristic Function

- Heuristic functions are usually obtained by finding a simple approximation to the problem
- Simple heuristics often work well (but not very well)
- Examples:
  - N-puzzle: Manhattan distance
    - Simplification: assumes that we can move each tile independently. Ignores interactions between tiles
    - A very general idea: ignore certain interactions