Search Problems

In this overview we study a special case of the MDP problems previously considered. In particular we assume:

- Deterministic transitions
- Fully observable state
- Goal based formulation
- Costs instead of rewards
- The state space is assumed to be large or even infinite

Example: Romania

On holiday in Romania; currently in Arad.
Flight leaves tomorrow from Bucharest

Formulate goal:
be in Bucharest

Formulate problem:
states: various cities
actions: drive between cities

Find solution:
sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Example: vacuum world

Single-state, start in \#5. Solution?
[Right, Suck]

Conformant, start in \{1, 2, 3, 4, 5, 6, 7, 8\}
e.g., Right goes to \{2, 4, 6, 8\}. Solution?
[Right, Suck, Left, Suck]

Contingency, start in \#5
Murphy’s Law: Suck can dirty a clean carpet
Local sensing: dirt, location only.
Solution?
[Right, if dirt then Suck]
Single-state problem formulation

A problem is defined by four items:

- **initial state**: e.g., "at Arad"
- **successor function** $S(x) = \text{set of action-state pairs}$
  - e.g., $S(\text{Arad}) = \{\langle \text{Arad} \rightarrow \text{Zerind}, \text{Zerind} \rangle, \ldots\}$
- **goal test**, can be
  - explicit, e.g., $x = \text{at Bucharest}$
  - implicit, e.g., $\text{NoDirt}(x)$
- **path cost** (additive)
  - e.g., sum of distances, number of actions executed, etc.
  - $c(x,a,y)$ is the step cost, assumed to be $\geq 0$

A solution is a sequence of actions
leading from the initial state to a goal state

Example: vacuum world state space graph

- **states**: integer dirt and robot locations (ignore dirt amounts etc.)
- **actions**: $\text{Left, Right, Suck, NoOp}$
- **goal test**: no dirt
- **path cost**: 1 per action (0 for $\text{NoOp}$)

Example: The 8-puzzle

- **states**: integer locations of tiles (ignore intermediate positions)
- **actions**: move blank left, right, up, down (ignore unjamming etc.)
- **goal test**: goal state (given)
- **path cost**: 1 per move

[Note: optimal solution of $n$-Puzzle family is NP-hard]

Example: robotic assembly

- **states**: real-valued coordinates of robot joint angles
- **parts of the object to be assembled**
- **actions**: continuous motions of robot joints
- **goal test**: complete assembly with no robot included!
- **path cost**: time to execute

Tree search algorithms

Basic idea:
- offline, simulated exploration of state space
  - by generating successors of already-explored states
    - (a.k.a. expanding states)

Function: $\text{Tree-Search}(\text{problem}, \text{strategy})$ returns a solution, or failure

1. Initialize the search tree using the initial state of $\text{problem}$
2. Loop:
   - If there are no candidates for expansion return failure
   - Choose a leaf node for expansion according to $\text{strategy}$
   - If the node contains a goal state then return the corresponding solution
   - Else expand the node and add the resulting nodes to the search tree
3. End

Tree search example
function Tree-Search(problem, fringe) returns a solution, or failure
fringe ← Insert(Make-Node(Initial-State(problem)), fringe)
loop do
  if fringe is empty then return failure
  node ← Remove-Front(fringe)
  if Goal-Test(problem, State(node)) then return node
  fringe ← InsertAll(Expand(node, problem), fringe)
end loop

function Expand(node, problem) returns a set of nodes
successors ← the empty set
for each action, result in Successor-Fn(problem, State[node]) do
  s ← an empty Node
  Parent-Node[s] ← node;
  Action[s] ← action;
  State[s] ← result
  Path-Cost[s] ← Path-Cost[node] + Step-Cost(State[node], action, result)
  Depth[s] ← Depth[node] + 1
  add s to successors
return successors

Search strategies

A strategy is defined by picking the order of node expansion.

Strategies are evaluated along the following dimensions:
- Completeness — does it always find a solution if one exists?
- Time complexity — number of nodes generated/expanded
- Space complexity — maximum number of nodes in memory
- Optimality — does it always find a least-cost solution?

Time and space complexity are measured in terms of:
- $b$ — maximum branching factor of the search tree
- $d$ — depth of the least-cost solution
- $m$ — maximum depth of the state space (may be $\infty$)

Uninformed search strategies

Uninformed strategies use only the information available in the problem definition.

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search

Properties of breadth-first search

Complete? Yes (if $b$ is finite)
Time? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in $d$
Space? $O(b^{d+1})$ (keeps every node in memory)
Optimal? Yes (if cost = 1 per step), not optimal in general
Space is the big problem; can easily generate nodes at 100MB/sec so 24hrs = 8640GB.

Uniform-cost search (Dijkstra’s Algorithm)

Expand least-cost unexpanded node

Implementation:
- fringe = queue ordered by path cost, lowest first

Equivalent to breadth-first if step costs all equal

Complete? Yes, if step cost $\geq c$
Time? $\phi$ of nodes with $g \leq$ cost of optimal solution, $O(b^{C^*/c})$ where $C^*$ is the cost of the optimal solution
Space? $\phi$ of nodes with $g \leq$ cost of optimal solution, $O(b^{C^*/c})$
Optimal? Yes — nodes expanded in increasing order of $g(n)$
Depth-first search

Expand deepest unexpanded node

Implementation:
\[ fringe = \text{LIFO queue, i.e., put successors at front } \]

Properties of depth-first search

- Complete? No: fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path
  - \( \Rightarrow \) complete in finite spaces

- Time? \( O(b^d) \): terrible if \( m \) is much larger than \( d \)
  - but if solutions are dense, may be much faster than breadth-first

- Space? \( O(bm) \), i.e., linear space!

Optimal? No

Depth-limited search

= depth-first search with depth limit \( l \)
  - i.e., nodes at depth \( l \) have no successors

Recursive implementation:

function Depth-Limited-Search(problem, limit)
returns soln/fail/cutoff

Recursive-DLS(node, problem, limit)
returns soln/fail/cutoff

cutoff-occurred? ← false
if Goal-Test(problem, State[node]) then return node
else if Depth[node] = limit then return cutoff
else for each successor in Expand(node, problem) do
  result ← Recursive-DLS(successor, problem, limit)
  if result = cutoff then cutoff-occurred? ← true
    else if result ≠ failure then return result
  if cutoff-occurred? then return cutoff else return failure

Iterative deepening search

- \( l = 0 \)

function Iterative-Deepening-Search(problem) returns a solution
inputs: problem

for depth ← 0 to \( \infty \) do
  result ← Depth-Limited-Search(problem, depth)
  if result ≠ cutoff then return result
end
Properties of iterative deepening search

Complete? Yes
Time? \((d + 1)b^d + db^{d-1} + \ldots + b^0 = O(b^d)\)
Space? \(O(bd)\)
Optimal? Yes, if step cost = 1
Can be modified to explore uniform-cost tree

Numerical comparison for \(b = 10\) and \(d = 5\), solution at far right leaf:

\[ N(\text{IDS}) = 50 + 100 + 3,000 + 20,000 + 100,000 = 123,450 \]
\[ N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100 \]

IDS does better because other nodes at depth \(d\) are not expanded
BFS can be modified to apply goal test when a node is generated

Repeated states

Failure to detect repeated states can turn a linear problem into an exponential one!

Graph search

function Graph-Search \((\text{problem, fringe})\) returns a solution, or failure

\[
\text{closed} \leftarrow \text{an empty set}
\]

\[
\text{fringe} \leftarrow \text{INSERT}(\text{Make-Node}(\text{Initial-State}[\text{problem}]), \text{fringe})
\]

loop do

if fringe is empty then return failure

\[
\text{node} \leftarrow \text{REMOVE-FRONT}(\text{fringe})
\]

if Goal-Test(\text{problem}, State[\text{node}]) then return node

if State[\text{node}] is not in \text{closed} then

\[
\text{add State}[\text{node}] \text{ to closed}
\]

\[
\text{fringe} \leftarrow \text{INSERT-ALL}(\text{EXPAND}(\text{node}, \text{problem}), \text{fringe})
\]
end

Uninformed Search — Summary

Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored

Variety of uninformed search strategies
Iterative deepening search uses only linear space and not much more time than other uninformed algorithms
Graph search can be exponentially more efficient than tree search

Informed Search Algorithms

- Best-first search
- A* search
- Heuristics

Summary of algorithms

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Breadth-First</th>
<th>Uniform-First</th>
<th>Depth-Limited</th>
<th>Iterative Deepening</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete?</td>
<td>Yes(^*)</td>
<td>Yes(^*)</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Time</td>
<td>(b^{d+1})</td>
<td>(b^{l+1})</td>
<td>(b^l)</td>
<td>(b^l)</td>
</tr>
<tr>
<td>Space</td>
<td>(b^{d+1})</td>
<td>(b^{l+1})</td>
<td>(b^l)</td>
<td>(b^l)</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes(^*)</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Best-first search

Idea: use an evaluation function for each node

⇒ Expand most desirable unexpanded node

Implementation:

\textit{fringe} is a queue sorted in decreasing order of desirability

Special cases:

greedy search

\(A^*\) search

---

Greedy search

Evaluation function \(h(n)\) (heuristic)

\(=\) estimate of cost from \(n\) to the closest goal

E.g., \(h_{SLD}(n)\) = straight-line distance from \(n\) to Bucharest

Greedy search expands the node that appears to be closest to goal

---

Properties of greedy search

\textit{Complete}?: No—can get stuck in loops, e.g.,

Iasi \(\rightarrow\) Neamt \(\rightarrow\) Iasi \(\rightarrow\) Neamt \(\rightarrow\)

\textit{Time}?: \(O(b^m)\), but a good heuristic can give dramatic improvement

\textit{Space}?: \(O(b^m)\)—keeps all nodes in memory

\textit{Optimal}?: No

---

A* search

Idea: avoid expanding paths that are already expensive

Evaluation function \(f(n) = g(n) + h(n)\)

\(g(n)\) = cost so far to reach \(n\)

\(h(n)\) = estimated cost to goal from \(n\)

\(f(n)\) = estimated total cost of path through \(n\) to goal

\(A^*\) search uses an admissible heuristic

i.e., \(h(n) \leq h^*(n)\) where \(h^*(n)\) is the true cost from \(n\).

(Also require \(h(n) \geq 0\), so \(h(G) = 0\) for any goal \(G\).)

E.g., \(h_{SLD}(n)\) never overestimates the actual road distance

\textit{Theorem}: \(A^*\) search is optimal
**A* search example**

```
A  Z
   / \
  B  C
 /   \
D   E
```

**Optimality of A***(standard proof)**

Suppose some suboptimal goal \( G_2 \) has been generated and is in the queue. Let \( n \) be an unexpanded node on a shortest path to an optimal goal \( G_1 \).

\[
f(G_2) = g(G_2) \quad \text{since} \quad h(G_2) = 0
\]
\[
> g(G_1) \quad \text{since} \quad G_2 \text{ is suboptimal}
\]
\[
\geq f(n) \quad \text{since} \quad h \text{ is admissible}
\]

Since \( f(G_2) > f(n) \), \A* will never select \( G_2 \) for expansion.

**Optimality of A***(more useful)**

Lemma: \A* expands nodes in order of increasing \( f \) value
Gradually adds "\( f \)-contours" of nodes (cf. breadth-first adds layers)
Contour \( i \) has all nodes with \( f = f_i \), where \( f_i < f_{i+1} \)

**Properties of A**

- Complete? Yes, unless there are infinitely many nodes with \( f \leq f(G) \)
- Time? Exponential in \( \text{relative error in } h \times \text{length of soln.} \)
- Space? Keeps all nodes in memory
- Optimal? Yes—cannot expand \( f_{i+1} \) until \( f_i \) is finished
- \A* expands all nodes with \( f(n) < C^* \)
- \A* expands some nodes with \( f(n) = C^* \)
- \A* expands no nodes with \( f(n) > C^* \)

**Proof of lemma: Consistency**

A heuristic is consistent if

\[ h(n) \leq c(n, a, n') + h(n') \]

If \( h \) is consistent, we have

\[
f(n') = g(n') + h(n')
\]
\[
= g(n) + c(n, n', n') + h(n')
\]
\[
\geq g(n) + h(n)
\]
\[
= f(n)
\]

i.e., \( f(n) \) is nondecreasing along any path.

**Admissible heuristics**

E.g., for the 8-puzzle:

- \( h_1(n) \) — number of misplaced tiles
- \( h_2(n) \) — total Manhattan distance
  (i.e., no. of squares from desired location of each tile)

<table>
<thead>
<tr>
<th>Start State</th>
<th>Goal State</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 3 4 1 5 6 8 7</td>
<td>1 2 3 4 5 6 7 8</td>
</tr>
</tbody>
</table>

\[
h_1(S) = 6
\]
\[
h_2(S) = 14
\]

7 4 3 1 5 2 8 6
Dominance

If \( h_2(n) \geq h_1(n) \) for all \( n \) (both admissible)
then \( h_2 \) dominates \( h_1 \) and is better for search

Typical search costs:
- \( d = 14 \) IDS ≈ 3,473,941 nodes
- \( A^*(h_1) = 539 \) nodes
- \( A^*(h_2) = 113 \) nodes
- \( d = 24 \) IDS ≈ 54,000,000,000 nodes
- \( A^*(h_1) = 39,135 \) nodes
- \( A^*(h_2) = 1,641 \) nodes

Given any admissible heuristics \( h_a, h_b \),
\[ h(n) = \max(h_a(n), h_b(n)) \]
is also admissible and dominates \( h_a, h_b \)

Relaxed problems

Admissible heuristics can be derived from the exact
solution cost of a relaxed version of the problem
If the rules of the 8-puzzle are relaxed so that a tile can move anywhere,
then \( h_1(n) \) gives the shortest solution
If the rules are relaxed so that a tile can move to any adjacent square,
then \( h_2(n) \) gives the shortest solution

Key point: the optimal solution cost of a relaxed problem
is no greater than the optimal solution cost of the real problem

Informed Search — Summary

Heuristic functions estimate costs of shortest paths
Good heuristics can dramatically reduce search cost

Greedy best-first search expands lowest \( h \)
  - incomplete and not always optimal
\( A^* \) search expands lowest \( g + h \)
  - complete and optimal
  - also optimally efficient (up to tie-breaks, for forward search)

Admissible heuristics can be derived from exact solution of relaxed problems

Incomplete information

Disjunctive effects, e.g., \( \text{Inflate}(x) \) causes
\( \text{Inflate}(x) \lor \text{SlowHiss}(x) \lor \text{Burst}(x) \lor \text{BrokenPump} \lor \ldots \)

Conditional planning

If the world is nondeterministic or partially observable
then percepts usually provide information,
i.e., split up the belief state

Solutions

Conformant
Devise a plan that works regardless outcome
  Such plans may not exist

Conditional planning
Subplan for each contingency, e.g.,
  \( \text{Check}(\text{Tire}_1), \text{if Intact}(\text{Tire}_1) \text{ then Inflate}(\text{Tire}_1) \text{ else CallAAA} \)
  Expensive because it plans for many unlikely cases

Monitoring/Replanning
Assume normal states, outcomes
  Check progress during execution, replan if necessary
  Unanticipated outcomes may lead to failure (e.g., no AAA card)

(Really need a combination; plan for likely/serious eventualities,
deal with others when they arise, as they must eventually)
Conditional planning contd.

Conditional plans check (any consequence of KB +) percept

\[ \ldots \text{if } C \text{ then } \text{Plan}_A, \text{ else } \text{Plan}_B, \ldots \]

Execution: check \( C \) against current KB, execute “then” or “else”

Need some plan for every possible percept.

(Cf. game playing: some response for every opponent move)

AND–OR tree search: Example

Double Murphy: sucking or arriving may dirty a clean square

\[ \text{Left Suck} \quad \text{Right Suck} \quad \text{GOAL} \quad \text{LOOP} \]

Triple Murphy: also sometimes stays put instead of moving

\[ \text{[L1 : Left if AtR then L1 else [if CleanL then [] else Suck]]} \]

or \[ [\text{while AtR do [Left if CleanL then [] else Suck]}] \]

“infinite loop” but will eventually work unless action always fails

Incomplete Information — Summary

Various forms, we only discussed disjunctive outcomes (cf. probabilistic outcomes in MDP).

Solutions: Conformant planning, Conditional Plans, RePlanning.

Search: AND-OR search trees (cf. updates in RL).

Online Search Problems

- Deterministic, fully observable environment
- Cannot search “offline” (can see immediate step and its cost but not beyond)
- Have a heuristic function \( h(s) \)
- Without assumptions easy to fall into unavoidable dead-ends.
- Assume “safely explorable” (cf. ergodic in MDP)

Random Walks for Online Search

- Will eventually reach goal (assuming finite state space)
- But it can be slow
Learning Real-Time A* (LRTA*)

- Hill climbing "with memory"—updates heuristic value
- \( h(s) \) is an admissible heuristic
- Initially \( H(s) = h(s) \)
- With moves \( H(s) = e(s, a, s') + H(s') \)
- Initialization to \( h(s) \) guarantees exploration (cf. optimistic initialization)

Learning Real-Time A* (LRTA*)

Function LRTA\_ALGORITHM returns an action

\begin{itemize}
  \item \( s \)—start state
  \item \( Q \)—table of state estimates ordered by state, initially empty
  \item \( R \)—table of state-action estimates ordered by state, action, initially empty
  \item \( A \)—the previous state and action, initially null
\end{itemize}

// pick best action and update hash
\( a = \text{GREEDYACTION}(s) \)
\( s' = \text{PICKNEXTSTATE}(a) \)
\( Q(s, a, s') = \text{update hash}(s', a, Q(s, a, s')) \)
\( R(s', a, s) = \text{min}(R(s', a, s), e(s, a, s') + Q(s, a, s')) \)
\( H(s) = H(s) + e(s, a, s') + R(s', a, s) \)
\( A = (s, a) \)

Repeat:
\[ V_a(s) = \max_{a'} \left( Q(s, a', s') + \sum_{s' \in S} \Pr(s' | s, a') \left[ \text{max}_{a''} \{ Q(s, a'', s') \} \right] \right) \]

end

RTDP Algorithm (as Previously Presented)

Repeat:
pick start state \( s \)
Repeat: [in state \( s \)]

\[ V_a(s) = \max_{a'} \left( Q(s, a', s') + \sum_{s' \in S} \Pr(s' | s, a') \left[ \text{max}_{a''} \{ Q(s, a'', s') \} \right] \right) \]

end

Choose action \( a \) using policy \( P \)
Sample \( s' \) from \( \Pr(s' | s, a) \)
\( s' \)

Note: This performs \( B(V) \) on the \( Q() \) representation

RTDP Algorithm (optimistic exploration) [BG03]

\begin{algorithm}
\begin{algorithmic}
\STATE RTDP(\( s \) : state)
\STATE \textbf{begin}
\STATE \quad \textbf{repeat} RTDP\_TRIAL(\( s \))
\STATE \quad \textbf{end}
\STATE \textbf{end}
\STATE RTDP\_TRIAL(\( s \) : state)
\STATE \textbf{begin}
\STATE \quad \textbf{while} \( \neg \text{GOAL}() \) \textbf{do}
\STATE \quad \quad // pick best action and update hash
\STATE \quad \quad \( a = \text{GREEDYACTION}(s) \)
\STATE \quad \quad \( s' = \text{PICKFROMACTIONS}(a) \)
\STATE \quad \quad // stochastically simulate next state
\STATE \quad \quad \textbf{end}
\STATE \textbf{end}
\end{algorithmic}
\end{algorithm}

Algorithm gets close to optimal quickly but can be slow to converge because it keeps on exploring the best states and actions that have already converged.
Checking for Solved States [BG03]

```plaintext
CHECKSOLVED(s, status : [false])
begin
if status == SOLVED
begin return status
end

status = EMPTYSTACK
LRTDP(s)
if status == SOLVED
begin PUSH(s)
end
while (status != EMPTYSTACK)
do
if residual(s) > ϵ then
else continue

endif

// expand state
foreach (s' such that P(s')(s, a) > 0) do
if --SOLVED{s'} == SOLVED{open(u) and visited(s')}
LRTDP{open(u) and visited(s')}
endif
Code: "LRTDP" Labeled RTDP [BG03]
```

Checking for Solved States [BG03]

```plaintext
if ϵ == ϵ_{up} then
foreach s ∈ C(s)
do
L if status == SOLVED
else
// update states with residuals and ancestors
LRTDP{open(u) and visited(s')}
if status == SOLVED
return ϵ_{up}
end
```

Labeled RTDP [BG03]

```plaintext
LRTDP(s : state, e : [false])
begin
while ¬SOLVED do LRTDP_TRIAL(s, e)
end
```

A trial ends at a solved state (not just goal).
Solved states are updated backward on path.
If not solved, an update makes at least ϵ-progress.

Labeled RTDP [BG03]

```plaintext
Theorem 6 Provided that the goal is reachable from every state
and the initial value function is admissible, then LRTDP
trials cannot be trapped into loops, and hence must termi-
inate in a finite number of steps.

Theorem 7 Provided that the goal is reachable from every state
and the initial value function is admissible and mono-
tonic, then LRTDP solves the model M1-M7 in a number of
trials bounded by ϵ^{-1} \sum_{s ∈ S} V^0(s) - h(s).
```

Online Search — Summary

Setting similar to the one in RL.
LRTA* similar to RDTP variant with optimistic initialization and greedy
policy.
Labeled RTDP improves convergence of RTDP.