Lecture 2
Solving Problems by Searching

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

Slides by Stuart Russell and Peter Norvig
Agents are used to provide a consistent viewpoint on various topics in the field AI.

Essential concepts:

- Agents interact with environment by means of sensors and actuators. A rational agent does “the right thing” $\equiv$ maximizes a performance measure.
  - PEAS

- Environment types: observable, deterministic, episodic, static, discrete, single agent

- Agent types: table driven (rule based), simple reflex, model-based reflex, goal-based, utility-based, learning agent
Structure of Agents

Agent = Architecture + Program

- Architecture
  - operating platform of the agent
  - computer system, specific hardware, possibly OS

- Program
  - function that implements the mapping from percepts to actions

This course is about the program, not the architecture
Outline

1. Problem Solving Agents
2. Search
3. Uninformed search algorithms
4. Informed search algorithms
5. Constraint Satisfaction Problem
Outline

1. Problem Solving Agents

2. Search

3. Uninformed search algorithms

4. Informed search algorithms

5. Constraint Satisfaction Problem
Outline

♦ Problem-solving agents
♦ Problem types
♦ Problem formulation
♦ Example problems
♦ Basic search algorithms
Problem-solving agents

Restricted form of general agent:

```plaintext
function Simple-Problem-Solving-Agent( percept ) returns an action
    static: seq, an action sequence, initially empty
    state, some description of the current world state
    goal, a goal, initially null
    problem, a problem formulation

    state ← Update-State( state, percept )
    if seq is empty then
        goal ← Formulate-Goal( state )
        problem ← Formulate-Problem( state, goal )
        seq ← Search( problem )
        action ← Recommendation( seq, state )
        seq ← Remainder( seq, state )
    return action
```

Note: this is offline problem solving; solution executed “eyes closed.”

Online problem solving involves acting without complete knowledge.
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: Romania
State space Problem formulation

A problem is defined by five items:

1. **initial state** e.g., “at Arad”
2. **actions** defining the other states, e.g., \( \text{Go}(\text{Arad}) \)
3. **transition model** \( \text{res}(x, a) \)
   e.g., \( \text{res}(\text{In}(\text{Arad}), \text{Go}(\text{Zerind})) = \text{In}(\text{Zerind}) \)
   alternatively: set of action–state pairs:
   \( \{(\text{In}(\text{Arad}), \text{Go}(\text{Zerind})), \text{In}(\text{Zerind})\}, \ldots \} \)
4. **goal test**, can be
   explicit, e.g., \( x = \text{“at Bucharest”} \)
   implicit, e.g., \( \text{NoDirt}(x) \)
5. **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
Selecting a state space

Real world is complex
⇒ state space must be abstracted for problem solving

(Abstract) state = set of real states

(Abstract) action = complex combination of real actions
e.g., “Arad → Zerind” represents a complex set of possible routes, detours, rest stops, etc.
For guaranteed realizability, any real state “in Arad” must get to some real state “in Zerind”

(Abstract) solution =
set of real paths that are solutions in the real world

Each abstract action should be “easier” than the original problem!

Atomic representation
Vacuum world state space graph

Example

**states??**: integer dirt and robot locations (ignore dirt amounts etc.)
**actions??**: Left, Right, Suck, NoOp
**transition model??**: arcs in the digraph
**goal test??**: no dirt
**path cost??**: 1 per action (0 for NoOp)
Example: The 8-puzzle

**states??**: integer locations of tiles (ignore intermediate positions)
**actions??**: move blank left, right, up, down (ignore unjamming etc.)
**transition model??**: effect of the actions
**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
Problem types

Deterministic, fully observable, known, discrete $\implies$ state space problem
Agent knows exactly which state it will be in; solution is a sequence

Non-observable $\implies$ conformant problem
Agent may have no idea where it is; solution (if any) is a sequence

Nondeterministic and/or partially observable $\implies$ contingency problem
percepts provide new information about current state
solution is a contingent plan or a policy
often interleave search, execution

Unknown state space $\implies$ exploration problem (“online”)
Example: vacuum world

State space, start in #5. Solution??
\[ \text{[Right, Suck]} \]

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

Contingency, start in #5
Murphy’s Law: Suck can dirty a clean carpet
Local sensing: dirt, location only.
Solution??
\[ \text{[Right, if dirt then Suck]} \]
Example Problems

- **Toy problems**
  - vacuum cleaner agent
  - 8-puzzle
  - 8-queens
  - cryptarithmetic
  - missionaries and cannibals

- **Real-world problems**
  - route finding
  - traveling salesperson
  - VLSI layout
  - robot navigation
  - assembly sequencing
Outline

1. Problem Solving Agents

2. Search

3. Uninformed search algorithms

4. Informed search algorithms

5. Constraint Satisfaction Problem
Objectives

- Formulate appropriate problems in optimization and planning (sequence of actions to achieve a goal) as search tasks:
  initial state, operators, goal test, path cost

- Know the fundamental search strategies and algorithms
  - uninformed search
    - breadth-first, depth-first, uniform-cost, iterative deepening, bi-directional
  - informed search
    - best-first (greedy, A*), heuristics, memory-bounded

- Evaluate the suitability of a search strategy for a problem
  - completeness, optimality, time & space complexity
Searching for Solutions

- Traversal of some search space from the initial state to a goal state with a legal sequence of actions as defined by operators.

- The search can be performed on:
  - On a search tree derived from expanding the current state using the possible operators: **Tree-Search algorithm**
  - A graph representing the state space: **Graph-Search algorithm**
Search: Terminology
Example: Route Finding
Tree search example
General tree search

function TREE-SEARCH(problem) returns a solution, or failure
initialize the frontier using the initial state of problem
loop do
  if the frontier is empty then return failure
  choose a leaf node and remove it from the frontier
  if the node contains a goal state then return the corresponding solution
  expand the chosen node, adding the resulting nodes to the frontier

function GRAPH-SEARCH(problem) returns a solution, or failure
initialize the frontier using the initial state of problem
initialize the explored set to be empty
loop do
  if the frontier is empty then return failure
  choose a leaf node and remove it from the frontier
  if the node contains a goal state then return the corresponding solution
  add the node to the explored set
  expand the chosen node, adding the resulting nodes to the frontier
  only if not in the frontier or explored set
Implementation: states vs. nodes

A **state** is a (representation of) a physical configuration
A **node** is a data structure constituting part of a search tree
   includes **state**, **parent**, **action**, **path cost** $g(x)$
States do not have parents, children, depth, or path cost!

The **Expand** function creates new nodes, filling in the various fields using the Transition Model of the problem to create the corresponding states.
Implementation: general tree search

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 ← a new 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
  end for
  return successors
Search strategies

A strategy is defined by picking the order of node expansion

```
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)
```

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 path 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 \( \infty \))
Outline

1. Problem Solving Agents
2. Search
3. Uninformed search algorithms
4. Informed search algorithms
5. Constraint Satisfaction Problem
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
Bidirectional Search
Breadth-first search

Expand shallowest unexpanded node (shortest path in the frontier)

Implementation:

*fringe* is a FIFO queue, i.e., new successors go at end
Properties of breadth-first search

**Complete??** Yes (if \( b \) is finite)

**Optimal??** Yes (if cost = 1 per step); not optimal in general

**Time??** \( 1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1}) \), i.e., exp. in \( d \)

**Space??** \( b^{d-1} + b^d = O(b^d) \) (explored + frontier)

**Space** is the big problem; can easily generate nodes at 100MB/sec so 24hrs = 8640GB.
Uniform-cost search

Expand first least-cost path
(Equivalent to breadth-first if step costs all equal)

Implementation:

fringe = priority queue ordered by path cost, lowest first,

Complete?? Yes, if step cost ≥ ϵ
Optimal?? Yes—nodes expanded in increasing order of \( g(n) \)
Time?? # of nodes with \( g \leq \) cost of optimal solution, \( O(b^{1+⌈C^*/ε⌉}) \)
where \( C^* \) is the cost of the optimal solution
Space?? # of nodes with \( g \leq \) cost of optimal solution, \( O(b^{1+⌈C^*/ε⌉}) \)
Depth-first search

Expand deepest unexpanded node

**Implementation:**

\[\text{fringe} = \text{LIFO queue, i.e., put successors at front}\]
Properties of depth-first search

**Complete??** No: fails in infinite-depth spaces, or spaces with loops
Modify to avoid repeated states along path
⇒ complete in finite spaces

**Optimal??** No

**Time??** $O(b^m)$: 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!
Depth-limited search

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

**Recursive implementation:**

```python
def Depth-Limited-Search(problem, limit):
    return Recursive-DLS(Make-Node(Initial-State[problem]), problem, limit)

def Recursive-DLS(node, problem, limit):
    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

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

for depth ← 0 to ∞ do
    result ← Depth-Limited-Search(problem, depth)
    if result ≠ cutoff then return result
end
Iterative deepening search

Limit = 3

Problem Solving Agents
Search
Uninformed search algorithms
Informed search algorithms
Constraint Satisfaction Problem
Properties of iterative deepening search

Complete?? Yes  
Optimal?? Yes, if step cost = 1  
Can be modified to explore uniform-cost tree  
Time?? $(d + 1)b^0 + db^1 + (d - 1)b^2 + \ldots + b^d = O(b^d)$  
Space?? $O(bd)$

Numerical comparison in time for $b = 10$ and $d = 5$, solution at far right leaf:

\[
N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450 \\
N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100
\]

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**  
Iterative lengthening not as successful as IDS
Bidirectional Search

- Search simultaneously (using breadth-first search) from goal to start from start to goal
- Stop when the two search trees intersects
Difficulties in Bidirectional Search

- If applicable, may lead to substantial savings
- Predecessors of a (goal) state must be generated
  Not always possible, eg. when we do not know the optimal state explicitly
- Search must be coordinated between the two search processes.
- What if many goal states?
- One search must keep all nodes in memory
### Summary of algorithms

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Breadth-First</th>
<th>Uniform-Cost</th>
<th>Depth-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, if $l \geq d$</td>
<td>Yes</td>
</tr>
<tr>
<td>Time</td>
<td>$b^{d+1}$</td>
<td>$b^{[1+C^*/\epsilon]}$</td>
<td>$b^m$</td>
<td>$b^l$</td>
<td>$b^d$</td>
</tr>
<tr>
<td>Space</td>
<td>$b^{d+1}$</td>
<td>$b^{[1+C^*/\epsilon]}$</td>
<td>$b^m$</td>
<td>$b^l$</td>
<td>$b^d$</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes*</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes*</td>
</tr>
</tbody>
</table>

**Criterion**
- Complete?
- Time
- Space
- Optimal?
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.
Outline

1. Problem Solving Agents
2. Search
3. Uninformed search algorithms
4. Informed search algorithms
5. Constraint Satisfaction Problem
**Review: Tree search**

```plaintext
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] applied to State(node) succeeds return node
        fringe ← InsertAll(Expand(node, problem), fringe)
```

A strategy is defined by picking the **order of node expansion**
Informed search strategy

Informed strategies use agent’s background information about the problem map, costs of actions, approximation of solutions, ...

- best-first search
  - greedy search
  - A* search

- local search (not in this course)
  - Hill-climbing
  - Simulated annealing
  - Genetic algorithms
  - Local search in continuous spaces
Best-first search

Idea: use an evaluation function for each node
    – estimate of “desirability”

⇒ Expand most desirable unexpanded node

Implementation:
fringe is a queue sorted in decreasing order of desirability

Special cases:
    greedy search
    A* search
Problem Solving Agents
Search
Uninformed search algorithms
Informed search algorithms
Constraint Satisfaction Problems

Romania with step costs in km
Greedy search

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

\[ h = \text{estimate of cost from } n \text{ to the closest goal} \]

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

Greedy search expands the node that appears to be closest to goal
Greedy search example
Properties of greedy search

**Complete??** No—can get stuck in loops, e.g., from Iasi to Fargas

\[ \text{Iasi} \rightarrow \text{Neamt} \rightarrow \text{Iasi} \rightarrow \text{Neamt} \rightarrow \]

Complete in finite space with repeated-state checking

**Optimal??** No

**Time??** \( O(b^m) \), but a good heuristic can give dramatic improvement

**Space??** \( O(b^m) \)
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

Theorem: A* search is optimal
A* search example

Problem Solving Agents
Search
Uninformed search algorithms
Informed search algorithms
Constraint Satisfaction Problems

Zerind → Arad → Sibiu → Timisoara → Bucharest

447 = 118 + 329

Arad → Sibiu → Fagaras

646 = 280 + 366

Sibiu → Oradea

671 = 291 + 380

Oradea → Bucharest

450 = 450 + 0

Bucharest → Craiova

526 = 366 + 160

Craiova → Pitesti

615 = 455 + 160

Pitesti → Sibiu

553 = 300 + 253

Sibiu → Rimnicu Vilcea

591 = 338 + 253

Rimnicu Vilcea → Bucharest

607 = 414 + 193

Bucharest → Craiova

418 = 418 + 0

Craiova → Rimnicu Vilcea

671 = 291 + 380

Rimnicu Vilcea → Zerind

449 = 75 + 374
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 $G_1$.

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

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 $f_i < f_{i+1}$

![Diagram of A* search algorithm](image-url)
A* vs. Depth search

Breadth-first

Expand by depth-layers

A*

Expands by f-contours

Uninformed search algorithms

Informed search algorithms

Constraint Satisfaction Problem
Properties of A*

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

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 $f(n) > C^*$

Time?? Exponential in [relative error in $h \times$ length of sol.]

Space?? Keeps all nodes in memory
Proof of lemma: Consistency

A heuristic is **consistent** if

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

If \( h \) is consistent, we have

\[
\begin{align*}
  f(n') &= g(n') + h(n') \\
  &= g(n) + c(n, a, n') + h(n') \\
  &\geq g(n) + h(n) \\
  &= f(n)
\end{align*}
\]

I.e., \( f(n) \) is nondecreasing along any path.
Admissible heuristics

E.g., for the 8-puzzle:

\[ h_1(n) = \text{number of misplaced tiles} \]
\[ h_2(n) = \text{total Manhattan distance} \]

(i.e., no. of squares from desired location of each tile)

Start State

\[
\begin{array}{ccc}
7 & 2 & 4 \\
5 & 6 & \\
8 & 3 & 1 \\
\end{array}
\]

Goal State

\[
\begin{array}{ccc}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & \\
\end{array}
\]
Admissible heuristics

E.g., for the 8-puzzle:

\[ h_1(n) = \text{number of misplaced tiles} \]
\[ h_2(n) = \text{total Manhattan distance} \]
(i.e., no. of squares from desired location of each tile)

\[
\begin{align*}
h_1(S) &= 6 \\
h_2(S) &= 4 + 0 + 3 + 3 + 1 + 0 + 2 + 1 = 14
\end{align*}
\]
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 $\approx 54,000,000,000$ 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 $h_a$, $h_b$
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.
Memory-Bounded Heuristic Search

- Try to reduce memory needs
- Take advantage of heuristic to improve performance
  - Iterative-deepening A* (IDA*)
  - SMA*
Iterative Deepening $A^*$

- Uniformed Iterative Deepening (repetition)
  - depth-first search where the max depth is iteratively increased

- IDA$^*$
  - depth-first search, but only nodes with $f$-cost less than or equal to smallest $f$-cost of nodes expanded at last iteration
  - was the "best" search algorithm for many practical problems
Properties of IDA*

Complete?? Yes
Time complexity?? Still exponential
Space complexity?? linear
Optimal?? Yes. Also optimal in the absence of monotonicity
Simple Memory-Bounded $A^*$

Use all available memory

- Follow $A^*$ algorithm and fill memory with new expanded nodes
- If new node does not fit
  - remove stored node with worst $f$-value
  - propagate $f$-value of removed node to parent
- SMA$^*$ will regenerate a subtree only when it is needed, the path through subtree is unknown, but cost is known
Propeties of SMA*

Complete?? yes, if there is enough memory for the shortest solution path
Time?? same as A* if enough memory to store the tree
Space?? use available memory
Optimal?? yes, if enough memory to store the best solution path

In practice, often better than A* and IDA* trade-off between time and space requirements
Recursive Best First Search

function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
  return RBFS(problem, MAKE-NODE(problem.INITIAL-STATE), ∞)

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  successors ← []
  for each action in problem.ACTIONS(node.STATE) do
    add CHILD-NODE(problem, node, action) into successors
  if successors is empty then return failure, ∞
  for each s in successors do /* update f with value from previous search, if any */
    s.f ← max(s.g + s.h, node.f)
  loop do
    best ← the lowest f-value node in successors
    if best.f > f_limit then return failure, best.f
    alternative ← the second-lowest f-value among successors
    result, best.f ← RBFS(problem, best, min(f_limit, alternative))
    if result ≠ failure then return result
Recursive Best First Search

(a) After expanding Arad, Sibiu, and Rimnicu Vilcea

(b) After unwinding back to Sibiu and expanding Fagaras

(c) After switching back to Rimnicu Vilcea and expanding Pitesti
Outline

1. Problem Solving Agents

2. Search

3. Uninformed search algorithms

4. Informed search algorithms

5. Constraint Satisfaction Problem
Constraint Satisfaction Problem (CSP)

Standard search problem:
- state is a “black box”—any old data structure that supports goal test, eval, successor

CSP:
- state is defined by variables $X_i$ with values from domain $D_i$

  goal test is a set of constraints specifying allowable combinations of values for subsets of variables

Simple example of a formal representation language

Allows useful general-purpose algorithms with more power than standard search algorithms
Standard search formulation

States are defined by the values assigned so far

- **Initial state:** the empty assignment, `{}`
- **Successor function:** assign a value to an unassigned variable that does not conflict with current assignment.
  - fail if no legal assignments (not fixable!)
- **Goal test:** the current assignment is complete

1) This is the same for all CSPs!
2) Every solution appears at depth $n$ with $n$ variables
   - use depth-first search
3) Path is irrelevant, so can also use complete-state formulation
4) $b = (n - \ell)d$ at depth $\ell$, hence $n!d^n$ leaves!!!
Backtracking search

Variable assignments are **commutative**, i.e.,

\[
\begin{align*}
[WA = \text{red} \text{ then } NT = \text{green}] & \quad \text{same as} \quad [NT = \text{green} \text{ then } WA = \text{red}] \\
\end{align*}
\]

Only need to consider assignments to a single variable at each node

\[\implies b = d \text{ and there are } d^n \text{ leaves}\]

Depth-first search for CSPs with single-variable assignments is called **backtracking** search

Backtracking search is the basic uninformed algorithm for CSPs

Can solve \(n\)-queens for \(n \approx 25\)
problem-solving-agents

Search

Uninformed search algorithms

Informed search algorithms

Constraint Satisfaction Problem

Backtracking search

function Backtracking-Search(csp) returns solution/failure
  return Recursive-Backtracking({}, csp)

function Recursive-Backtracking(assignment, csp) returns soln/failure
  if assignment is complete then return assignment
  var ← Select-Unassigned-Variable(Variables[csp], assignment, csp)
  for each value in Order-Domain-Values(var, assignment, csp) do
    if value is consistent with assignment given Constraints[csp] then
      add {var = value} to assignment
      result ← Recursive-Backtracking(assignment, csp)
      if result ≠ failure then return result
  remove {var = value} from assignment
  return failure
Summary

Uninformed Search
- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search
- Bidirectional Search

Informed Search
- best-first search
- greedy search
- A* search
- Iterative Deepening A*
- Memory bounded A*
- Recursive best first

Constraint Satisfaction and Backtracking