Lecture 2
Solving Problems by Searching

Marco Chiarandini

Deptartment of Mathematics & Computer Science
University of Southern Denmark

Slides by Stuart Russell and Peter Norvig

Last Time

- Agents are used to provide a consistent viewpoint on various topics in the field AI
- Essential concepts:
  - Agents intereact with environment by means of sensors and actuators.
    A rational agent does "the right thing" ≡ maximizes a performance measure
    ➨ PEAS
  - Environment types: observable, deterministic, episodic, static, discrete, single agent
  - Agent types: table driven, 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

In this course, emphasis on the program,
not on the architecture

Course Overview

- ✔ Introduction
  - ✔ Artificial Intelligence
  - ✔ Intelligent Agents
- Search
  - Uninformed Search
  - Heuristic Search
- Adversarial Search
  - Minimax search
  - Alpha-beta pruning
- Knowledge representation and Reasoning
  - Propositional logic
  - First order logic
  - Inference
- Uncertain knowledge and Reasoning
  - Probability and Bayesian approach
  - Bayesian Networks
  - Hidden Markov Chains
  - Kalman Filters
- Learning
  - Decision Trees
  - Maximum Likelihood
  - EM Algorithm
  - Learning Bayesian Networks
  - Neural Networks
  - Support vector machines

Outline

Outline

Objectives

- Formulate appropriate problems in optimization and planning (sequence of actions to achive 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, time & space complexity, optimality

Search

Search:
process of looking for a (or the best) sequence of actions, that leads to a goal (specific state of the environment), starting from an initial state

- Used in problem solving agent: aka planning
- Hypothesis on the environment
  - Static
  - Discrete
  - Deterministic
  - Fully observable

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

Problem formulation

Abstraction of real states and actions

A problem is defined by four items:

states and initial state e.g., "at Arad"

successor function $S(x)$ = set of action–state pairs
e.g., $S(Arad) = \{\langle Arad \rightarrow Zerind, Zerind\rangle, \ldots\}$

goal test, can be
explicit, e.g., x = "at Bucharest"
implicit, e.g., $NoDirt(x)$

path cost (additive)
e.g., sum of distances, number of actions executed, etc.
$c(x, a, y)$ is the step cost, assumed to be ≥ 0

State Space

Graph representation of states and successor function (operators), with the cost (if any)

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

8-Queens

Incremental formulation

- States
  arrangement of up to 8 queens on the board
- Operators
  add a queen to any square
- Goal test
  all queens on board no queen attacked
- Path cost
  irrelevant (all solutions equally valid)

Complete-state formulation

- States
  arrangement of 8 queens on the board
- Operators
  move a queen to a different square
- Goal test
  no queen attacked
- Path cost
  irrelevant (all solutions equally valid)

Searching for Solutions

- Traversal of some search space
  from the initial state to a goal state
  legal sequence of actions as defined by operators
- The search can be performed on
  - A graph representing
    the state space
    Graph-Search algorithm
  - Or on a search tree derived from
    expanding the current state using the possible operators
    Tree-Search algorithm

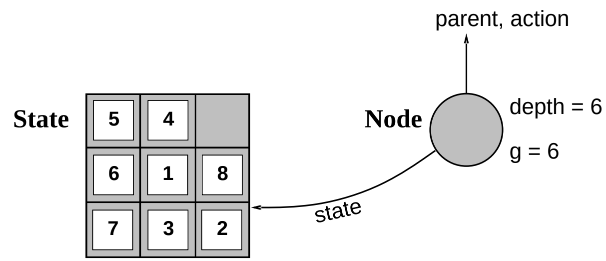
Tree search algorithms

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

function Tree-Search(*problem, strategy*) **returns** a solution, or failure
  initialize the search tree using the initial state of *problem*
  loop do
    if there are no candidates for expansion **then return** failure
    choose a leaf node for expansion according to *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
  end

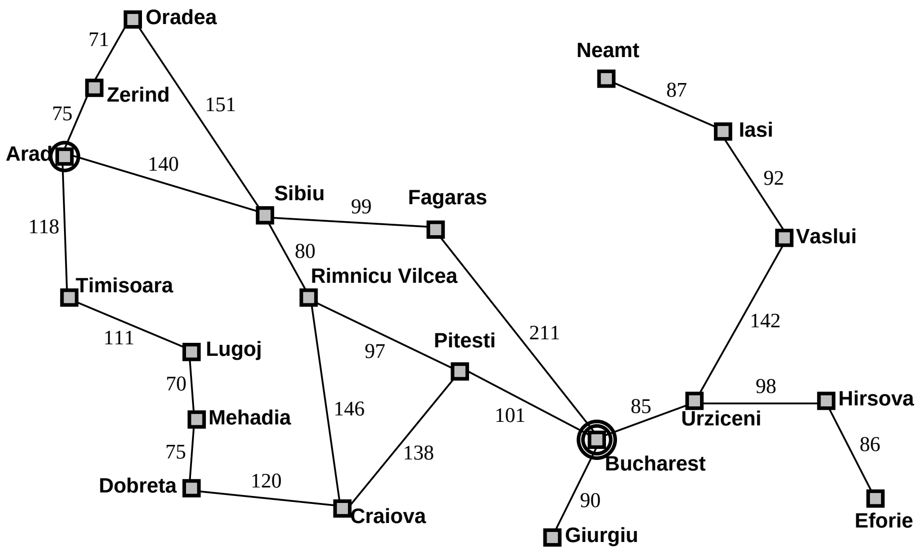
Implementation: states vs. nodes

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

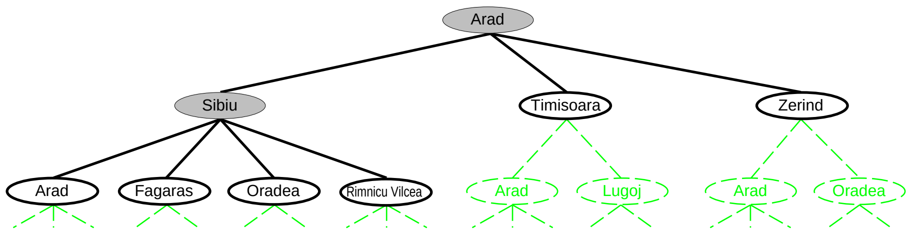


The Expand function creates new nodes, filling in the various fields and using the SuccessorFn of the problem to create the corresponding states.

Example: Route Finding



Tree search example



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)
```

```
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
   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-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 ∞)

Outline

Uninformed search strategies

Uninformed strategies use only the information available in the problem definition

Breadth-first search

Uniform-cost search

Depth-first search

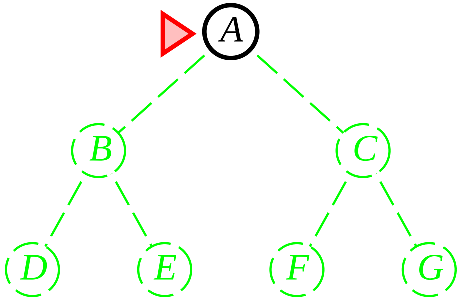
Depth-limited search

Iterative deepening search

Bidirectional Search

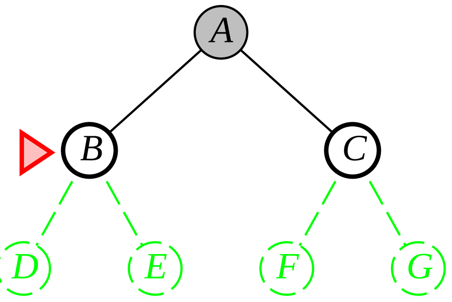
Breadth-first search

Expand shallowest unexpanded node
Implementation:
fringe is a FIFO queue, i.e., new successors go at end



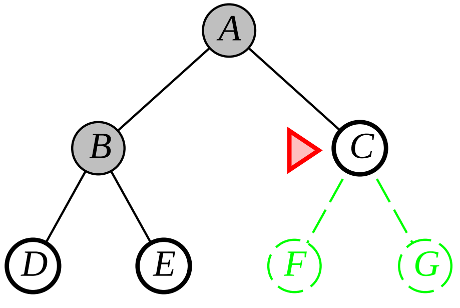
Breadth-first search

Expand shallowest unexpanded node
Implementation:
fringe is a FIFO queue, i.e., new successors go at end



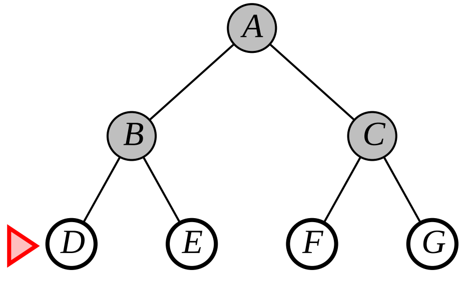
Breadth-first search

Expand shallowest unexpanded node
Implementation:
fringe is a FIFO queue, i.e., new successors go at end



Breadth-first search

Expand shallowest unexpanded node
Implementation:
fringe is a FIFO queue, i.e., new successors go at end



Properties of breadth-first search

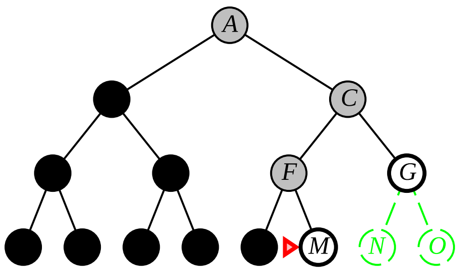
Complete?? Yes (if b is finite)
Time?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in d
Space?? $O(b^{d+1})$ (keeps every node in memory)
Optimal?? Yes (if cost = 1 per step); not optimal in general
Space is the big problem; can easily generate nodes at 100MB/sec
so 24hrs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

Uniform-cost search

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 $\geq \epsilon$
Time?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$
where C^* is the cost of the optimal solution
Space?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$
Optimal?? Yes—nodes expanded in increasing order of $g(n)$

Depth-first search

Expand deepest unexpanded node
Implementation:
fringe = 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?? $O(b^m)$: 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!
Optimal?? No

Depth-limited search

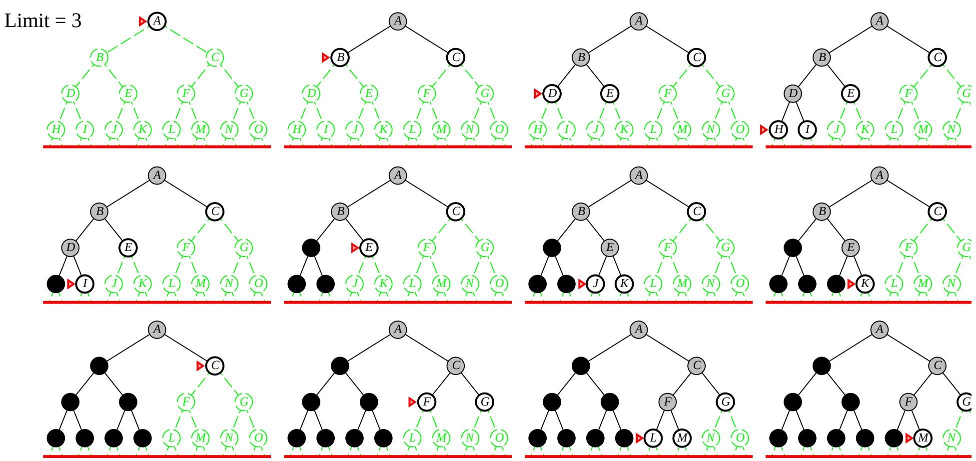
= depth-first search with depth limit l,
i.e., nodes at depth l have no successors
Recursive implementation:

```
function Depth-Limited-Search( problem, limit) returns soln/fail/-cutoff
    Recursive-DLS(Make-Node(Initial-State[problem]), problem, limit)

function 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

```
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 $l = 0$



Properties of iterative deepening search

Complete?? Yes
Time?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
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:

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

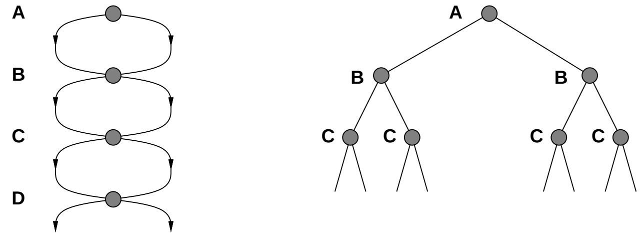
IDS does better because other nodes at depth d are not expanded
BFS can be modified to apply goal test when a node is **generated**

Summary of algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes*	Yes*	No	Yes, if $l \geq d$	Yes
Time	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d
Space	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd
Optimal?	Yes*	Yes	No	No	Yes*

Repeated states

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



Graph search

```
function Graph-Search( problem, fringe) returns a solution, or failure
    closed ← an empty set
    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
        if State[node] is not in closed then
            add State[node] to closed
            fringe ← InsertAll(Expand(node, problem), fringe)
    end
```


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 and Search
2. Uninformed search algorithms
3. Informed search algorithms
   Local search algorithms
4. Constraint Satisfaction Problem

Review: 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] 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
  - Hill-climbing
  - Simulated annealing
  - Genetic algorithms (briefly)
  - Local search in continuous spaces (very briefly)

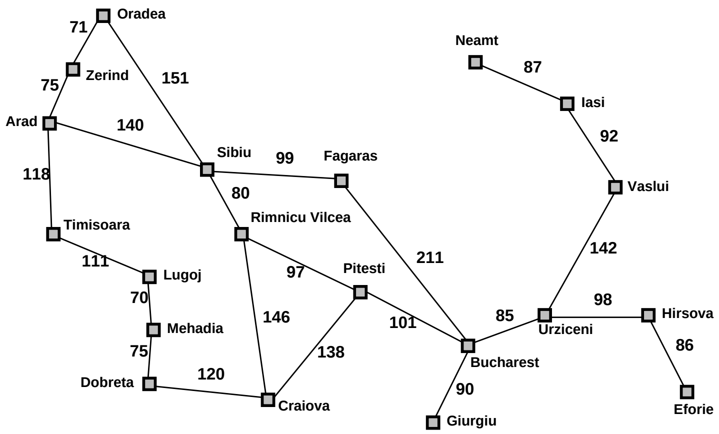
Best-first search

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

Romania with step costs in km



Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy search

Evaluation function $h(n)$ (**h**euristic)
= estimate of cost from n to the closest goal

E.g., $h_{\mathrm{SLD}}(n)$ = straight-line distance from n 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
Iasi → Neamt → Iasi → Neamt →
Complete in finite space with repeated-state checking
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
Evaluation function $f(n) = g(n) + h(n)$

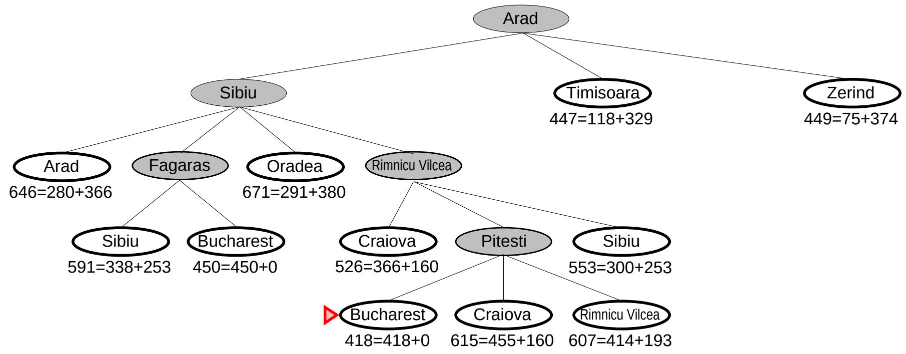
$g(n)$ = cost so far to reach n
$h(n)$ = estimated cost to goal from n
$f(n)$ = estimated total cost of path through n to goal

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

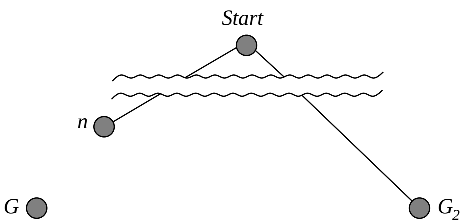
Theorem: A* search is optimal

A* search example



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.

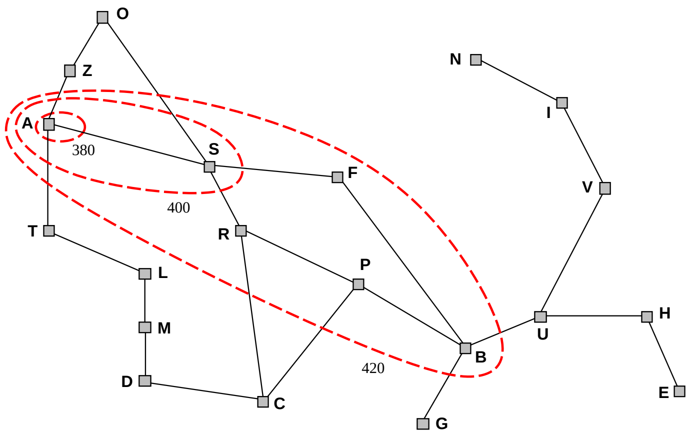


$$
\begin{aligned}
f(G_2) &= g(G_2) && \text{since } h(G_2) = 0 \\
&> g(G_1) && \text{since } G_2 \text{ is suboptimal} \\
&\geq f(n) && \text{since } h \text{ is admissible}
\end{aligned}
$$

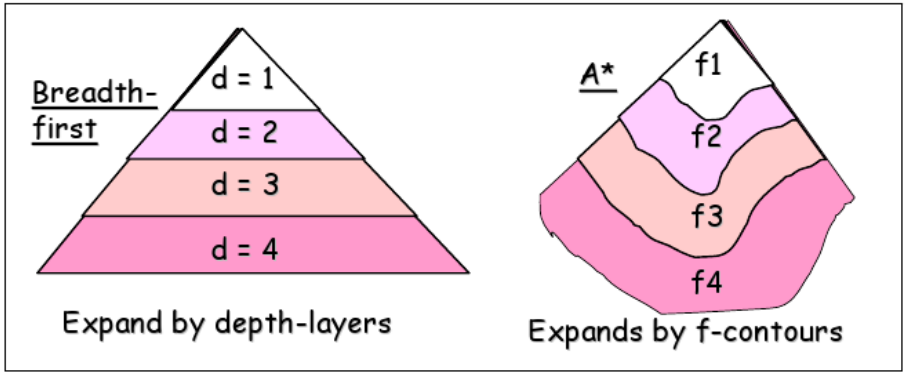
Since $f(G_2) > f(n)$, A* will never select G_2 for expansion

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 $f_i < f_{i+1}$



Astar vs. Depth search



Problem Solving and Search
Uninformed search algorithms
Informed search algorithms
Constraint Satisfaction Problem

Properties of A*

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$
Time?? Exponential in [relative error in h × 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 $f(n) > C^*$

Problem Solving and Search
Uninformed search algorithms
Informed search algorithms
Constraint Satisfaction Problem

Proof of lemma: Consistency

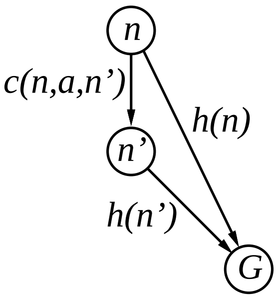
A heuristic is consistent if

$$h(n) \leq c(n, a, n') + h(n')$$

If h is consistent, we have

$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n, a, n') + h(n') \\ &\geq g(n) + h(n) \\ &= f(n) \end{aligned}$$

I.e., $f(n)$ is nondecreasing along any path.

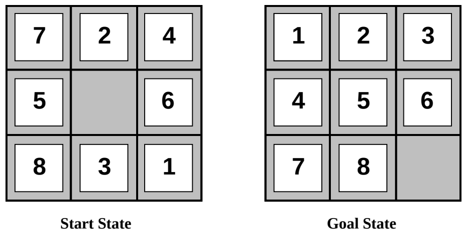


Problem Solving and Search
Uninformed search algorithms
Informed search algorithms
Constraint Satisfaction Problem

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)

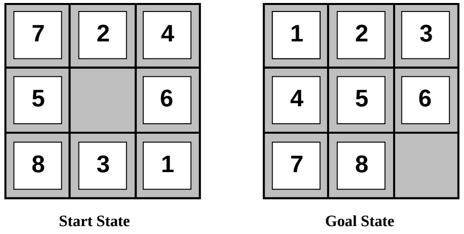


$h_1(S) =$??
$h_2(S) =$??

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)



$h_1(S) =$?? 6
$h_2(S) =$?? 4+0+3+3+1+0+2+1 = 14

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

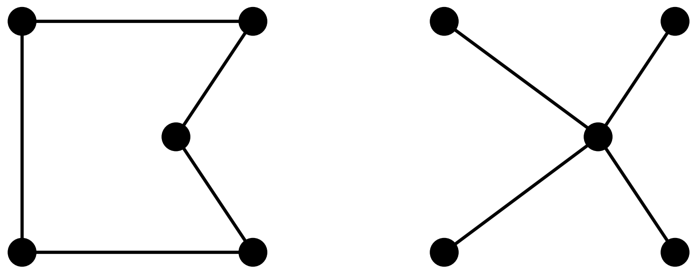
Relaxed problems

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

Relaxed problems contd.

Well-known example: travelling salesperson problem (TSP)
Find the shortest tour visiting all cities exactly once



Minimum spanning tree can be computed in $O(n^2)$
and is a lower bound on the shortest (open) tour

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

Outline

- ♢ Hill-climbing
- ♢ Simulated annealing
- ♢ Genetic algorithms (briefly)
- ♢ Local search in continuous spaces (very briefly)

Iterative improvement algorithms

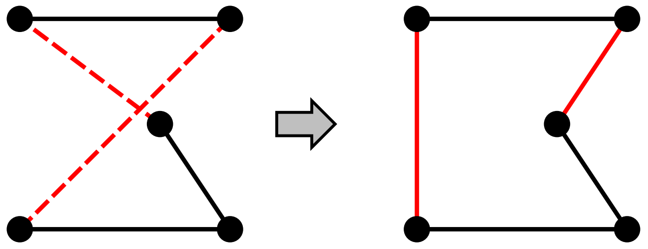
Then state space = set of "complete" configurations;
find **optimal** configuration, e.g., TSP
or, find configuration satisfying constraints, e.g., timetable

In such cases, can use iterative improvement algorithms;
keep a single "current" state, try to improve it

Constant space, suitable for online as well as offline search

Example: Travelling Salesperson Problem

Start with any complete tour, perform pairwise exchanges

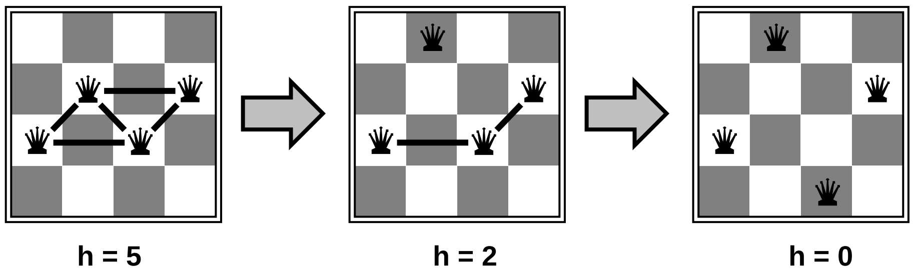


Variants of this approach get within 1% of optimal very quickly with thousands of cities

Example: n-queens

Put n queens on an $n \times n$ board with no two queens on the same row, column, or diagonal

Move a queen to reduce number of conflicts



Almost always solves n-queens problems almost instantaneously for very large n, e.g., $n = 1 million$

Hill-climbing (or gradient ascent/descent)

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*
  end

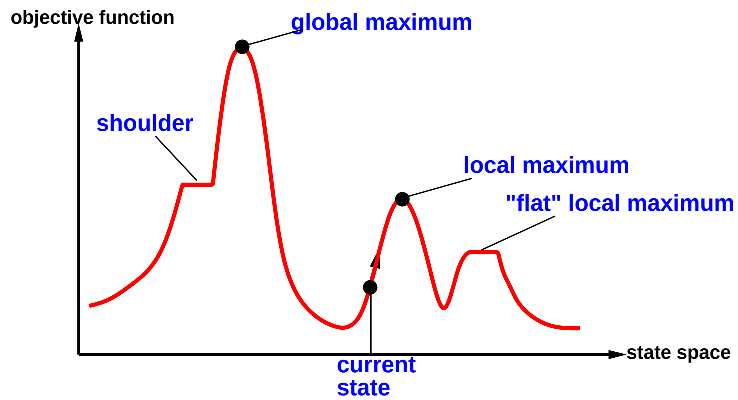
Example: n-queens

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	♛	13	16	13	16
♛	14	17	15	♛	14	16	16
17	♛	16	18	15	♛	15	♛
18	14	♛	15	15	14	♛	16
14	14	13	17	12	14	12	18

- Current cost 17
- 8 possible successor

Hill-climbing contd.

Useful to consider state space landscape



Random-restart hill climbing overcomes local maxima—trivially complete
Random sideways moves 😊escape from shoulders ☹loop on flat maxima

Simulated annealing

Idea: escape local maxima by allowing some "bad" moves
but gradually decrease their size and frequency

```
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, temp. 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^(Δ E/T)
```


Problem Solving and Search
Uninformed search algorithms
Informed search algorithms
Constraint Satisfaction Problem

Properties of simulated annealing

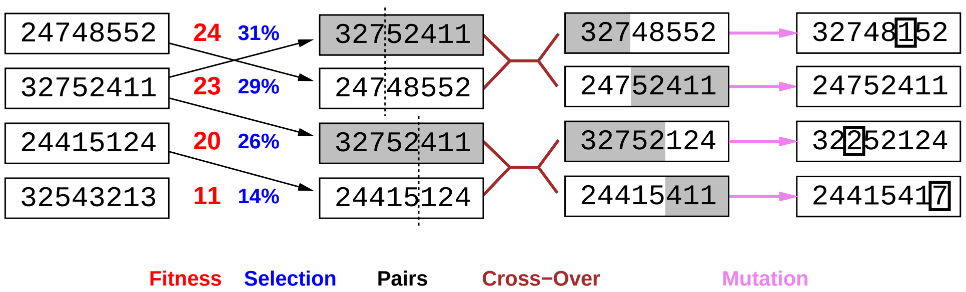
At fixed "temperature" T, state occupation probability reaches Boltzman distribution

$$p(x) = \alpha e^{\frac{E(x)}{kT}}$$

T decreased slowly enough $\Longrightarrow$ always reach best state x^*
because $e^{\frac{E(x^*)}{kT}}/e^{\frac{E(x)}{kT}} = e^{\frac{E(x^*)-E(x)}{kT}} \gg 1$ for small T
Is this necessarily an interesting guarantee??
Devised by Metropolis et al., 1953, for physical process modelling
Widely used in VLSI layout, airline scheduling, etc.

Problem Solving and Search
Uninformed search algorithms
Informed search algorithms
Constraint Satisfaction Problem

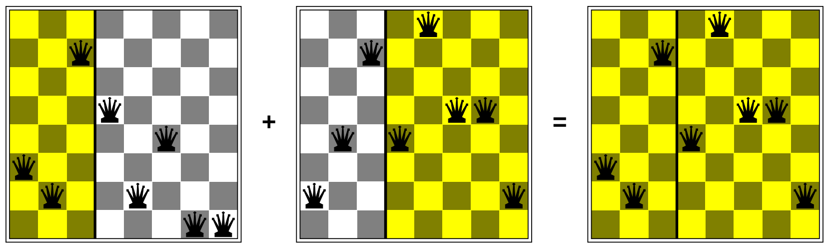
Genetic algorithms



Problem Solving and Search
Uninformed search algorithms
Informed search algorithms
Constraint Satisfaction Problem

Genetic algorithms contd.

GAs require states encoded as strings (GPs use programs)
Crossover helps **iff substrings are meaningful components**



GAs $\neq$ evolution: e.g., real genes encode replication machinery!

Continuous state spaces

Suppose we want to site three airports in Romania:
– 6-D state space defined by (x_1, y_2), (x_2, y_2), (x_3, y_3)
– objective function $f(x_1, y_2, x_2, y_2, x_3, y_3) =$
sum of squared distances from each city to nearest airport

Discretization methods turn continuous space into discrete space, e.g., empirical gradient considers $\pm\delta$ change in each coordinate

Gradient methods compute

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3} \right)$$

to increase/reduce f, e.g., by $\mathbf{x} \leftarrow \mathbf{x} + \alpha \nabla f(\mathbf{x})$

Sometimes can solve for $\nabla f(\mathbf{x}) = 0$ exactly (e.g., with one city).

Newton–Raphson (1664, 1690) iterates $\mathbf{x} \leftarrow \mathbf{x} - \mathbf{H}_f^{-1}(\mathbf{x}) \nabla f(\mathbf{x})$ to solve $\nabla f(\mathbf{x}) = 0$, where $\mathbf{H}_{ij} = \partial^2 f / \partial x_i \partial x_j$

Outline

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.
  $\Longrightarrow$ 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
$\Longrightarrow$ use depth-first search
3) Path is irrelevant, so can also use complete-state formulation
4) $b = (n - \ell)d$ at depth ℓ, hence $n!d^n$ leaves!!!!

Backtracking search

Variable assignments are commutative, i.e.,

[$WA = red$ then $NT = green$] same as
[$NT = green$ then $WA = red$]

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

$\Longrightarrow$ $b = d$ and there are d^n 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 $n \approx 25$

Backtracking search

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

function Recursive-Backtracking(assignment, csp) returns soln/fail-
ure
   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*
- local search
  - Hill-climbing
  - Simulated annealing
  - Genetic algorithms (briefly)
  - Local search in continuous spaces (very briefly)

Constraint Satisfaction and Backtracking