TDT4136 Introduction to Artificial Intelligence

Lecture 4: (A* and) Local search

Chapter (3/)4 in the textbook.

Pinar Ozturk

Norwegian University of Science and Technology
2021

Outline

- A* Search
- Local search algorithms
  - incremental vs iterative search
  - Hill Climbing, Simulated annealing, local beam, genetic algorithms
- Search in non-deterministic environments
- Search in partially-observable environments

Uninformed Search disadvantageous

- Last week: Uninformed search
- Uninformed search, systematically searching the search space blindly - not questioning where the goal may be in the space, and not using any domain-specific knowledge
- Search space is often very large. Time/space problems with such exhaustive search - think of chess.
- In such situations, better to use algorithms which does a more informed search

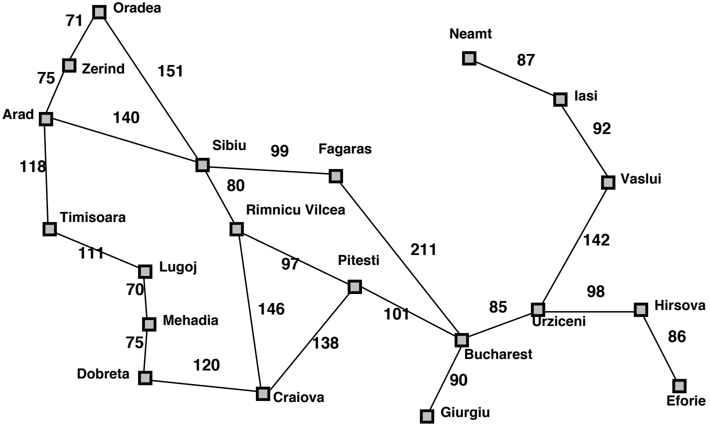
A* search

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 of the cheapest path from n to goal node
$f(n)$ = estimated cost of the cheapest solution through n to goal

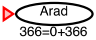
Heuristic-example

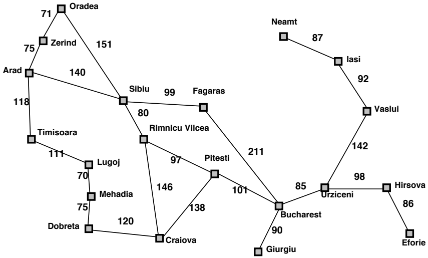


Straight–line distance
to Bucharest

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

A* search example

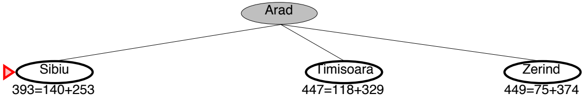


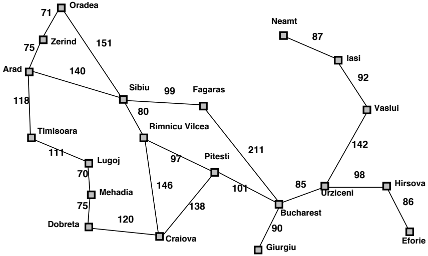


Straight−line distance to Bucharest

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

A* search example

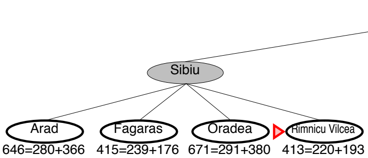


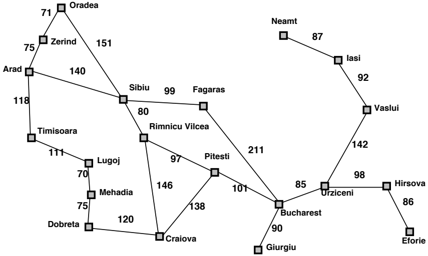


Straight–line distance to Bucharest

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

A* search example

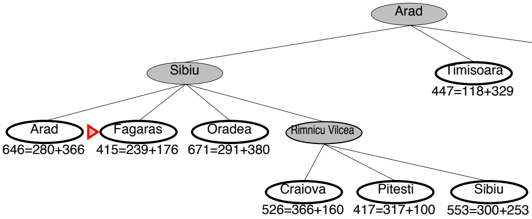




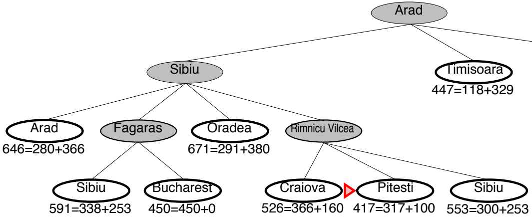
Straight–line distance
to Bucharest

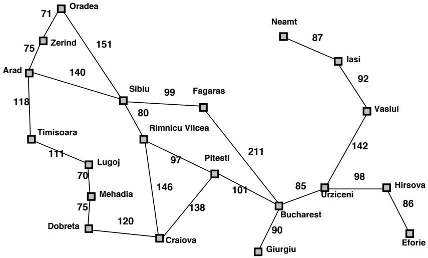
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* search example



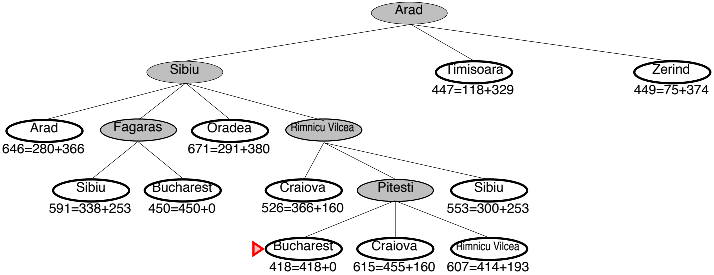
A* search example

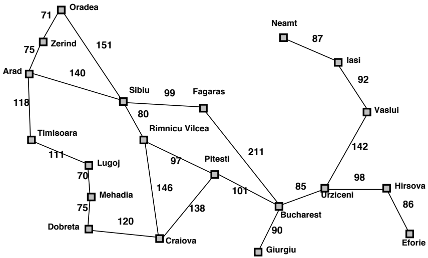




Straight−line distance
to Bucharest

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





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

Optimal path?

Properties of A*

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$.
I.e., if all step costs are $> \epsilon$ and b is finite.

Time??

Properties of A*

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

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

Space??

Properties of A*

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

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

Properties of A*

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

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

Space?? Main drawback. Keeps all nodes in memory

Optimal??

Optimality of A*

A* is optimal if

- the branching factor is finite
- arc costs are strictly positive
- for tree search: h is admissible and is non-negative
- for graph search: h is consistent(monotonic) and non-neg

Admissible: Does not overestimate the cost from node *n* to the goal node (Also require $h(n) \geq 0$, so $h(G) = 0$ for any goal G.)

Properties of A*

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

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

Space?? Keeps all nodes in memory

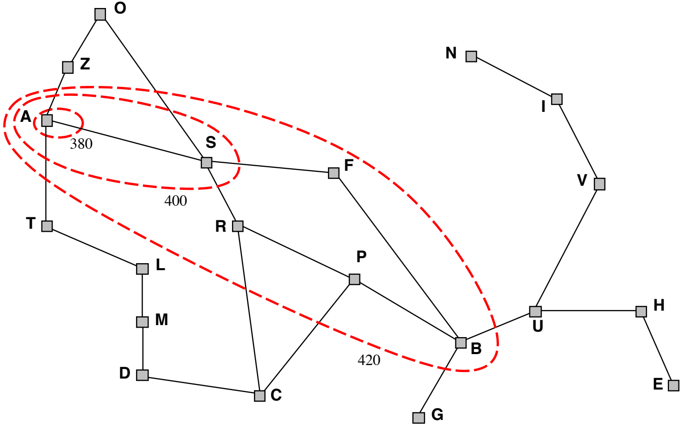
Optimal?? Yes—cannot expand f_{i+1} until f_i is finished

A* expands all nodes with $f(n) < C^*$ (where C* is the cost of optimal solution path)
A* expands some nodes with $f(n) = C^*$
A* expands no nodes with $f(n) > C^*$

Optimality of A*

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 $f \leq f_i$, where $f_i < f_{i+1}$



Optimality of A* Proof

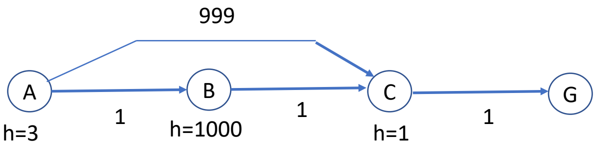
How to prove that A* with admissible heuristics cannot return a suboptimal path.

Proof by contradiction:

- Suppose C* is the optimal cost and A* with an admissible heuristic returns a suboptimal path.
- Suppose **n** is a node on the optimal path.
- If A* returns an nonoptimal path it means that *n* was not expanded.
- This means that:

$$
\begin{aligned}
f(n) &> C^\star && \text{otherwise } n \text{ would be expanded} \\
f(n) &= g(n) + h(n) && \text{by definition} \\
f(n) &= g^\star(n) + h(n) && \text{because n is on the optimal path} \\
f(n) &\leq g^\star(n) + h^\star(n) && \text{because of admissibility,} \\
f(n) &\leq C^\star && \text{by definition } \mathrm{C}^\star = \mathrm{g}^\star(\mathrm{n}) + \mathrm{h}^\star(\mathrm{n})
\end{aligned}
$$

Example – Admissibility-Optimality



Admissible?
Finds the optimal solution?

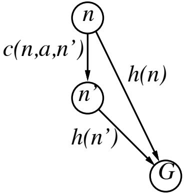
A* is optimally efficient

Optimal Efficiency: No other optimal algorithm using the same heuristic information is guaranteed to expand fewer nodes than A* - except it may be unlucky about how it breaks ties between nodes with f(n)$< C*$.

This is because any algorithm that does not expand all nodes with f(n)$< C*$ has the risk of missing the optimal solution.

Consistency -"Inequality of triangle"

A heuristic is **consistent** / **monotonous** if



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

If h is consistent, we have

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

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

A* graph search algorithm - pseudocode

Assumption: heuristic is consistent (hence admissible)

Theorem: If h(n) is consistent, A* Graph Search is optimal

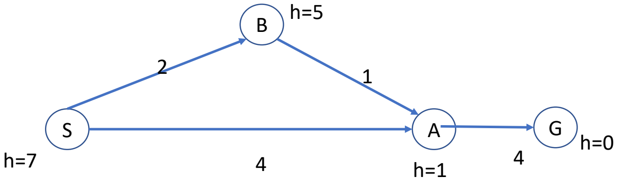
```
Start.g = 0;
Start.h = heuristic(Start)
FRONTIER = {Start}
CLOSED = {empty set}
    WHILE FRONTIER is not empty
            N = FRONTIER.popLowestF()
            IF state of N= GOAL RETURN N
            add  N to CLOSED
            FOR all children M of N not in CLOSED:
                        M.parent = N
                        M.g = N.g + cost(N,M)
                        M.h = heuristic(M)
                        add M to FRONTIER
            ENDFOR
    ENDWHILE
```

CLOSED is the list of nodes that are already expanded, i.e. REACHED minus FRONTIER (in the 4th ed of the textbook)

Graph search alg. does not re-expand nodes already expanded.

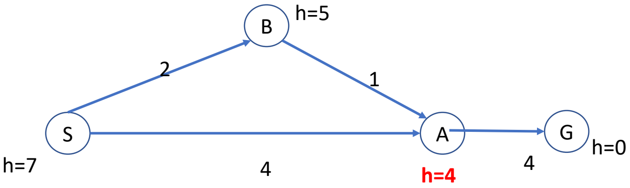
Would it find the optimal solution?

Example - More on A* graph search



Re-expansion of nodes in “closed” – How to avoid?

Example cont. Graph with consistent heuristic



Now heuristic is consistent
Node *A* does not now need to be taken out from closed and into frontier.

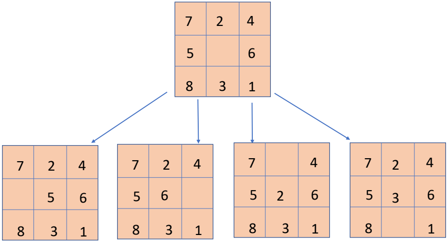
- When a new node N is generated:
  - If N is in *Closed* then discard N
  - If N is already in the frontier, then keep N with least f value.

Admissible heuristics

E.g., for the 8-puzzle:

Goal state is: Upper left tile is empty, in the rest of the grid: numbers 1-8 are in natural order (example from the book) .

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

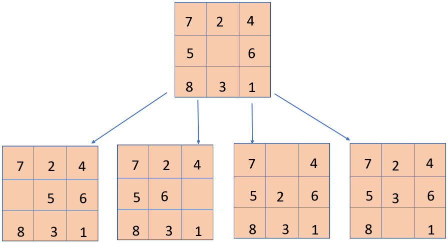


$h_1(S) =??$
$h_2(S) =??$

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)



$h_1(S) =$?? 8
$h_2(S) =$?? $3+1+2+2+2+3+3+2 = 18$
The shortest solution cost is 26 actions long

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

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

Local Search

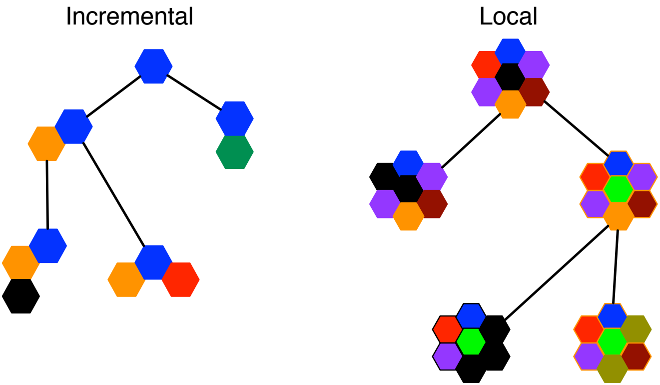
In many optimization problems, path is irrelevant;
the goal state itself is the solution

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

Incremental versus Local Search

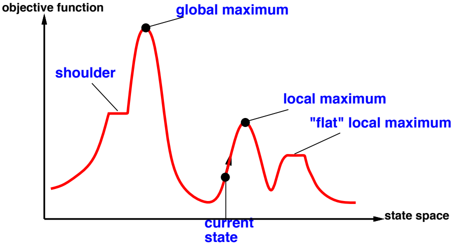


Properties of Local Search

- Low space complexity - only need to save one (or a set) of current solutions, NOT paths back to the start state.
- Time complexity varies, though recent work indicates major improvements over incremental search for problems with densely-packed optimal solutions.
- Satisficing - can often find *reasonably good* solutions quickly.
- Requires representations that are easy to *tweak* to generate search-space neighbors.
- Uses an **objective function** to evaluate solutions. Similar to a heuristic but for complete solutions
- Often portrayed as movement in a **landscape**.

State-space Landscape

Useful to consider **state space landscape**



Each point in the landscape represent a state in the "world", and has an "elevation".

If the elevation correspond to an objective function, then the aim is to find the highest peak. Then this is **Hill Climbing**

If elevation corresponds to the cost, then the aim is to find the lowest point. This is called **gradient descent.**

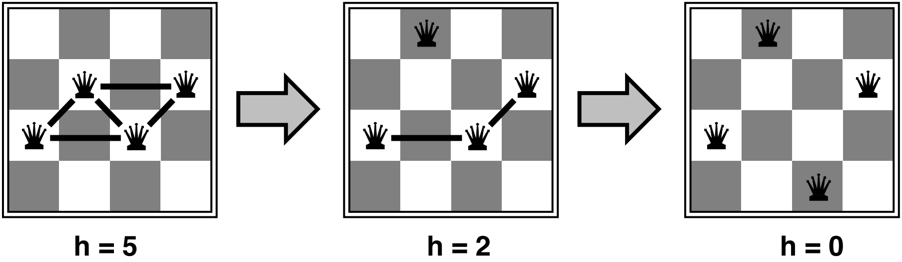
The negative of the cost function can be used as the objective function.

Example: n-queens

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

Heuristic cost$=$ number of conflicting pairs of queens

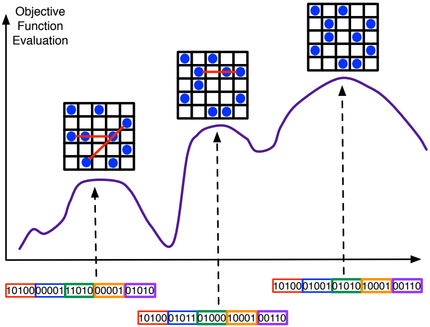
Move a queen (on the same column) to reduce number of conflicts.



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

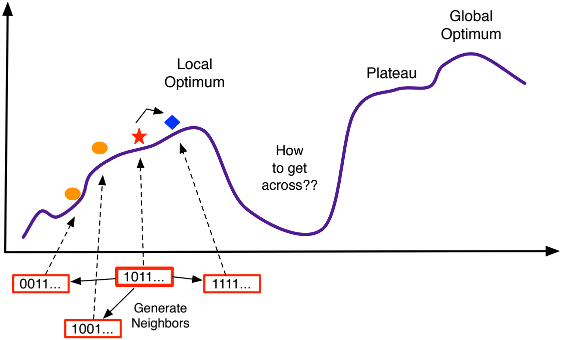
Hill-climbing

function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum
  current ← *problem*.INITIAL
  while *true* **do**
    neighbor ← a highest-valued successor state of *current*
    if VALUE(*neighbor*) ≤ VALUE(*current*) **then return** *current*
    current ← *neighbor*

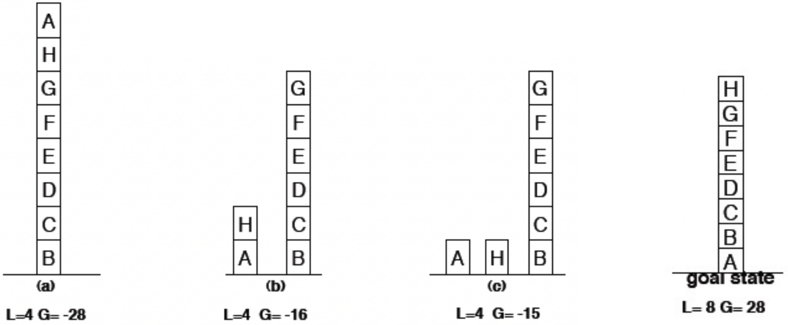


Hill-climbing - cont.

"Like climbing Everest
in thick fog with amnesia"



Effect of Evaluation Function in Hill climbing



Local evaluation function: Add 1 point for every block that is resting on the thing it is supposed to be resting on. Subtract 1 point for every block that is sitting on the wrong thing.

Global Eval function: For each block that has the correct support structure (i.e., the complete structure underneath it is exactly as it should be), add 1 point for every block in the support structure. For each block that has an incorrect support structure, subtract one point for every block in the existing support structure.

Hill-Climbing

Properties

- Greedy: always moves to states with immediate benefits (i.e., $\uparrow$ evals).
- Quick on smooth landscapes.
- Easily gets stuck on rough landscapes (e.g, the 8-puzzle state with h=1 in the previous slide)

Simulated annealing is a similar algorithm which tries to solve this problem.

Simulated annealing

Gradient Descent.

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

function SIMULATED-ANNEALING(*problem*, *schedule*) **returns** a solution state
  current ← *problem*.INITIAL
  for $t = 1$ **to** ∞ **do**
    $T \leftarrow$ *schedule*(t)
    if $T = 0$ **then return** *current*
    next ← a randomly selected successor of *current*
    $\Delta E \leftarrow$ VALUE(*current*) – VALUE(*next*)
    if $\Delta E > 0$ **then** *current* ← *next*
    else *current* ← *next* only with probability $e^{-\Delta E/T}$

Local beam search

Idea: keep k states instead of 1; choose top k of all their successors

Not the same as k searches run in parallel!
Searches that find good states recruit other searches to join them

Problem: quite often, all k states end up on same local hill

Idea: choose k successors randomly, biased towards good ones

Observe the close analogy to natural selection!

Genetic algorithms

- A successor state is generated by combining two parent states
- Start with k randomly generated states (population)
- A state is represented as a string over a finite alphabet (often a string of 0s and 1s)
- Evaluation function (fitness function). Higher values for better states.
- Produce the next generation of states by selection, crossover, and mutation

Genetic algorithms

Example: 8-queens

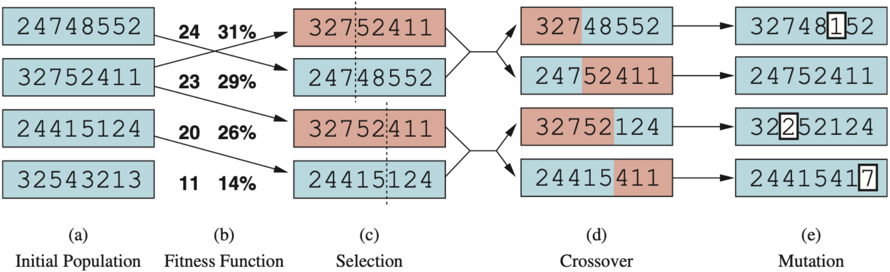


Figure 4.5 A genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by a fitness function in (b) resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

Fitness function (i.e, objective fn): number of non-attacking pairs
Selection: Probability of being regenerated in the next generation

Searching with non-deterministic actions

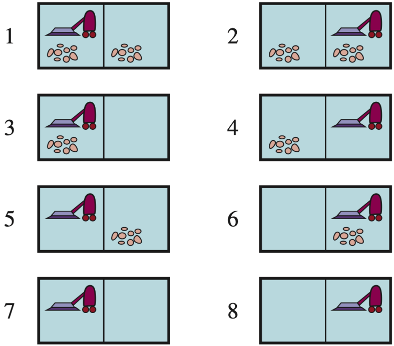
- So far, we have assumed that the actions are deterministic.
- In the real-world, things do not always go as expected.
- To account for different possible outcomes, we need to come up with a contingency plan instead of a single path of actions.

Example : the erratic vacuum world

We consider a vacuum world where the Suck action. has a non-deterministic effect:

- When applied to a dirty square, it cleans the square, but sometimes it. cleans an adjacent square too.
- When applied to a clean square, it may deposit dirt on the square.

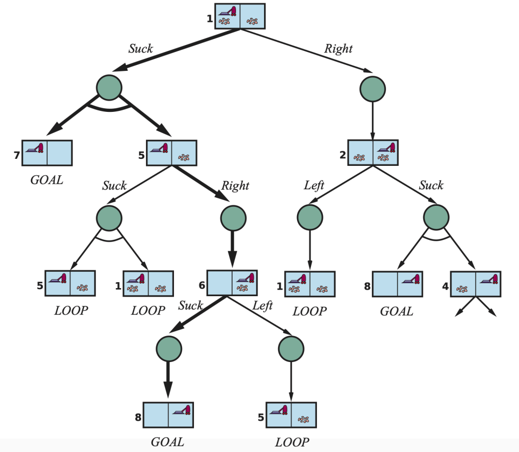
Example - cont



- States are "belief states" now, subset of actual states, e.g., {State 5, State 7}
- The state transition model can be defined to return a set of possible states, not a single outcome:
  RESULT(1, Suck)= {5,7}

- If the agent starts at State 1 then the following conditional plan solves the problem (State 7 and 8 are goal states):
  $[Suck, if\ \ State = 5\ then\ [Right, Suck]\ else\ []\ ]$

Example -cont



- starts to constructs a tree with AND (shown as circles) nodes representing "belif states" of the agent and OR nodes representing actions.
- solution is a subtree
- the usual tree search algorithms can be used for finding contingency AND-OR plans.

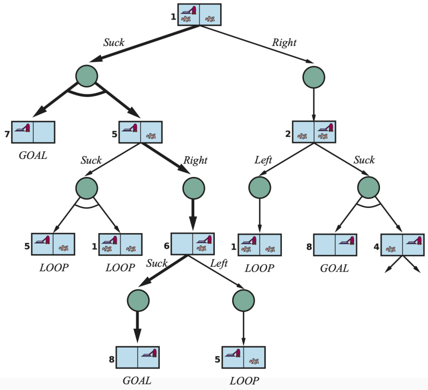
Search on AND-OR graphs for nondeterministic environments

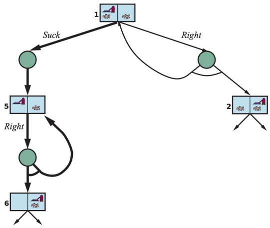
```
function AND-OR-SEARCH(problem) returns a conditional plan, or failure
  return OR-SEARCH(problem, problem.INITIAL, [])

function OR-SEARCH(problem, state, path) returns a conditional plan, or failure
  if problem.IS-GOAL(state) then return the empty plan
  if IS-CYCLE(path) then return failure
  for each action in problem.ACTIONS(state) do
      plan ← AND-SEARCH(problem, RESULTS(state, action), [state] + path])
      if plan ≠ failure then return [action] + plan]
  return failure

function AND-SEARCH(problem, states, path) returns a conditional plan, or failure
  for each s_i in states do
      plan_i ← OR-SEARCH(problem, s_i, path)
      if plan_i = failure then return failure
  return [if s_1 then plan_1 else if s_2 then plan_2 else ...if s_{n−1} then plan_{n−1} else plan_n]
```

Loops and Cyclic plans





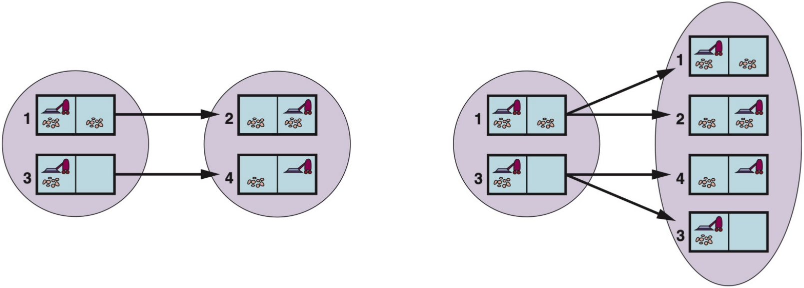
Solution (Slippery vacuum):
[*Suck*, *L*1 : *Right*, *if State* = 5 *then L*1 else Suck]

Solution (Erratic vacuum):
[*Suck*, *if State* = 5 *then* [*Right*, *Suck*] *else* []]

Searching in Partially Observable Environments

- So far, we have assumed that the agent knows exactly the state of its environment.
- In reality, an agent receives partial, possibly noised, observations.
- Therefore, the state can only be estimated - "belief state space".
- In this case, the agent needs to remember all its history of actions and observations in order to track the state.
- Solution for entirely sensorless problems: a sequence of actions
- Solution for a "partially-observing sensor": a contingency plan.

Predicting the next state with sensorless agents



Predicting the next belief state when the action *Right* is taken in deterministic (left) and non-deterministic (right)situations

Belief state space for sensorless, deterministic environments

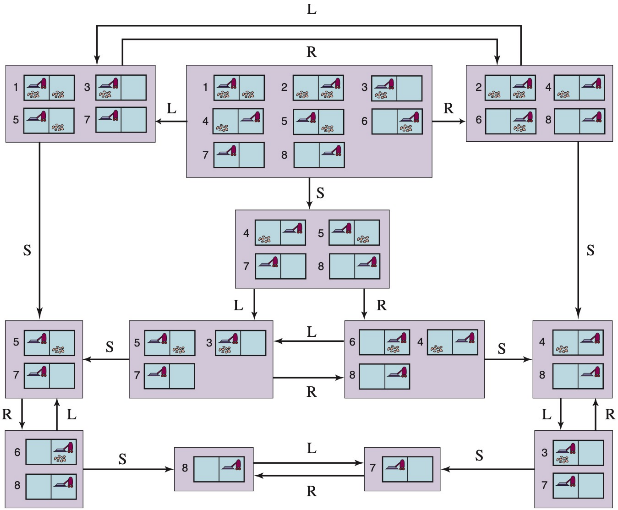
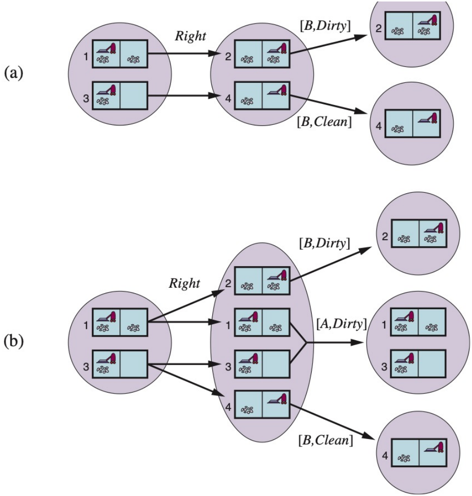


Figure 4.13 The reachable portion of the belief-state space for the deterministic, sensorless vacuum world. Each rectangular box corresponds to a single belief state. At any given point, the agent has a belief state but does not know which physical state it is in. The initial belief state (complete ignorance) is the top center box.

Transition model for local sensing agent



Initial belief state is $\{1, 3\}$. (a) In deterministic world, (b) Non-deterministic world.

Example AND-OR search tree for the local sensing deterministic. vacuum world

Suppose the Initial percept is $[A, dirty]$

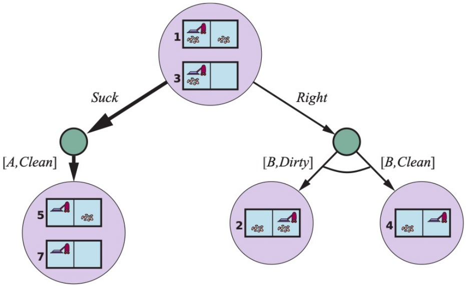


Figure 4.15 The first level of the AND–OR search tree for a problem in the local-sensing vacuum world; *Suck* is the first action in the solution.

Notice that a solution is a conditional plan - because there is perception in local-sensing agents.

Summary