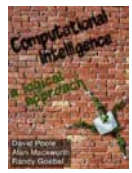
Chapter 4 (part 2): Searching

- Heuristic Search (4.5)
- Refinement to Search Strategies (4.6)
- Constraint Satisfaction Problems (4.7)



D. Poole, A. Mackworth, and R. Goebel, *Computational Intelligence: A Logical Approach*, Oxford University Press, January 1998

Heuristic Search (4.5)

- Previous methods do not take into account the goal (where they are trying to go) until they stumble on a goal node.
- Often there is extra knowledge that can be used to guide the search: heuristics.
- We use *h(n)* as an estimate of the distance from node *n* to a goal node, h(n) is a positive real value.
- *h(n)* is an underestimate if it is less than or equal to the actual cost of the shortest path (straight line) from node *n* to a goal.
- *h(n)* uses only readily obtainable information about a node.

Heuristic Search (4.5) (cont.)

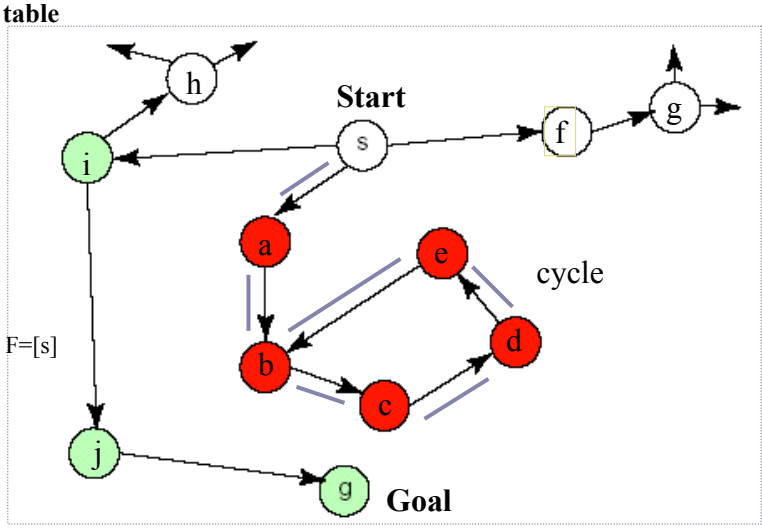
Best-First Search

- **Idea:** always choose the node n on the frontier with the smallest h-value (Min h(n)).
- It treats the frontier as a priority queue ordered by h.
- It uses space exponential in path length.
- It isn't guaranteed to find a solution, even if one exists.
- It doesn't always find the shortest path.

Lookup table

h[a,10]
h[b,7]
h[c,5]
h[d,8]
h[e,10]
h[f,15]
h[g,16]
h[i,13]
h[h,14]

Frontier F=[s]
F=[a]
F=[b]
F=[c]
F=[d]
F=[e]
F=[b]
Looping



Best-first Search does not always converge

Heuristic Search (4.5) (cont.)

- Heuristic Depth-first Search
  - It's a way to use heuristic knowledge (problem-specific information) in depth-first search.
  - Idea: order the neighbors of a node (by h) before adding them to the front of the frontier.
  - Locally chooses which sub-tree to develop, but still does depth-first search. It explores all paths from the node at the head of the frontier before exploring paths from the next node.
  - Space is linear in path length. It isn't guaranteed to find a solution. It can get led up the garden path.

Example of Heuristic Depth-First Search

- Start= s
- Goal=g
- Initially: F=[s]

Neighbors of s=[i,f,a]

Heuristic:

h(a)=10

h(i)=13

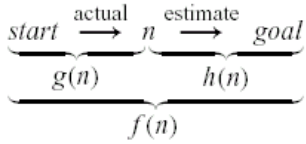
h(f)=15

F=[a,i,f] (sorted with respect to the values of h)

Perform a depth-first search starting from a then sort again with respect to heuristic, if failure backtrack to i and then to f.

Heuristic Search (4.5) (cont.)

- A* Search
  - A* search takes the path to a node and heuristic value into account.
  - Let g(n) be the cost of the path (shortest path) found to node n.
  - Let h(n) be the estimate of the cost from n to a goal.
  - Let f(n) = g(n) + h(n). It is an estimate of a path from the start to a goal via n.



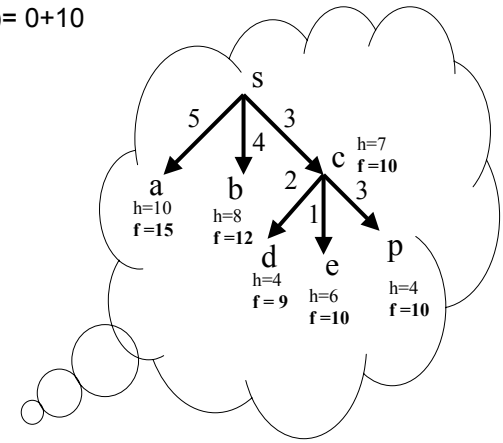
Heuristic Search (4.5) (cont.)

- A* Search Algorithm
  - A* is a mix of lowest-cost-first and best-first search.
  - It treats the frontier as a priority queue ordered by f (n).
  - It always chooses the node n on the frontier with the lowest estimated distance from the start to a goal node constrained to go via the node n (Minimize f).

Example of A*

- If s=start node f(s)=g(s) + h(s)= 0+10
  Neighbors of s=[a,b,c]
  h(a)=10 g(a)=5 f(a)=15
  h(b)=8 g(b)=4 f(b)=12
  h(c)=7 g(c)=3 f(c)=10

  Initially: F=[s_{20}] replaced by its neighbors
  F=[c_{10},b_{12},a_{15}]



Neighbors of c=[d,e,p]
F=[d_9,e_{10},p_{10},b_{12},a_{15}]
etc.. The function f should be minimized at each iteration, you may need to backtrack if failing.
The sub-optimal path obtained so far is: [s,c,d]

Heuristic Search (4.5) (cont.)

- Admissibility of *A**
  - If there is a solution, *A** always finds an optimal solution —the first path to a goal selected— if
    - the branching factor is finite
    - arc costs are bounded above zero (there is some ϵ> O such that all of the arc costs are greater than ϵ), and
    - *h(n)* is an underestimate of the length of the shortest path from *n* to a goal node.

Summary (Heuristics)

- Methods based on a function h(n) that underestimates the actual cost of the shortest-cost path from n to a goal
- Best-first search
  - Idea: Choose an element of the frontier that appears closest to the goal
  - The h-values are a-priori known, they are less or equal to the exact distance

Summary (Heuristics) (Cont.)

- Best-first search (cont.)
  - Algorithm (BFS)

    - Step 1: Choose a starting node S & a goal node G;
      initialize: F = [S]
    - Step 2: Compute N(s) (neighbors of S) and assign
      F ← N(S) % h values stored in F
    - Step 3: Select & Store the node x ∈ F such that h(x) min
    - Step 4: Compute N(x)
    - Step 5: If x = G then {Print path; stop}
      else {F ← F V N(x) (expand frontier) go to step 3}

Summary (Heuristics) (Cont.)

Heuristic Depth-first Search

- Idea: Ordering the neighbors before adding them to the front of the frontier
- Algorithm (HDFS)

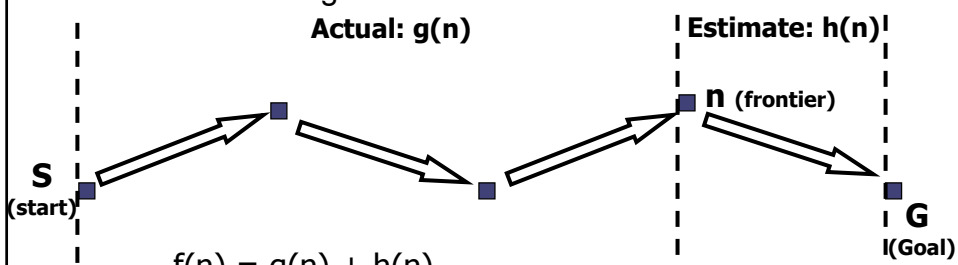
- Step 1: Choose a starting node S & a goal node G and assign F = [S]
- Step 2: Compute N(S) (neighbors of S) & assign F ← N(S)
- Step 3: Sort N(S) with respect to h (heuristic function)
- Step 4: Select and store the node x such that h(x) min (best neighbor)
- Step 5: If x ≠ G then { assign S ← x go to step 2} else {print path stop}

] Perform depth-first search

Summary (Heuristics) (Cont.)

A* Search

- Idea: It uses g(n), the length (lowest cost) of the path found from a start node to frontier node n, as well as the heuristic function h(n), the estimated path length from node n to the goal



f(n) = g(n) + h(n)
≅ total path length

Summary (Heuristics) (Cont.)

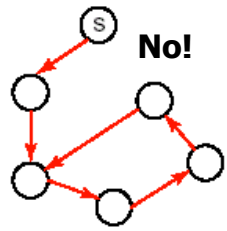
- A* search (cont.)
  - Algorithm (A*)
    - Step 1: Choose a starting node S & a goal node G, assign F ← [S]
    - Step 2: Compute f(S) (f-value of S)
    - Step 3: Compute N(S) and assign F ← N(S)
    - Step 4: Compute f(x) $\forall$ x $\in$ F (f(x) = g(x) + h(x))
    - Step 5: Select and store node x; x $\in$ F such that f(x) min
    - Step 6: Assign F ← N(x) % replace x by its neighbors
    - Step 7: If x = G stop else go to step 5

Summary (Heuristics) (Cont.)

- Admissibility of A*
  - A* finds an optimal path, if one exists, and the first path P* found to a goal is optimal
  - P* is called the admissibility of A*

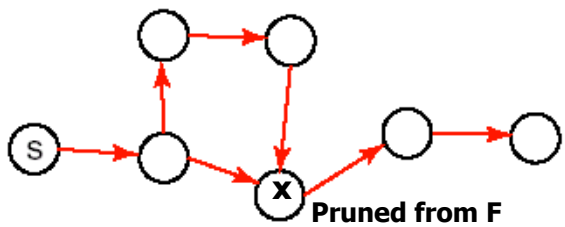
Refinements to Search Strategies (4.6)

- Cycle checking



  - You can prune a node *n* that is on the path from the start node to *n*. This pruning cannot remove an optimal solution.
  - This checking can be done either when the node is added to the frontier or when the node is selected

Refinements to Search Strategies (4.6) (cont.)



- Multiple-path Pruning
  - You can prune a node *n* that you have already found a path to.
  - Multiple-path pruning subsumes a cycle check.
  - This entails storing all nodes you have found paths to.

Refinements to Search Strategies (4.6) (cont.)

- Dynamic Programming
  - Idea: for statically stored graphs, build a table of *dist(n)* the actual distance of the shortest path from node *n* to a goal.
  - This can be built backwards from the goal:

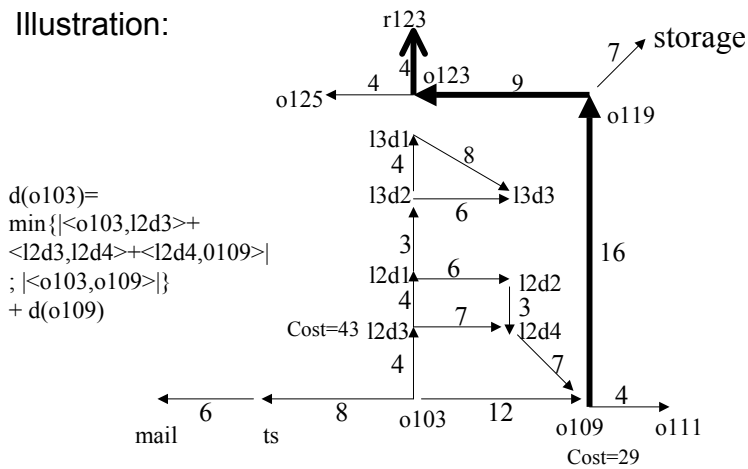
m=succ(n)

$$dist(n) = \begin{cases} 0 & \text{if } is\_goal(n), \\ \min_{\langle n,m\rangle \in A}(|\langle n, m\rangle| + dist(m)) & \text{otherwise.} \end{cases}$$

  - This can be used locally to determine what to do.
  - There are two main problems:
    - You need enough space to store the graph.
    - The *dist* function needs to be recomputed for each goal.

3

Illustration:



Refinements to Search Strategies (4.6) (cont.)

- Dynamic Programming (cont.)
  - Since r123 is a goal, we have:
  - dist (r123) = 0
    Continuing with a shortest first search from r123, you obtain:
  - dist (o123) = 4
    dist (o119) = 13
    dist (o1O9) = 29
    dist (l2d4) = 36
    dist (l2d2) = 39
    dist (o1O3) = 41
    dist (l2d3) = 43
    dist (l2d1) = 47
  - **Remark:** to determine the shortest path from o1O3 to r123, you have to compare 4 + 43 (cost via l2d3) with 12 + 29 (going directly to o1O9). Therefore, the robot should go directly to o1O9 from o1O3.

Refinements to Search Strategies (4.6) (cont.)

- Constraint Satisfaction Problems (CSP)
  - Multi-dimensional Selection Problems
    - Given a set of variables, each with a set of possible values (a domain), assign a value to each variable that either
      - satisfies some set of constraints:
        satisfiability problems — “hard constraints”
      - minimizes some cost function, where each assignment of values to variables has some cost:
        optimization problems — “soft constraints”
    - Many problems are a mix of hard and soft constraints.

Refinements to Search Strategies (4.6) (cont.)

- Relationship to Search
  - The path to a goal isn't important, only the solution is (only the goal is of interest!).
  - Many algorithms exploit the multi-dimensional nature of the problems.
  - There are no predefined starting nodes.
  - Often these problems are huge, with thousands of variables, so systematically searching the space is infeasible.
  - For optimization problems, there are no well-defined goal nodes.

Refinements to Search Strategies (4.6) (cont.)

- Posing a Constraint Satisfaction Problem
  - A CSP is characterized by
    - A set of variables $V_1, V_2, \ldots, V_n$.
    - Each variable V_i has an associated domain $\mathbf{D}_{Vi}$ of possible values.
    - For satisfiability problems, there are constraint relations on various subsets of the variables which give legal combinations of values for these variables.
    - A solution to the CSP is an *n*-tuple of values for the variables that satisfies all the constraint relations.

Refinements to Search Strategies (4.6) (cont.)

- Example: scheduling activities
  - Variables: *A*, *B*, *C*, *D*, *E* that represent the starting times of various activities.
    A: Starting time for activity A
  - Domains:
    - $\mathbf{D}_A = \{1, 2, 3, 4\}$
    - $\mathbf{D}_B = \{1, 2, 3, 4\}$,
    - $\mathbf{D}_C = \{1, 2, 3, 4\}$
    - $\mathbf{D}_D = \{1, 2, 3, 4\}$,
    - $\mathbf{D}_E = \{1, 2, 3, 4\}$

Refinements to Search Strategies (4.6) (cont.)

- Constraints:

time (1).
time (2).
time (3).
time (4).
1<2. 1<3. 1<4. 2<3. 2<4. 3<4.
1≠2. 1≠3. 1≠ 4. 2≠3. 2≠4. 3≠4.
2≠1. 3≠1. 4≠1. 3≠2. 4≠2. 4≠3.
1=1. 2=2. 3=3. 4=4.

Find all values of A,B,C,D,E that obey the following constraint: The query takes the following form:

? time (A) ∧ time (B) ∧ time (C) ∧ time (D) ∧ time (E) ∧

constraints

$$(B \neq 3) \wedge (C \neq 2) \wedge (A \neq B) \wedge (B \neq C) \wedge$$
$$(C < D) \wedge (A = D) \wedge (E < A) \wedge (E < B) \wedge$$
$$(E < C) \wedge (E < D) \wedge (B \neq D).$$

(B ≠ C) means that activities B and C cannot be done simultaneously!

Refinements to Search Strategies (4.6) (cont.)

Solving CSP's

- The finite constraint satisfaction problem is NP-hard. We can:
  - Try to find algorithms that work well on typical cases even though the worst case may be exponential
  - Try to find special cases that have efficient algorithms
  - Try to find efficient approximation algorithms
  - Develop parallel and distributed algorithms (each processor deals with each variable value change)
- Methods such as hill climbing, randomized strategies (simulated annealing) and genetic algorithm are more appropriate for theses problems!

Refinements to Search Strategies (4.6) (cont.)

CSP as Graph Searching

- A CSP can be seen as a graph-searching algorithm:
  - A node corresponds to an assignment of a value to all of the variables, and the neighbors of a node corresponds to changing one variable value to a local value:

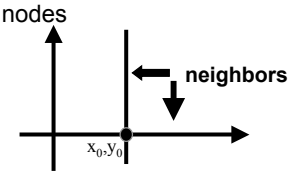
    $N(x_0, y_0) = \{(x_1, y_0), (x_0, y_1)\}$ such that
    $d(x_o, x_1)$ min; $d(y_0, y_1)$ min
  - Totally order the variables, $V_1, \ldots, V_n$.

    A node assigns values to the first j variables.

    The neighbors of node $\{V_1|v_1, \ldots, V_j|v_j\}$ are the nodes $\{V_1|v_1, \ldots, V_j|v_j, V_{j+1}|v_{j+1}\}$ for each $v_{j+1} \in \mathbf{D}_{Vj+1}$ but neighbors.

    The start node is the empty assignment {}.

    A goal node is a total assignment that satisfies the constraints.



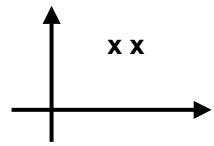
Refinements to Search Strategies (4.6) (cont.)

- Hill Climbing

  - Many search spaces are too big for systematic search.
  - A useful method in practice for some consistency and optimization problems is hill climbing:
    - Assume a heuristic value for each assignment of values to all variables.
    - Maintain a single node corresponding to an assignment of values to all variables.
    - Select a neighbor of the current node that improves the heuristic value to be the next current node.

Refinements to Search Strategies (4.6) (cont.)

- Selecting Neighbors in Hill Climbing

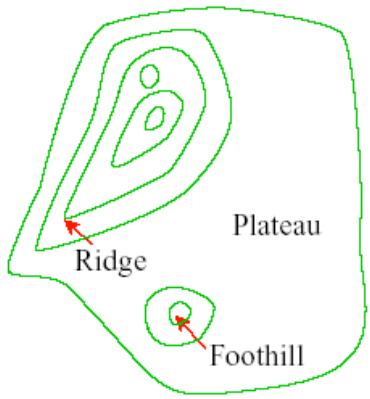


  - When the domains are unordered, the neighbors of a node correspond to choosing another value for one of the variables.
  - When the domains are ordered, the neighbors of a node are the adjacent values for one of the dimensions.
  - If the domains are continuous, you can use gradient descent (minimize the cost): change each variable proportional to the gradient of the heuristic function in that direction.
  - The value of variable V_i *goes from* v_i *to* $v_i \overset{+}{-} \eta \frac{\partial h}{\partial V_i}$
  - The direction -∇h leads us abruptly to the minimum of h

$$\nabla h = \left[\frac{\partial h}{\partial V_1}, \frac{\partial h}{\partial V_2}, \ldots, \frac{\partial h}{\partial V_n} \right]^T$$

Problems with Hill Climbing

- **Foothills** local maxima (minima) that are not global maxima (minima) ($\nabla h = 0$)
- **Plateaus** heuristic values are uninformative (no direction is better; evaluation function is flat)
- **Ridges** the top is reached with ease (with steeply sloping) but the top may slope only very gently toward a **peak** (search oscillate from side to side of the peak, making little progress)



Refinements to Search Strategies (4.6) (cont.)

- Randomized Algorithms
  - Consider two methods to find a maximum value:
    - Hill climbing, starting from some position, keep moving uphill, & report maximum value found
    - Pick values at random & report maximum value found
  - Combinations:
    - random-restart hill climbing
    - two-phase search: random search, then hill climbing (we hope that the random phase will find the mountain & then the hill climbing will find the peak!)

Refinements to Search Strategies (4.6) (cont.)

- Randomized Algorithms (cont.)
  - Simulated Annealing (SA)
    - Allow the search to take some downhill steps to escape the local maximum
    - Instead of picking the best move, however SA picks a random move, if the move improves the situation, it is executed, otherwise, SA makes the move with some probability
    - This probability decreases exponentially with the "badness" of the move - ∇h is worsened

Refinements to Search Strategies (4.6) (cont.)

- Randomized Algorithms (cont.)
  - Simulated Annealing (SA) (cont.)
    - A second parameter T is also used to determine the probability. At higher values of T, "bad" moves are more likely to be allowed
    - As T tends to zero, they become more & more unlikely, until the algorithm behaves more or less like hill-climbing

Refinements to Search Strategies (4.6) (cont.)

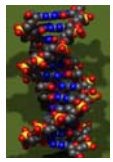
- Evolutionary Algorithms
  - Beam Search (BS) (stochastic)
    - Similar to hill-climbing, but k nodes are being maintained instead of just one
    - At each stage of the algorithm, you determine the set of all of the neighbors of all of the current nodes, and select the k-nearest of these & repeat with this new set of nodes
    - If the selection is performed randomly with a probability that favors nodes with higher heuristic (or fitness) function ⇒ Stochastic BS
    - SBS is similar to asexual reproduction
      Each node leads to its local mutation & then you proceed with survival of the fittest

Refinements to Search Strategies (4.6) (cont.)

- Evolutionary Algorithms (cont.)
  - Genetic Algorithm 
    - Similar to SBS but the new elements of the population are combinations of pair of nodes
    - Choose pairs of nodes, and then create new off-spring by taking values for the off-spring's variables from one of the parents and some from the other parent
    - Similar to DNA splice in sexual reproduction
    - Cross-over & mutation are the two operations involved

Refinements to Search Strategies (4.6) (cont.)

- Evolutionary Algorithms (cont.)
  - Genetic Algorithm (cont.)
    - Algorithm:



Step 1: Select randomly pairs of nodes privileging the fitter individuals

Step 2: For each pair, perform a cross-over

Step 3: Randomly mutate values by choosing other values for some randomly selected variables

Step 4: Repeat until k node are created & go to step 1