Informatics 2D: Tutorial 2

Informed Search and Constraint Satisfaction

Week 3

1 Informed Search

We saw in the lectures a graph representing the road map of part of Romania, see Figure 1. The cost of a path is the distance via the road, as given on the graph. We also have a table of straight-line distances from each town to Bucharest.

(a) Show that using greedy best-first search with the straight-line heuristic function, h_{SLD}, does not give an optimal solution when looking for a path from Arad to Bucharest.

(b) Suppose that you have the following straight-line distances from Fagaras to: Neamt 140km, Iasi 175km, Vaslui 175km, Urziceni 180km, Hirsova 230km, Giurgiu 220km, Pitesti 50km, Rimnicu Vilcea 50km, Craiova 180km, Sibiu 60km. What happens when you try to use greedy best-first search to find a path from Iasi to Fagaras?

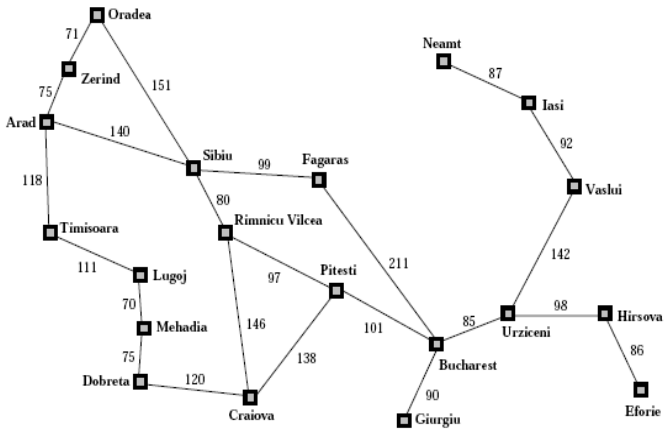
(c) We can use A^* search in this problem; h_{SLD} is an admissible heuristic that can be combined with the actual distance of the path so far to get a new heuristic f. Show that f finds an optimal solution in part (a) and solves the problem in part (b).

Answers

For both greedy best-first search and $A^*search$ you should go through (at least some of) the search tree, and show which nodes are expanded. Below are the search graphs for greedy best-first search and A^* search for the Arad to Bucharest problem (taken from R&N).

(a) Greedy best-first search should find the route Arad $\rightarrow$ Sibiu $\rightarrow$ Fagaras $\rightarrow$ Bucharest, which is 450km. But the route Arad $\rightarrow$ Sibiu $\rightarrow$ Rimnicu Vilcea $\rightarrow$ Pitesti $\rightarrow$ Bucharest is 418km.

(b) Greedy best-first search should loop between Neamt and Iasi, since the heuristic value of Neamt is less than the heuristic value of Vaslui.



Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Dobreta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Figure 1: The Romania map from R&N with a table of straight-line distances to Bucharest

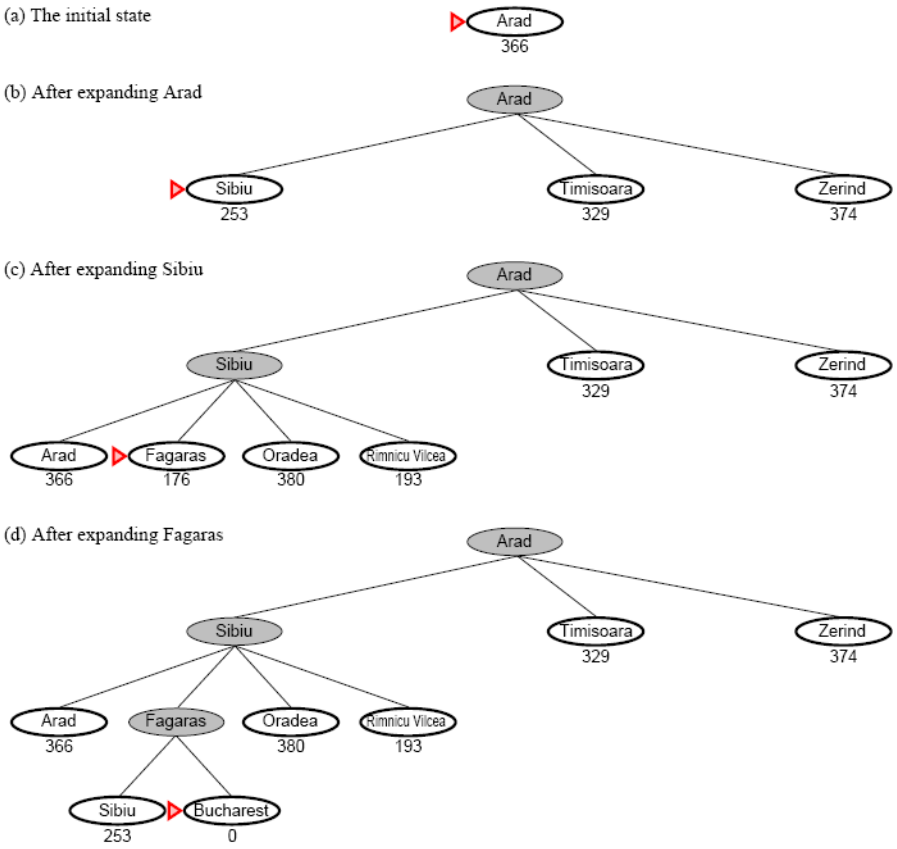


Figure 2: Search graph for greedy best-first search for the Arad to Bucharest problem (taken from R&N).

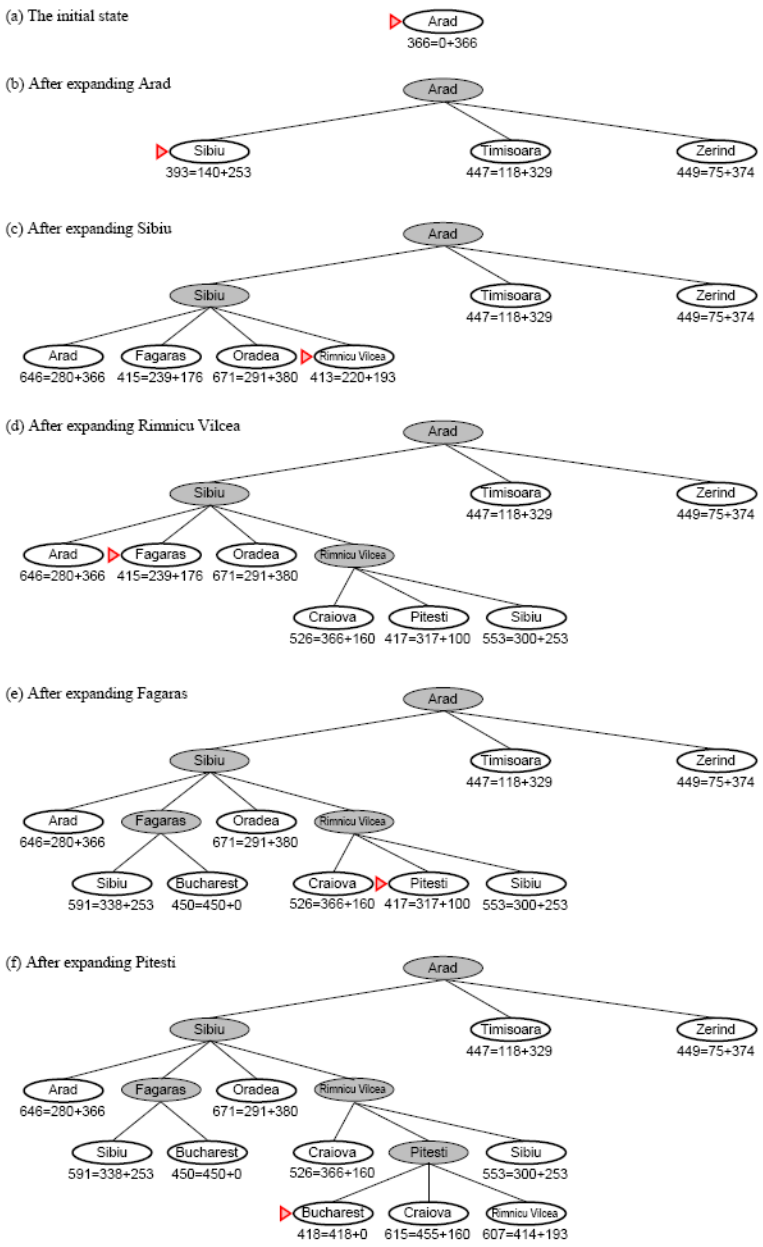


Figure 3: Search graph for A* search for the Arad to Bucharest problem (taken from R&N)

(c) A^* search should find the route Arad $\rightarrow$ Sibiu $\rightarrow$ Rimnicu Vilcea $\rightarrow$ Pitesti $\rightarrow$ Bucharest for part (a), and the route Iasi $\rightarrow$ Vaslui $\rightarrow$ Urziceni $\rightarrow$ Bucharest $\rightarrow$ Fagaras for part (b).

For (b) the A^* values are (for the initial transition):
f(Neamt) = 140 + 87 = 227
f(Iasi) = 175 = 175
f(Vaslui) = 175 + 92 = 267
f(Urziceni) = 180 + 234 = 414
f(Bucharest) = 176 + 319 = 495
f(Hirsova) = 230 + 332 = 562
f(Giurgui) = 220 + 319 = 539
f(Pitesti) = 50 + 420 = 470
f(Fagaras) = 0 + 530 = 530
f(Rimnicu Vilcea) = 50 + 517 = 567
f(Craiova) = 180 + 558 = 738

2 Heuristics

(Taken from R&N Chapter 3)

Sometimes there is no good evaluation function for a problem, but there is a good comparison method: a way to tell whether one node is better than another, without assigning numerical values to either. Show that this is enough to do a Best-First search. Is there an analog for A^*?

Answers

If we assume the comparison function is transitive, then we can sort a list of nodes using it, and choose the node that is at the head of the list.

A^* relies on the division of the total cost estimate $f(n)$ into the cost-so-far and the cost- to-go. If we have comparison operators for each of these, then we can prefer to expand a node that is better than other nodes on both comparisons. Unfortunately, there may not be a node with these properties. In which case, the tradeoff between $g(n)$ and $h(n)$ cannot be realized without numerical values.

3 The Crop Allocation Problem

Consider the following problem in bio-dynamic farming (where some crops grow better next to particular crops)[1] for the specific land division shown in Figure 4.

[1] Adapted from an original problem set by Mellish & Fisher

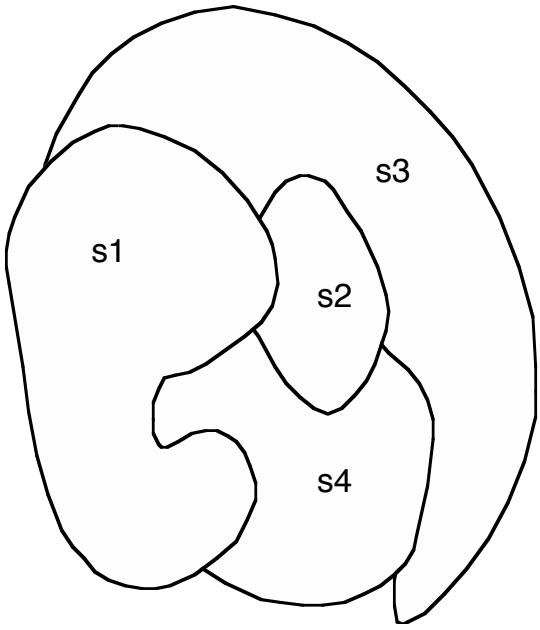


Figure 4: The Bio-Dynamic Farming Problem

The figure shows the allocation of a piece of land for planting four different crops using the constraints of bio-dynamic farming. In this kind of farming, the idea is that there are groups of crops that develop better if set in particular arrangements. Also the balance of nutrients in the soil is used to decide what to plant where. Here are the constraints according to the current levels of nutrients in the soil:

1. Sector 1 (s1) can be planted with one of the following crops: {cabbage, kale, broccoli, cauliflower }
2. Sector 2 (s2) can be planted with one of the following crops: {cabbage, kale, broccoli}
3. Sector 3 (s3) can be planted with one of the following crops: {kale}
4. Sector 4 (s4) can be planted with one of the following crops: {kale, broccoli}

The constraint here is that we do not want two sectors that are adjacent to each other to be planted with the same crops

How does this look when expressed as a constraint satisfaction problem (CSP)? What are the stages that the AC-3 algorithm goes through in obtaining arc consistency for this example? (see Figure 5 for the AC-3 algorithm)

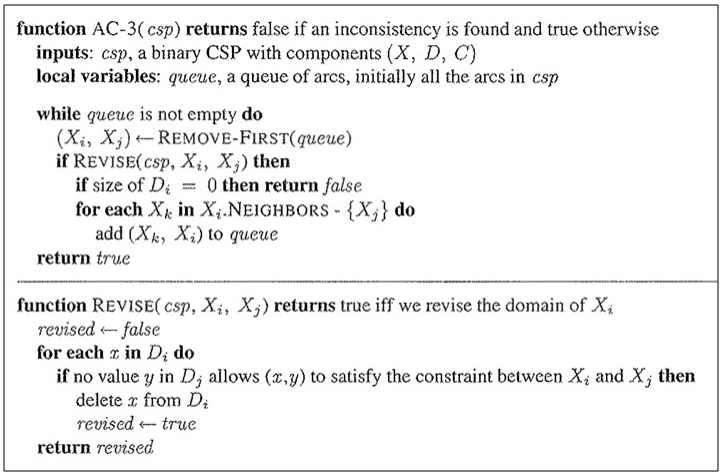


Figure 5: The AC-3 algorithm

Answer

Variables: s1, s2, s3 and s4

Domains.

domain(s1, [cabbage, kale, broccoli, cauliflower]).
domain(s2, [cabbage, kale, broccoli]).
domain(s3, [kale]).
domain(s4, [kale, broccoli]).

Constraints:

s1 $\neq$ s2 $\neq$ s3 $\neq$ s4

AC-3

The initial queue is:

[s1→s2, s1→s3, s1→s4, s2→s3, s2→s4, s3→s4, s2→s1, s3→s1, s4→s1, s3→s2, s4→s2, s4→s3]

The stages of the following can be shown incrementally on the board. You will want to show the values lists (domains) for the variables being updated as this all takes place.

Don't feel that you have to see this question through to the end, if the students understand the algorithm.

queue is

[s1→s2, s1→s3, s1→s4, s2→s3, s2→s4, s3→s4, s2→s1, s3→s1, s4→s1, s3→s2, s4→s2, s4→s3]

Revise (s1→s2) = False

queue is

[s1→s3, s1→s4, s2→s3, s2→s4, s3→s4, s2→s1, s3→s1, s4→s1, s3→s2, s4→s2, s4→s3]

Revise (s1→s3) = True, domain(s1,[cabbage,broccoli,cauliflower])
add []

queue is

[s1→s4, s2→s3, s2→s4, s3→s4, s2→s1, s3→s1, s4→s1, s3→s2, s4→s2, s4→s3]

Revise (s1→s4) = False

queue is

[s2→s3, s2→s4, s3→s4, s2→s1, s3→s1, s4→s1, s3→s2, s4→s2, s4→s3]

Revise (s2→s3) = True, domain(s2,[cabbage,broccoli])
add [s1→s2]

queue is

[s2→s4, s3→s4, s2→s1, s3→s1, s4→s1, s3→s2, s4→s2, s4→s3, s1→s2]

Revise (s2→s4) = False

queue is

[s3→s4, s2→s1, s3→s1, s4→s1, s3→s2, s4→s2, s4→s3, s1→s2]

Revise (s3→s4) = False

queue is

[s2→s1, s3→s1, s4→s1, s3→s2, s4→s2, s4→s3, s1→s2]

Revise (s2→s1) = False

queue is

[s3→s1, s4→s1, s3→s2, s4→s2, s4→s3, s1→s2]

Revise (s3→s1) = False

queue is [s4→s1, s3→s2, s4→s2, s4→s3, s1→s2]

Revise (s4→s1) = False

queue is

[s3→s2, s4→s2, s4→s3, s1→s2]

Revise (s3→s2) = False

queue is

[s4→s2, s4→s3, s1→s2]

Revise (s4→s2) = False

queue is

[s4→s3, s1→s2]

Revise (s4→s3) = True, domain(s4,[broccoli])
add [s1→s4, s2→s4]

queue is

[s1→s2, s1→s4, s2→s4]

Revise (s1→s2) = False

queue is

[s1→s4, s2→s4]

Revise (s1→s4) = True, domain(s1,[cabbage,cauliflower])
add [s2→s1, s3→s1]

queue is

[s2→s4, s2→s1, s3→s1]

Revise (s2→s4) = True, domain(s2,[cabbage])
add [s1→s2, s3→s2]

queue is

[s2→s1, s3→s1, s1→s2, s3→s2]

Revise (s2→s1) = False

queue is

[s3→s1, s1→s2, s3→s2]

Revise (s3→s1) = False

queue is

[s1→s2, s3→s2, s4→s2]

Revise (s1→s2) = True, domain(s1,[cauliflower])
add [s3→s1, s4→s1]

queue is

[s3→s2, s4→s2, s3→s1, s4→s1]

Revise (s3→s2) = False

queue is

[s4→s2, s3→s1, s4→s1]

Revise (s4→s2) = False

queue is

[s3→s1, s4→s1]

Revise (s3→s1) = False

queue is

[s4→s1]

Revise (s4→s1) = False

queue is []