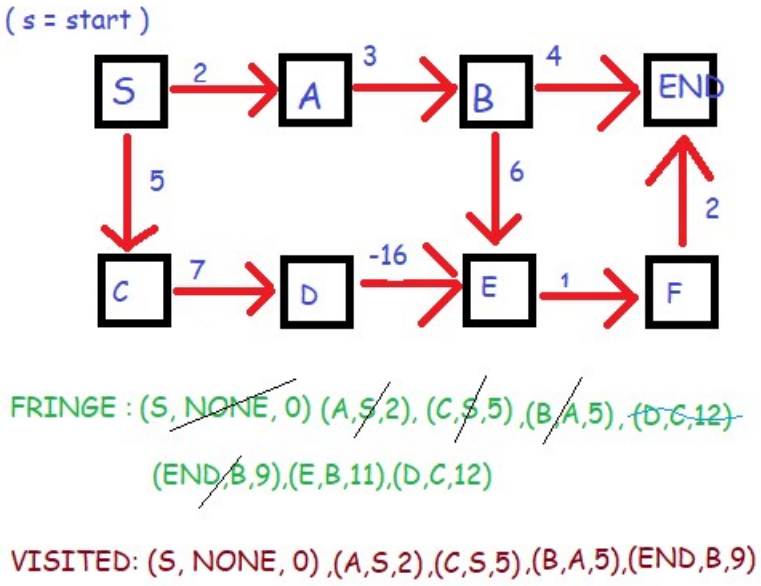
Thomas Mulvey
3/16/2019
CS_158 Midterm
Dr. Doboli

Midterm 1
CSC 158, Spring 2019

1. (10 pts) Give an example of a state-space with negative costs (some arcs have less than 0 costs). Show that UCS applied to that state-space is not optimal.



UCS is not optimal with negative weights. A path "p1" may have a high current g(n) (start to current node) cost and may not be visited in the Priority Queue. Later, "p1" could have a negative weight to make it the optimal one, but a path "p2" with a lower g(n) could be completed before it. Here is an example, where the State is represented as (node, parent_node, cost so far). The path S->A->B->END was found with cost 9. The path S->C->D->E->F was ignored due to the first path's cost was less than S->C->D. The path S->C->D->E->F cost is 5+7-16+1+2 = -1, which is the optimal path. The path S->A->B->END was not the optimal path but was found first.

2. (10 pts) (a) Compare iterative deepening search (IDS) with DFS in terms of space complexity and memory complexity.

DFS:

<u>Time Complexty</u>: If accessing a node in the tree is O(1) with a branching factor of "b" and with max depth D, then the total # of nodes in the tree is b*b*b…*b = b^M = O(b^M) when visiting every node in the tree in the worst case.

It should also be known, instead of describing the tree's branching factor and max depth, the time complexity can also be represented as # of nodes and # of edges visited = O(V + E) where V is the # of nodes and E is the # of edges, because in the worst case, all of them are visited. Those are just two different ways of writing it.

<u>Space Complexity</u>: Let the length of the solution path = m. For every node in this path

Thomas Mulvey
3/16/2019
CS_158 Midterm
Dr. Doboli

m, you must store its siblings so you know where you can go next. For every node, there is b extra nodes to store in the stack, resulting in O=(b*m) space complexity (linear).

IDS:

<u>Time Complexty</u>: Assuming we have a well-balanced tree with depth =d and branching factor b. The nodes on the max depth given are expanded once, the depth-1 are expanded twice, etc., etc. This can be written as

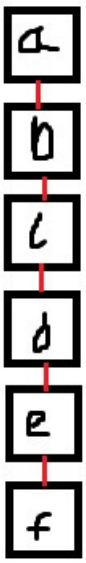
(d)*b + (d-1)b^2 + … + 3b^(d-2) + 2b^(d-1) + b^d
== Summation [(D+1-i)*b^i] from i=0 to i=d
== O(b^d)

<u>Space Complexity</u>: The exact same situation as DFS. Given a depth d, the space complexity is =O(b*d). Both IDS and DFS can be =O(depth_of_sol) time complexity by storing the siblings in an array/linked_list, and you can keep only the pointer to the first element in the list. Either way, the space complexity is linear for both.

IDS is good for when you have a limited amount of memory to use and won't accept a solution after depth = k. You want the shallowest solution like BFS finds. Slower-performance is accepted here.

(b) Describe a situation in which iterative deepening search performs worse than depth-first search (it visits more nodes) = $O(n^2)$ versus O(n).

IDS performs =O(n^2) for an unbalanced tree (linear list of nodes). See below.



DFS will visit a->b->c->d->e->f , just N nodes.

IDS will visit a, a->b, a->b->c, a->b->c->d,
a->b->c->d->e, a->b->c->d->e->f . That is
1+2+3+4+5+6 nodes.
n + (n-1) + … + (n-d+1) nodes total.

$$= \sum_{i=d}^{0} (i)$$

Ignoring the constants of the # of depth expansions (assuming large n values), we have n + n + … + n = n^2.

Thus in an unbalanced tree (linear tree), IDS performs worse (n^2) than DFS (n).

Thomas Mulvey
3/16/2019
CS_158 Midterm
Dr. Doboli

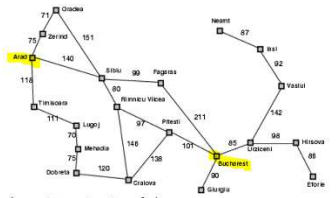
3. (30 pts) Sometimes it is better to search from both start and end position of a state space – bidirectional search. You expand nodes from the start state forward (successors) and you expand nodes from the end state backwards (predecessors). When expanding a node from the forward fringe check if it is in the backward fringe and stop if you find it.

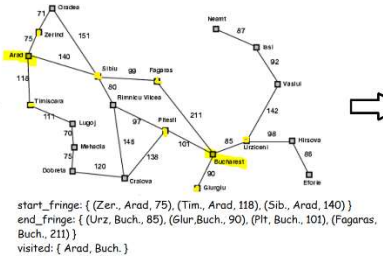
a) (10 pts) Write the pseudo-code of a bidirectional search algorithm using UCS starting from both ends: start and end.

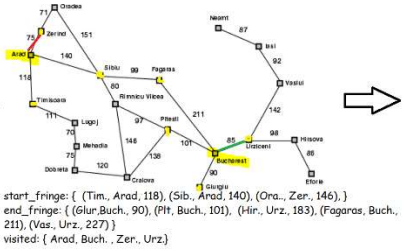
```
function bidirectoin(Start_State, End_State):
  start_set <- set()
  end_set <- set()

  start_set.put(Start_State)
  end_set.put(End_State)
  WHILE start_set not empty and end_set not empty; do
    If start_set not empty:
      s <- start_set.get()
      if s == End_State or s in end_set:
        return SUCCESS
      forall children of s:
        if children not in start_set:
          start_set.add(children)

    If end_set not empty:
      e <- end_set.get()
      if e == Start_State or e in start_set:
        return SUCCESS
      forall children of e:
        if children not in end_set:
          end_set.add(children)

  return FAILURE
```

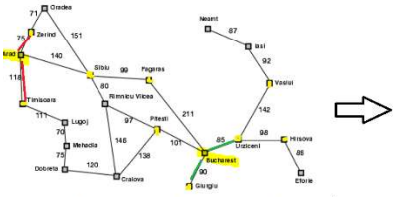
Thomas Mulvey
3/16/2019
CS_158 Midterm
Dr. Doboli

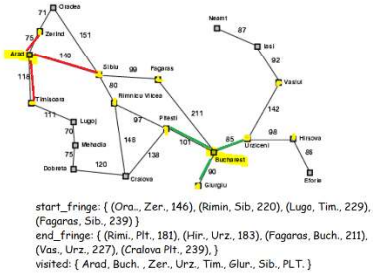
b) (10 pts) Apply the bidirectional search algorithm to the map of Romania – starting node – Arad, end node Bucharest.

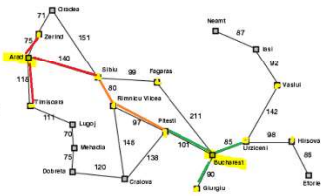












c) (5 pts) Comment on the bidirectional search optimality, time and space complexity. Compare it to the simple UCS.

Simple UCS will have =O(b^d) space complexity, where b is branching factor and d is distance. The time complexity is also =O(b^d). The time complexity for bidirectional search is =O(b^(d/2) + b^(d/2)). The space complexity is also =O(b^(d/2)).

d) (5 pts) Can you use A* algorithm with bidirectional search? If yes, explain how. Hint: what heuristics you needs, etc.

A* can be merged with a bidirectional search. The heuristic would be how far away a child from the start state is to the end state, and how far away a child from end state is from the start state. In the case of Romania map, we would want straight line distance from child to opposite state, for both start and end state. In the best case, these two states meet in the middle. Otherwise, they branch out separately and you run slower than regular A*.

4. (10 pts) Which algorithm would you use to find the path between two web pages? Would you consider bidirectional search? Why or why not?

I'm assuming we are using a system like Wikipedia, where many things are linked together. A simple BFS could take forever, and DFS would most likely just not stop diving deeper and deeper into the first link. I would consider A* from both ends, where the heuristic of one side

Thomas Mulvey
3/16/2019
CS_158 Midterm
Dr. Doboli

is how close a linked article's title is to the other. It would be hard to get a good, admissible heuristic but would be interesting to try and optimize. Could use a word2vec to compare article names and what not. The costs for links are all the same though, so its really just BFS with heuristics.

5. (25 pts) Apply the constraint satisfaction problem framework to the Sudoku game below:

		1	
	3		4
3		4	
	2		

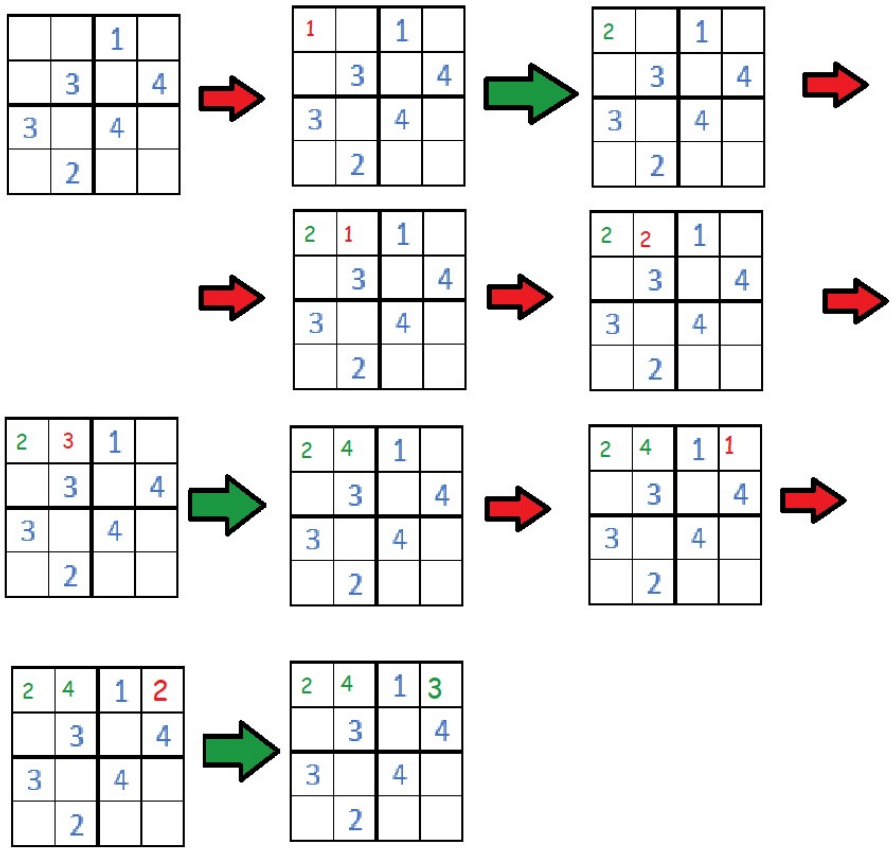
The purpose of the game is to fill the empty squares with digits 1 to 4 such that each of the digits (1,2,3 and 4) appears only once on each row, column and a 2x2 box (indicated by bold lines).

a) (5 pts) What are the variables and what is their domain of value (label the variables). What are the constraints?

Variables: Empty spots on the grid [row][col] .
Domains: {1-4} (in this case of a 4x4 board. A 9x9 would be 1-9)
Constraints: In each 2x2 section of the grid, can only place {1-4} with no repeats. On the entire grid, each row needs values {1-4}, each unique again. The same goes for the columns.

Thomas Mulvey
3/16/2019
CS_158 Midterm
Dr. Doboli

b) (10 pts) Use the backtracking algorithm for the Sudoku problem above for the first three variable assignment. Show all your steps.



The backtracking will basically brute force each square until it is valid. In my case above, I chose to work row by row. This can be improved greatly b keeping tracking of domains.

c) (10 pts) Use the forward-checking algorithm (with the minimum remaining values, most constrained variable and least constraining value heuristics) for the Sudoku problem for the first four variable assignments. Explain the choice of each variable and each value. Compare the backtracking with forward-checking in terms of performance.

Thomas Mulvey
3/16/2019
CS_158 Midterm
Dr. Doboli

	c1	c2	c3	c4
r1			1	
r2		3		4
r3	3		4	
r4		2		

DOMAINS:

[r1][c1] : {1,2,3,4}
[r1][c2] : {1,2,3,4}
[r1][c3] : {1,2,3,4}
[r1][c4] : {1,2,3,4}
[r2][c1] : {1,2,3,4}
[r2][c2] : {1,2,3,4}
[r2][c3] : {1,2,3,4}
[r2][c4] : {1,2,3,4}
[r3][c1] : {1,2,3,4}
[r3][c2] : {1,2,3,4}
[r3][c3] : {1,2,3,4}
[r3][c4] : {1,2,3,4}
[r4][c1] : {1,2,3,4}
[r4][c2] : {1,2,3,4}
[r4][c3] : {1,2,3,4}
[r4][c4] : {1,2,3,4}

AFTER INITIAL STATE'S DOMAIN WIPEOUT

[r1][c1] : { 2,4}
[r1][c2] : { 4}
[r1][c4] : {2,3}
[r2][c1] : {1,2 }
[r2][c3] : {2 }
[r3][c2] : {1 }
[r3][c4] : {1,2}
[r4][c1] : {1,4}
[r4][c3] : {3}
[r4][c4] : {1,3}

---

	c1	c2	c3	c4
r1			1	
r2		3		4
r3	3		4	
r4		2		

With the Min remaning values, Most Constrained Var and Least Constrained val heuristics,

[r1][c2], [r2][c3], and [r3][c2] all have 1 remaining value. They all 4 when altered will all effect 3 othe domains. We will pick [r3][c2] because it will make the lower left 2x2 square have only 1 variable left.

---

	c1	c2	c3	c4
r1			1	
r2		3		4
r3	3	1	4	
r4		2		

new domains after placing [r3][c2] = 1:

[r1][c1] : { 2,4}
[r1][c2] : { 4}
[r1][c4] : {2,3}
[r2][c1] : {1,2 }
[r2][c3] : {2 }
[r3][c4] : { 2}
[r4][c1] : { 4}
[r4][c3] : {3}
[r4][c4] : {1,3}

Thomas Mulvey
3/16/2019
CS_158 Midterm
Dr. Doboli

	c1	c2	c3	c4
r1			1	
r2		3		4
r3	3	1	4	
r4		2		

4 domains are tied for the least sized domain heuristic: [r1][c2], [r2][c3], [r3][c4], & [r4][c1]

The next choice will be [r4][c1] because it will alter the most domains, and finish the lower most square:

---

	c1	c2	c3	c4
r1			1	
r2		3		4
r3	3	1	4	
r4	4	2		

new domains after placing [r3][c2] = 1:

[r1][c1] : { 2, }
[r1][c2] : { 4}
[r1][c4] : {2,3}
[r2][c1] : {1,2 }
[r2][c3] : {2 }
[r3][c4] : { 2}
[r4][c3] : {3}
[r4][c4] : {1,3}

---

	c1	c2	c3	c4
r1	2		1	
r2		3		4
r3	3	1	4	
r4	4	2		

as you can see, most of the domains all the domains have 1-2 values.

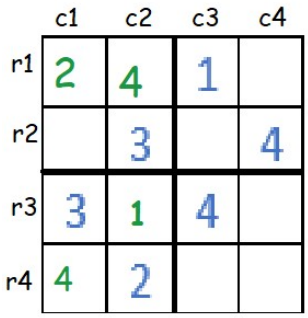
all domains of size 1 (besides [r3][c4]) alter 3 other domains. Now we need to find the least constrained val.

[r1][c1] is the least constrained from 3 other vars, so that will be picked next

new domains after placing [r1][c1] = 1

[r1][c2] : { 4}
[r1][c4] : { 3}
[r2][c1] : {1, }
[r2][c3] : {2 }
[r3][c4] : { 2}
[r4][c3] : {3}
[r4][c4] : {1,3}

Thomas Mulvey
3/16/2019
CS_158 Midterm
Dr. Doboli



The most constrained variables are [r2][c2] and [r1][c2] (5). Pick a random one of these for the 4th variable assignment. Ex: [r1][c2] = 4

post domain wipeout:

[r1][c4] : { 3}
[r2][c1] : {1, }
[r2][c3] : {2 }
[r3][c4] : { 2}
[r4][c3] : {3}
[r4][c4] : {1,3}

The backtracking is much slower than the forward checking. It has to branch out for every failed attempt at placing a value. The forward checking only branched once in the first step and that is it. It might branch out again for [r4][c4], but that is it. The first placement of backtracking already had more branch widening. The domain wipeout and the heuristics help vastly for the performance of forward-checking.

Thomas Mulvey
3/16/2019
CS_158 Midterm
Dr. Doboli

6. (15 pts) Write the pseudo-code of the alpha-beta algorithm applied to the tic-tac-toe game – add any functions you might need.

```
function AlphaBeta(state, depth, alpha, beta, player)
   if at lowest depth: #node has no children
      return state.value
   children = Moves(state, player)
   if player == 1  // MAX
       for all move in children:
          beta = max(beta ,  AlphaBeta(Play(state,move), depth+1, alpha, beta, -player) )
          if alpha >= beta
             break
       return alpha
   else
      for all move in children:
         alpha = min(alpha , AlphaBeta(Play(state,move), depth+1, alpha, beta, -player) )
         if beta <= alpha
            break
      return beta

function Moves(state,player):
   #given whos turn, genereate all possible places to place a piece
   if player == 1:
      generate x moves
   else
      generate o moves
   return moves

function Play(state, move):
   # given a state, place the move
   if move is valid:
      state = state.board + move
   return state
```