CS156 Midterm Fall 2014 Version 1

Name:
StudID:

Problem	Grade
1	4
2	4
3	4
4	4
5	4

20

1. Consider the following situation for four light switches on the control panel of a nuclear power plant: (a) The first and last can never both all on. (b) At least one light must be on. Express this as a propositional logic knowledge base.

L_1 - First of the four lights. True if on, false if off

L_2 - Second of the four lights. True if on, false if off

L_3 - Third of the four lights. True if on, false if off

L_4 - Fourth (i.e last) of the four lights. True if on, false if off

$R_a: \neg(L_1 \wedge L_4)$

$R_b: L_1 \vee L_2 \vee L_3 \vee L_4$

$\wedge$ - similar to "and" in boolean algebra

$\vee$ - Similar to "or" in boolean algebra

$R_a \wedge R_b$

$(\neg(L_1 \wedge L_4)) \wedge (L_1 \vee L_2 \vee L_3 \vee L_4)$ 4

DFS - Depth First Search | BFS - Breadth first search | IDDFS - Iterative Deepening Depth first Search

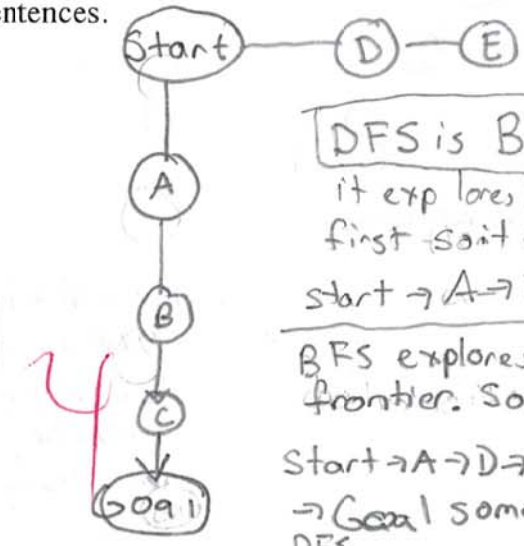
See back for third

2. For each of the following three uninformed search algorithms give an example graph on which it will perform better than the other two with respect to time or space. (a) breadth-first search, (b) depth-first search, (c) iterative-deepening depth first search. To receive any credit you need to explain why your graph works in coherent English sentences.

Goal BFS has best time complexity since it explores uniform frontier so it does start → A → E → B → Goal. (4 states)

DFS is much slower (time complexity) and expands Start → A → B → C → D → E → Goal so more steps than BFS. (6 states)

IDDFs needs to visit more states than BFS since it restarts. It visits Start → A → E restart Start → A → B → E → Goal (so two more states than BFS)



DFS is Best since it explores leftmost first so it explores start → A → B → C → D

BFS explores in uniform frontier. So it explor Start → A → D → B → E → C → Goal so more states t DFS.

IDDFs – Does DFS in one level increme so it does Start → A → D. restarts then start → A → B → D → E restarts then start → A → B → C → D then restart then does start → A → B → Goal so many more move than DFS

3. Consider uniform cost search, *IDA**-search, and local beam search. Briefly explain how each algorithm works and whether it is an informed or uninformed search algorithm (0.5pts each). Then give a context where you might prefer one algorithm over the other two (0.5pts each).

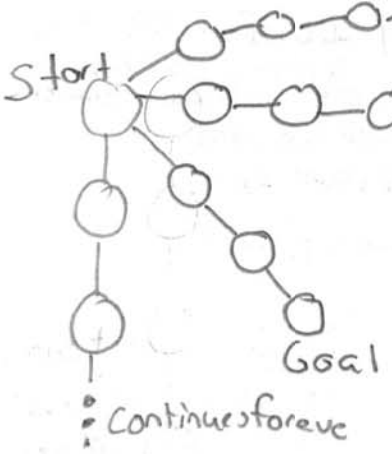
Uniform Cost Search – Uninformed
IDA* – Informed
Local Beam – Informed Local search

Uniform cost search expands nodes based off their distance from the start. Hence it's evaluation function $f(n) = c(n)$ (cost from start to node n)

IDA* – Is Iterative deepening A*. A* evaluation function ($f(n)$) which defines how nodes are expanded) is $f(n) = c(n) + h(n)$ where $c(n)$ is cost from inital st to node n while $h(n)$ is heuristic function estimating cost from node n to the goal.

Local Beam search is a local search with k (where k is an integer) number of states. In each round, all successors of the k states are generated and the best k states (based off utility function which defines solution quality) go to next round / generation. This repeats until a is found.

See back for more



IDDFS performs best in terms of memory. DFS takes left most path with infinite time and space but never completes.

BFS expodes all the four side branches, so its memory use is 12 states, $O(b^d)$. IDDFS memory usage is $O(d)$ which in this case is d (the depth of the solution which is 3)

b is initial branching which is 4 in this case and since no subsequent branching is multiply by d

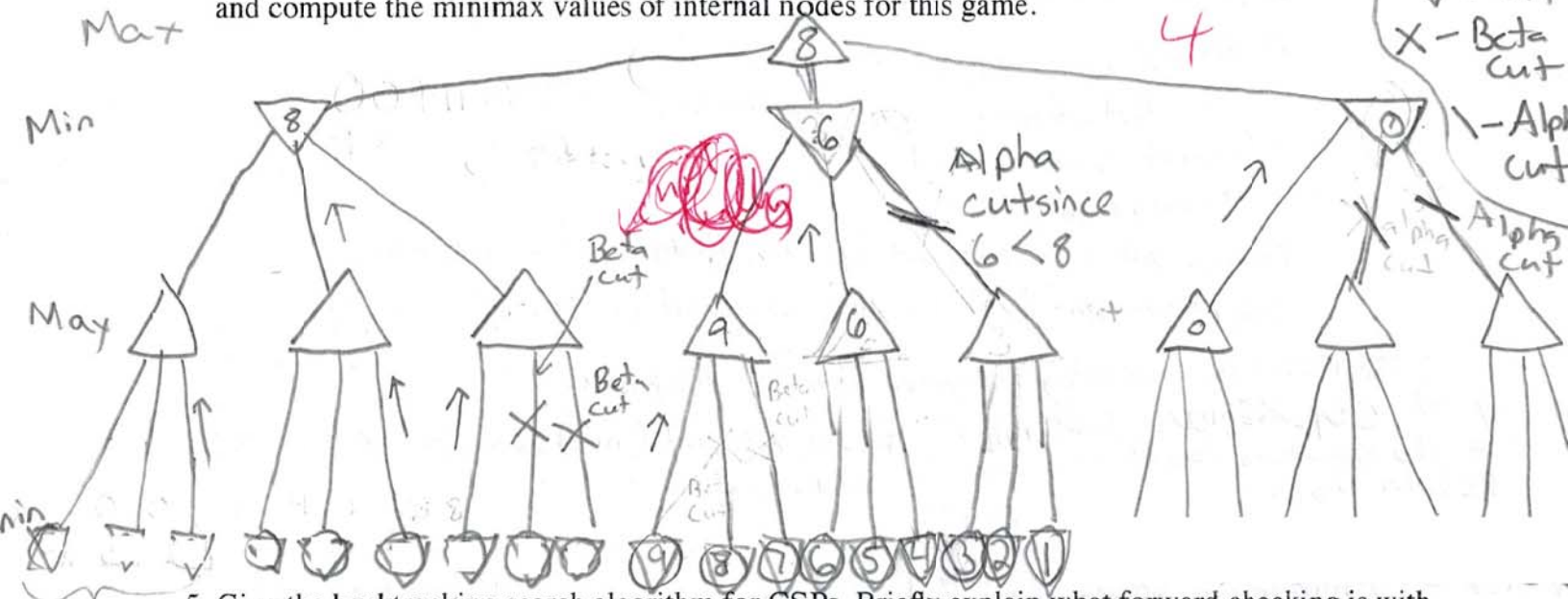
---

Uniform Cost may perform better since it only keeps K states in memory. As such, if $K \geq 2$, if local beam search can solve the problem but band d are larger (e.g. greater than 2) it will be more memory to use IDA* and UCS.

If heuristic and utility function are not admissible over estimate costs UCS may perform better as correct solutions may be ignored (example $h = \infty$)

If μ is less than $f(n)$ eval function cost, IDA* performs exactly like A* so it performs better if $h(n)$ is perfectly accurate. If local beam hit a set of local maximum in that case it would terminate and find no solution result in A* being complete and fastest.

4. Write down the first 9 digits of your Student ID with leading 0's. Next to these on the same line write the numbers 9, 8, 7, 6, 5, 4, 3, 2, 1. Next write the 9 digits of your Student ID with minus signs in front of them. Let these be the minimax payoff values in the leaves of a complete ternary tree representing a game. The root of the tree is MAX's move. Show all α and β-cuts and compute the minimax values of internal nodes for this game.



5. Give the backtracking search algorithm for CSPs. Briefly explain what forward checking is with respect to this algorithm and then explain the MRV heuristic.

```
def BACTRACKINGALGORITM(csp):
  #Returns CSP solution or none
  Initial_assignment = {} #empty
  return Backtrack(csp, initialassignment)

def BackTrack(csp, assignment):
  #If assignment is complete, return it
  if(csp.CHECK_ASSIGNMENT_complete(csp,assignment)):
     return assignment

  #Select the next variable to assign (i.e. give a value)
  next_var = csp.SELECT_UNASSIGNED_VARIABLE(assignment)

  #order the domain values, (Domain is set of legal values for the variable)
  domain = csp.ORDER_DOMAIN_VALUES( next_var)
  #Check all domain values
  for d_i in domain:
    #Check if this variable assignment is consistent (i.e. satisfies all constraints)
    if(csp.CHECK_ASSIGNMENT_consistent(assignment, next_var, d:)):
       # set variable to value di
       assignment.set_variable(next_var, d:)
```

4

```
    inference = csp.INFERENCES(assignment) # Returns inferences if a valid
                                           # inference exists or if a domain
                                           # becomes empty returns "None"

    if(inference is not None):
        csp.APPLY_INFERENCES(assignment, inference)
        # Recurse and assignment variable
        result= BACKTRACK(csp, assignment)
        if(result is not None):
            return result
        # No valid assignment for this assignment so undo inference
        csp.REMOVE_INFERENCES(assignment, inference)
    # Remove variable assignment from for loop
    csp.REMOVE_VARIABLE_ASSIGNMENT(next_var, d_i):
# No solution found for domain so return none
return None
```

MRV - Minimum Remaining Value heuristic. Also known as the "Fail First" heuristic, It entails selecting the unassigned variable with the smallest domain (i.e. least legal values) next for assignment. It corresponds to the method "SELECT_UNASSIGNED_VARIABLE" in my code.

Forward checking - Method "INFERENCES" in my code and after a variable is assigned a variable, you perform inference checking (e.g. arcconsistency) to eliminate illegal values from the remaining unassigned values. This will reduce/prune the search tree by eliminating those values with no chance of being valid, consistent (not violating any constraints) assignments