13/01/2021

Computer Science BSc
Intelligent Systems 1 (Exam)
Closed Examination

Exam Candidate Number:
Y3878747

Question 1:

(i) a. [A, B, C, D, E, F, G, H, I}

b. [A, B, C, E, H, I, F, G, D]

c. {A, B, C, E, F, G, D]

d. {A, B, C, E, F, G, D, H, I}

where the initial limit is 2
and is increased by 1 each time.

(ii) The data structure that represents the fringe for a DFS is a LIFO (last-in-first-out) stack. New nodes are pushed onto the top ~~od~~ of the stack, and nodes are removed by 'popping' the node off the top of the stack.

~~In the case of a list implementation~~

ordered contents of the fringe having just visited node C:

[D, E, F, G]

↖ this is the top of the stack

↑
this is the bottom of the stack.

Question 2:

(i) problem representation: - State representation:

each state can be represented by a list of lists. The first list in the outermost list, represents the leftmost pole: P_1 in the puzzle. The rest of the lists represent the remaining number of poles all the way up to the last list representing the right most pole P_n, with ~~the~~ any pole P_j such that the list before it represents P_{j-1}, and the list after it represents P_{j+1} {bar the special cases of P_1 and P_n}. The order of this 2D list cannot be changed and must remain static. where $j \in \{1, \ldots n\}$
The lists within the 2D list may contain elements or elements D_i, where $i \in \{1, \ldots, m\}$. Each element D_i represents a disk in the real world problem, and the P_j D_i is in, represents ~~the pole~~ the pole that the disk (that D_i is representing) is on. An empty 1D list represents a pole which has no disks on it. The leftmost element of a 1D list represents the bottommost disk, and the rightmost element represents the ~~right~~ topmost disk on the pole. *

initial state:

$$[[D_1, D_2 \ldots, D_m], [], \ldots, []]$$

path cost: number of times a disk is removed from a pole and subsequently placed on another pole.

goal test: $IsOnRightMostPole(D_1) \wedge \ldots \wedge IsOnRightMostPole(D_m)$

Successor function:

a move is defined as such: $Move(D_i, P_j)$, where D_i is the disk being moved to the pole P_j.
the move is allowed if D_i is the topmost disk of the pole in which it currently resides, and if P_j either is empty or P_j's topmost disk is D_h, such that $h > i$.

* constraints: any D_i can only be in one P_j at a time - no duplicates. only the topmost element of a LIFO list can be removed and placed on another list at any one time - this counts as a move. any D_i can only have D_x to the left of it, where $x < i$; similarly any D_i can only any D_i can only have D_y to the right of it, such that $y > i$.

Scanned with CamScanner

(ii) BFS is best for this problem, especially as opposed to DFS, as BFS is complete AND finds the most optimal solution.

(iii) One possible admissible heuristic could be to use the number of disks that are on the rightmost pole, P$_1$

(iv) A heuristic h(n) is defined as admissible, if the remaining steps in the problem f(n) is always greater than or equal to the heuristic. i.e. a heuristic that is admissible is always an optimistic underestimate.

The Move operation always moves a topmost disk from one pole to another if it is an allowable move. In the case that the move moves a disk from any pole that is not P$_1$, h(n) converges with f(n). On the other hand, in the case that the Move operation is carried out on the topmost disk on pole P$_1$, h(n) will change by 1. Hence h(n) is admissible.

Question 3:

(i) gradient descent algorithm: $x_{n+1} = x_n - \eta \cdot f'(x_n)$

when ~~##~~ $f(n) = x^2 + 4$, $f'(n) = 2x$ and $\eta = 0.05$, algorithm becomes: $x_{n+1} = x_n - 0.05(2x_n)$

$x_0 = 0.1$

$x_1 = 0.1 - (0.05)(2)(0.1) = 0.09$

$x_2 = 0.081$

$x_3 = 0.0729$

(ii) if η is increased, a problem that may be encountered is that the algorithm may end up overshooting any local minimums, instead continuing to jump across the same region i.e. it never converges. So it is better to choose η such that it isn't big enough to never converge, nor small enough that it never gets to ~~through~~ an approximate of the solution in a reasonable amount of time.

(iii) Depending on x_0, the gradient descent minimising algorithm will converge to its local minimum - as in the minimum which is closest to that initial value specifically. Other starting points will converge to their local minima (which may be different if ~~a~~ the graph of a function has more than one global minima), or may be even diverge off into infinity, never reaching a solution.

Page 4

INT 1
exam
Y3878747

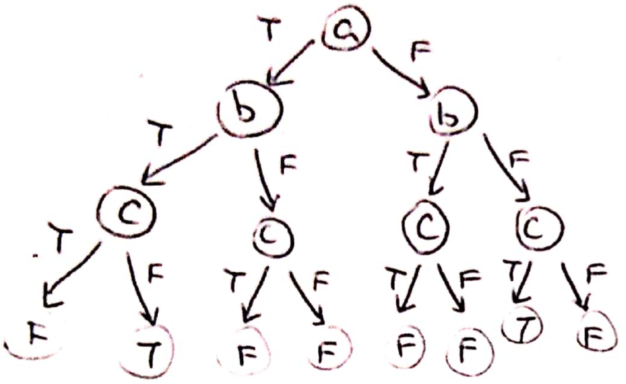
Question 4:

(i)

$(a \Leftrightarrow b) \wedge (c \Leftrightarrow \neg a) \equiv (a \Rightarrow b) \wedge (b \Rightarrow a) \wedge (c \Rightarrow \neg a) \wedge (\neg a \Rightarrow c)$

$\equiv (\neg a \vee b) \wedge (\neg b \vee a) \wedge (\neg c \vee \neg a) \wedge (a \vee c)$ Q.E.D.

(ii) The following is the DPLL tree ~~[illegible]~~ using the most simple version of DPLL for ~~[illegible]~~ L: (without early termination and without unit clause propogation)



(iii) No, as DPLL is complete, enumerating over every possible option. WalkSAT on the other hand, sacrifices ~~[illegible]~~ DPLL's completeness, in favour of getting to an answer quicker, and works best for large satisfiable instances. DPLL eventually gives an answer, unlike WalkSAT oftimes, and it has given its answer in that the SAT problem in question ~~[illegible]~~ cannot be solved in any variation of the model, therefore this SAT problem is unsatisfiable.

(iv) WalkSAT is a SAT solving algorithm ~~[illegible]~~ incorperating ~~[illegible]~~ simmulated anneal.ing. It assigns random True / False values to all variables, testing whether ~~[illegible]~~ the problem is satisfied. (An objective function could count broken clauses number.) If not, random broken clause picked. with probability p, flip a variable randomly, otherwise flip variable which improves objective function.

Page 5|

INT 1
exam
Y3878747

Question 5:

(i) no cats allowed to walk on any tables:

$\forall x \forall y\ IsCat(x) \land IsTable(y) \land WalkedOn(x,y) \Rightarrow RuleBreaker(x)$

my dining room has paw prints of a cat on it:

$\exists x \exists y \exists z\ IsTable(y) \land Has(y,z) \land ArePawPrints(z) \land Prints(x,z) \land IsCat(x)$

$IsTable(DiningTable) \land Has(DiningTable, TablePrints) \land ArePawPrints(TablePrints) \land Prints(AnCat,$ ~~Paw~~ $TablePrints) \land IsCat(AnCat)$

the only way paw prints can get on a table is by pets walking on it:

$IsTable(y) \land Has(y,z) \land ArePawPrints(z) \Leftrightarrow WalkedOn(p,y) \land IsPet(p)$

all my pets leave paw prints when they walk.

$\forall x\ IsMyPet(x) \Rightarrow ArePawPrints(z) \land WalkedOn(x,y) \land Has(y,z) \land IsWalkableOnObject(y)$

$IsTable(x) \Rightarrow IsWalkableOnObject(y)$

of all my pets, I only have one cat and she is called Pixel:

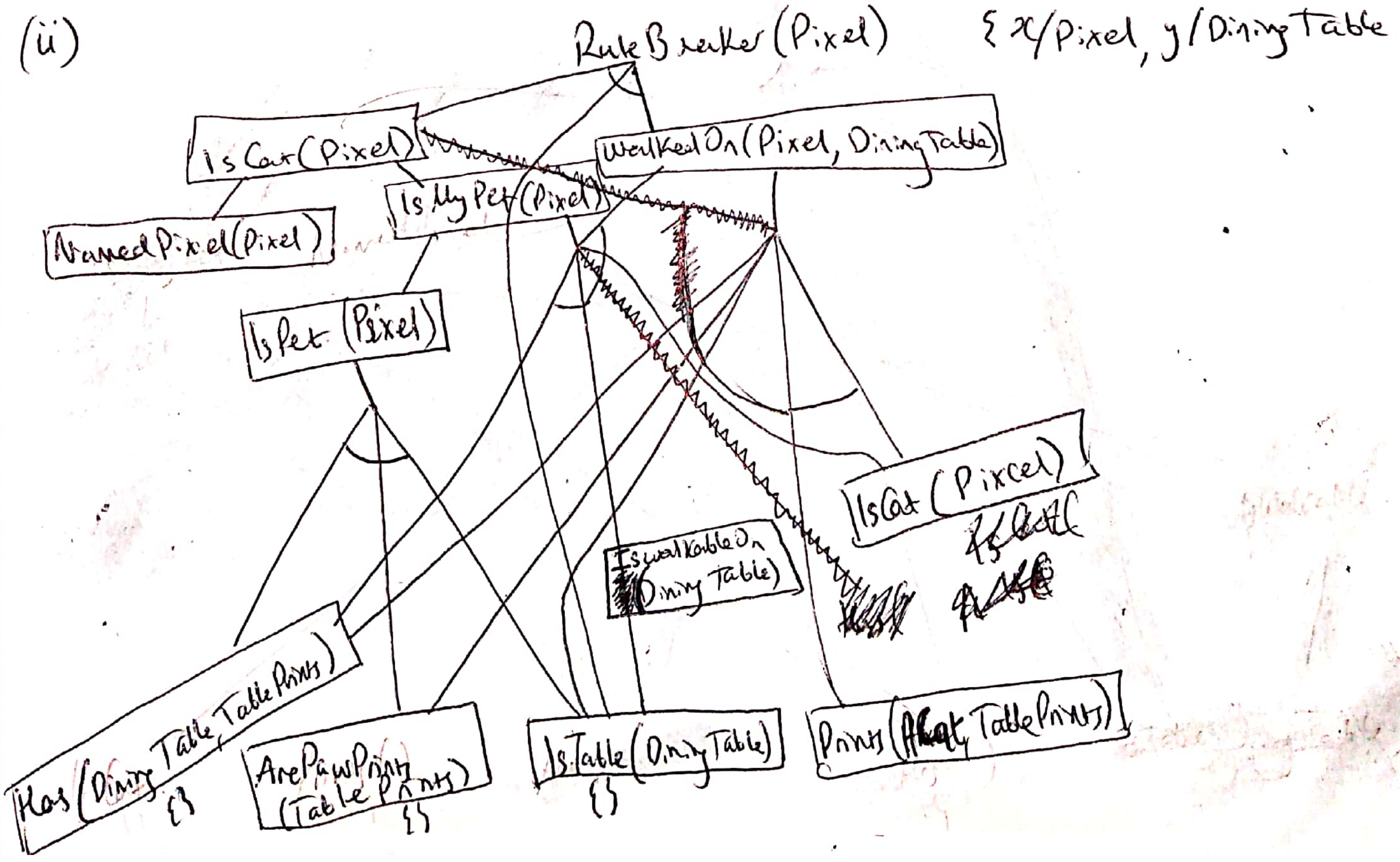
$\exists x\ IsMyPet(x) \Leftrightarrow IsCat(x) \land NamedPixel(x)$

$IsMyPet(Pixel) \Leftrightarrow IsCat(Pixel) \land NamedPixel(Pixel)$

$IsMyPet(x) \Rightarrow IsPet(x)$

$IsMyPet(Pixel) \Rightarrow IsPet(Pixel)$

(ii)


RuleBreaker(Pixel)
{x/Pixel, y/DiningTable
IsCat(Pixel)
WalkedOn(Pixel, DiningTable)
IsMyPet(Pixel)
NamedPixel(Pixel)
IsPet(Pixel)
IsCat(Pixcel)
IsWalkableOn DiningTable)
Has(DiningTable, TablePrints)
{}
ArePawPrints (TablePrints)
{}
IsTable(DiningTable)
{}
Prints(AnCat, TablePrints)