Norwegian University of Science and Technology
Department of Computer and Information Science

EXAMINATION IN LOGIC AND REASONING SYSTEMS (TDT4136)

SOLUTION GUIDE

Problem 1 (25%)

a) 1. $\forall x \;\; Seabird(x) \implies Bird(x)$
2. $\forall x \;\; Landbird(x) \implies Bird(x)$
3. $\forall x \;\; Bird(x) \implies Wings(x)$
4. $\forall x \;\; Bird(x) \implies Fly(x)$
5. $\forall x \;\; Seabird(x) \implies Eat(x, Fish)$
6. $Seabird(Bob)$
7. $\forall x \;\; Eagle(x) \implies Landbird(x)$
8. $\forall x \;\; Swallow(x) \implies Landbird(x)$
9. $Eagle(Sam)$

Alternatively, 3 and 4 could be joined to $\forall x \;\; Bird(x) \implies (Wings(x) \wedge Fly(x))$ but this would have no effect on clausal form in the next subtask. The two first sentences in the text do not need to be formulated, as they define the domain.

b) Sentence 6 is dropped.

New sentences:
10. $Penguin(Bob)$
11. $\forall x \;\; Penguin(x) \implies Seabird(x)$
12. $\forall x \;\; Penguin(x) \implies \neg Fly(x)$

Need to convert sentences 1, 4, 11 and 12 to clausal form.

Step 1. Eliminate implications
1. $\forall x \;\; \neg Seabird(x) \vee Bird(x)$
4. $\forall x \;\; \neg Bird(x) \vee Fly(x)$

11. $\forall x \ \neg Penguin(x) \vee Seabird(x)$
12. $\forall x \ \neg Penguin(x) \vee \neg Fly(x)$

Step 2. Move $\neg$ inwards (not needed here).

Step 3. Standardize variables (not needed here).

Step 4. Skolemize (not needed here).

Step 5. Drop $\forall$
1. $\neg Seabird(x) \vee Bird(x)$
4. $\neg Bird(x) \vee Fly(x)$
11. $\neg Penguin(x) \vee Seabird(x)$
12. $\neg Penguin(x) \vee \neg Fly(x)$

Step 6. CNF form, distribute $\wedge$ over $\vee$ (not needed here).

Step 7. Clause form, split CNF $\wedge$ (not needed here).

Resulting sentences in clause form:
1. $\neg Seabird(x) \vee Bird(x)$
4. $\neg Bird(x) \vee Fly(x)$
10. $Penguin(Bob)$
11. $\neg Penguin(x) \vee Seabird(x)$
12. $\neg Penguin(x) \vee \neg Fly(x)$

One resolution proof (multiple exist):
$(10, 11)$ 13. $Seabird(Bob)$ $\{x/Bob\}$
$(10, 12)$ 14. $\neg Fly(Bob)$ $\{x/Bob\}$
$(1, 13)$ 15. $Bird(Bob)$
$(4, 15)$ 16. $Fly(Bob)$
$(14, 16)$ 17. []

Another one:
$(1, 11)$ 13. $Bird(x) \vee \neg Penguin(x)$
$(4, 13)$ 14. $\neg Penguin(x) \vee Fly(x)$
$(12, 14)$ 15. $\neg Penguin(x)$ $\{x/Bob\}$
$(10, 15)$ 16. []

Either way, the new information leads to an inconsistency in our knowledge base.

c) Figure 1 shows the resulting semantic net.

d) Figure 2 shows the resulting semantic net.

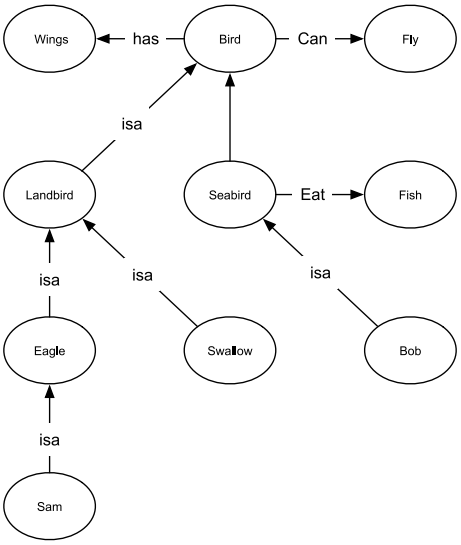

Figure 1: Semantic net from a)

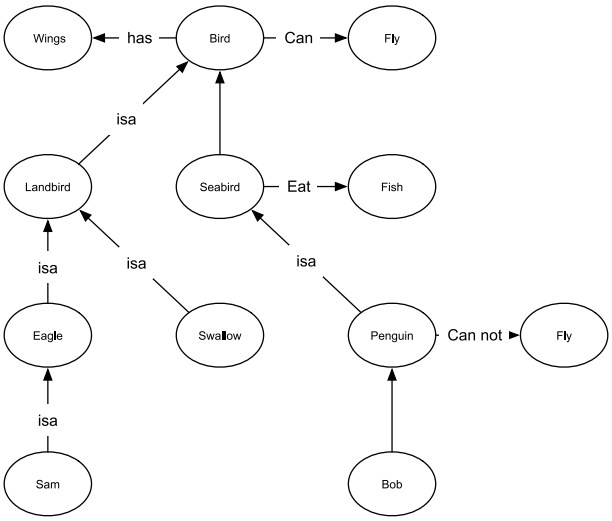

Figure 2: Semantic net from b)

Problem 2 (15%)

a) Starting with a top node, each possible move is depicted down to a maximum depth, or a terminal node (win or lose position). For the leaf nodes at the bottom level, each node is assigned an evaluation function value.

The valuation function assigns the values from the point of view of one of the players (called Max). The valuation function must assign high values to win positions and low

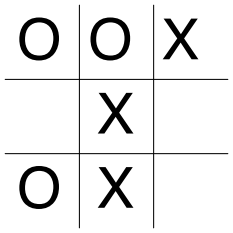

Figure 3: Tic-Tac-Toe

values to losing positions, and a value in between for non-terminal nodes. A draw is typically assigned a value of 0.

Then, all values are backed up so that a Max node (Max to move) gets the maximum of its daughter nodes (Min nodes), while each Min node gets the minimum value of its daughter nodes (Max nodes).

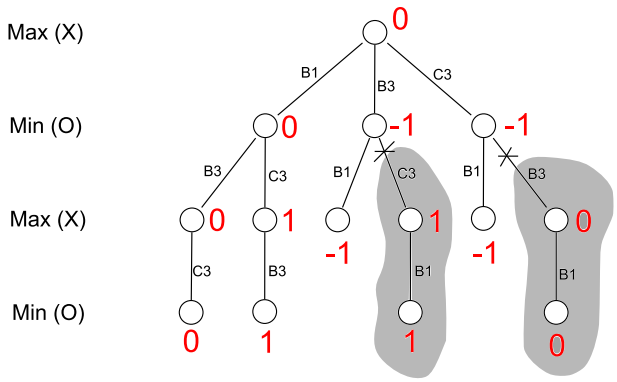

Figure 4: Minimax tree of figure 3. Grey areas would be pruned away by $\alpha - \beta$ pruning.

b) Figure 4 shows the resulting minimax tree for the game situation in figure 3. Here, -1 is given for a loss, 0 for a draw and +1 for a win. As we assume our opponent is playing optimally, the backed up value of 0 shows that we can't win from this position.

c) Alpha-Beta pruning is a method where a node is not evaluated further if it can be proved that the node will never influence the choice. This is done by assigning provisional values (alpha values at Max nodes, beta values at Min nodes) when a value is backed up from below.

These values (alpha and beta) are then used to see if further processing of a branch is necessary. For instance, if a Min node with a beta value β is below a Max node with an alpha value α, and $\beta <= \alpha$ then Max will never choose that node anyway, and any further analysis of the subnodes will not change that.

This is shown in figure 4, where the pruned nodes are marked in grey. Note: it is possible to arrange the evaluation of nodes here so that no nodes are pruned away. If so, this should be commented.

Problem 3 (20%)

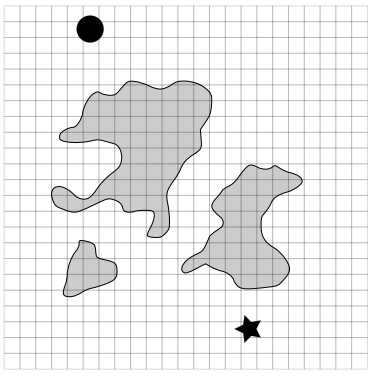

Figure 5: Oil spill. The response vessel needs to get as fast as possible from its current position (star) to the oil spill (circle) by finding a route around land (grey areas).

a) A heuristic search problem consists of

- a set of nodes containing states
- a start node
- a goal node (or goal condition)
- a successor function
- a cost function for one step
- a heuristic function as an estimate for cost from a node to a goal node

For this problem, the states are positions on the grid. Start and goal nodes are the star and circle, respectively. The successor function produces the north, east, south and west cells of the current position, the cost is 1 per move (except for land which should be infinite), and the heuristic function is found below.

The goal is to find the cheapest route from start node to goal node.

Note: if one considers land passable but with a very high cost, there would be a difference between the fastest and shortest route here. We assume land is completely impassable for this task, so the cheapest route is both fastest and shortest.

b) Admissible heuristics means technically that H is less or equal to the real minimal cost. The benefit is that we are guaranteed a minimal cost plan if it exists.

Monotone heuristics means technically that the cost of a step is higher than the reduction in heuristics, which means that the total heuristic is (monotonically) non-decreasing.

Benefits of monotone heuristics is that we can regard the first found path to nodes as the minimal route to them, and can thus discard further search for any nodes that is detected earlier.

These two concepts guarantees finding the optimal path to the goal.

c) In general, a good approach when generating heuristics is to find an optimal solution to a relaxed problem, as such a solution would be an admissible heuristic for the original problem. Here, we can relax two restrictions: (1) moving on land and, additionally, (2) cardinal direction movement.

The first relaxation leads to the manhattan distance heuristic (h_1) and the second leads to the euclidian distance or straight line distance (h_2) heuristic. As optimal solutions to relaxed problems, they are both admissible and consistent.

d) One heuristic (h_1) dominates the other (h_2) when $h_1(n) \geq h_2(n)$ for all n. As long as both heuristics are consistent, we can in general expect h_1 to expand fewer nodes than h_2, leading to a more efficient search for the goal node. It is thus generally better to use heuristics with higher values, provided they are consistent and computation time is feasible.

For this problem, the manhattan distance (h_1) gives larger values than the euclidian distance (h_2). At start, $h_2(n) = \sqrt{19^2 + 10^2} \approx 21.5$ while $h_1(n) = 19 + 10 = 29$. As both are consistent, we can reasonably expect that the manhattan distance heuristic will lead to a more efficient search for the goal, as the euclidian distance heuristic will expand more nodes (even some in the opposite direction of the goal!).

Problem 4 (20%)

	1	2	3	4
A				
B	3			2
C				1
D	4			

Figure 6: In Sudoku each digit can only appear once in each row, column and 2-by-2 box.

a) A CSP problem is a type of state-search problem defined by

- a set of variables
- a set of domains (legal values) for these variables
- a set of constraint relations between the variables

A solution to the CSP is an assignment of values from the domain to each variable so that each variable has a value and no constraints are violated.

b) For this problem, the variables are the various cells: {A1,A2,A3,A4,B1,...,D4}

The common domain is {1,2,3,4}.

The constraints are on groups of variables as each row, column and box must have different values. This can be represented using the `AllDiff` global constraint:

```
AllDiff(A1,A2,A3,A4)          (row constraint)
AllDiff(B1,B2,B3,B4)
...
AllDiff(A1,B1,C1,D1)          (column constraint)
...
AllDiff(A1,A2,B1,B2)          (box constraint)
...
```

This is visualized in the constraint graph in figure 7. Here, the circles are the variables and the squares are the 4-ary constraints. Each variable is thus part of 3 such constraints. The constraints themselves are all `AllDiff`, meaning that all variables in each constraint must have different values from the chosen domain.

c) The most basic method of solving a CSP is backtracking search. The method is to simply assign possible domain values to variables one at a time, test the constraints and backtrack when constraints are violated.

A major and necessary improvement is to carefully select the order of the variables to instantiate (heuristics "minimum-remaining-value" and the "degree heuristic" can be used here), and also to carefully select the order of the values to try out ("least-constraining-value" is useful).

Forward checking means that each variable is assigned an apriori set of legal values, and that each assignment leads to a subsequent elimination of all other assignments that become impossible. For instance, whenever a variable X is assigned, for each variable Y that is connected to X by a constraint, delete from Y's domain any value that is inconsistent with the value chosen for X.

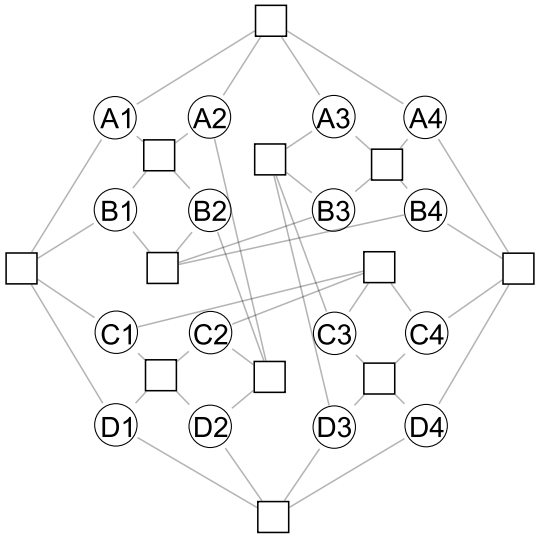

Figure 7: Constraint graph of a 4-by-4 Sudoku problem

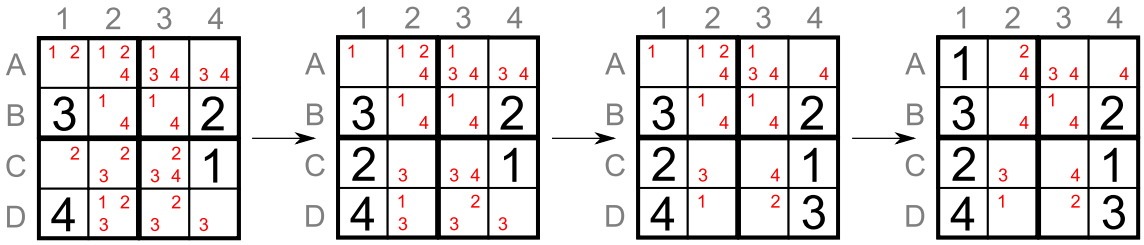

Figure 8: Forward checking for Sudoku

The first three steps for this problem are shown in figure 8. Numbers in red represent the legal domain values for each variable. The first step shows the initial configuration with legal domain values after the initial assignment. Using the "minimum-remaining-value" heuristic, we can see that either C1 or D4 is a good choice for the next assignment. Assigning 2 to C1, we delete this value from the variables in associated constraints, leading to the situation in the next step. Following this method, a solution is found quickly and without the need to backtrack in this case (this does not necessarily apply to harder Sudokus however!). The solution is found after 12 steps.

d) A variable in a CSP is arc-consistent if every value in its domain satisfies the variable's constraints. That is, X is arc-consistent with another variable Y if for every value in X's domain there is a value in Y's domain that satisfies the constraint.

For instance, from the situation in the first step of figure 8, variable C1 is arc-consistent with C2, but not the other way around. This is because the domain of C1 would be empty after an assignment of 2 in C2.

We can apply the arc consistency algorithm AC-3 to ensure that the domains of each variable is arc-consistent before we start assigning values to variables. Without going in to details, for this simple problem the result would be a single domain value for each variable, thus effectively reducing the search to a simple allocation of values.

It is not required to calculate the actual cost here, just mention that the cost to perform this step can sometimes be high, worst case is $O(cd^3)$ (c is number of constraints, d the domain size). For this simple problem, depending on the order of checking constraints, it can be higher than the forward checking method above.

More generally however, running an arc-consistency check leads to faster searches because each variable will have smaller domains.

Problem 5 (20%)

a) Using PDDL notation.

```
Action(Move(x,y),
       PRE: At(Robot,x),
       EFF: ¬At(Robot,x) ∧ At(Robot,y))
Action(Pickup(b,x),
       PRE: At(Robot,x) ∧ At(b,x) ∧ Empty,
       EFF: ¬At(b,x) ∧ ¬Empty ∧ Holding(b))
Action(Drop(b,x),
       PRE: At(Robot,x) ∧ Holding(b),
       EFF: ¬Holding(b) ∧ Empty ∧ At(b,x))
```

STRIPS is also fine, but then the effect part would be split in two: one for the DELETE list and one for the ADD list.

b) Initial condition: `Init(Holding(P`$_1$`) ∧ ¬Empty)`

Goal condition: `Goal(Holding(P`$_2$`))`

This problem text is a bit vague with details on the possibility of having multiple boxes at the same location. As the text states that we can disregard the `At` state, we assume that multiple boxes can be at the same location without the need for stacking.

Figure 9 shows the planning graph down to level S_2, which is when the goal condition appears. The planning graph also levels off at this point. The action $Pickup(P_1)$ is omitted from level A_1 to make the figure less cluttered. Red links are mutexes at each level, between actions or states.

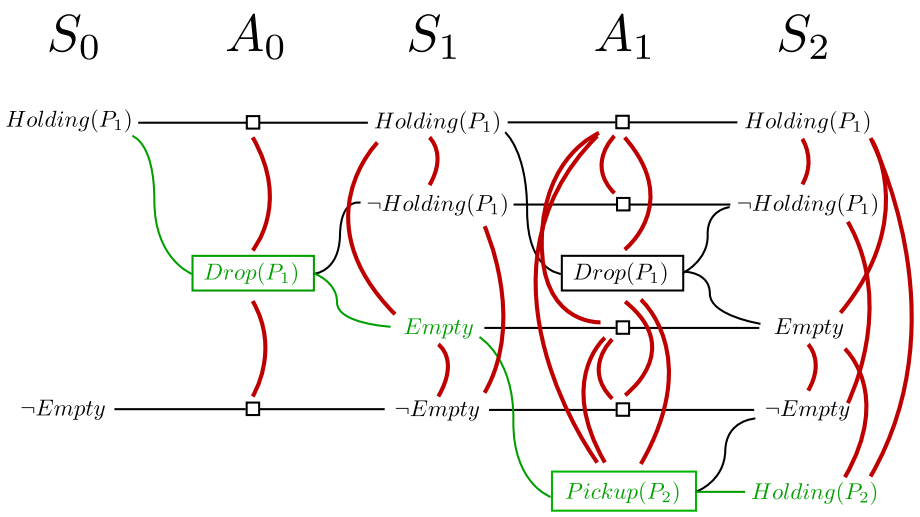

Figure 9: Planning graph

The important part with a planning graph is to show how the agent moves from state to state. This is done by using applicable actions as well as the persistence action (no-op), which is marked with a small empty squares in the graph.

It is not required to show all the mutexes as this can be quite time consuming, but the mutex between $Holding(P_1)$ and $Holding(P_2)$ due to the mutex between preconditions $Holding(P_1)$ and $Empty$ is important.

c) Firstly, for there to exist a plan in the planning graph, the goal states must not be mutex with each other at the last level. This is not an existance guarantee for a valid solution however, as any states or actions along the path from the start to the goal states must also not be in any mutex relation with each other.

Roughly speaking, to extract a solution, start at the level where the goal states are not in mutex with each other. Then search backwards (state search) to find a path where actions are not in mutex with each other. If one is able to arrive at the start state S_0, a valid plan exists and that path can be extracted. Alternatively, this could also be stated as a constraint satisfaction problem.

For this task however, this path is fairly straight forward. The resulting plan is shown in green in figure 9.