Software Verification Homework 4

Yosef Goren

January 3, 2023

1 BDD constructions

a. Denote the set of vertices: $V = \{\overline{x}_i \mid i \in [n]\}$
Let $E(\overline{v}) := E_0(\overline{v}) \wedge E_1(\overline{v})$.

$$A'(\overline{v}) := A(\overline{v}) \wedge \left(\bigwedge_{i=0}^{n} (E(\overline{v}, \overline{x}_i) \Rightarrow B(\overline{x}_i)) \right)$$

The idea is that $A(\overline{v})$ means the 'accepted' node has to be from A, and rest of the expression means that all of it's neighbors have to be in B. It is equivalent to satisfying the following formula:

$$(\overline{v} \in V) \wedge (\forall \overline{x} \in V, E(\overline{x}, \overline{v}) \to B(\overline{x}))$$

b.

$$V_{1,2}(\overline{v}, \overline{v}') := \left(\bigvee_{i=0}^{n} E_1(\overline{v}, \overline{x}_i) \wedge E_0(\overline{x}_i, \overline{v}') \right) \vee \left(\bigvee_{i=0}^{n} E_0(\overline{v}, \overline{x}_i) \wedge E_1(\overline{x}_i, \overline{v}') \right)$$

For the vertices $\overline{v}, \overline{v}'$ to have a path of length 2 and weight 1 between them, there must either be a path of length 2 with weight 1 were the edge connected to $\overline{v}$ is 1 and the other edge is 0, or the other way around.
The primary operator of the expression above describes this fact; to be more specific - the left side of the expression describes the case where there is a path of length 2 were the node connected to $\overline{v}$ has weight 1, and so on.

c. The algorithm works as follows:

```
$T(v) \leftarrow \emptyset$
$T'(v) \leftarrow A(v)$
$l \leftarrow 0$ (*).
while $T'(V) \neq T(v)$ do
    while $T'(V) \neq T(v)$ do
        $T(v) \leftarrow T'(v)$
        $T'(v) \leftarrow T'(v) \wedge (\bigvee_{i=0}^{n} E_0(v, v_i))$
        **if** $B(v) \wedge T'(v) \neq \emptyset$ **then**
            **return** $l$
        **end if**
    **end while**
    $T(v) \leftarrow T'(v)$
    $T'(v) \leftarrow T'(v) \wedge (\bigvee_{i=0}^{n} E_1(v, v_i))$
    $l \leftarrow l + 1$
    **if** $B(v) \wedge T'(v) \neq \emptyset$ **then**
        **return** $l$
    **end if**
**end while**
**return** $-1$
```

(*) l represents the maximal minimum path from A to any node in the working set T.

The idea of the algorithm is to start with the set of states of A, then in each iteration expand our working set; if we can expand without increasing the weight of the newly formed paths - then we do that (expand only with edges of weight 0), if we cannot expand without increasing the weight - we expand with edges of weight 1.
After each such exapnsion we check if we have reached a state from B, and if so - we know we have a found the 'first' path into B in terms of path weight - in other words - we have found a minimal path.

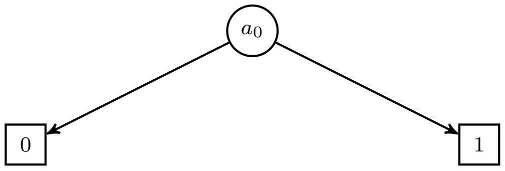
Note that while there is a nested loop in this algorithm, it's total number of iterations is bounded by n since in each iteration the length (*) of the maximal minimum path increases by once, and since the maximal length of any simple (and hence any minimal) path is bounded by n - the total number of iterations is bounded by n.

(*) length in terms of in terms of distance, not weight.

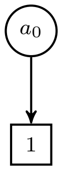
2 BDD operations

a. **1.** True. $D \subseteq D'$.
Proof:
Let $x \in D$. Denote the path of x on B with $a_1, a_2, ..., a_n$.
Since $x \in D$ then $B(x) = 1$, meaning the path must end with 1. If $\exists i \in [n] : a_i = u$. then on the evaluation of x on B', the path will be $a_1, a_2, ..., a_{i-1}, u, 1$. Thus $B'(x) = 1$ and so $x \in D'$ in this case.
Otherwise, $x \notin D$. Thus the path $a_1, ..., a_n$ is unchanged in B' w.r. to B. Hence $B'(x)$ evaluates on the exact same path - which we know ends with 1. Hence $B'(x) = 1$ too, so $x \in D'$.
Meaning in all cases $x \in D'$.
So $D \subseteq D'$.

2. False. Counter example:
Consider kripke structure $(S = \{0,1\}, R = \{(s,s) \mid s \in S\}, L = \{(s,\emptyset) \mid s \in S\})$. Consider $D = \{1\}$.
Let B be the BDD representing D:



Now consider u as a_0. This would mean B' is: aa asdas



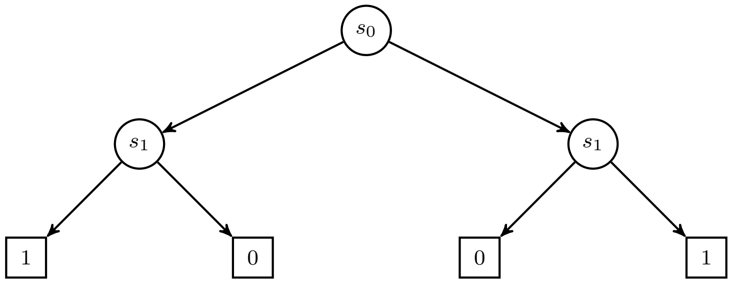
(Technically the a_0 would be reduced...).
So now $B'(0) = 1$ also, hence $D' = \{0,1\} \not\subseteq D$.

b. **i.** True. $R \subseteq R'$.
For the same reason as before, if we have a BDD B_i representing set S_i for $i \in \{0,1\}$ and were $B_1 = cut(B_0)$ then regardless of the context $S_0 \subseteq S_1$ since we only increase the set of inputs that evaluate to 1.

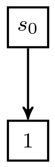
ii. False. $R' \not\subseteq R$.
Consider a similar counterexample (as in a.2.):
The structure:

$$(S = \{0,1\}, R = \{(s,s) \mid s \in S\}, L = \{(s,\emptyset) \mid s \in S\})$$

The BDD of $R(s_0, s_1)$ be B: (if 0 go to left son)



Let $u = s_0$ and $B' = cut(B, u)$:



(After reduction just the leaf with 1 remains).
Hence $R' = \{(0,0),(0,1),(1,0),(1,1)\} \not\subseteq R$

No. This is since the change might create transisions that don't even go into an existing vertex (state).
For example, if we have $S := \{0,1,2\}$ represented in binary as $S := \{00,01,10\}$, and say we have some legal transition set $R \subseteq S \times S$, then we cat it's tree at the root node (like we did before), we will now have the set of all transitions on a pair of strings (xy, zw) meaning:

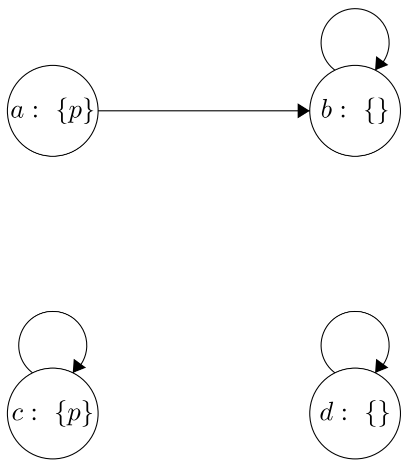
$$R' = \{(00,00),(00,01),(00,10),...,(11,10),(11,11)\}$$

$$= \{(0,0),(0,1),...,(3,2),(3,3)\}$$

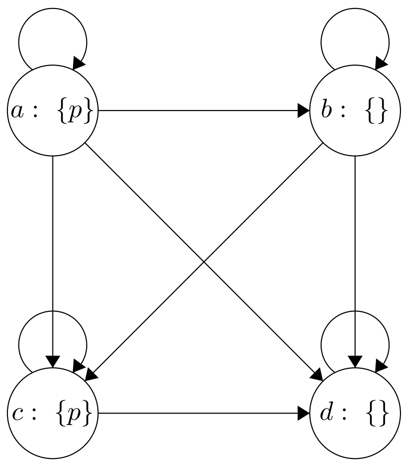
Among the rest, R' contains $(3,2)$ for example - which makes no sense (and is not 'legal'); since $3 \notin S$!

c. **1.** Yes. As described in **b.1.** - $R' \subseteq R$, meaning that all transisions that existed in R also exist in R'. Moreover - this means that any path in R is also a path in R'.
Hence the set of paths considered to evaluate $M', s \models AGp$ (*) contains the set of paths considered to evaluate $M, s \models AGp$; (**) so if all paths in $(*)$ satisfy the condition, then all the paths in $(**)$ also do; in other words - if $M', s \models AGp$ is satisfied, so is $M, s \models AGp$.

2. No. Take for example the structure M:



Now cut the transitions set at the root node (which will cause all transitions to exist) like before to get the structure M':



If we define $s := a$, then it is easy to see how $M', s \models EGp$ with the path $a \to c \to c \to c....$
On the other hand, the only path that starts from s (a) goes through b, hence it does not satisfy Gp.

3 D&D

Solution for **a.+b.**
For the purpose of part **b.** we have assumed that the knight can only carry one princess at a time and each dragon takes the princess away if reached.

In the solution we use the following notations:

1. $\{v_i \mid i \in [n]\}$: the set of vertices ('squares').

2. $V(v)$: a BDD representing the set of vertices.

3. $E(v, v')$: a BDD representing the **REVERSE** set of edges (a transition from v into v').

4. S: a BDD representing the set of starting vertecies.

5. F: a BDD representing the set of final vertecies.

6. D: a BDD representing the set of vertices with a dragon on them.

7. P: a BDD representing the set of vertices with a princess on them.

8. $N := V \wedge \neg(S \vee F \vee D \vee P)$ (the set of 'normal' vertices).

9.
$$\Phi_U(v) := \bigvee_{i=1}^{n} (U(v) \wedge E(v, v_i))$$

Φ_U is simply a 'macro' for ease of readability.
Given a set U, Φ_U is the set of vertecies that are directly connected to a vertex in U.
The evaluation $\Phi_U(v)$ is 1 iff there is a vertex in U that is connected to v.

We present a single algorithm that can handle multiple dragons.
The structure of the algorithm is that we hold multiple sets that are represented as BDDs, and we manipulate these sets in each iteration s.t. when the algorithm ends, one of them will contain the set of all valid paths.

In our algorithm we look for reverse paths, meaning paths from a final state to a starting state - on a graph with reversed edges.

The main idea of the algorithm is that we will start with the set of finishing squares, and in each iteration 'expand' to all neighboring squares, but if we encounter a dragon we will have to move each node that was reached through a dragon into a new set of 'debt' paths (more accurately - squares that were reached from an final square thorugh a dragon square). Now if we step from a square with debt into a starting square, we know the path is not valid, but if we step from a square with no debt into a starting square, then we know we have a path that started from a final square and ended in a starting square were each dragon we have passed through has be 'paid' for with a princess.
If we encounter a princess from the debt set - we can move into the non-debt set, if we encounter a princess from the non-debt set - we have 'no use' for her since all dragons in the rest of the path have already been paid for (we have no debt!), hence we stay in the non-debt set.

These transitions between the sets are described with the following atomata:

$$\begin{tikzcd}
\mathit{Debt} \arrow[loop above, "{F, N}"] \arrow[r, bend right, "P"'] & \mathit{Cur} \arrow[loop above, "{P, N, F, S}"] \arrow[l, bend right, "D"'] \arrow[r, "S"'] & \mathit{Valid}
\end{tikzcd}$$

The algorithm :

```
$Cur \leftarrow F, Debt \leftarrow \emptyset, Valid \leftarrow \emptyset$
for $\_ \in [n]$ do
    Do the following 3 assigments atomically (*):
    $Cur \leftarrow (\Phi_{Cur} \wedge (P \vee N \vee F \vee S)) \vee (\Phi_{Debt} \wedge P)$
    $Debt \leftarrow (\Phi_{Cur} \wedge D) \vee (\Phi_{Debt} \wedge (F \vee N))$
    $Valid \leftarrow (\Phi_{Cur} \wedge S)$
end for
return $Valid$
```

(*) making a set of assignments atomically means that all rvalues are evaluated before any lvalues are assigned.