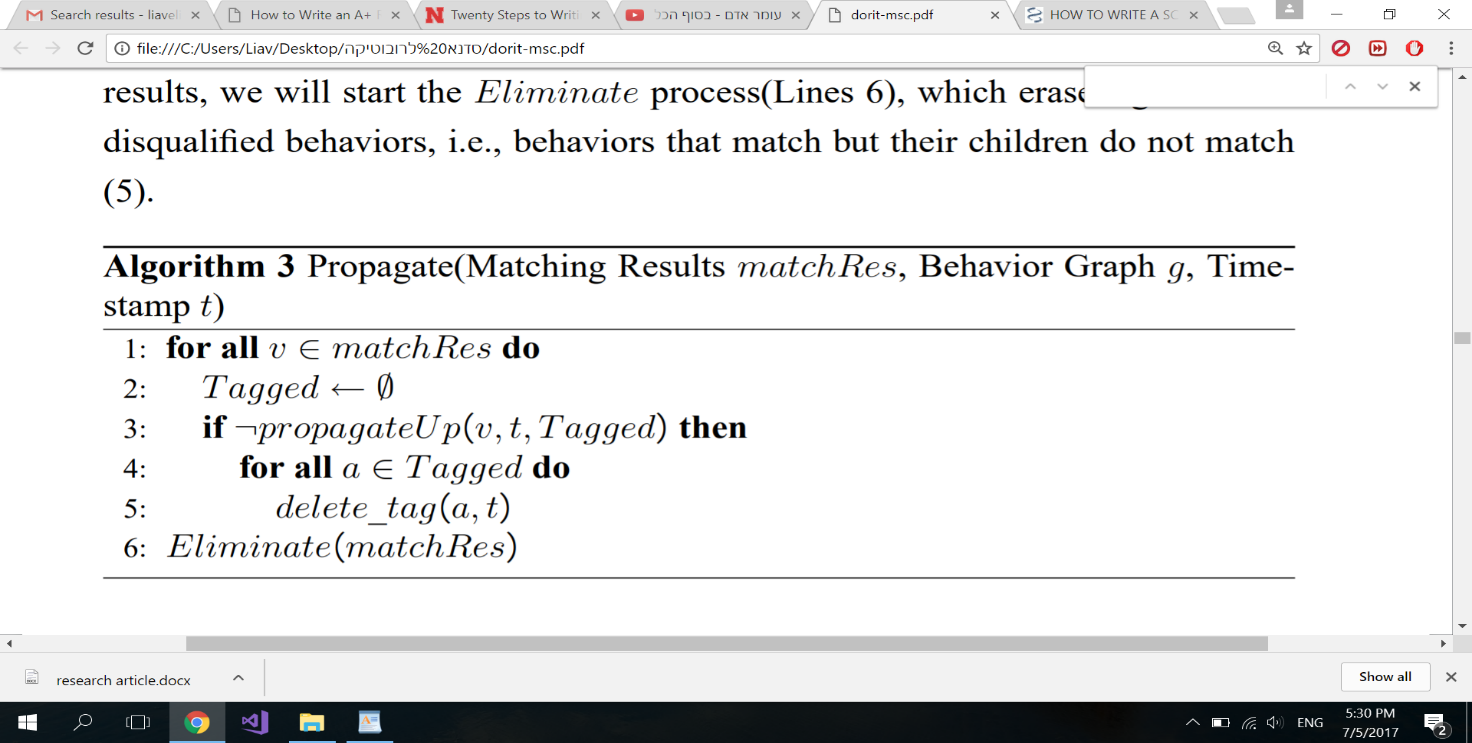
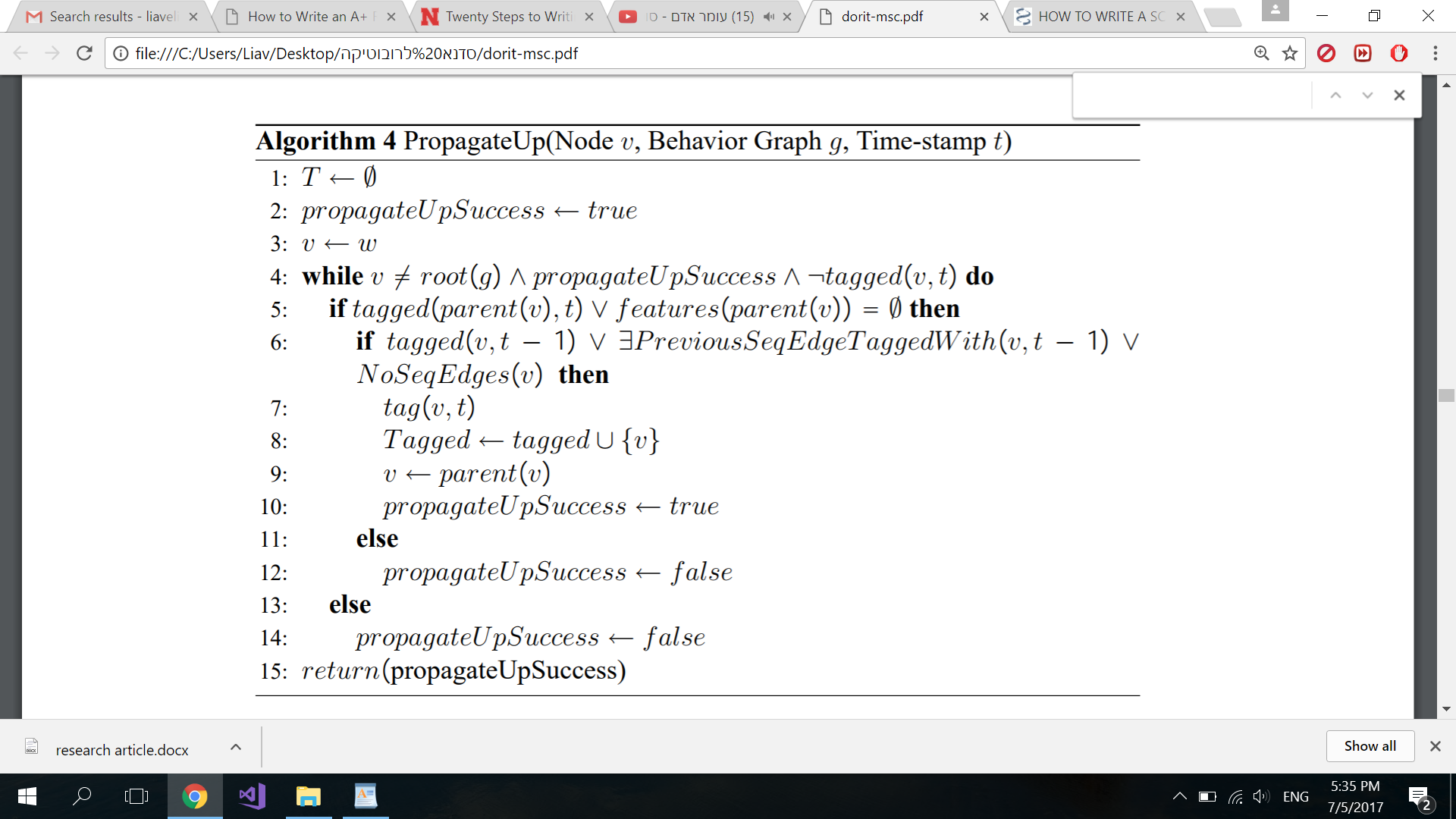
**SYMBOLIC BEHAVIOR RECOGNITION**

It is important for robots to model other robots’ unobserved plans, goals and behaviors, based on their observable actions. This process of modeling others based on observations is known as behavior- or plan-recognition. Behavior-recognition algorithms work by first matching observed actions to a template model (called the plan- or behavior-library), and then propagating the implications of matching actions to determine possible hypotheses that explain the observed behavior.

**The Dorit’s algorithms**

**CSQ-**  
This query answers the question what are the hypotheses regarding the current state of the robot. The answer that the algorithms give is complete, but not accurate, because it refers to every behavior separately.  
The algorithms execute in two phases: (i) matching phase; (ii) tagging and propagating phase.





**HSQ** -

This query answers the question: what were all possible sequences of behaviors that the robot executed from time t = 0 until the current time t + k, k > 0.

We use a connected graph G’ , called Hypotheses Graph, whose vertices correspond to successfully-tagged behavior paths in the behavior graph (i.e., hypotheses). Edges in G’ connect hypothesis vertices tagged with time stamp t to hypothesis vertices tagged with time stamp t + 1. G’ is therefore built in levels, where each level represents hypotheses that hold in each time stamp. For each set of observations made at time t0, we add to G’ a level t0 all possible hypotheses that were tagged t = t0 and propagated successfully in the behavior graph. We then create edges between vertices x1, . . . , xn in level t to vertices y1, . . . , ym in level t − 1 in the following manner: If xi is not part of a sequence (i.e., it is a first child), then we connect xi to each vertex yj (j=1...m); otherwise, if xi is part of a sequence, we connect xi to yj (j=1...m) if any of the behaviors in yj has a sequential edge to any behavior in xi. If xi is equal to yj , we connect them, since we assume that we have durations.

**Complexity**WORST CASE - O(T \* 2^L \* 2^L \* log L) = O(T \* 2^2L \* log L)  
WORST CASE (totally order) - O(T \* 2^L \* 2^L \* log L) = O(T \* 2^2L \* log L)  
BEST CASE – O( T \* 1 \* 1) = O(T)

**My algorithm**

I worked on improving the HSQ algorithm by adding an array of nodes for every node. This array contains all nodes that are able to come before this node. Therefore, we can give up on creating the G’ graph and find the optional hypothesis in a different way. My algorithm goes through all the nodes that were tagged at time t and checks if its array contains a node that was tagged at time t-1. If yes, I continue in this way and when I arrived at a node that was tagged at time 0, I add this hypothesis to a link list.

**WCCB** – WhoCanComeBefore - For every node u in the FDT we save an array of nodes such   
 that for each node v in the array - v is prev(u) or v is   
 prev(parents(u)).  
**HSQB** –History of States Query Back.  
**CSE** – Check Sequential Edge.

function HSQB (Time t, Observations) // t – the last time, observation – an array of   
 behaviors(get time t and return all the nodes   
 with the same behavior).

stacks <- a link list of Stacks

for each node u in Observations[t] do  
 A<- a new stack  
 call CSE(u, t, stcks, A)

reurn stacks

------------------------------------------------------------------------------------------------------------------

function CSE(node u, Time t, stacks, Stack A)   
 // condition stop to the recursion   
 if t = t0:

A.push(u)  
 Stacks.SetNext(A)  
 return

For node b in WCCB(u) do  
 if b signed t-1  
 B<- a new stack  
 B.Addall(A)  
 B.push(u)  
 Func\_2(b, t-1, stacks, B)  
 if u is root  
 stacks2<- a new link list of stacks   
 Stacks2 <- HSQB(t-1)  
 for Stack S in Stacks2  
 B<- a new stack  
 B.addall(A)  
 B.addall(S)   
 Stacks.SetNext(B)

**Complexity**WORST CASE - O(T \* 2^L \* 2^L) = O(T \* 2^2L)  
WORST CASE (totally order) - O(T \* 2^L\* 1) = O(T \* 2^L)  
BEST CASE – O( T \* 1 \* 1) = O(T)

WORST CASE - O(T \* 2^L \* R)  
WORST CASE (totally order) - O(T \* 2^L\* R)  
BEST CASE – O( T \* 1 \* R) = O(T \* R)

L – the library size.  
T – the number of behaviors.  
R – the size of WCCB.

In the Worst Case we have 2 options,

* the size of the array WCCB is maximum(R = 2^L).
* the size of the array WCCB is 1 (R = 1) – totally order.

In the best case the size of the array WCCB is 0 or 1 (R = 0/1).

Just an example.

Position -> turn -> score-kick

