1 Ideas for project 3

Groups presented two main ideas for concurrent insertion in a binary decision diagram. The first one (Algorithm 1) is to encapsulate the vertex creation into a JAVA synchronized section. This way will ensure that only one thread can execute this section. The second one (Algorithm 2) is to use JAVA concurrent hashmap. All nodes of the tree are hashed. The insertion of a new node is made by making use of the map.putIfAbsent() function.

Algorithm 1 Function makeVertex
Ensure: The concurrent insertion of vertex $B$
Require: A BDD and a vertex $B$
1: synchronized {Ensure that only one thread can execute this sub-section}
2: if $B$ isMember of the pool then
3:   return it
4: else
5:   insertToPool($B$)
6: end if
7: return pool

Algorithm 2 Function insertNode
Ensure: The concurrent insertion of a node $B$
Require: A concurrent hashmap map and a node $B$
1: map.putIfAbsent($B$) {ensure that only one thread can insert the node $B$ in the hashmap map}
2: return map
figure 1: concurrent execution schedule

2 Structures for testing our implementation

Our implementation must provide at least two functions. The first function is a swap function. This function interchange the variable \(x_i\) with the next subsequence variable in the order of variables. For instance, considering the variable order \(x_1 \ x_2 \ x_4 \ x_5 \ x_3\), swap(2) produces the following ordering result \(x_1 \ x_4 \ x_2 \ x_5 \ x_3\). In practice, the swap function is used to reduce size of BDD. The second function (Algorithm 3) must compare two boolean expression by returning the size of BDD generated if both boolean expressions are equivalent. Otherwise, it must return a 0. The swap function is executed on a single thread only to respect the order of swap. The function compareBDD can be executed on the others threads. Figure 1 illustrates one possible concurrent execution.

Algorithm 3 Function compareBDD

Ensure: The size of the BDD
Require: Two boolean expressions \(e_1\) and \(e_2\)
1: if \(e_1\) is equivalent to \(e_2\) then
2: return The size of the BDD construct using expression \(e_1\)
3: else
4: return 0
5: end if

3 Performance issues of implementation

Two performance issues of the implementation

• As little synchronization as possible
• Save as much BDD structure as possible across swaps