

DSA

Assignment 2 – Binary Decision Diagrams

Arman Vardanyan

The code provided appears to implement a Binary Decision Diagram (BDD) and includes functions to build BDDs, create BDDs from boolean functions, evaluate inputs using BDDs, and perform testing and evaluation. In my project this are the main parts/functions which I implemented:

1. buildBDD: This recursive function builds a BDD by constructing the nodes and connecting them based on the given array of binary values. It takes a parent node and an array as input and returns the constructed BDD.

```
36 BDD* buildBDD(BDD* parentNode, char* array) {
37
38
39
40 if (strlen(array) > 0) {
41
42
43     BDD* leftChild = malloc(sizeof(BDD)); // Create a new node for the left child
44     BDD* rightChild = malloc(sizeof(BDD)); // Create a new node for the right child
45
46
47     char* leftArray = malloc(sizeof(char) * strlen(array)); // Allocate memory for the left subarray
48     char* rightArray = malloc(sizeof(char) * strlen(array)); // Allocate memory for the right subarray
49
50     strcpy(leftArray, array, strlen(array) / 2); // Copy the left half of the array
51     leftArray[strlen(array) / 2] = '\0'; // Add a null terminator to the left subarray
52     strcpy(rightArray, array + strlen(array) / 2, strlen(array) - strlen(array) / 2); // Copy the right half of the array
53     rightArray[strlen(array) / 2] = '\0'; // Add a null terminator to the right subarray
54
55     parentNode->value = malloc(sizeof(char) * strlen(array) + 1); // allocate memory for the value of the parent node
56     int length = strlen(array);
57
58
59     // Copy the values from the original array to the parent node's value
60     for (int i = 0; i < length; i++)
61     {
62         parentNode->value[i] = array[i];
63     }
64     parentNode->value[strlen(array)] = '\0'; // Add a null terminator to the parent node's value
65
66     parentNode->left = buildBDD(leftChild, leftArray); // Recursively build the left subtree
67     parentNode->right = buildBDD(rightChild, rightArray); // Recursively build the right subarray
68
69     return parentNode; // Return the constructed node in the BDD
70 }
71
72 return parentNode; // Return the parent node if the array is empty
73 }
```

2. **BDD_create**: This function creates a BDD from a boolean function represented by a binary vector. It calculates the total number of nodes in the BDD by halving the vector length iteratively. It returns the created BDD.

```
79
80 BDD* BDD_create(BF* bfunkcia) {
81
82     int totalNodes = 0;           // Variable to track the total number of nodes in the BDD
83     int vectorLength = bfunkcia->length; // length of the boolean function's vector array
84
85     NODE* startNode = malloc(sizeof(NODE)); // Create a new node as the starting node of the BDD
86     BDD* bdd = malloc(sizeof(BDD));        // Create a new BDD struct
87
88     startNode = buildBDD(startNode, bfunkcia->vector_array); // Build the BDD recursively
89
90     while (vectorLength != 1)
91     {
92         totalNodes += vectorLength;
93         vectorLength /= 2;
94     }
95     // Calculate the total number of nodes in the BDD by halving the vector length iteratively
96
97     bdd->head = startNode; // Set the start node as the head of the BDD
98     bdd->nodes = totalNodes; // Set the total number of nodes in the BDD
99     bdd->variables = log2(strlen(bfunkcia->vector_array)); // Calculate the number of variables in the BDD
100
101     return bdd; // Return the created BDD
102 }
103
```

3. **BDD_use**: This function uses a BDD to evaluate inputs and returns the result. It performs a lookup operation in the BDD based on the given inputs by traversing the BDD nodes according to the input values. It returns the value stored in the final node.

```
104
105
106 char BDD_use(BDD* bdd, char* inputs) {
107
108     signed char result = -1; // Initialize the result value as -1 (indicating an error)
109     if (bdd->head == NULL) // Return the result if the BDD is empty (no head node)
110         return result;
111
112     NODE* currentNode = bdd->head; // Start from the head of the BDD
113
114     for (int i = 0; i < bdd->variables; i++)
115     {
116
117         if (inputs[i] == '0')
118             currentNode = currentNode->left; // Traverse to the left child if the input is '0'
119
120         if (inputs[i] == '1')
121             currentNode = currentNode->right; // Traverse to the right child if the input is '1'
122
123     }
124
125     return *currentNode->value; // Return the value stored in the final node
126 }
127
128
```

4. information: This function prints information about the test, including the number of variables and functions tested, the total processing time, and the average processing time.
5. Binary_increment: This function increments a binary combination represented as a string of '0's and '1's. It performs a binary increment operation on the combination by iterating from the rightmost digit and changing '0' to '1' or '1' to '0' until a non-carry operation is encountered.

```
141 char* Binary_increment(char* combination, int size) {
142
143
144
145     int sizeOfCombination = size;
146     char* combinationCopy = malloc(sizeOfCombination * sizeof(char)); // Create a copy of the combination
147
148     strcpy(combinationCopy, combination, sizeOfCombination);
149
150     for (int i = sizeOfCombination - 1; i >= 0; i--)
151     {
152         if (combinationCopy[i] == '0')
153         {
154             combinationCopy[i] = '1'; // If the current digit is '0', increment it to '1'
155             return combinationCopy; // Return the incremented combination
156         }
157
158         combinationCopy[i] = '0'; // If the current digit is '1', set it to '0' and continue
159     }
160
161     strcpy(combination, combinationCopy, sizeOfCombination); // Copy the incremented combination back to the original combination
162     free(combinationCopy); // Free the memory allocated for the copy
163
164     return combination;
165 }
```

6. vectorGenerator: This function generates a random binary vector for boolean functions. It takes the number of variables and the number of boolean functions as input and returns a randomly generated binary vector.

```
168
169 char* vectorGenerator(int powered_number, int bool_functions) {
170
171     char* result = malloc(sizeof(char) * (powered_number + 1)); // Allocates memory for the binary vector
172     srand(time(NULL)); // Seed the random number generator with the current time
173
174     for (int i = 0; i < powered_number; i++) {
175         int random_number = rand() % (2 + bool_functions); // Generate a random number in the range [0, 2 + bool_fun
176         switch (random_number) {
177             case 0:
178                 result[i] = '0'; // Set the corresponding digit in the binary vector to '0'
179                 break;
180             default:
181                 result[i] = '1'; // Set the corresponding digit in the binary vector to '1'
182                 break;
183         }
184     }
185     result[powered_number] = '\0'; // Null-terminate the binary vector string
186     return result; // Return the generated binary vector
187
188 }
189
190 // Performs testing based on the number of variables and boolean functions
191
192
193
194
```

7. testing: This function performs testing based on the number of variables and boolean functions provided. It generates random vector arrays using vectorGenerator, creates BDDs using BDD_create, evaluates the BDDs for each key, and measures the processing time. It calls the information function to display the test results.
8. main: This is the main function that handles user input for selecting the test scenario. It prompts the user to choose a test case and calls the testing function accordingly.

The code is correct in terms of implementing the BDD construction, evaluation, and testing logic. In Code i used recursion to build the BDD and it provide the creation of BDDs from boolean functions. The testing function generates random vector arrays and evaluates the BDDs for each key, providing information about the processing time for each test case. The code provides a way to create, evaluate, and test Binary Decision Diagrams.

Test Results:

Test 1: 2 variables, 200 functions

Time for the whole test: 0.011 seconds

Average time per function: 0.000 seconds

Test 2: 2 variables, 300 functions

Time for the whole test: 0.013 seconds

Average time per function: 0.000 seconds

Test 3: 5 variables, 200 functions

Time for the whole test: 0.020 seconds

Average time per function: 0.000 seconds

Test 4: 5 variables, 300 functions

Time for the whole test: 0.030 seconds

Average time per function: 0.000 seconds

Test 5: 10 variables, 200 functions

Time for the whole test: 0.770 seconds

Average time per function: 0.004 seconds

Test 6: 10 variables, 300 functions

Time for the whole test: 1.404 seconds

Average time per function: 0.005 seconds

Test 7: 13 variables, 200 functions

Time for the whole test: 6.302 seconds

Average time per function: 0.032 seconds

Test 8: 13 variables, 300 functions

Time for the whole test: 9.902 seconds

Average time per function: 0.033 seconds

Test 9: 15 variables, 200 functions

Time for the whole test: 14.329 seconds

Average time per function: 0.072 seconds

Observations:

The processing time remains relatively low throughout all the tests, even with increasing variables and functions.

The average time per function is consistently very low, indicating efficient processing.

As the number of variables and functions increases, there is a gradual increase in processing time.

Overall, the tests demonstrate that the BDD implementation performs well, with low processing times and average times per function