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.

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.

```c
BDD* BDD_create(BF* bfunkcia) {

    int totalNodes = 0;                      // Variable to track the total number of nodes in the BDD
    int vectorLength = bfunkcia->length;     // Length of the boolean function's vector array

    NODE* startNode = malloc(sizeof(NODE));;    // Create a new node as the starting node of the BDD
    BDD* bdd = malloc(sizeof(BDD));             // Create a new BDD struct

    startNode = buildBDD(startNode, bfunkcia->vector_array);     // Build the BDD recursively

    while (vectorLength != 1)
    {
        totalNodes += vectorLength;
        vectorLength /= 2;
    }
    // Calculate the total number of nodes in the BDD by halving the vector length iteratively

    bdd->head = startNode;                 // Set the start node as the head of the BDD
    bdd->nodes = totalNodes;               // Set the total number of nodes in the BDD
    bdd->variables = log2(strlen(bfunkcia->vector_array));  // Calculate the number of variables in the BDD

    return bdd;          // Return the created BDD
}
```

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.

```c
char BDD_use(BDD* bdd, char* inputs) {

    signed char result = -1;                         // Initialize the result value as -1 (indicating an error)
    if (bdd->head == NULL)
        return result;                               // Return the result if the BDD is empty (no head node)


    NODE* currentNode = bdd->head;                     // Start from the head of the BDD

    for (int i = 0; i < bdd->variables; i++)
    {


        if (inputs[i] == '0')
            currentNode = currentNode->left;              // Traverse to the left child if the input is '0'

        if (inputs[i] == '1')
            currentNode = currentNode->right;               // Traverse to the right child if the input is '1'

    }
    return *currentNode->value;                         // Return the value stored in the final node
}
```

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.

```c
char* Binary_increment(char* combination, int size) {


    int sizeOfCombination = size;
    char* combinationCopy = malloc(sizeOfCombination * sizeof(char));              // Create a copy of the combination

    strncpy(combinationCopy, combination, sizeOfCombination);

    for (int i = sizeOfCombination - 1; i >= 0; i--)
    {
        if (combinationCopy[i] == '0')
        {
            combinationCopy[i] = '1';                       // If the current digit is '0', increment it to '1'
            return combinationCopy;                              // Return the incremented combination
        }

        combinationCopy[i] = '0';                              // If the current digit is '1', set it to '0' and continue
    }

    strncpy(combination, combinationCopy, sizeOfCombination);          // Copy the incremented combination back to the original combina
    free(combinationCopy);                                             // Free the memory allocated for the copy

    return combination;
}
```

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.

```c
char* vectorGenerator(int powered_number, int bool_functions) {


    char* result = malloc(sizeof(char) * (powered_number + 1));          // Allocate memory for the binary vector
    srand(time(NULL));                                                    // Seed the random number generator with the current time

    for (int i = 0; i < powered_number; i++) {
        int random_number = rand() % (2 + bool_functions);                // Generate a random number in the range [0, 2 + bool_fun
        switch (random_number) {
        case 0:
            result[i] = '0';                                              // Set the corresponding digit in the binary vector to '0'
            break;
        default:
            result[i] = '1';                                              // Set the corresponding digit in the binary vector to '1
            break;
        }
    }
    result[powered_number] = '\0';                                        // Null-terminate the binary vector string

    return result;                                                        // Return the generated binary vector


}

// Performs testing based on the number of variables and boolean functions
```

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