Synthesis & Optimization of Digital Circuits - Project Documentation

Arvind S Kumar
Sripathi M
Shantanu Vijay

November 5, 2017

Overview

Synthesis and optimisation of digital circuits involves implementing scripts that perform various tasks such as graph building, searching, logical operations and recursion. As part of the projects, lots of useful functions have been developed using **Python 3.0**.

For Project 1, Binary Decision Diagrams were implemented. The programs render a given logical function as a COBDD, ROBDD or ITE Recursion.

For Project 2, a search for minimum cost cover was carried out given an implicant and cover table along with costs for each cover.

For Project 3, many implicant operations such as Sharp, Disjoint Sharp, Consensus, Co-factors, Union, Intersection & Complementation were implemented. Another part was the Unate Recursive Paradigm, which can be used to find if a function is a Tautology or not.

Binary Decision Diagrams

Salient Features

- Builds COBDD, ROBDD and ITE for a given function in PCN format.
- Uses a computationally efficient algorithm to get the leaf nodes of BDD.
- Variable ordering is taken into account.
- Simple UI that can be used to build all graphs and edit the input contents.

Modules

- Complete Order Binary Decision Diagram (COBDD)
- Reduced Order Binary Decision Diagram (ROBDD)
- If-Then-Else Recursion (ITE)

Software

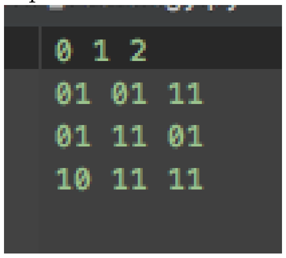
- Graphviz
- Numpy 1.13
- Python 3 Data Structures: Text Files, Stacks & Dictionaries

Usage

- Change contents of `PCN_data.txt` with the first line being variable ordering and the remaining lines corresponding to implicants in PCN format.
- Run `COBDD.py` to generate a COBDD graph stored in `COBDD.pdf`
- Run `ROBDD.py` to generate a COBDD graph stored in `ROBDD.pdf`
- Run `ITE.py` to generate If-then-else recursion which is stores in a text file `ITE.txt`
- Run `UI.py` to open up an easy to use User Interface to perform the above functions by clicking buttons. Just edit the path to input file and application used to open it.

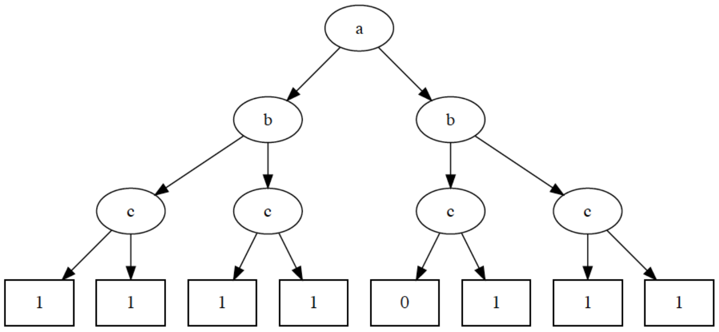
Example

- Input Format

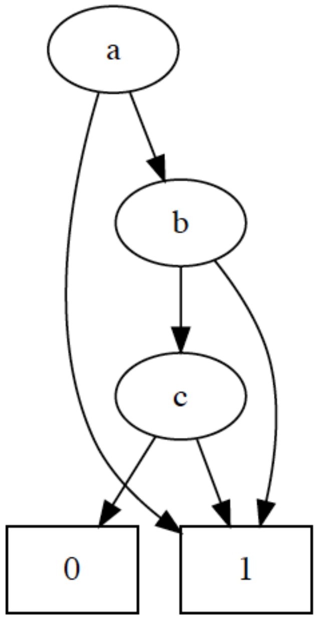


```
0 1 2
01 01 11
01 11 01
10 11 11
```

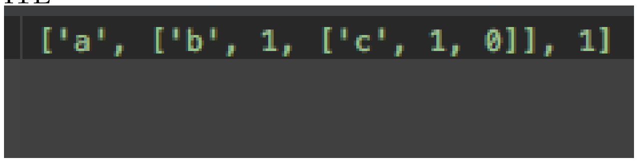
- COBDD



- ROBDD

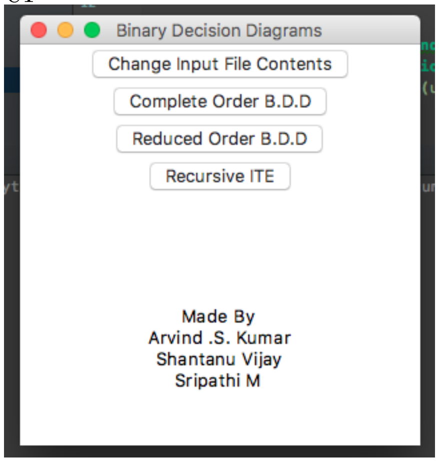


- ITE



```
['a', ['b', 1, ['c', 1, 0]], 1]
```

- UI



Made By
Arvind .S. Kumar
Shantanu Vijay
Sripathi M

Implementation

Input

Function: `readPCN.readData()`
Returns: Logic Function *(numpy matrix)*, Number of Variables *(int)*, Variable Ordering *(list)*
Description: Reads the input file and returns the matrix representing the function in the specified order of splitting. First line of the text file is splitting order. It is followed by implicants in PCN format. All of this is read into splitVarOrder and data_matrix considering corner cases such as void cubes.

- Reordering Logic Function matrix to account for splitting order.

```
    data_out = np.copy(data_matrix)
    for i, j in zip(splitVarOrder, range(len(splitVarOrder))):
        data_out[:, j] = data_matrix[:, int(i)]
```

- Corner Case: If input cubes are all 00s, output will be a flag signal, which will force BDDs and ITEs to generate 0 as output.

Leaf Nodes of BDD

Function: `OBDD.get_leaf_nodes()`
Return: Leaf Nodes *(numpy array)*

Description:

- Returns the leaf nodes of the BDD using a novel algorithm that has a better time complexity than standard procedures. In essence, it calculates the indices at which '1' is present in an array of leaf nodes.

```
def get_leaf_nodes():
    max_no = 26  # Max No. Of Variables
    answer = np.zeros(2 ** no_of_splitting_var)

    for pcn_row_1 in pcn_mat:
        pcn_row = list(pcn_row_1)
        s = Stack()  # To Hold Occurrence of Don't Care in ...
            the BDD
        s.push(1)
        for i in range(len(pcn_row)):
            if pcn_row[i] == "11":
                s.push(i+max_no)

        ini_pos = 0
        for i in range(len(pcn_row)):
            ini_pos += weight(i, no_of_splitting_var, ...
                pcn_row[i])

        answer[int(ini_pos)] = s.peek()
        while s.peek() != 1:
            popkey = s.pop()
            indices = np.argwhere(answer == popkey)
            indices = indices.reshape(len(indices))
            for i in indices:
                answer[i] = popkey
                answer[i + dist(popkey-max_no+1, ...
                    no_of_splitting_var) + 1] = popkey
            answer = np.where(answer == popkey, s.peek(), ...
                answer)
    return answer
```

- Finding Position of 1 in leaf node array based on cube. Function to Hash to a Particular Location in Leaf Node Array. col_no corresponds to the variable, n_size is number of variables and pcode is PCN.

```
def weight(col_no, n_size, pcode):
    if pcode == "10" or pcode == "11":
        return 0
    elif pcode == "01":
        return (2 ** n_size) / (2 ** (col_no+1))
```

- In case a variable is don't care in an implicant, then, the position of 1 is decided by this distance formula. Here n is number of variables and col_no corresponds to the variable that is don't care.

```python
def dist(col_no, n):
    return (2 ** (n-col_no)) - 1
```

Building the COBDD

Function: `OBDD.build()`
Return: COBDD as a Heap *(numpy array)*
Description: Takes the leaf nodes from the previous function and builds a heap, since we know this is a binary tree with both children filled.

Rendering COBDD

Function: `COBDD.cobdd()`
Output: Visualizes the COBDD *(COBDD.pdf)*
Description: Uses the above built heap to render it as a *.pdf* file using Graphviz. The resulting visual graph is stored in `COBDD.pdf`

- Building Edges using Graphviz

```python
def add_edges(graph, edges):
    for e in edges:
        if isinstance(e[0], tuple):
            graph.edge(*e[0], **e[1])
        else:
            graph.edge(*e)
    return graph
```

- Building Nodes using Graphviz

```python
def add_nodes_graph(graph1, nodes1):  # Function To Add ...
    Node to Grpah

    # Iterating over all nodes in the list
    for n in nodes1:
        if isinstance(n, tuple):
            graph1.node(n[0], **n[1])
        else:
            graph1.node(n)
    return graph1
```

- We then create a python lost of nodes and edges using the heap.

- Rendering the Graph

```
    graph = Digraph(format="pdf")
    g = add_edges_graph(add_nodes_graph(graph, nodes), edges)
    g.render('COBDD', view=True)
```

Building & Rendering ROBDD

Function: `ROBDD.robdd()`
Output: Visualizes the ROBDD *(ROBDD.pdf)*
Description: Uses the built heap to render it as a *.pdf* file using Graphviz. The resulting visual graph is stored in `ROBDD.pdf`. The procedure is same as that of COBDD, except an additional reduction algorithm creates the id_dict variable, that contains all node and edge information. Now, the COBDD(heap) is converted to ROBDD(Node, Edge Relations List) based on the following conditions:

- Create the table and assign identifier for a node visited the first time.
- If left and right child are same, node is not considered.
- If 2 nodes have same left and right children, they are given same id (merged).

```
        node_list = OBDD.build()
        id_dict = list()
        id_dict.append(("0", "-", "-"))
        id_dict.append(("1", "-", "-"))

        for i in range(int(len(node_list)/2)-1, -1, -1):
            left = node_list[2*i + 1]
            right = node_list[2*i + 2]
            temp = (node_list[i], left, right)
            if left == right:
                node_list[i] = str(left)
            elif temp not in id_dict:
                id_dict.append(temp)
                node_list[i] = str(len(id_dict)-1)
            else:
                node_list[i] = str(id_dict.index(temp))
```

Building ITE Recursion

Function: `ITE.ite_master()`
Output: ITE String *(ITE.txt)*
Description: Working is same as that of ROBDD. Terminating condition is if a node's children are 0 or 1. The following recursive function does the job when we pass the root node along with it's children information.

```python
def ite(tup):
    if tup == 1 or tup == 0:
        return tup
    else:
        tup[1] = ite(id_dict[int(tup[1])])
        tup[2] = ite(id_dict[int(tup[2])])
        return tup
```

Minimum Cost Cover

Salient Features

- Finds minimum cost cover of a function given in PCN format.
- Each implicant can be assigned a cost so that, the cost is minimized.

Modules

- Minimum Cost Cover Search
- Finding Essential Primes

Software

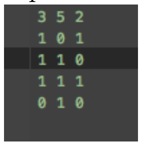
- Numpy 1.13
- Python 3 Data Structures: Text Files & Lists

Usage

- Change contents of `Implicant_Table_Costs.txt` with the first line being implicant costs and the remaining lines corresponding to whether an minterm(column) is present in a cover(row) or not in a fashion similar to how it was presented in the lectures.
- Run `MinCostCover.MinCostCover(filename)` with `Implicant_Table_Costs.txt` as the argument
- Output is printed to the console in a formatted fashion. It prints the essential primes, cost of each cover and finally prints the cover that has the minimum cost.
- To make the program print minimum cover, assign all weights to 1.

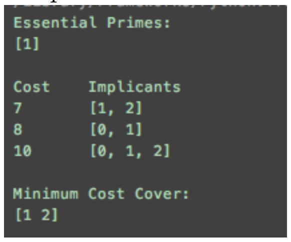
Example

- Input Format



```
3 5 2
1 0 1
1 1 0
1 1 1
0 1 0
```

- Output Format



```
Essential Primes:
[1]

Cost    Implicants
7       [1, 2]
8       [0, 1]
10      [0, 1, 2]

Minimum Cost Cover:
[1 2]
```

Implementation

Input

File: `Implicant_Table_Costs.txt`
Description: First line of the file is costs relative to 1. If all costs are same, then minimum cover is found. Remaining part of the file is a matrix. Rows correspond to Cover, columns correspond to minterms. Values in each cell is 1 or 0 depending on whether or not the cover has the particular minterm. Reading input is done in a fashion similar to Binary Decision Diagrams.

Removing Essential Primes

Function: `MinCostCover.RemEssPrimes(filename)`
Input: Text file in the above specified format.
Output: Matrix after removing essential prime rows *(numpy matrix)*, prime implicants *(list)*, cost *(numpy array)*
Description: Finds essential prime implicants. If any column has only one 1, then the implicant can be covered by that cover only. We use this information to remove certain rows corresponding to minterms that are already covered. Finally, a reduced matrix is returned.

Minimum Cost Search

Function: `MinCostCover.MinCostCover(filename)`
Input: Text file in the above specified format.
Output: Prints essential primes, cost of each cover and finally prints the cover that has the minimum cost.

Description: It uses `MinCostCover.RemEssPrimes(filename)` to obtain a reduced matrix. It runs through the whole space of possible implicant combinations and calculates costs of those that cover the function, thereby not leaving out any possible combination.
Cover checking is done in the following way. If after removing all covers' rows, if any minterm is left uncovered (any column sums to 1), then, we update flag and continue our search.

```
        for i in range(1, 2 ** no_of_col):
            ... Some other computations ...
            for ij in list(np.where(chosen == 1)[0]):
                col_sum += good_matrix[:, ij]
            flag2 = 0
            for st in col_sum:
                if st == 0:
                    flag2 = 1  # Indicates not covering.
            if flag2 == 1:
                continue
```

Operations On Implicants

Modules

- Sharp
- Disjoint Sharp
- Consensus
- Co-factor with respect to implicants
- Union
- Intersection
- Complementation
- Tautology Check using URP

Software

- Numpy 1.13

Usage & Implementation

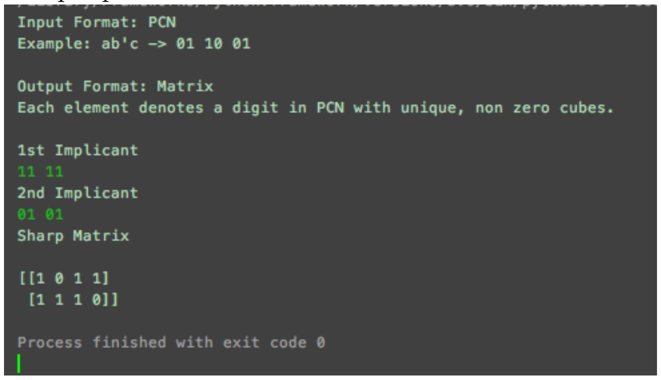
Note: All scripts prompt for input when run. So, just run the python scripts and provide input in the specified format for each operation.

A support function called Check_void.check() checks if in any operation result there is a void cube, repeat of a cube or containment and removes them.

For smooth running, inputs such as 11 11 for both implicants should be avoided. But nevertheless, void cubes can be given as input and that will be taken care of.

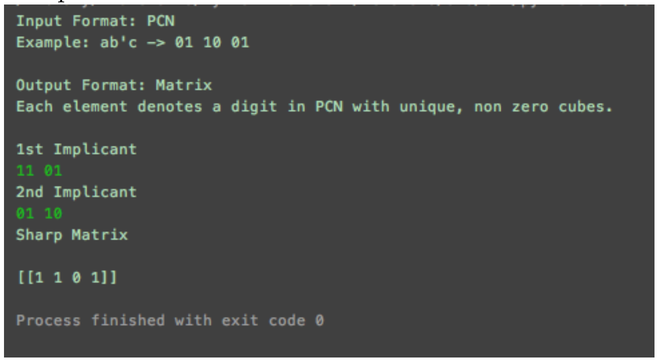
Sharp

Script: `Sharp.py`
Input: a,b *(Implicants in PCN Format with space between each variable)*
Output: Sharp Matrix *(Numpy matrix in PCN Format)*
Examples:
Sharp Operation



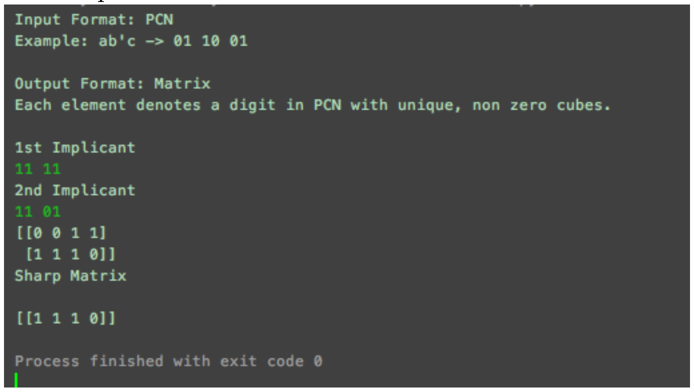
```
Input Format: PCN
Example: ab'c -> 01 10 01

Output Format: Matrix
Each element denotes a digit in PCN with unique, non zero cubes.

1st Implicant
11 11
2nd Implicant
01 01
Sharp Matrix

[[1 0 1 1]
 [1 1 1 0]]

Process finished with exit code 0
```

Sharp with containment



```
Input Format: PCN
Example: ab'c -> 01 10 01

Output Format: Matrix
Each element denotes a digit in PCN with unique, non zero cubes.

1st Implicant
11 01
2nd Implicant
01 10
Sharp Matrix

[[1 1 0 1]]

Process finished with exit code 0
```

Sharp with some Void Cubes

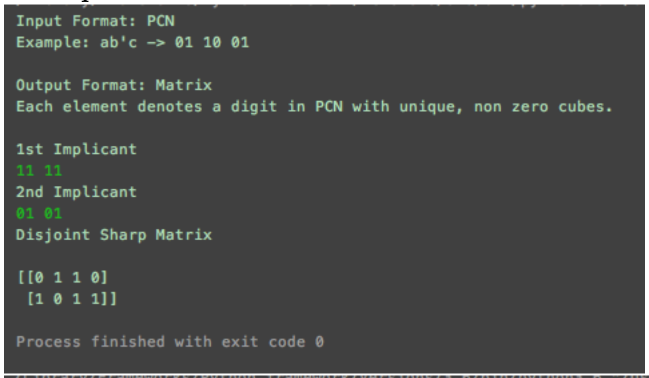


```
Input Format: PCN
Example: ab'c -> 01 10 01

Output Format: Matrix
Each element denotes a digit in PCN with unique, non zero cubes.

1st Implicant
11 11
2nd Implicant
11 01
[[0 0 1 1]
 [1 1 1 0]]
Sharp Matrix

[[1 1 1 0]]

Process finished with exit code 0
```

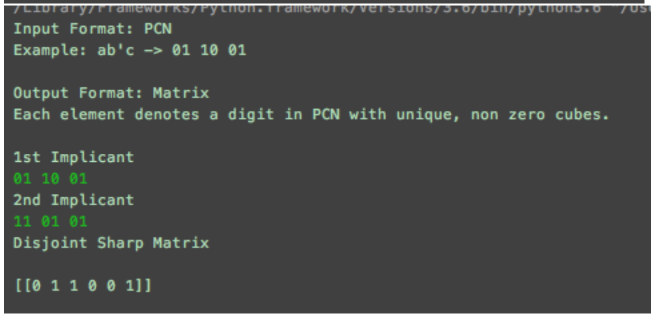
Description: Returns sharp of two implicants in PCN format. Removes all the void cubes and repetitive cubes. Also checks containment.

Disjoint Sharp

Script: `Dis_Sharp.py`
Input: a,b *(Implicants in PCN Format with space between each variable)*
Output: Disjoint Sharp Matrix *(Numpy matrix in PCN Format)*
Examples:



```
Input Format: PCN
Example: ab'c -> 01 10 01

Output Format: Matrix
Each element denotes a digit in PCN with unique, non zero cubes.

1st Implicant
11 11
2nd Implicant
01 01
Disjoint Sharp Matrix

[[0 1 1 0]
 [1 0 1 1]]

Process finished with exit code 0
```

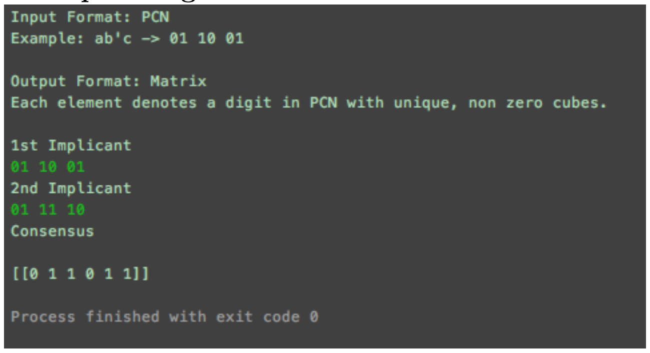


```
Input Format: PCN
Example: ab'c -> 01 10 01

Output Format: Matrix
Each element denotes a digit in PCN with unique, non zero cubes.

1st Implicant
01 10 01
2nd Implicant
11 01 01
Disjoint Sharp Matrix

[[0 1 1 0 0 1]]
```

Description: Returns disjoint sharp of two implicants in PCN format. Removes all the void cubes and repetitive cubes. Also checks containment.

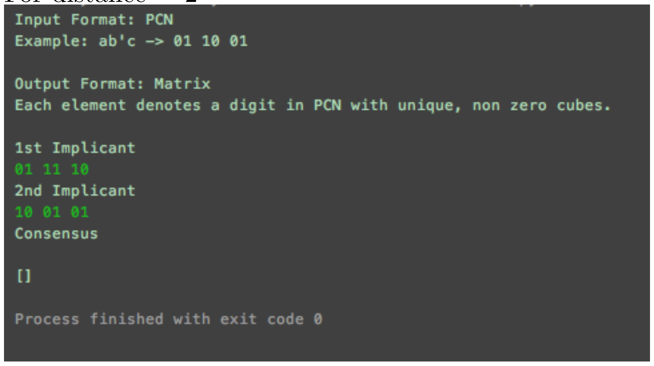
Consensus

Script: `Consensus.py`
Input: a,b *(Implicants in PCN Format with space between each variable)*
Output: Consensus Matrix *(Numpy matrix in PCN Format)*
Example Usage:



```
Input Format: PCN
Example: ab'c -> 01 10 01

Output Format: Matrix
Each element denotes a digit in PCN with unique, non zero cubes.

1st Implicant
01 10 01
2nd Implicant
01 11 10
Consensus

[[0 1 1 0 1 1]]

Process finished with exit code 0
```

For distance = 2



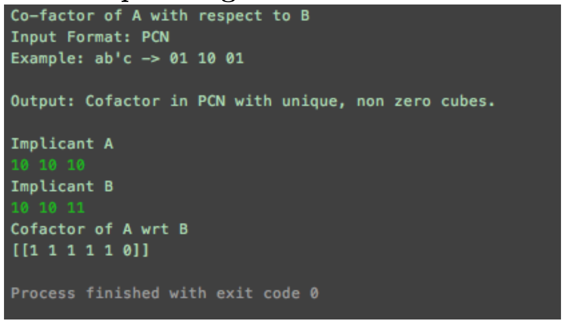
```
Input Format: PCN
Example: ab'c -> 01 10 01

Output Format: Matrix
Each element denotes a digit in PCN with unique, non zero cubes.

1st Implicant
01 11 10
2nd Implicant
10 01 01
Consensus

[]

Process finished with exit code 0
```

Description: Returns consensus of two implicants in PCN format. Removes all the void cubes and repetitive cubes. Also checks containment.

Co-factor with respect to implicants

Script: `Cofactor.py`
Inputs: a *(Single Implicant Function in PCN Format with space between each variable)*
b *(Implicant in PCN Format with space between each variable)*
Output: Cofactor Matrix *(Numpy matrix in PCN Format)*

Example Usage:

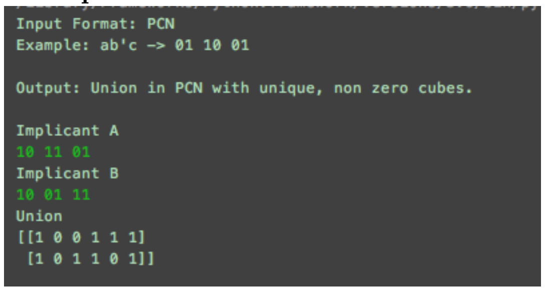


```
Co-factor of A with respect to B
Input Format: PCN
Example: ab'c -> 01 10 01

Output: Cofactor in PCN with unique, non zero cubes.

Implicant A
10 10 10
Implicant B
10 10 11
Cofactor of A wrt B
[[1 1 1 1 1 0]]

Process finished with exit code 0
```

Description: Returns cofactor of A with respect to B in PCN format. Removes all the void cubes and repetitive cubes. Also checks containment.

Union

Script: `Union.py`
Inputs: a, b *(Implicants in PCN Format with space between each variable)*
Output: Union Matrix *(Numpy matrix in PCN Format)*
Example:



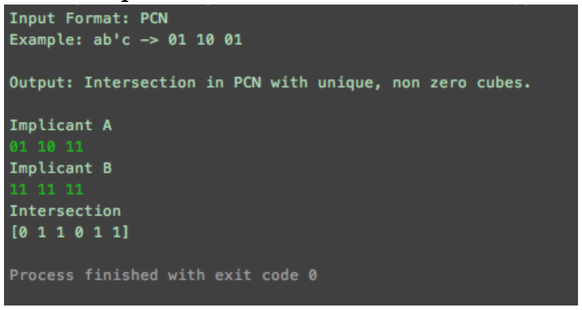
```
Input Format: PCN
Example: ab'c -> 01 10 01

Output: Union in PCN with unique, non zero cubes.

Implicant A
10 11 01
Implicant B
10 01 11
Union
[[1 0 0 1 1 1]
 [1 0 1 1 0 1]]
```

Description: Returns Union in PCN format. Removes all the void cubes and repetitive cubes. Also checks containment.

Intersection

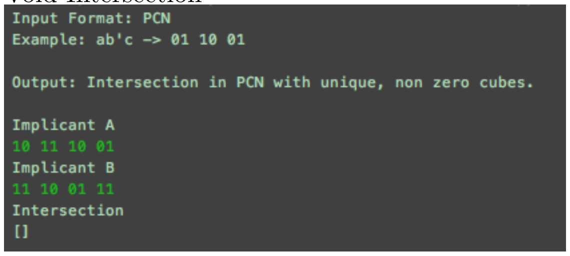
Script: `Intersection.py`
Inputs: a, b *(Implicants in PCN Format with space between each variable)*
Output: Intersection Array *(Numpy matrix in PCN Format)*

Examples:



```
Input Format: PCN
Example: ab'c -> 01 10 01

Output: Intersection in PCN with unique, non zero cubes.

Implicant A
01 10 11
Implicant B
11 11 11
Intersection
[0 1 1 0 1 1]

Process finished with exit code 0
```

Void Intersection

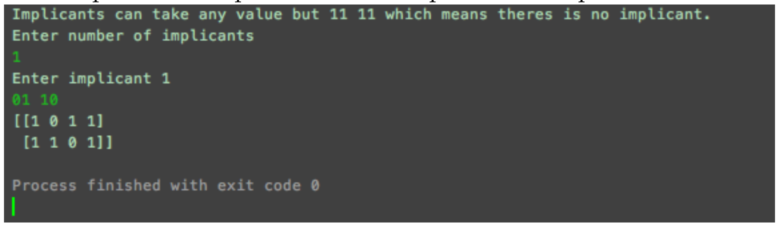


```
Input Format: PCN
Example: ab'c -> 01 10 01

Output: Intersection in PCN with unique, non zero cubes.

Implicant A
10 11 10 01
Implicant B
11 10 01 11
Intersection
[]
```

Description: Returns Intersection array in PCN format. Prints empty array if intersection is void.

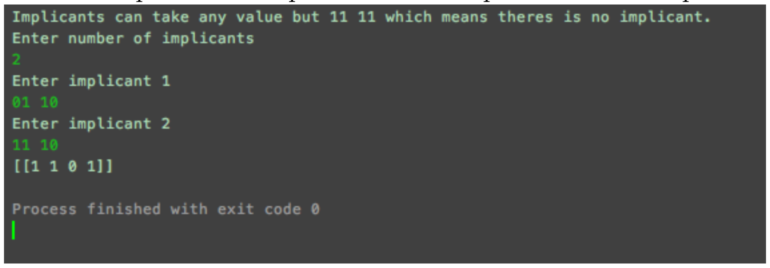
Complementation

Script: `Complementation.py`
Inputs: number of implicants *(int)*, Function *(Set of Implicants in PCN Format with space between each variable)*
Output: Complemented function *(Numpy matrix in PCN Format)*
Example Usage:
One Implicant Complementation: Input: ab' Output: a'+b



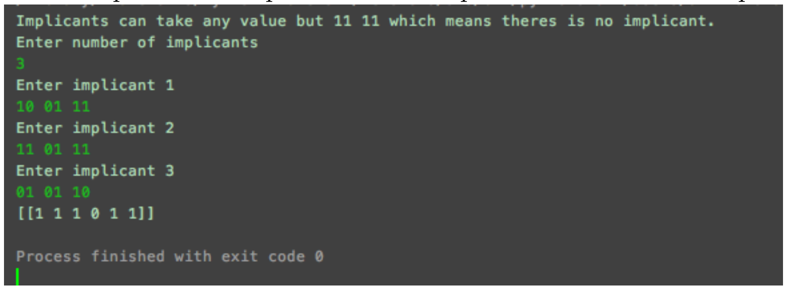
```
Implicants can take any value but 11 11 which means theres is no implicant.
Enter number of implicants
1
Enter implicant 1
01 10
[[1 0 1 1]
 [1 1 0 1]]

Process finished with exit code 0
```

Two Implicants Complementation: Input ab' + b' Output: b



```
Implicants can take any value but 11 11 which means theres is no implicant.
Enter number of implicants
2
Enter implicant 1
01 10
Enter implicant 2
11 10
[[1 1 0 1]]

Process finished with exit code 0
```

Three Implicants Complementation: Input a'b + b + abc' Output: b'

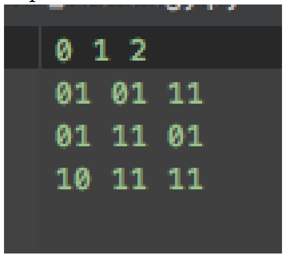


```
Implicants can take any value but 11 11 which means theres is no implicant.
Enter number of implicants
3
Enter implicant 1
10 01 11
Enter implicant 2
11 01 11
Enter implicant 3
01 01 10
[[1 1 1 0 1 1]]

Process finished with exit code 0
```

Description: Returns complemented function in PCN format.

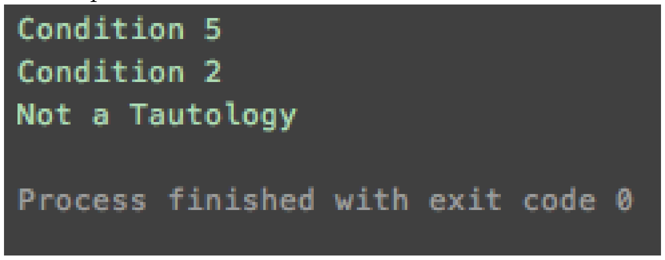
Unate Recursive Paradigm

Script: `URP_Tautology.py`
Inputs: `PCN_data.txt` *(1st line corresponds to splitting order. Remaining lines correspond to the function in PCN format)*
Output: Tautology or not *(String)*
Example Usage:
Input:



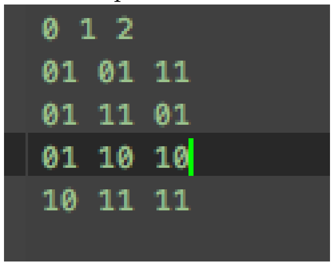
```
0 1 2
01 01 11
01 11 01
10 11 11
```

Output:



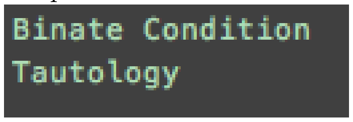
```
Condition 5
Condition 2
Not a Tautology

Process finished with exit code 0
```

Binate Input:



```
0 1 2
01 01 11
01 11 01
01 10 10
10 11 11
```

Output:



```
Binate Condition
Tautology
```

Description: Returns if a function is tautology or not using Unate Recursive Paradigm. In case, the function is Binate, it calls the Binary Decision Diagram Module to find it the function is tautology or not.