МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ

УНИВЕРСИТЕТ им. Р.Е.АЛЕКСЕЕВА

Институт радиоэлектроники и информационных технологий

Кафедра информатики и систем управления

ОТЧЕТ

по курсовой работе

по дисциплине

Надежность и качество АСО и У.

по теме

Анализ надежности с помощью деревьев отказов. Автоматизация построения дерева отказов.

РУКОВОДИТЕЛЬ:

_____ Соколова Э.С.

(подпись) (фамилия, и.,о.)

СТУДЕНТ:

_____ Напылов Е.И.

(подпись) (фамилия, и.,о.)

_М22-ИВТ-4_____

(шифр группы)

Работа защищена «___» _____

С оценкой _____

Нижний Новгород 2022

Содержание

1. Постановка задачи

В настоящее время автоматизированные системы состоят из огромного числа блоков. Рано или поздно какой-нибудь из блоков выходит из строя. В результате этого может выйти из строя и вся система целиком. Для уменьшения вероятности таких нежелательных событий необходимо проводить анализ надежности систем. Существуют различные методы анализа надежности. У многих из них есть один общий недостаток - отсутствие наглядности. Метод деревьев отказа лишен данного недостатка и позволяет наглядно отследить причинно-следственные связи для анализа надежности системы. Построение деревьев отказов является достаточно сложной задачей, поэтому было бы неплохо автоматизировать данный процесс.

Цель работы - разработка программы для построения деревьев отказов на основе надежностной схемы системы.

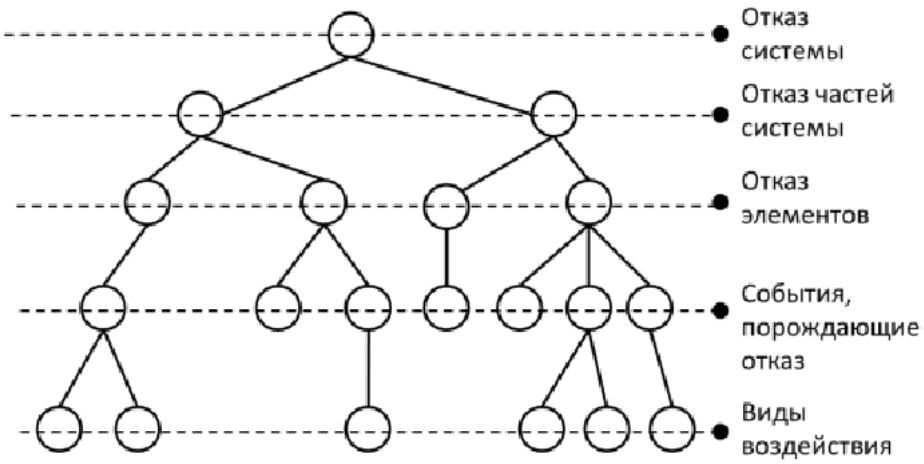
Задачи:

1. Изучение метода анализа надежности систем с помощью деревьев отказа.
2. Разработка алгоритма автоматического построения деревьев отказов на ЯП Python.
3. Разработка визуализатора дерева отказов с помощью ЯП Python и библиотеки Graphviz.

2. Деревья отказов

Дерево отказов лежит в основе логико-вероятностной модели причинно-следственных связей отказов системы с отказами ее элементов и другими событиями. Дерево отказов состоит из последовательностей и комбинаций неисправностей, и таким образом представляет собой многоуровневую структуру причинных взаимосвязей, полученных в результате прослеживания ситуаций в обратном порядке, для того чтобы отыскать возможные причины их возникновения.

Дерево отказов состоит из определенного списка элементов. Листьями являются события, связанные с отказом компонент, программными ошибками, человеческими ошибками и т.д. Корнем является ключевое событие, т.е. отказ системы. Промежуточные вершины - события, которые являются отказами блоков, вызванных причинами из предыдущего уровня.



Для каждого события-листа рассчитывается вероятность его возникновения. Затем рассчитывается вероятность возникновения отказа на блоках следующего уровня, и т.д. В итоге вычисление вероятности доходит до корня дерева, т.е. до того события, ради которого выполняется анализ.

Деревья отказов строятся на основе следующего алгоритма:

1. Определение конечного события, например отказ всей системы.
2. Определение причин или событий для определения того, что может вызвать событие-корень.
3. Этапы 1 и 2 повторяются до тех пор, пока не получаются события-листья, которые являются нерасщепляемыми.

Преимуществами метода анализа дерева отказов являются:

1. Универсальность - разнообразие причин различного рода
2. Подход “сверху вниз” позволяет рассматривать только то, что влияет на конечный результат
3. Подходит для сложных систем
4. Показывает узкие места системы
5. Наглядное представление
6. Выявляет проблемные места
7. Позволяет анализировать сложные логические взаимосвязи

В тоже время существует несколько недостатков:

1. Большие временные и денежные затраты
2. Элементы имеют только 2 состояния - исправное и неисправное
3. Сложно учитывать резервирование элементов
4. Требуются высококвалифицированные специалисты
5. Описывает систему только в определенный момент времени
6. Нет возможности работать с обратной связью

3. Разработка программы для автоматического построения дерева отказов

3.1 Входные данные

Во-первых, необходимо разобраться с входными данными и их форматом. Для построения дерева надежности требуются: схема надежности, состоящая из блоков (устройств), список названий блоков и список причин выхода из строя для каждого блока.

Схема надежности представляется в виде графа. Для хранения графа используется список смежности. На языке программирования Python для списка смежности удобно использовать словарь. Например, для схемы надежности, имеющей вид:

словарь будет иметь вид:

```
G = {
    '1': ['2', '3'],
    '2': ['4'],
    '3': ['5'],
    '4': ['6', '7'],
    '5': ['7'],
    '6': ['8'],
    '7': ['8'],
    '8': []
}
```

Список смежности не имеет особой ценности без списка названий блоков, соответствующих каждой вершине. Для хранения названий используется словарь вида:

```
names = {
    'system': '"Система очистки воздуха"',
    '1': '"Фильтр грубой очистки"',
    '2': '"Центробежный вентилятор №1"',
    '3': '"Центробежный вентилятор №2"',
    '4': '"Фильтр мелкой очистки №1"',
    '5': '"Фильтр мелкой очистки №2"',
    '6': '"Обеззараживатель №1"',
    '7': '"Обеззараживатель №2"',
    '8': '"Распределитель чистого воздуха"',
}
```

У каждого блока помимо “следственных” причин выхода из строя существуют собственные причины. Для их хранения также используется словарь:

```
reasons = {
    'r_1': ['Заблокирован', 'Пробоина'],
    'r_2': ['Отсутствие питания'],
    'r_3': ['Отсутствие питания'],
    'r_4': ['Сильно загрязнен'],
    'r_5': ['Сильно загрязнен'],
    'r_6': ['Неисправность УФ ламп', 'Отсутствие питания'],
    'r_7': ['Неисправность УФ ламп', 'Отсутствие питания'],
    'r_8': ['Пробоина'],
}
```

3.2 Поиск путей распространения неисправности

Предположим, что выход из строя любого из предыдущих блоков ведет к выходу строя текущего блока. В таком случае неисправность будет распространятся от первого блока к последнему. Для поиска таких путей используется рекурсивный алгоритм:

```
def find_paths_algorithm(a, b, G, paths=[], q=[], visited=set()):
    if a == b:
        paths.append(copy(q))
        return paths
    for u in G[a]:
        if u not in visited:
            q.append(u)
            visited.add(u)
            paths = find_paths_algorithm(u, b, G, paths, q, visited)
            q.pop()
            visited.remove(u)
    return paths

def find_all_paths(v1, v2, G):
    return [x[::-1]+[v1] for x in find_paths_algorithm(v1, v2, G)]
```

В результате работы данного алгоритма получается список путей:
[['8', '6', '4', '2', '1'], ['8', '7', '4', '2', '1'], ['8', '7', '5', '3', '1']]

3.3 Построение дерева распространения неисправностей

Основная сложность построения дерева неисправностей заключается в том, что во время построения необходимо учитывать вершины, которые можно слить в одну и те, которые нельзя слить в одну. Для определения таких вершин используется алгоритм, который для каждого уровня находит пары вершин, которые можно соединить:

```
largest_path = find_largest_path(paths)
for level in range(1, len(largest_path)):
    all_pairs = get_all_pairs_on_level(paths, level)
    to_merge_paths_ids = set()
    to_merge_then_add = []
    for pair in all_pairs:
        pair_can_be_merged = is_can_be_merged(paths[pair[0]["path_id"]], paths[pair[1]["path_id"]], level)
        if pair_can_be_merged:
            to_merge_then_add.append(pair)
            to_merge_paths_ids.add(pair[0]['path_id'])
            to_merge_paths_ids.add(pair[1]['path_id'])
    not_to_merge_paths_ids = list(set([i for i in range(len(paths))]).difference(to_merge_paths_ids))
    to_merge_paths_ids = list(to_merge_paths_ids)
```

Слияние возможно только для тех вершин из списка путей, которые имеют одного и того же предка на предыдущем уровне дерева Для предка необходимо выполнение того же самого условия. Таким образом, получаем рекурсию, которую легко развернуть в цикл. Для проверки условия используется функция:

```
def is_can_be_merged(path1, path2, vertex_index):
    for level in range(vertex_index, 0, -1):
        if not (path1[level] == path2[vertex_index] and path1[level-1] == path2[level-1]):
            return False
    return True
```

Затем полученные вершины добавляются в список смежности целевого дерева:

```
for path_id in not_to_merge_paths_ids:
    if paths[path_id][level-1] in G.keys():
        G[paths[path_id][level-1]].append(paths[path_id][level])
    else:
        G[paths[path_id][level-1]] = [paths[path_id][level]]

for path_id in to_merge_paths_ids:
    if paths[path_id][level-1] in G.keys():
        if paths[path_id][level] not in G[paths[path_id][level-1]]:
            G[paths[path_id][level-1]].append(paths[path_id][level])
    else:
        G[paths[path_id][level-1]] = [paths[path_id][level]]
```

На самом деле после этого шага список еще нельзя назвать списком дерева, поскольку в нем имеется конфликт имен. Для решения этой проблемы вершины переименовываются с помощью дополнительных индексов ({1} -> 1_1, 1_2, …).

3.4 Графическое представление дерева отказов

На предыдущем этапе работы программы получается список смежности дерева распространения ошибок. В нем не хватает названий блоков и внутренних причин выхода из строя этих блоков. Решение данной задачи реализовано в функции, которая кроме того строит графическое представление дерева отказов.

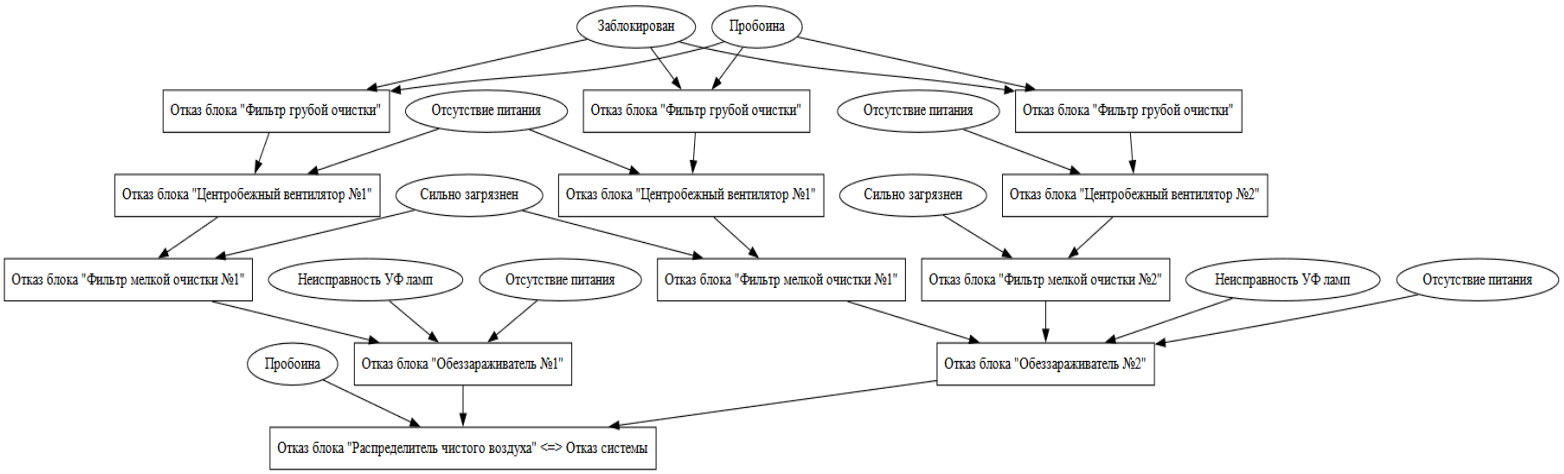
Для построения графического представления дерева отказов была использована библиотека Graphiz. На мой взгляд, данная библиотека является лучшим решением для построения ориентированных графов - удобная, имеет промежуточный язык представления графов, имеет утилиту для построения изображения из промежуточного представления и многое другое.

Построение графического представления реализовано в функции:

```
def plot_graph(G, paths, names, reasons, debug=False):
    dot = graphviz.Digraph(comment='Fault Tree')
    for reasons_block in reasons.keys():
        counter = 0
        for reason in reasons[reasons_block]:
            dot.node(f'{reasons_block}_{counter}', reason)
            counter += 1
    for key in G.keys():
        if debug:
            dot.node(key, key, shape='box')
            for child in G[key]:
                dot.node(child, child)
                dot.edge(key, child)
        else:
            if key == paths[0][0]:
                dot.node(key, 'Отказ блока ' + names[key] + ' <=> Отказ системы', shape='box')
            else:
                dot.node(key, 'Отказ блока ' + names[key.split('_')[0]], shape='box')
            for child in G[key]:
                dot.node(child, 'Отказ блока ' + names[child.split('_')[0]], shape='box')
                dot.edge(key, child)
    for key in G.keys():
        for child in G[key]:
            child_clear = child.split('_')[0]
            for i in range(len(reasons[f'r_{child_clear}'])):
                v1 = child
                v2 = f'r_{child_clear}_{i}'
                dot.edge(v1, v2)
    last_block = paths[0][0]
    for i in range(len(reasons[f'r_{last_block}'])):
        dot.edge(last_block, f'r_{last_block}_{i}')
    dot.view('out/tree.gz')
```

В результате работы данной функции имеем 2 файла - представление дерева отказов с помощью промежуточного языка DOT и графическое представление в формате PDF. PDF-файл автоматически открывается в программе по-умолчанию.

Итоговое дерево отказов имеет вид:



Подробнее с данным деревом можно ознакомиться в приложении 2.

Таким образом, было построено дерево отказов, которое может быть использовано для анализа надежности системы. Промежуточное представление на языке (DOT) может использоваться для быстрого внесения корректировок в дерево без проведения затратной процедуры (для реальных больших систем) построения с нуля. Например, по данной схеме можно сказать, что проблемными местами являются блоки “Фильтр грубой очистки” и “Распределитель чистого воздуха”. С помощью языка DOT можно быстро внести правки - зарезервировать данные блоки и рассчитать показатели надежности для новой системы без повторного построения дерева.

Стоит отметить, что разработанная программа не идеальна. На данный момент текущая версия программы не способна расставлять логические операции между узлами дерева. Кроме того существуют небольшие проблемы с отображением графического представления дерева - алгоритм построения не всегда успешно справляется с задачей приближения к плоской укладке графа, именно поэтому было решено соединять одинаковые листья без их дублирования с последующими событиями.

4. Заключение

В результате работы изучен метода анализа надежности с помощью деревьев отказов. Были выделены основные преимущества и недостатки данного метода. На мой взгляд, главное преимущество заключается в наглядном представлении. В то же время метод деревьев отказов имеет ряд недостатков, из которых наиболее выделяется высокая затратность по времени и денежным средствам.

После изучения теоретических аспектов была разработана программа, позволяющая автоматически строить деревья отказов. Для построения необходимо подать на вход алгоритма список смежности схемы надежности, список названий блоков и список внутренних причин выхода из строя каждого компонента системы. В результате работы программы генерируется дерево отказов в двух форматах - промежуточном языке DOT из Graphviz и наглядное графическое представление в виде PDF файла дерева. На данный момент разработанная программа пока еще далека от идеала, она имеет ряд недостатков, которые необходимо решить в дальнейшем.

5. Приложения

Приложение 1. Представление дерева отказов на промежуточном языке DOT

```
// Fault Tree
digraph {
    r_1_0 [label="Заблокирован"]
    r_1_1 [label="Пробоина"]
    r_2_0 [label="Отсутствие питания"]
    r_3_0 [label="Отсутствие питания"]
    r_4_0 [label="Сильно загрязнен"]
    r_5_0 [label="Сильно загрязнен"]
    r_6_0 [label="Неисправность УФ ламп"]
    r_6_1 [label="Отсутствие питания"]
    r_7_0 [label="Неисправность УФ ламп"]
    r_7_1 [label="Отсутствие питания"]
    r_8_0 [label="Пробоина"]
    8 [label="Отказ блока \"Распределитель чистого воздуха\" <=> Отказ системы" shape=box]
    6 [label="Отказ блока \"Обеззараживатель №1\"" shape=box]
    6 -> 8
    7 [label="Отказ блока \"Обеззараживатель №2\"" shape=box]
    7 -> 8
    6 [label="Отказ блока \"Обеззараживатель №1\"" shape=box]
    4 [label="Отказ блока \"Фильтр мелкой очистки №1\"" shape=box]
    4 -> 6
    7 [label="Отказ блока \"Обеззараживатель №2\"" shape=box]
    "4_1" [label="Отказ блока \"Фильтр мелкой очистки №1\"" shape=box]
    "4_1" -> 7
    5 [label="Отказ блока \"Фильтр мелкой очистки №2\"" shape=box]
    5 -> 7
    4 [label="Отказ блока \"Фильтр мелкой очистки №1\"" shape=box]
    2 [label="Отказ блока \"Центробежный вентилятор №1\"" shape=box]
    2 -> 4
    5 [label="Отказ блока \"Фильтр мелкой очистки №2\"" shape=box]
    3 [label="Отказ блока \"Центробежный вентилятор №2\"" shape=box]
    3 -> 5
    2 [label="Отказ блока \"Центробежный вентилятор №1\"" shape=box]
    1 [label="Отказ блока \"Фильтр грубой очистки\"" shape=box]
    1 -> 2
    3 [label="Отказ блока \"Центробежный вентилятор №2\"" shape=box]
    "1_2" [label="Отказ блока \"Фильтр грубой очистки\"" shape=box]
    "1_2" -> 3
    "4_1" [label="Отказ блока \"Фильтр мелкой очистки №1\"" shape=box]
    "2_1" [label="Отказ блока \"Центробежный вентилятор №1\"" shape=box]
    "2_1" -> "4_1"
    "2_1" [label="Отказ блока \"Центробежный вентилятор №1\"" shape=box]
    "1_1" [label="Отказ блока \"Фильтр грубой очистки\"" shape=box]
    "1_1" -> "2_1"
    r_6_0 -> 6
    r_6_1 -> 6
    r_7_0 -> 7
    r_7_1 -> 7
    r_4_0 -> 4
    r_4_0 -> "4_1"
    r_5_0 -> 5
    r_2_0 -> 2
    r_3_0 -> 3
    r_1_0 -> 1
    r_1_1 -> 1
    r_1_0 -> "1_2"
    r_1_1 -> "1_2"
    r_2_0 -> "2_1"
    r_1_0 -> "1_1"
    r_1_1 -> "1_1"
    r_8_0 -> 8
}
```

Приложение 2. Графическое представление дерева отказов (PDF)