Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика, искусственный интеллект и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

ОТЧЕТ

по лабораторной работе № 3

по курсу «Экспертные системы»

на тему: «Обратный поиск в глубину в графах И-ИЛИ»

Студент ИУ7-31М (Группа) _____ (Подпись, дата) Клименко А. К. (И. О. Фамилия)

Преподаватель _____ (Подпись, дата) Русакова З. Н. (И. О. Фамилия)

2024 г.

Введение

Цель лабораторной работы — приобретение практических навыков реализации алгоритма поиска в глубину в графах И-ИЛИ.

Задачи работы:

1. Разработать программу, реализующую алгоритм поиска в глубину в графах И-ИЛИ.

2. Протестировать программу для различных графов.

1 Теоретический раздел

1.1 Алгоритм поиска в глубину в графах И-ИЛИ

В нем определяется самое левое дерево решения (возможно не оптимальное). Если не глубокий, то его хорошо использовать. Используется идея стека и идея списков. Вводятся следующие списки:

1. Список открытых вершин.
2. Список закрытых вершин.
3. Список открытых и закрытых правил.
4. В методе поиска в глубину - список запрещенных вершин.
5. Список всех правил.

Разработаем класс, полями которого объявляем эти списки, вводим два флага – есть решение, и нет решения. В конструктор класса передаем список базы правил, целевую вершину, массив заданных входных вершин. В конструкторе входные вершины записываем в список закрытых вершин, целевую вершину помещаем в голову стека (который является списком открытых вершин). Флаги решения выставляем в единицу. Все вершины имеют флаги 0, а флаги входных вершин устанавливаем в 1.

Можно выделить 4 метода:

1. Потомки. В этом методе определяем первое правило, которое раскрывает текущую подцель. Возвращает признак нахождения правила (0 или 1).
2. Поиск - основной метод, вызывает все остальные методы.
3. Бэктрекинг - в нем отсекаются запрещенные вершины и правила.
4. Разметка - в нем формируется список закрытых правил, составляющих дерево решения и список доказанных/закрытых вершин.

1.2 Алгоритм определения потомков

В этом методе по принципу стека формируются списки открытых вершин и правил. Текущая подцель выбирается из головы стека открытых вершин. В цикле по базе правил определяем первое правило, выходная вершина которого совпадает с подцелью, то есть раскрывает эту вершину. Номер правила записываем в голову стека открытых правил. Следующая задача – сформировать новые подцели и записать в стек открытых вершин. Для этого надо определить, какие вершины выбранного правила (из входных) не входят в закрытые (сначала это заданные входные вершины). Эти вершины добавляем в голову стека вершин. А в процессе определения таких вершин флаги входных вершин текущего правила (которое входит в список закрытых) ставим в единицу и никуда не пишем. Условие выбора правила: выходная вершина совпадает с подцелью и метка правила не выставлена (мы выбираем его первый раз) и дополнительно можно проверять, что выходная вершина правила не является запрещенной. Метку выбранного правила надо поставить в единицу.

Проверяем следующее – если все входные вершины правила попали в закрытые, то новых подцелей нет, и если выходная вершина является целевой, то флаг решения в 0 (мы нашли решение). Иначе, если выход – нецелевая вершина, то вызываем разметку.

В разметки методе выходную вершину доказанного правила надо переписать в закрытые и удалить из открытых.

1.3 Алгоритм возврата

В этом случае потомков мы не нашли – функция потомков возвратит 0. Это значит, что текущая подцель должна объявляться как запрещенная вершина. Само правило тоже должно быть запрещено. Мы должны при этом текущую подцель удалить из стека открытых правил, правило из головы стека объявить запрещенным, удалить и поместить в запрещенные. Помимо этого надо из стека удалить все подцели, которые являлись входными вершинами запрещенного правила. После этого снова вызываем метод потомки – ищем альтернативный путь для последней вершины. И аналогично – если альтернативы нет, то снова возврат.

Если в результате получаем, что в стеке открытых вершин осталась

только целевая вершина, а потомков нет, то флаг нет решения сбрасываем в 0.

1.4 Алгоритм разметки. Построение дерева решения. Формирование закрытых правил

В разметке нужно подниматься снизу вверх. Последнее правило становится доказанным, мы его переносим в закрытые правила. В список закрытых вершин добавляем нашу подцель и удаляем их из стека открытых вершин и правил. Можно поставить флаг у этой вершины. Если выходная вершина целевая, то флаг fy = 0.

На следующем шаге в цикле мы должны вершину, которую мы доказали, найти в списке входных вершин правила из головы стека, флаг ее поставить в единицу и проверить, выполняется ли покрытие ее входов закрытыми вершинами. Если выполняется, то на следующий шаг цилка. Если покрытия нет – выход из цикла.

В разметке введем флаг, который означает что модуль доказан (флаг1).

```
1. Пока fy == 1 и флаг1 == 1:
2. Метку правила из головы стека ставим в доказанную
3. Флаг вершины подцели ставим в доказанную
4. Флаг выходной вершины модуля тоже в единицу
5. Если выходная вершина доказанного правила == цель, то fy = 0
6. Иначе:
7. Правило из головы стека открытых правил пишем в закрытые и
   удаляем из списка открытых
8. Вершину из списка открытых вершин перенести в закрытые, а
   потом удалить
9. В новом правиле из головы стека среди входных определить
   вершину, флаг ее поставить в единицу и проверить, все ли
   входные вершины доказаны.
10. Если не доказаны, то флаг1 = 0
```

2 Практический раздел

2.1 Базовые структуры вершины и правила

```
struct Node {
    int number;
    bool forbidden = false;
    bool closed = false;
};

struct Rule {
    std::vector<int> srcNodes;
    int dstNode;
    int number;
    bool visited = false;
    bool forbidden = false;
    unsigned int openIndex = 0;
};
```

2.2 Класс поиска в графе

```
class GraphSearch {
public:
    GraphSearch(std::list<Rule> rules, std::vector<int> srcNode,
       int dstNode);
    std::list<int> DoDepthFirstSearch();

private:
    int DescendantsDFS(int node);
    void Backtrack(int node);
    void Mark(int node);

    std::list<Rule> m_rules;
    std::map<int, Node> m_nodes;
    std::map<int, Rule *> m_rulesRef;
    std::list<int> m_openNodes;
    std::list<int> m_openRules;
    std::list<int> m_closedNodes;
    std::list<int> m_closedRules;
    std::vector<int> m_srcNodes;
    bool m_foundSolution = false;
```

```
    bool m_noSolution = false;
};
```

2.3 Реализация методов класса поиска

```
GraphSearch::GraphSearch(std::list<Rule> rules, std::vector<int>
   srcNodes, int dstNode) : m_rules(std::move(rules)),
   m_srcNodes(std::move(srcNodes)) {
  if (m_srcNodes.size() == 0)
    m_noSolution = true;
  else if (std::find(m_srcNodes.begin(), m_srcNodes.end(),
     dstNode) != m_srcNodes.end())
    m_foundSolution = true;
  else {
    for (auto node : m_srcNodes) {
      m_nodes[node].closed = true;
      m_closedNodes.push_back(node);
    }
    for (auto &rule : m_rules)
      m_rulesRef[rule.number] = &rule;
    m_openNodes.push_back(dstNode);
  }
}

std::list<int> GraphSearch::DoDepthFirstSearch() {
  while (!m_foundSolution && !m_noSolution) {
    int node = m_openNodes.back();
    int count = DescendantsDFS(node);
    if (m_foundSolution) {
      Mark(node);
      if (m_foundSolution) break;
    } else if (count == 0 && !m_openNodes.empty()) {
      Backtrack(node);
    } else if (m_openNodes.empty()) {
      m_noSolution = true;
      break;
    }
  }
  return m_noSolution ? {} : m_closedRules;
}
```

```
int GraphSearch::DescendantsDFS(int node) {
  int count = 0;
  for (auto &rule : m_rules) {
    if (rule.visited || rule.dstNode != node || rule.forbidden)
      continue;
    bool all_closed = true;
    for (auto srcNode : rule.srcNodes) {
      const Node &nodeRef = m_nodes[srcNode];
      if (nodeRef.forbidden) {
        rule.forbidden = true;
        break;
      }
      all_closed = all_closed && nodeRef.closed;
    }
    if (rule.forbidden) continue;
    m_openRules.push_back(rule.number);
    if (all_closed) {
      m_foundSolution = true;
      break;
    }
    m_rulesRef[rule.number]->openIndex = m_openNodes.size();
    for (auto srcNode : rule.srcNodes)
      if (!m_nodes[srcNode].closed)
        m_openNodes.push_back(srcNode);
    rule.visited = true;
    ++count;
  }
  return count;
}

void GraphSearch::Backtrack(int node) {
  while (true) {
    // node at top of open node stack needs to be forbidden
    m_nodes[node].forbidden = true;
    // then rule at top of open rules stack needs to be forbidden
    if (m_openRules.empty()) {
      m_noSolution = true;
      break;
    }
    int ruleNum = m_openRules.back();
    Rule *ruleRef = m_rulesRef[ruleNum];
```

```
    int dstNode = ruleRef->dstNode;
    ruleRef->forbidden = true;
    //  all input nodes of that rule must be removed from open
       nodes
    for (const auto &rule : m_rules) {
      if (rule.number == ruleNum) {
        while (ruleRef->openIndex < m_openNodes.size())
          m_openNodes.pop_back();
        break;
      }
    }
    m_openRules.pop_back();
    // new open rule must be searched to close previous node in
       open stack
    // 1. check if previous open rule resolves current top of
       open nodes
    // 2. if true - continue with this rule
    //    if not - remove this rule and corresponding nodes and
       retry
    if (m_openRules.empty())
      m_noSolution = true;
    else if (m_rulesRef[m_openRules.back()]->dstNode == dstNode)
      break;
    node = dstNode;
  }
}

void GraphSearch::Mark(int node) {
  // 0. mark given node as closed
  // 1. pop all rules that have same dst node
  // 2. if current top of opened rules has closed inputs - repeat
  // 3. dont forget to update m_foundSolution variable if real
     solution was found!
  while (m_foundSolution) {
    m_nodes[node].closed = true;
    m_closedNodes.push_back(node);
    m_closedRules.push_back(m_openRules.back());
    while (!m_openRules.empty() &&
       m_rulesRef[m_openRules.back()]->dstNode == node)
      m_openRules.pop_back();
    // real solution found
```

```
    if (m_openRules.empty())
      return;
    // remove all open nodes up until given node
    while (m_openNodes.back() != node)
      m_openNodes.pop_back();
    m_openNodes.pop_back();
    // check current top
    int rule = m_openRules.back();
    bool all_closed = true;
    for (auto srcNode : m_rulesRef[rule]->srcNodes) {
      if (!m_nodes[srcNode].closed) {
        all_closed = false;
        break;
      }
    }
    if (all_closed)
      node = m_rulesRef[rule]->dstNode;
    else m_foundSolution = false;
  }
}
```

2.4 Сборка ПО

Для сборки программы используется `CMake`. Стандартная последовательность команд для сборки и запуска программы:

```
mkdir build && cd build
cmake ..
# сборка и запуск программы
make app
./app
# опциональная сборка и запуск юнит-тестов
make unittests
./unittests
```

Альтернативно, можно собрать и запустить программу с помощью `makefile` скрипта в корневой папке:

```
make app
./app
```

2.5 Тестирование ПО

Для тестирования реализации алгоритма поиска была составлена база знаний, соответствующая графу, представленному на рисунке 2.1.

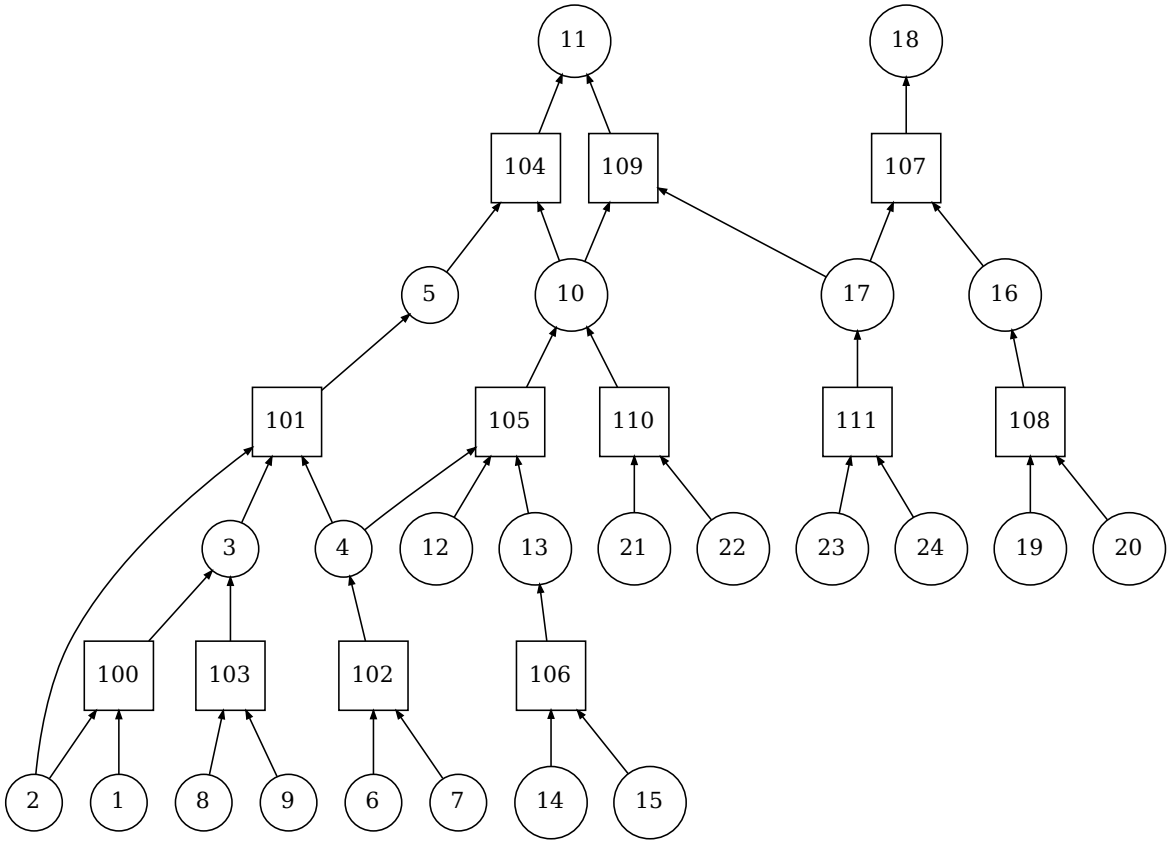

Рисунок 2.1 – Граф составленной базы знаний

Таблица 2.1 – Тестовые данные для проверки реализации алгоритма

Входные вершины	Выходная вершина	Ожидаемый результат	Фактический результат
$1, 2, 4, 8, 9$	5	$100, 101$	$100, 101$
$2, 3, 4, 12, 13, 21, 22, 23, 24$	11	$111, 105, 109$	$111, 105, 109$
$8, 9, 6, 7, 12$	10	$\varnothing$	$\varnothing$
$2, 3, 4, 12, 14, 15, 23, 24$	11	$111, 106, 105, 109$	$111, 106, 105, 109$

Выводы

В ходе лабораторной работы была разработана программа, осуществляющая обратный поиск в глубину в графе И-ИЛИ.