|  | **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 ИУ, Информатика и системы управления

КАФЕДРА ИУ7, Программное обеспечение ЭВМ и информационные технологии

**Отчет**

***По лабораторной работе №4:***

***Процедура резолюции с унификацией***

Студент ИУ7-32М **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ А. А. Андреев**

(Группа) (Подпись, дата) (И.О.Фамилия)

Преподаватель **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ З.Н. Русакова**

(Подпись, дата) (И.О.Фамилия)

*2024 г.*

**ЗАДАНИЕ**

Разработать программу, реализующую алгоритм решения методом резолюции в логике высказываний.

Цели:

* Ознакомиться с основными концепциями логики высказываний, включая атомы и дизъюнкты.
* Реализовать алгоритм метода резолюции для доказательства логической выводимости.
* Создать программу, которая может решать задачи на основе данного метода.

Программа должна включать следующие компоненты:

* Определение атомов:
  + Класс Atom, который будет хранить имя атома и его значение (истинность).
  + Функция для отрицания атома, проверки равенства двух атомов и вывода атома.
* Создание и работа с дизъюнктами:
  + Класс Clause, который содержит несколько Atom.
  + Функция для добавления атомов в дизъюнкт, поглощения (объединения) атомов и вывода содержимого дизъюнкта.
* Алгоритм резолюции:
  + Функция для выполнения базовой и полной резолюции (BasicResolution и FullResolution).
  + Алгоритмы должны использовать объединение дизъюнктов, чтобы выявить противоречия и доказать выводимость (вывод пустого дизъюнкта).

# Содержание

[Содержание 4](#_Toc181966597)

[ВВЕДЕНИЕ 5](#_Toc181966598)

[Структура программы 6](#_Toc181966599)

[Работа программы 7](#_Toc181966600)

[ПРИЛОЖЕНИЕ А 8](#_Toc181966601)

# ВВЕДЕНИЕ

Метод резолюции используется в автоматическом доказательстве теорем и программировании логики, чтобы доказать, что данное множество высказываний является логически посылаемым из других высказываний.

# Структура программы

Данная программа реализует алгоритм решения методом резолюции в логике высказываний. Метод резолюции используется в автоматическом доказательстве теорем и программировании логики, чтобы доказать, что данное множество высказываний является логически посылаемым из других высказываний.

Программа организована в нескольких файлах с различными обязанностями:

* atom.h и atom.cc: Определяют атомы - основные составляющие части высказываний. Атомы имеют имена и знак (истина или ложь). Реализованы операции, такие как отрицание атома, проверка равенства и вывод атома.
* clause.h и clause.cc: Представляют дизъюнкты, которые состоят из одного или нескольких атомов. Дизъюнкт рассматривается как логическое или между его атомами. Реализованы функции для поглощения атомов (объединение без дубликатов с учетом противоречий) и вывода дизъюнктов.
* resolution.h и resolution.cc: Здесь реализованы основные функции для разрешения, такие как FullResolution и BasicResolution. Эти функции применяют метод резолюции, чтобы доказать либо противоречие (пустой дизъюнкт), либо исчерпание возможностей поиска новых следствий.
* resolvent.h: Содержит логику для создания резольвенты, основываясь на двух противоположных парах, то есть замена (¬𝐴, 𝐵) и (𝐶,𝐴) на (𝐵, 𝐶).
* main.cc: Точка входа в программу. Инициализирует некоторые атомы и дизъюнкты, а затем вызывает метод полного перебора (FullResolution) с данными аксиомами и инвертированными целями, чтобы попытаться доказать истину.
* Файлы сборки (CMakeLists.txt, cmake\_install.cmake, Makefile): Скрипты и файлы CMake/Make обеспечивают автоматизированную сборку программы, помогая в управлении зависимостями и компиляции.

# Работа программы

Инициализация: В main.cc создаются атомы и на их основе определяются начальные дизъюнкты. Эти дизъюнкты служат аксиомами, которые будут использоваться для доказательства.

Приклад алгоритма резолюции:

* В методе FullResolution вводные аксиомы и инвертированная цель (необходимо доказать, что цель приводит к противоречию вместе с аксиомами) добавляются в "стек дизъюнктов".
* Происходит перебор всех пар дизъюнктов для создания резольвенты. Если среди них найдена пустая (контрарная пара), утверждение считается доказанным (т.е. истина доказана, так как нашлось противоречие).
  + Процесс перебора продолжается до достижения максимального количества итераций или пока не будет добавлено ни одного нового дизъюнкта.
  + Результат выводится пользователю, указывая, был ли найден доказательство или нет.
* Вывод результатов: После выполнения алгоритма, результат выражен в терминах ResolutionResult, что позволяет информировать пользователя о статусе доказательства: найдено противоречие, превышено число итераций и т.д.

# Тестирование программы

Для проведения тестирования программы возьмем следующие входные данным:

|  |
| --- |
| // Создаем несколько атомов с их именами и значениями истинности  const Atom a = {"A", true};  const Atom b = {"B", true};  const Atom c = {"C", true};  const Atom d = {"D", true};  const Atom not\_a = {"A", false};  const Atom not\_b = {"B", false};  const Atom not\_c = {"C", false};  const Atom not\_d = {"D", false}; |

Листинг 1 – Создание нескольких атомов с их именами и значеними истиности

|  |
| --- |
| // Создаем несколько дизъюнктов из атомов  const Clause ax1 = {a, b};  const Clause ax2 = {not\_a, c};  const Clause ax3 = {not\_b, d};  const Clause ax4 = {not\_c};  const Clause ax5 = {not\_d};  const Clause ax6 = {not\_a}; |

Листинг 2 – Создание нескольких дизъюнктов из атомов

|  |
| --- |
| // Собираем все аксиомы в вектор  const std::vector<Clause> axioms = {ax1, ax2, ax3, ax4, ax5}; |

Листинг 3 – Сбор все аксиомы в вектор

|  |
| --- |
| // Проводим полную резолюцию на базе аксиом (axioms) с противоречием ax6  // Указываем лимит итераций равный 1000  const auto res = FullResolution(axioms, ax6, 1000);  // Выводим результат резолюции на экран  std::cout << '\n' << ToString(res); |

Листинг 4 – Полная резолюция

|  |
| --- |
| [Полный перебор] Входные данные:  Аксиомы:  1. {A, B}  2. {¬A, C}  3. {¬B, D}  4. {¬C}  5. {¬D}  Цель: A  [Итерация 0] Множество дизъюнктов: {{A, B}, {¬A, C}, {¬B, D}, {¬C}, {¬D}, {¬A}}  Найдена резольвента дизъюнктов {A, B} и {¬A, C}: {B, C}  Найдена резольвента дизъюнктов {A, B} и {¬B, D}: {A, D}  Найдена резольвента дизъюнктов {A, B} и {¬A}: {B}  [Итерация 8] Множество дизъюнктов: {{A, B}, {¬A, C}, {¬B, D}, {¬C}, {¬D}, {¬A}, {B, C}, {A, D}, {B}}  Найдена резольвента дизъюнктов {¬A, C} и {A, B}: {C, B}  Найдена резольвента дизъюнктов {¬A, C} и {¬C}: {¬A} [игнор]  Найдена резольвента дизъюнктов {¬A, C} и {A, D}: {C, D}  [Итерация 18] Множество дизъюнктов: {{A, B}, {¬A, C}, {¬B, D}, {¬C}, {¬D}, {¬A}, {B, C}, {A, D}, {B}, {C, B}, {C, D}}  Найдена резольвента дизъюнктов {¬B, D} и {A, B}: {D, A}  Найдена резольвента дизъюнктов {¬B, D} и {¬D}: {¬B}  Найдена резольвента дизъюнктов {¬B, D} и {B, C}: {D, C}  Найдена резольвента дизъюнктов {¬B, D} и {B}: {D}  Найдена резольвента дизъюнктов {¬B, D} и {C, B}: {D, C} [игнор]  [Итерация 32] Множество дизъюнктов: {{A, B}, {¬A, C}, {¬B, D}, {¬C}, {¬D}, {¬A}, {B, C}, {A, D}, {B}, {C, B}, {C, D}, {D, A}, {¬B}, {D, C}, {D}}  Найдена резольвента дизъюнктов {¬C} и {¬A, C}: {¬A} [игнор]  Найдена резольвента дизъюнктов {¬C} и {B, C}: {B} [игнор]  Найдена резольвента дизъюнктов {¬C} и {C, B}: {B} [игнор]  Найдена резольвента дизъюнктов {¬C} и {C, D}: {D} [игнор]  Найдена резольвента дизъюнктов {¬C} и {D, C}: {D} [игнор]  [Итерация 46] Множество дизъюнктов: {{A, B}, {¬A, C}, {¬B, D}, {¬C}, {¬D}, {¬A}, {B, C}, {A, D}, {B}, {C, B}, {C, D}, {D, A}, {¬B}, {D, C}, {D}}  Найдена резольвента дизъюнктов {¬D} и {¬B, D}: {¬B} [игнор]  Найдена резольвента дизъюнктов {¬D} и {A, D}: {A}  Найдена резольвента дизъюнктов {¬D} и {C, D}: {C}  Найдена резольвента дизъюнктов {¬D} и {D, A}: {A} [игнор]  Найдена резольвента дизъюнктов {¬D} и {D, C}: {C} [игнор]  Найдена резольвента дизъюнктов {¬D} и {D}: {}  ИСТИНА  Доказано, найдена контрарная пара% |

Рисунок 5 – Вывод программы

# ПРИЛОЖЕНИЕ А

|  |
| --- |
| 1. #**pragma** once // Директива препроцессора для предотвращения множественного включения заголовочного файла 2. #**include** <iosfwd> // Включает предварительную декларацию потоковых классов ввода-вывода 3. #**include** <string> // Включает библиотеку для работы со строками 4. // Определение структуры Atom, которая используется для моделирования атомов с именем и знаком 5. **struct** **Atom** { 6. std::string name; // Имя атома 7. bool sign; // Знак атома: true для положительного, false для отрицательного 8. }; 9. // Функция, которая принимает структуру Atom и возвращает новую структуру Atom с противоположным знаком 10. Atom **Negate**(const Atom& atom); 11. // Переопределение оператора равенства для сравнения двух объектов типа Atom 12. bool **operator**==(const Atom& lhs, const Atom& rhs); 13. // Переопределение оператора вывода в поток для объектов типа Atom 14. std::ostream& **operator**<<(std::ostream& os, const Atom& atom); |

Листинг 6 – atom.h

|  |
| --- |
| 1. #**include** "atom.h" // Подключаем заголовочный файл, содержащий объявление структуры Atom 2. #**include** <ostream> // Подключаем заголовочный файл для работы с потоками вывода 3. // Функция Negate принимает Atom и возвращает новый Atom с обратным атомным знаком 4. Atom **Negate**(const Atom& atom) { 5. **return** {atom.name, !atom.sign}; // Создаем новый Atom с тем же именем, но переворачиваем знак 6. } 7. // Переопределяем оператор равенства для структур Atom 8. bool **operator**==(const Atom& lhs, const Atom& rhs) { 9. // Сравниваем название и знак двух атомов, возвращая true, если оба совпадают 10. **return** lhs.name == rhs.name && lhs.sign == rhs.sign; 11. } 12. // Переопределяем оператор вывода в поток для структур Atom 13. std::ostream& **operator**<<(std::ostream& os, const Atom& atom) { 14. **if** (!atom.sign) { // Если атом находится в отрицательной форме 15. os << "¬"; // Выводим символ отрицания 16. } 17. **return** os << atom.name; // Выводим имя атома 18. } |

Листинг 7 – atom.cc

|  |
| --- |
| 1. #**pragma** once 2. #**include** <iosfwd> 3. #**include** <vector> 4. #**include** "atom.h" 5. **class** **AbsorbedClause**; 6. /// Дизъюнкт. 7. **class** **Clause** : **public** std::vector<Atom> { 8. **public**: 9. **using** vector::vector; 10. **explicit** **Clause**(AbsorbedClause&& absorbed\_clause) **noexcept**; 11. /// Приведение дизъюнкта к виду, где атомы не повторяются, или к истине, если 12. /// внутри него есть пара типа: ~A, A. 13. /// 14. /// @returns дизъюнкт с неповторяющимися атомами или пустой дизъюнкт в случае 15. /// истины. 16. [[nodiscard]] AbsorbedClause **Absorb**() const; 17. }; 18. /// Дизъюнкт без повторов. 19. **class** **AbsorbedClause** : **public** std::vector<Atom> { 20. **public**: 21. **using** vector::vector; 22. }; 23. std::ostream& **operator**<<(std::ostream& os, const Clause& clause); 24. std::ostream& **operator**<<(std::ostream& os, const AbsorbedClause& clause); |

Листинг 8 – clause.h

|  |
| --- |
| 1. #**include** "clause.h" 2. #**include** <unordered\_map> // Для хранения информации о уже обработанных атомах 3. #**include** <utility> 4. #**include** <ostream> 5. #**include** <string> 6. #**include** <vector> 7. #**include** "atom.h" 8. #**include** "io\_join.h" 9. // Конструктор, который инициирует Clause из AbsorbedClause, перемещая его данные 10. Clause::Clause(AbsorbedClause&& absorbed\_clause) **noexcept** 11. : std::vector<Atom>(std::move(absorbed\_clause)) {} 12. // Метод, который возвращает дизъюнкт без дублирующихся атомов. 13. // Если обнаружены противоречащие атомы (одинаковое имя, разные знаки), возвращается пустой. 14. AbsorbedClause **Clause::Absorb**() const { 15. AbsorbedClause absorbed; // Дизъюнкт без дублей 16. std::unordered\_map<std::string, bool> processed\_atoms; // Отслеживание обработанных атомов 17. **for** (**auto**&& atom : \***this**) { 18. // Проверка на существование атома в processed\_atoms 19. **if** (**auto** it = processed\_atoms.find(atom.name); 20. it != processed\_atoms.end()) { 21. const **auto**& [name, sign] = \*it; 22. // Если знаки отличаются, значит имеется противоречие, возвращаем пустой дизъюнкт 23. **if** (atom.sign != sign) { 24. **return** {}; 25. } 26. // Если знаки совпадают, пропускаем добавление, так как атом уже учтен 27. **continue**; 28. } 29. // Атом добавляется в дизъюнкт, если его ещё не было 30. absorbed.push\_back(atom); 31. processed\_atoms.emplace(atom.name, atom.sign); 32. } 33. **return** absorbed; 34. } 35. // Переопределенный оператор вывода для Clause 36. std::ostream& **operator**<<(std::ostream& os, const Clause& clause) { 37. os << "{"; 38. join(clause, os); // Функция join используется для форматирования вывода 39. **return** os << "}"; 40. } 41. // Переопределенный оператор вывода для AbsorbedClause 42. std::ostream& **operator**<<(std::ostream& os, const AbsorbedClause& clause) { 43. // Приведение AbsorbedClause к базовому типу Clause для использования переопределенного оператора вывода 44. **return** os << reinterpret\_cast<const Clause&>(clause); 45. } |

Листинг 9 – clause.cc

|  |
| --- |
| 1. #**pragma** once 2. #**include** <experimental/iterator> 3. #**include** <ostream> 4. // Общая шаблонная функция для объединения элементов диапазона [begin, end) с использованием заданного разделителя 5. **template** <**typename** It> 6. void **join**(It begin, It end, std::ostream& os, const char\* delimiter = ", ") { 7. // std::copy копирует элементы из входного диапазона в выходной поток 8. // std::experimental::make\_ostream\_joiner позволяет вставлять элементы в поток вывода os с указанным разделителем 9. std::copy(begin, end, std::experimental::make\_ostream\_joiner(os, delimiter)); 10. } 11. // Перегрузка функции join для контейнеров, принимает контейнер с элементами для объединения 12. **template** <**typename** Container> 13. void **join**(const Container& container, 14. std::ostream& os, 15. const char\* delimiter = ", ") { 16. // Вызов первой версии функции join с итераторами контейнера. 17. // std::begin(container) и std::end(container) используются для получения итераторов начала и конца контейнера. 18. join(std::begin(container), std::end(container), os, delimiter); 19. } |

Листинг 10 – io\_join.h

|  |
| --- |
| 1. #**pragma** once 2. #**include** <utility> 3. #**include** <vector> 4. #**include** "clause.h" 5. /// Перечисление возможных результатов процесса резолюции. 6. **enum class** **ResolutionResult** { 7. kProvenByFoundingEmptyClause, // Доказано нахождением пустой клаузулы. 8. kMaxIterationsExceeded, // Превышено максимальное число итераций. 9. kNoProofFound, // Доказательство не найдено. 10. kNoNewClausesAdded, // Не добавлены новые клаузулы. 11. }; 12. /// Преобразует результат резолюции в строковое представление. 13. std::string\_view **ToString**(ResolutionResult result); 14. /// Полный перебор для поиска доказательства. 15. ResolutionResult **FullResolution**(const std::vector<Clause>& axioms, 16. const Clause& inverted\_target, 17. size\_t max\_iterations); 18. /// Опорное множество для поиска доказательства. 19. ResolutionResult **BasicResolution**(const std::vector<Clause>& axioms, 20. const Clause& inverted\_target, size\_t max\_iterations); |

Листинг 11 – resolution.h

|  |
| --- |
| 1. #**include** "resolution.h" 2. #**include** <cstddef> 3. #**include** <iostream> 4. #**include** <string\_view> 5. #**include** <utility> 6. #**include** <vector> 7. #**include** "atom.h" 8. #**include** "clause.h" 9. #**include** "io\_join.h" // Подключение заголовочного файла для функций объединения в потоке вывода 10. #**include** "resolvent.h" 11. **namespace** { 12. /// Функция добавляет элементы вектора `what` в конец вектора `to` 13. **template** <**typename** T> 14. void **Append**(std::vector<T>& to, const std::vector<T>& what) { 15. to.insert(to.end(), what.begin(), what.end()); 16. } 17. /// Функция конкатенации двух векторов 18. **template** <**typename** T> 19. std::vector<T> **Concatenate**(const std::vector<T>& first, 20. const std::vector<T>& second) { 21. std::vector<T> result; 22. result.reserve(first.size() + second.size()); // Предварительное выделение памяти для результирующего вектора 23. Append(result, first); 24. Append(result, second); 25. **return** result; 26. } 27. void **PrintInput**(const std::vector<Clause>& axioms, 28. const Clause& inverted\_target) { 29. std::cout << "Входные данные:\n"; 30. std::cout << "\tАксиомы:\n"; 31. **for** (size\_t i = 0; i < axioms.size(); ++i) { 32. std::cout << "\t\t" << i + 1 << ". " << axioms[i] << '\n'; // Печать списка аксиом 33. } 34. std::cout << "\tЦель: " << Negate(inverted\_target.front()) << '\n'; // Печать инверсной цели 35. } 36. } // namespace 37. std::string\_view **ToString**(ResolutionResult result) { 38. **switch** (result) { 39. **case** ResolutionResult::kProvenByFoundingEmptyClause: 40. **return** "ИСТИНА\nДоказано, найдена контрарная пара"; 41. **case** ResolutionResult::kMaxIterationsExceeded: 42. **return** "ЛОЖЬ\nВыход по числу итераций"; 43. **case** ResolutionResult::kNoProofFound: 44. **return** "ЛОЖЬ\nНе найдено доказательство"; 45. **case** ResolutionResult::kNoNewClausesAdded: 46. **return** "ЛОЖЬ\nНовых дизъюнктов не добавилось"; 47. } 48. } 49. // Основная функция разрешения, использующая полный перебор 50. ResolutionResult **FullResolution**(const std::vector<Clause>& axioms, 51. const Clause& inverted\_target, 52. size\_t max\_iterations) { 53. std::cout << "[Полный перебор] "; 54. PrintInput(axioms, inverted\_target); 55. **auto** clause\_stack = Concatenate(axioms, {inverted\_target}); // Инициализация стека дизъюнктов 56. **for** (size\_t iteration = 0, i = 0; i < clause\_stack.size() - 1; 57. ++i) { // Перебор всех дизъюнктов 58. std::cout << "[Итерация " << iteration << "] Множество дизъюнктов: {"; 59. join(clause\_stack, std::cout); 60. std::cout << "}\n"; 61. **for** (size\_t j = 0; j < clause\_stack.size(); ++j) { // Перебор пар дизъюнктов 62. **if** (i == j) { 63. **continue**; // Пропускаем пары с одинаковыми индексами 64. } 65. ++iteration; 66. **if** (iteration > max\_iterations) { 67. **return** ResolutionResult::kMaxIterationsExceeded; // Проверка на превышение максимального числа итераций 68. } 69. const **auto**& clause1 = clause\_stack[i]; 70. const **auto**& clause2 = clause\_stack[j]; 71. // Попытка создать резольвенту 72. **auto** [resolvent, res] = CreateResolvent(clause1, clause2); 73. **if** (res == ResolventResult::kEmptyClause) { // Если получен пустой дизъюнкт, 74. std::cout << "Найдена резольвента дизъюнктов " << clause1 << " и " 75. << clause2 << ": {}\n"; // Вывод сообщения о найденной резольвенте 76. **return** ResolutionResult::kProvenByFoundingEmptyClause; // Возвращаем результат в случае успеха 77. } 78. **if** (res == ResolventResult::kOk) { // Если резольвента ненулевая, 79. std::cout << "Найдена резольвента дизъюнктов " << clause1 << " и " 80. << clause2 << ": " << resolvent; 81. **auto** ignore = false; 82. **for** (const **auto**& clause : clause\_stack) { 83. **if** (clause == resolvent) { 84. std::cout << " [игнор]"; // Если резольвента уже существует, игнорируем её 85. ignore = true; 86. **break**; 87. } 88. } 89. **if** (!ignore) { 90. clause\_stack.push\_back(resolvent); // Добавляем новую резольвенту в стек 91. } 92. std::cout << '\n'; 93. } 94. } 95. } 96. **return** ResolutionResult::kNoProofFound; // Возврат результата, если доказательство не найдено 97. } 98. // Осуществляется опорное множество 99. ResolutionResult **BasicResolution**(const std::vector<Clause>& axioms, 100. const Clause& inverted\_target, 101. size\_t max\_iterations) { 102. std::cout << "[Опорное множество] "; 103. PrintInput(axioms, inverted\_target); 104. std::vector<Clause> s2{inverted\_target}; // Инициализация вектора s2 с инверсной целью 105. **for** (size\_t iteration = 0; /\* nothing \*/; /\* nothing \*/) { 106. std::vector<Clause> clauses\_to\_append; 107. std::cout << "[Итерация " << iteration << "] Множество дизъюнктов: {"; 108. join(s2, std::cout); 109. std::cout << "}\n"; 110. **for** (const **auto**& clause2 : s2) { 111. **for** (const **auto**& clause1 : axioms) { 112. // Основная логика, подобная методу полного перебора 113. ++iteration; 114. **if** (iteration > max\_iterations) { 115. **return** ResolutionResult::kMaxIterationsExceeded; 116. } 117. **auto** [resolvent, res] = CreateResolvent(clause1, clause2); 118. **if** (res == ResolventResult::kEmptyClause) { 119. std::cout << "Найдена резольвента дизъюнктов " << clause1 << " и " 120. << clause2 << ": {}\n"; 121. **return** ResolutionResult::kProvenByFoundingEmptyClause; 122. } 123. **if** (res == ResolventResult::kOk) { 124. std::cout << "Найдена резольвента дизъюнктов " << clause1 << " и " << clause2 << ": " << resolvent; 125. **auto** ignore = false; 126. **for** (const **auto**& clause : Concatenate(s2, clauses\_to\_append)) { 127. **if** (clause == resolvent) { 128. std::cout << " [игнор]"; 129. ignore = true; 130. **break**; 131. } 132. } 133. **if** (!ignore) { 134. clauses\_to\_append.push\_back(std::move(resolvent)); // Добавляем резольвенту в список для включения 135. } 136. std::cout << "\n"; 137. } 138. } 139. } 140. **if** (clauses\_to\_append.empty()) { 141. **return** ResolutionResult::kNoNewClausesAdded; // Если новые дизъюнкты не были добавлены 142. } 143. Append(s2, clauses\_to\_append); // Добавляем новые дизъюнкты в s2 144. } 145. } |

Листинг 12 – resolution.h

|  |
| --- |
| 1. #**pragma** once 2. #**include** <utility> 3. #**include** <vector> 4. #**include** "clause.h" 5. /// Перечисление возможных результатов процесса резолюции. 6. **enum class** **ResolutionResult** { 7. kProvenByFoundingEmptyClause, // Доказано нахождением пустой клаузулы. 8. kMaxIterationsExceeded, // Превышено максимальное число итераций. 9. kNoProofFound, // Доказательство не найдено. 10. kNoNewClausesAdded, // Не добавлены новые клаузулы. 11. }; 12. /// Преобразует результат резолюции в строковое представление. 13. std::string\_view **ToString**(ResolutionResult result); 14. /// Полный перебор для поиска доказательства. 15. ResolutionResult **FullResolution**(const std::vector<Clause>& axioms, 16. const Clause& inverted\_target, 17. size\_t max\_iterations); 18. /// Опорное множество для поиска доказательства. 19. ResolutionResult **BasicResolution**(const std::vector<Clause>& axioms, 20. const Clause& inverted\_target, size\_t max\_iterations); |

Листинг 13 – resolvent.h

|  |
| --- |
| 1. #**include** "resolvent.h" 2. #**include** <string> 3. #**include** <unordered\_map> 4. #**include** <utility> 5. #**include** "clause.h" 6. **namespace** { 7. // Создает быстрый поиск для данной клаузулы, сопоставляя имена атамов и их знаки. 8. std::unordered\_map<std::string, bool> **MakeClauseFastLookup**( 9. const AbsorbedClause& clause) { 10. std::unordered\_map<std::string, bool> clause\_fast\_lookup; 11. **for** (**auto**&& atom : clause) { 12. clause\_fast\_lookup.emplace(atom.name, atom.sign); 13. } 14. **return** clause\_fast\_lookup; 15. } 16. /// Вспомогательная функция для CreateResolvent. 17. /// Ищет имя атома из первого дизъюнкта, для которого существует атом, 18. /// противоположный по знаку во втором дизъюнкте. 19. std::string **FindOppositeAtomsName**(const AbsorbedClause& first\_clause, 20. const AbsorbedClause& second\_clause) { 21. const **auto** second\_clause\_fast\_lookup = MakeClauseFastLookup(second\_clause); 22. **for** (**auto**&& atom : first\_clause) { 23. **if** (**auto** it = second\_clause\_fast\_lookup.find(atom.name); 24. it != second\_clause\_fast\_lookup.end()) { 25. const **auto**& [name, sign] = \*it; 26. **if** (atom.sign != sign) { 27. **return** name; 28. } 29. } 30. } 31. **return** {}; 32. } 33. } // namespace 34. // Создает резольвент двух дизъюнктов и возвращает его результатом. 35. std::pair<Clause, ResolventResult> **CreateResolvent**(const Clause& first, 36. const Clause& second) { 37. **auto** first\_absorbed\_clause = 38. first.Absorb(); // Приводим первый дизъюнкт к виду, где атомы не повторяются. 39. **auto** second\_absorbed\_clause = 40. second.Absorb(); // То же самое для второго дизъюнкта. 41. // Проверяем, если один из дизъюнктов пустой. 42. **if** (first\_absorbed\_clause.empty() || second\_absorbed\_clause.empty()) { 43. **return** {{}, ResolventResult::kEmptyClause}; 44. } 45. // Ищем противоположные атомы 46. **auto** mb\_opposite\_atoms\_name = 47. FindOppositeAtomsName(first\_absorbed\_clause, second\_absorbed\_clause); 48. **if** (mb\_opposite\_atoms\_name.empty()) { 49. // Противоположные пары не найдены 50. **return** {{}, ResolventResult::kOppositePairNotFound}; 51. } 52. // Если найдена противоположная пара атомов 53. // Заполняем резольвент, исключая противоположные атомы 54. Clause resolvent; 55. const **auto** append\_resolvent\_without\_opposite\_atom = 56. [&mb\_opposite\_atoms\_name, &resolvent](AbsorbedClause& absorbed\_clause) { 57. **for** (**auto**&& atom : absorbed\_clause) { 58. **if** (atom.name != mb\_opposite\_atoms\_name) { 59. resolvent.push\_back(std::move(atom)); 60. } 61. } 62. }; 63. append\_resolvent\_without\_opposite\_atom(first\_absorbed\_clause); 64. append\_resolvent\_without\_opposite\_atom(second\_absorbed\_clause); 65. **auto** absorbed\_resolvent = resolvent.Absorb(); // Поглощаем резольвент. 66. const **auto** result = absorbed\_resolvent.empty() ? ResolventResult::kEmptyClause : ResolventResult::kOk; 67. **return** {Clause{std::move(absorbed\_resolvent)}, result}; // Возвращаем окончательный резольвент и результат. 68. } |

Листинг 14 – resolvent.cc