1 (Алтана, DONE). Что такое булева формула? Приведите пример.

Ответ 1:

Определение из лекции 1

Пусть дано множество переменных V, скобки и логические связки $\vee$, $\wedge$, $\longrightarrow$, $\neg$. Определим булеву формулу по индукции:

- Все переменные из V – формулы
- если A – формула, то $\neg(A)$ – формула
- если A, B - формулы, то $(A) \vee (B)$, $(A) \wedge (B)$, $(A) \longrightarrow (B)$ – также формулы

2 (Каринэ, DONE). Что такое конъюнктивная нормальная форма(КНФ) булевой формулы?

Ответ 2:

Определение из лекции 1

Конъюнктивная нормальная форма (КНФ) – форма записи булевой формулы, при которой эта формула имеет вид конъюнкции дизъюнкций литералов.

С.В. Яблонский. Введение в дискретную математику.

Пусть $\sigma-$параметр, равный либо 0 либо 1. Тогда, обозначим:

$$x^{\sigma} = \begin{cases} \overline{x} & \text{при } \sigma = 0, \\ x & \text{при } \sigma = 1 \end{cases}$$

Разложение вида:

$$f(x_1, \ldots, x_n) = \bigwedge_{\substack{(\sigma_1,\ldots,\sigma_n) \\ f(\sigma_1,\ldots,\sigma_n)=0}} (x_1^{\overline{\sigma_1}} \vee \cdots \vee x_n^{\overline{\sigma_n}})$$

называется ***совершенной конъюнктивной нормальной формой (СКНФ)***.

3 (Паша, DONE). В чём заключается SAT задача?

Ответ 3:

SAT задача:

- На вход подается булева формула, содержащая только нумерованные переменные, $\vee, \wedge, \neg$.
- *Является ли формула выполнимой?*

SAT задача (некоторые критерии):

- Если дизъюнкты в формуле имеет длину не более 2, то сложность задачи распознавания принадлежит классу P.
- Иначе, это NP-полная задача (теорема Кука-Левина). Более того, исторически, это первая задача, чья NP-полнота была доказана.

следствие: Если мы научимся решать NP-полную задачу 'быстро', то мы сможем 'быстро' решать весь класс NP задач.

4 (Саит, DONE). Запишите в КНФ формулу $x_1 + \cdots + x_n \leqslant 2$.

Ответ 4:

На основе лекции 2.
Тут нужно реализовать $AtMostTwo$.
Общая формула для $AtMostK$ выглядит так:

$$AtMostK = \bigwedge_{1 \leqslant i_1 < i_2 < \cdots < i_{k+1} \leqslant n} (\overline{x_{i_1}} \vee \overline{x_{i_2}} \vee \ldots \vee \overline{x_{i_{k+1}}})$$

Поскольку такая формула требует, чтобы среди любых $k+1$ переменных хотя бы одна была ложна, то есть среди переменных $x_1, \ldots, x_n$ не может быть k истинных.

Тогда формула $AtMostTwo$ для $x_1, \ldots, x_n$ будет выглядеть так:

$$AtMostTwo = \bigwedge_{1 \leqslant i < j < k \leqslant n} (\overline{x_i} \vee \overline{x_j} \vee \overline{x_k})$$

Ответ: $\bigwedge\limits_{1 \leqslant i < j < k \leqslant n} (\overline{x_i} \vee \overline{x_j} \vee \overline{x_k})$.

5 (Алтана, DONE). Дана формула, примените к ней преобразование Цейтина.

Ответ 5:

Пример из лекции 2
В худшем случае преобразование булевой формулы в КНФ занимает 2^n. Преобразование Цейтина позволяет перевести булеву формулу в вид КНФ, которую можно подать на вход SAT-решателю, за меньшее количество операций.

Пример: дана формула $P \to (Q \wedge R)$.

1. Введем переменные для всех неатомарных подформул(в моем понимании неатомарные формулы – это формулы вида: $A \circ B$, где $\circ$ - знак какой-то булевой операции, а A и B - булевы переменные).

   То есть сначала вводим $T_2 : T_2 \leftrightarrow (Q \wedge R)$.

   Затем вводим $T_1 : T_1 \leftrightarrow P \to T_2$.

2. Каждую преобразуем в КНФ(берем полученные формулы из первого пункта и раскрываем до вида КНФ):

   $F_1 = T_1 \leftrightarrow P \to T_2 =$
   $= (T_1 \to (P \to T_2)) \wedge ((P \to T_2) \to T_1) =$
   $= (\neg T_1 \vee \neg P \vee T_2) \wedge (\neg(\neg P \vee T_2) \vee T_1) =$
   $= (\neg T_1 \vee \neg P \vee T_2) \wedge (P \wedge \neg T_2 \vee T_1) =$
   $= (\neg T_1 \vee \neg P \vee T_2) \wedge (P \vee T_1) \wedge (\neg T_2 \vee T_1)$

   $F_2 = T_2 \leftrightarrow (Q \wedge R) =$
   $= (T_2 \to (Q \wedge R)) \wedge ((Q \wedge R) \to T_2) =$
   $= (\neg T_2 \vee (Q \wedge R)) \wedge (\neg(Q \wedge R) \vee T_2) =$
   $= (\neg T_2 \vee Q) \wedge (\neg T_2 \vee R) \wedge (\neg Q \vee \neg R \vee T_2)$

3. Результирующей КНФ будет КНФ из конъюнкций переменной T_1, которая обозначает всю исходную формулу, и полученных во 2 пункте формул F_1 и F_2:

   $CNF = T_1 \wedge F_1 \wedge F_2$

6 (Каринэ, DONE). Что такое теория первого порядка? Приведите пример теории первого порядка.

Ответ 6:

По лекции 3

Теория первого порядка.

- **Сигнатура** $\Sigma = (R, F, C, \rho)$.

  $\boldsymbol{R}$ - множество **символов для отношений** (предикатов). Пример: $R = \{=\}$

  $\boldsymbol{F}$ - множество **функциональных символов**. Пример: $F = \{+\}$

  $\boldsymbol{C}$ - множество **констант**. Пример: $C = \{1, 2, \dots\}$

  ρ - **функция**, сопоставляющая элементам из R и F их арность *(например, у „+“ и „=“ арность = 2)*

- Переменные. Пример: $V = \{x_1, \dots, x_m\}$

- Логические операции: $\wedge, \vee, \to, \neg$

- Кванторы: $\forall, \exists$

- Скобки и запятые

- **Терм** - константы, переменные или $f(t_1, \dots, t_{\rho(f)})$, где f - символ арности $\rho(f)$, t_i—термы.

  Примеры термов: x_1, x_{11}, $x_2 + x_4$, $x_1 + 4$, $x_1 + (x_2 + x_4)$ - это всё термы.

- **Атом** - $p = (t_1, \dots, t_{\rho(p)})$ - предикатный символ (из R) арности $\rho(p)$, t_i - термы. (По простому: в предикатный символ вставляем термы).

  Примеры атомов: $x_1 = x_{11}$, $x_1 + (x_2 + x_4) = 10$

- **Формула** - либо атом, либо $\neg f_1$, $f_1 \vee f_2$, $f_1 \wedge f_2$, $f_1 \to f_2$, $\forall x f_1$, $\exists x f_2$, где f_1 и f_2 - формулы.

  Примеры формул: $(x_1 = x_{11}) \to (x_1 + (x_2 + x_4) = 10)$

Decision procedures, стр 15. Но там просто текст, неструктурированно написано.

Задать модель:

- Задать множество D - домен и функцию σ, которая каждому предикатному и функциональному символам сопоставляет предикат или функцию, а каждой константе сопоставляет элемент из множества D.
  Следующий абзац для понимания. Можете писать или не писать)

  К примеру, символ „+“ можно интерпретировать как функцию умножения, а символ „=“ понимать как не равно $\neq$. D - какое-то множество, например множество натуральных чисел или целых чисел или отрицательных/положительных целых чисел и т.д.

- Задать интерпретацию. Пусть s - функция, которая сопоставляет каждой переменной некоторое значение из D.

- Интерпретация формул относительно функции s - это индуктивное вычисление формул.

  Пояснение:
  Например, есть какая-то формула $(x_1 = x_{11}) \to (x_1 + (x_2 + x_4) = 10)$. Каждой переменной сопоставляем какое-то число из домена согласно функции s, потом по индукции все вычисляем. Интерпретация - посчитать значение формул на каком-то наборе.

  Пример теории первого порядка - **Арифметика Пресбургера** с $\Sigma = \{0, 1, +, =\}$.

7 (Паша, DONE). В чём заключается SMT задача?

Ответ 7:

Satisfiability Modulo Theories (SMT) - это задача определения выполнимости математической формулы. Она обобщает SAT задачу на более сложные формулы, включающие действительные числа, целые числа и/или различные структуры данных: списки, массивы, битовые векторы и строки. Название происходит от того факта, что эти выражения интерпретируются в рамках ("по модулю Modulo") определенной формальной теории в логике первого порядка с равенством (часто запрещающим кванторы)

8 (Саит, DONE). Что такое блокирующий дизъюнкт?

Ответ 8:

Есть в лекции 4, но там отсутствует звук. Также есть на 62 странице "Decision Procedures"

Блокирующий дизъюнкт t – это дизъюнкция отрицаний атомов, соответствующих данной теории $\overline{Th(\alpha)}$ (α – некоторая оценка формулы), при которой алгоритм $Deduction$ возвратил $UNSAT$. Этот дизъюнкт добавляется к изначальной формуле $B = B \wedge t$, тем самым *"блокируя"* данную оценку при следующих итерациях ленивого алгоритма решения SMT-задачи.

9 (Алтана, DONE). Опишите ленивый алгоритм решения SMT-задачи.

Ответ 9:

Дано:

- $at(\phi)$ - множество атомов в формуле ϕ
- $at_i(\phi)$ - определенный атом из этого множества.
- Атом a сопоставляем с булевой переменной $e(a)$ (такая процедура называется булевским кодированием).
- $e(t)$ - булева формула, полученная булевым кодированием каждого атома формулы t.
- α - некоторая(возможно, частичная) оценка формулы $e(\phi)$
- $$Th(at_i, \alpha) = \begin{cases} at_i & \alpha(at_i) = TRUE \\ \neg at_i & \alpha(at_i) = FALSE \end{cases}$$
- $Th(\alpha) = Th(at_i, \alpha | e(at_i) \in \alpha)$
- $\hat{Th}(\alpha)$ - конъюнкция всех элементов из $Th(\alpha)$

Пример: формула $\phi := x = y \vee y = z$

1. Атомы: $at_1 = (x = y)$, $at_2 = (y = z)$, $at_3 = (z = w)$
2. Проводим булевское кодирование: $e(x = y) = b_1$, $e(y = z) = b_2$, $e(\phi) = b_1 \vee b_2$
3. Пусть $\alpha := \{e(at_1) \mapsto FALSE, e(at_2) \mapsto TRUE\}$
4. $Th(at_1, \alpha) := \neg(x = y)$, $Th(at_2, \alpha) := (y = z)$
5. $Th(\alpha) := \{\neg(x = y), (y = z)\}$
6. $\hat{Th}(\alpha) := \neg(x = y) \wedge (y = z)$

Пусть нам дан алгоритм Deduction, который может решить $\hat{Th}(\alpha)$. **Ленивый алгоритм** решения таких задач заключается в следующем:

Algorithm 3.3.1: LAZY-BASIC

Input: A formula φ
Output: "Satisfiable" if φ is satisfiable, and "Unsatisfiable" otherwise

```
1. function LAZY-BASIC(φ)
2.     B := e(φ);
3.     while (TRUE) do
4.         ⟨α, res⟩ := SAT-SOLVER(B);
5.         if res ="Unsatisfiable" then return "Unsatisfiable";
6.         else
7.             ⟨t, res⟩ := DEDUCTION(T̂h(α));
8.             if res ="Satisfiable" then return "Satisfiable";
9.             B := B ∧ e(t);
```

2. Сначала вычислим $B = e(\phi)$
4. Отправляем результат вычисления B SAT-решателю.
5. Если получили "UNSAT то возвращаем "UNSAT".
7. Иначе вычисляем $Deduction(\hat{Th}(\alpha))$
8. Если Deduction вернул "SAT то возвращаем "SAT"
9. Если Deduction вернул "UNSAT то добавляем в B блокирующий дизъюнкт, который позволяет больше не учитывать данное решение(это решение нам не подходит и блокирующий дизъюнкт гарантирует, что мы больше его не получим).

10 (Каринэ, DONE). Что такое правило единичного дизъюнкта?

Ответ 10:

Взято из https://www.cs.princeton.edu/~zkincaid/courses/fall18/readings/SATHandbook-CDCL.pdf, стр. 129 и видео 6 лекции

Единичный дизъюнкт – это дизъюнкт, в котором все литералы кроме одного равны 0, а оставшемуся литералу значение не присвоено.
Правило единичного дизъюнкта – если дизъюнкт единичный, то единственному неопределенному литералу нужно присвоить значение 1, чтобы дизъюнкт стал выполнимым.

Пример:
Пусть дано частичное определение

$$\{x_1 \mapsto 1, x_2 \mapsto 0, x_4 \mapsto 1\}$$

Тогда
$(x_1 \vee x_3 \vee \neg x_4)$ - выполнимый дизъюнкт
$(\neg x_1 \vee x_2)$ - конфликтный дизъюнкт
$(\neg x_1 \vee \neg x_4 \vee x_3)$ - **единичный дизъюнкт**
$(\neg x_1 \vee x_3 \vee x_5)$ - неразрешенный дизъюнкт

11 (Паша, DONE). Опишите $CDCL$ алгоритм.

Ответ 11:

Conflict-driven clause learning

```
1. function CDCL
2.     while (TRUE) do
3.         while (BCP() = "conflict") do
4.             backtrack-level := ANALYZE-CONFLICT();
5.             if backtrack-level < 0 then return "Unsatisfiable";
6.             BackTrack(backtrack-level);
7.         if ¬DECIDE() then return "Satisfiable";
```

BCP() - (BackTrackingPropagation) оценивает все переменные из единичных дизъюнктов, если конфликт произошел, то будет произведена попытка отката

Decide() - подбрасывает монетку и делает предположение о какой-то переменной (если всё оценено, то возвращает *True*)

Analyze-Conflict() - строит блокирующий дизъюнкт, добавляет его к исходной формуле, и смотрит на какой уровень нужно откатиться

BackTrack() - откатывается на нужный уровень

12 (Саит, DONE). Что такое граф следствий?

Ответ 12:

На основе "Лекции 6. Оптимизации CDCL". И Decision Procedures, страница 33.

Граф следствий – это ациклический, ориентированный граф $G(V, E)$ с пометками на ребрах, такой, что:

- Вершины графа - переменные, определенные частичной оценкой.
- Будем обозначать вершины $x_i@dl$, если на уровне dl мы присвоили переменной x_i значение истина и $\neg x_i@dl$ - если присвоили ложь.
- Из вершины v_i идет ребро в вершину v_j , если переменная v_j оценена на основе дизъюнкта c и v_i входит в c. Это ребро помечается меткой c.
- Если есть конфликт, то ему соответствует вершина **K**. Пусть c - конфликтный дизъюнкт. Тогда к вершине **K** идут ребра от переменных, входящих в c и они помечаются меткой c.

Граф следствий служит для того, чтобы проиллюстрировать процесс распространения единиц *(Boolean constraint propagation, BCP())* в $CDCL$ алгоритме.

На рисунке 1 представлен пример графа следствий:

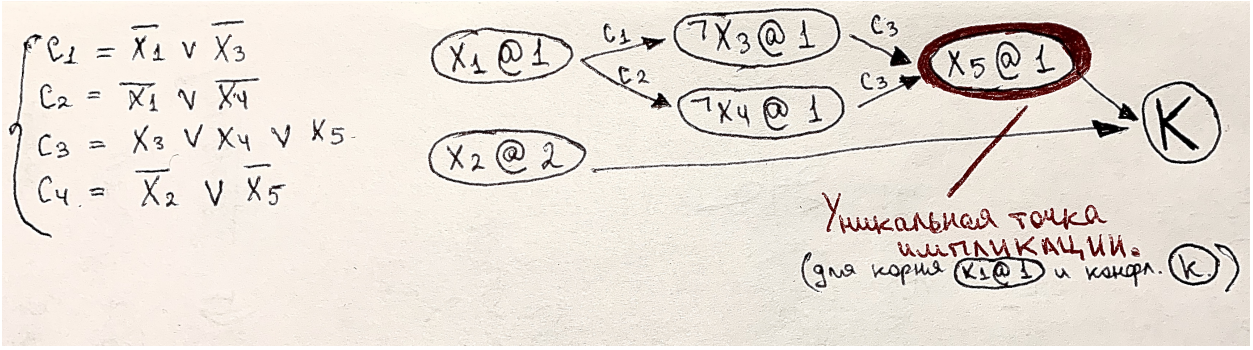


Рис. 1: Граф следствий. Красным цветом помечена уникальная точка импликации для корня x_1@1 и указанной вершины *"конфликт"*.

13 (Алтана, DONE). Что такое уникальная точка импликаций?

Ответ 13:

На основе "Лекции 6. Оптимизации CDCL"

Уникальная точка импликации – любая вершина графа следствий (импликаций), которая не является вершиной *«конфликт»* и которая находится на каждом пути, ведущему от конкретной корневой вершины к конкретной вершине *«конфликт»*.

На рисунке 1 представлен пример графа следствий и уникальной точки импликации.

14 (Каринэ, DONE). Опишите DPLL(T) алгоритм.

Ответ 14:

Лекция 7, Decision Procedures, страница 67.

Основная идея неформально:

Предположим, получили частичную оценку формулы. Теперь нужно проверить, непротиворечива ли она. Основная идея состоит в том, что в Lazy-CDCL Deduction вынесена из if и производится всегда, когда отрабатывает внутренний while и получается какая-то оценка. Если получилась невыполнимая формула, то делается back tracking и формула добавляется в блокирующий класс. (Получился алгоритм DPLL(T)). Это может увеличить время работы алгоритма. Для ускорения можно применять эвристики - вызывать Deduction не всегда, а если оценено определенное количество переменных. Этот солвер может искать минимально противоречивое ядро - количество атомов, которые нужно „запретить“

Более формальное описание:
Ленивый CDCL:

Algorithm 3.3.2: LAZY-CDCL

Input: A formula φ
Output: "Satisfiable" if the formula is satisfiable, and "Unsatisfiable" otherwise

```
1.  function LAZY-CDCL
2.      ADDCLAUSES(cnf(e(φ)));
3.      while (TRUE) do
4.          while (BCP() = "conflict") do
5.              backtrack-level := ANALYZE-CONFLICT();
6.              if backtrack-level < 0 then return "Unsatisfiable";
7.              else BackTrack(backtrack-level);
8.          if ¬DECIDE() then                          ▷ Full assignment
9.              ⟨t, res⟩:=DEDUCTION(T̂h(α));            ▷ α is the assignment
10.             if res="Satisfiable" then return "Satisfiable";
11.             ADDCLAUSES(e(t));
```

Ленивый CDCL алгоритм может быть улучшен вызовом Deduction перед булевым кодированием. Такой ранний вызов служит двум целям:

- Противоречивые частичные оценки исключаются на ранней стадии
- Значения литералов, которые еще не были оценены могут быть переданы обратно в SAT солвер, т.е. информация сохранена и может быть использована дальше.

Таким образом, мы приходим к алгоритму **DPLL(T)**

Дано:

- $at(\phi)$ - множество атомов в формуле ϕ
- $at_i(\phi)$ - определенный атом из этого множества.
- Атом a сопоставляем с булевой переменной $e(a)$ (такая процедура называется булевским кодированием).
- $e(t)$ - булева формула, полученная булевым кодированием каждого атома формулы t.
- α - некоторая(возможно, частичная) оценка формулы $e(\phi)$
- $$Th(at_i, \alpha) = \begin{cases} at_i & \alpha(at_i) = TRUE \\ \neg at_i & \alpha(at_i) = FALSE \end{cases}$$
- $Th(\alpha) = Th(at_i, \alpha | e(at_i), \alpha)$

- $\hat{Th}(\alpha)$ - конъюнкция всех элементов из $Th(\alpha)$

Algorithm 3.4.1: DPLL(T)

Input: A formula φ
Output: "Satisfiable" if the formula is satisfiable, and "Unsatisfiable" otherwise

```
1.  function DPLL(T)
2.      AddClauses(cnf(e(φ)));
3.      while (TRUE) do
4.          repeat
5.              while (BCP() = "conflict") do
6.                  backtrack-level := Analyze-Conflict();
7.                  if backtrack-level < 0 then return "Unsatisfiable";
8.                  else BackTrack(backtrack-level);
9.              ⟨t, res⟩:=Deduction(T̂h(α));
10.             AddClauses(e(t));
11.         until t ≡ TRUE;
12.         if α is a full assignment then return "Satisfiable";
13.         Decide();
```

Здесь Deduction вызывается в 9 строке, когда с помощью BCP уже не будет сделано никаких оценок. Далее находятся T-оцененные литералы и передаются в CDCL часть солвера в форме ограничений. Поэтому, в дополнение к следствиям в булевом домене, здесь также возникают следствия, связанные с теорией T. Соответственно, данная техника называется **распространение теорий.**

Требования к добавляемым дизъюнктам: они должны следовать из ϕ и ограничены конечным множеством атомов. Желательно, чтобы в случае неразрешимости $\hat{Th}(\alpha)$, $e(t)$ блокировала α; это не является обязательным, потому что блокировка или не блокировка α не влияет на корректность - Deduction завершается только когда α - полная оценка. В случае, когда $\hat{Th}(\alpha)$ выполнима, требуется чтобы t удовлетворяло одному из двух следующих условий для того, чтобы гарантировать остановку работы алгоритма:

- Добавление $e(t)$ в $\boldsymbol{\mathcal{B}}$ и вызов BCP ведет к оценке некоторого литерала.
- Когда Deduction не может найти дизъюнкт, удовлетворяющий первому условию, t и $e(t)$ эквивалентны TRUE.

Второй случай возникает, например, когда булевы переменные уже оценены, и, следовательно, формула является выполнимой. В таком случае выполняется условие в строке 11 и процедура продолжается с 13 строки, где снова вызывается Decide. Поскольку все переменные уже оценены, процедура возвращает „Satisfiable“.

Пример:

Рассмотрим случай двух булевых формул $e(x_1 \geqslant 10)$ и $e(x_1 < 0)$. После того, как первая формула была установлена в TRUE, процедура Deduction делает вывод, что формула

$$t := \neg(x_1 \geqslant 10) \vee \neg(x_1 < 0)$$

является T-допустимой. Тогда соответствующая t булева формула

$$e(t) := (\neg e(x_1 \geqslant 10) \vee \neg e(x_1 < 0))$$

добавляется в $\boldsymbol{\mathcal{B}}$, что приводит к незамедлительному следствию $\neg e(x_1 < 0)$ и, возможно, дальнейшим следствиям.

15 (Паша). Опишите разрешающую процедуру для теории равенства и неинтерпретируемых функций.

Ответ 15:

Алгоритм
Вход: ϕ^{UF} - конъюнкция предикатов равенства (я так понял неравенства тоже) над переменными и неинтерпретируемыми функциями.

$$\phi^{UF} := x_1 = x_2 \wedge x_2 = x_3 \wedge x_4 = x_5 \wedge x_5 \neq x_1 \wedge F(x_1) \neq F(x_3)$$

Выход: $SAT/UNSAT$.

1. Возьмем два терма t_1, t_2 (это могут быть как переменные, так и неинтерпретируемые функции) и отнесем их в один класс эквивалентности, если в ϕ^{UF} встречается предикат $t_1 = t_2$. Оставшиеся термы будут образовывать одноэлементные классы эквивалентности. (В данном пункте подразумевается, что мы будем как бы идти по предикатам и для каждой парочки создавать свой класс).

$$\{x_1, x_2\}, \{x_2, x_3\}, \{x_4, x_5\}, \{F(x_1)\}, \{F(x_3)\}$$

2. Пройдемся по получившимся классам. Если у каких-то классов есть общий терм, объединяем их в один общий класс. Повторяем до тех пор, пока мы можем это делать.

$$\{x_1, x_2, x_3\}, \{x_4, x_5\}, \{F(x_1)\}, \{F(x_3)\}$$

3. Пройдемся по получившимся классам. Если термы t_i, t_j лежат в одном классе, а также в ϕ^{UF} встречаются термы $F(t_i), F(t_j)$ (где F - некоторая неинтерпретируемая функция), то объединяем классы, представителями которых являются $F(t_i)$ и $F(t_j)$. Повторяем до тех пор, пока можем.

$$\{x_1, x_2, x_3\}, \{x_4, x_5\}, \{F(x_1), F(x_3)\}$$

4. Если в ϕ^{UF} присутствует предикат $t_i \neq t_j$, такой, что t_i, t_j принадлежат одному классу эквивалентности, то возвращаем $UNSAT$. Иначе - SAT.

16 (Саит, DONE). Какой метод лежит в основе разрешающей процедуры теории линейной арифметики?

Ответ 16:

На основе лекции 8.

Симплекс метод.

17 (Алтана). Что такое комбинация теорий?

Ответ 17:

Теория T – это множество "предложений"(формулы 1-го порядка над сигнатурой Σ, где все переменные связаны с квантором)

Пусть T_1, T_2 - теории над сигнатурами Σ_1, Σ_2 соответственно. Тогда **комбинацией теорий** $T_1 \oplus T_2$ назовем теорию над сигнатурой $\Sigma_1 \cup \Sigma_2$ над множеством аксиом $T_1 \cup T_2$

18 (Каринэ, DONE). Что такое выпуклые теории?

Ответ 18:

Определение из Decision procedures, стр. 230 и лекции 10

Σ - теория T является ***выпуклой***, если для любой Σ-формулы ϕ
$(\phi \Rightarrow \bigvee_{i=1}^{n} x_i = y_i)$ - T - тавтология для некоторого конечного $n > 1 \Rightarrow$
$(\phi \Rightarrow x_i = y_i)$ - T - тавтология для некоторого $i \in \{1, \ldots, n\}$
где $x_i, y_i, i \in \{1, \ldots, n\}$ - некоторые переменные.

Другими словами, в выпуклой теории T, если формула подразумевает дизъюнкцию равенств, то она подразумевает также хотя бы одно из этих равенств по отдельности.

19 (Паша). Что такое ограничение Нельсона-Оппена?

Ответ 19:

Ограничения Нельсона-Оппена:

1. $T_1, ..., T_n$ - бескванторные теории с равенством
2. Для каждой теории есть разрешающая процедура
3. Пересечение каждой пары сигнатур - пустое множество
4. Теории интерпретируются над бесконечным доменом (теории вида битовых векторов не подходят)

P.S. Интерпретация тоже задана *(хз ограничение или нет)*

20 (Саит, DONE). Что такое вычислительное дерево программы?

Ответ 20:

Есть в лекции 11, но там нет видео.

Вычислительное дерево программы – способ представления программы в виде бинарного дерева, при котором:

- Каждая вершина - выполнение условного оператора.
- Каждое ребро – выполнение последовательности команд, которые не являются условным оператором.

Таким образом, каждый путь от корня делит множество входных данных на классы эквивалентности.

21 (Алтана, DONE). Что такое путь исполнения программы?

Ответ 21:

Есть в лекции 11, но там нет видео.

Вычислительное дерево программы – способ представления программы в виде бинарного дерева, при котором:

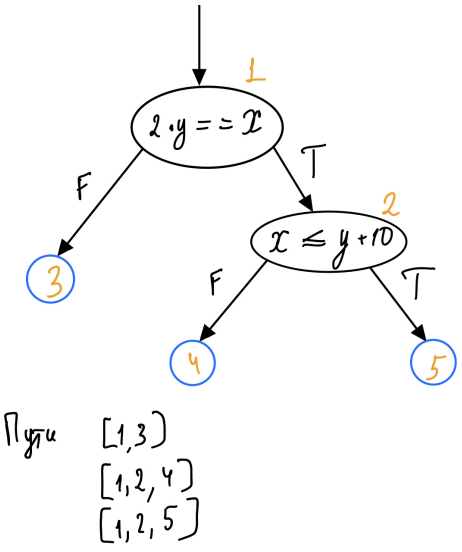
- Каждая вершина - выполнение условного оператора.
- Каждое ребро – выполнение последовательности команд, которые не являются условным оператором.

Путь исполнения программы – это путь от корня до какого-либо листа в вычислительном дереве программы.

22 (Паша, DONE). Дана программа, выпишите все пути исполнения программы.

Ответ 22:

```
1  void test(int x, int y) {
2      if (2*y == x) {
3          if (x <= y+10) {
4              printf("OK");
5          } else {
6              printf("not OK");
7              assert false;
8          }
9      } else {
10         print("OK");
11     }
12 }
```



23 (Саит, DONE). Дана программа, выпишите её path constaint.

Ответ 23:

На основе лекции 12.

Пример 1:

Дана следующая программа. Суть в том, что в данные разбиты на блоки, в начале каждого блока ссылка на начало следующего блока.

```
1  void ReadBlocks (int data[], int cookie) {
2      int i = 0;
3      while (true) {
4          int next;
5          next = data[i];
6          if (! (i < next && next < N)) return;
7          i = i + 1;
8          for (; i < next; i = i + 1) {
9              if (data[i] == cookie) { // skip data
10                 i = i + 1;
11             }
12             else { // process data
13                 Process(data [i]);
14             }
15         }
16     }
17 }
```

Рассмотрим ее следующую **трассу** (видимо в таком задании будет дана трасса):

```
1  i = 0;                      // 2, assignment
2  next = data[i];             // 5, assignment
3  i < next && next < N        // 6, branch
4  i = i + 1;                  // 7, assignment
5  i < next                    // 8, branch
6  data[i] != cookie           // 9, branch
7  Process(data[i]);           // 13, function call
8  i = i + 1;                  // 8, assignment
9  !(i < next)                 // 8, branch
10 next = data[i];             // 5, assignment
11 !(i < next && next < N)     // 6, branch
```

Далее по этой трассе можно построить ***SSA (Static single assignment)*** – для этого каждый раз, когда происходит новое присваивание, мы вводим новую перемнную. Все это нужно для того, чтобы представить трассу в виде формулы, которую можно будет отдать солверу и получить от него набор переменных, которые удовлетворяют данной трассе.

SSA для приведенной выше трассы:

```
1  i_1 = 0;                        // 2, assignment
2  next_1 = data_0[i_1];           // 5, assignment
3  i_1 < next_1 && next_1 < N_0    // 6, branch
4  i_2 = i_1 + 1;                  // 7, assignment
5  i_2 < next_1                    // 8, branch
6  data_0[i_2] != cookie_0         // 9, branch
7  Process(data_0[i_2]);           // 13, function call
```

```
8  i_3 = i_2 + 1;                  // 8, assignment
9  !(i_3 < next_1)                 // 8, branch
10 next_2 = data_0[i_3];           // 5, assignment
11 !(i_3 < next_2 && next_2 < N_0) // 6, branch
```

А уже по полученному SSA мы можем легко получить ***Path constraint***:

$$
\begin{array}{ll}
i_1 = 0 & \wedge \\
next_1 = data_0[i_1] & \wedge \\
(i_1 < next_1 \wedge next_1 < N_0) & \wedge \\
i_2 = i_1 + 1 & \wedge \\
i_2 < next_1 & \wedge \\
data_0[i_2] \neq cookie_0 & \wedge \\
i_3 = i_2 + 1 & \wedge \\
\neg(i_3 < next_1) & \wedge \\
next_2 = data_0[i_3] & \wedge \\
\neg(i_3 < next_2 \wedge next_2 < N_0) &
\end{array}
$$

Пример 2:

Можно анализировать программы на то, что все assert-ы в них никогда не срабатывают или находить примеры, на которых они срабатывают. Для этого достаточно добавить отрицание условия в соответствующем assert в path constraint и отдать все это солверу. Если получим SAT и набор переменных, то assert сработал, а если получим UNSAT, то все ок – assert никогда не срабатывает.

Рассмотрим следующую, другую трассу вышеприведенной программы:

```
1  i = 0;                  // 2, assignment
2  next = data[i];         // 5, assignment
3  i < next && next < N    // 6, branch
4  i = i + 1;              // 7, assignment
5  i < next                // 8, branch
6  data[i] = cookie        // 9, branch
7  i = i + 1;              // 10, assignment
8  i = i + 1;              // 8, assignment
9  !(i < next)             // 8, branch
10 0 <= i && i < N         // 5, assertion
```

Видим, что в последней строчке мы хотим видеть assert для $0 <= i \wedge i < N$, прежде чем обращаться к $next = data[i]$, рискуя выйти за границы массива $data$.
Окей, просто добавим отрицание этого assert к path constraint (шаг с построением SSA выполняем в уме):

$$
\begin{array}{ll}
i_1 = 0 & \wedge \\
next_1 = data_0[i_1] & \wedge \\
(i_1 < next_1 \wedge next_1 < N_0) & \wedge \\
i_2 = i_1 + 1 & \wedge \\
i_2 < next_1 & \wedge \\
data_0[i_2] = cookie_0 & \wedge \\
i_3 = i_2 + 1 & \wedge \\
i_4 = i_3 + 1 & \wedge \\
\neg(i_4 < next_1) & \wedge \\
\neg(0 \leqslant i_4 \wedge i_4 < N_0) &
\end{array}
$$

Пример 3:
Что делать, если у нас цикл ограниченной длины *(Bounded Program Analysis)*?
В таком случае мы можем развернуть цикл в несколько вложенных *if*-ов. После этого для того, чтобы переписать нашу программу, чтобы получить логические формулы – мы будем условия циклов записывать в качестве некоторых переменных, а также будем высчитывать значения необходимые для симуляции поведения того, что было бы если бы мы пошли бы по ветке *True*, а что, если бы по *False*. И далее просто пишем тернарники на переменные согласно *if*-ам:

```
1  if (i < next) {
2    if (data[i] == cookie)
3       i = i + 1;
4    else
5       Process(data[i]);
6
7    i = i + 1;
8
9    if (i < next) {
10     if (data[i] == cookie)
11        i = i + 1;
12     else
13        Process(data[i]);
14
15     i = i + 1;
16   }
17 }
```

```
1  γ1 = (i0 < next0);
2  γ2 = (data0[i1] == cookie0);
3  i1 = i0 + 1;
4
5
6  i2 = γ2 ? i1 : i0;
7  i3 = i2 + 1;
8
9  γ3 = (i3 < next0);
10 γ4 = (data0[i3] == cookie0);
11 i4 = i3 + 1;
12
13
14 i5 = γ4 ? i4 : i3;
15 i6 = i5 + 1;
16 i7 = γ3 ? i6 : i3;
17 i8 = γ1 ? i7 : i0;
```

Рис. 2: Path constraint для цикла, развернутого в *if*-ы.

Пример 4:
А что делать, если цикл неограниченной длины, т.е. мы не знаем через сколько именно витков они закончатся *(Unbounded Program Analysis)*?
В таком случае мы можем потерять точность в программе, но при этом заменить цикл

на *if*:

```
1 int i=0, j=0;
2
3 while(data[i] != '\n')
4 {
5   i++;
6   j=i;
7 }
8
9 assert(i == j);
```

⟶

```
1 int i=0, j=0;
2
3 if(data[i] != '\n')
4 {
5   i=*;
6   j=*;
7   i++;
8   j=i;
9 }
10
11 assume(data[i] == '\n')
12
13 assert(i == j);
```

Рис. 3: Замена неограниченного цикла.

То есть мы сказали, что в *if*-е, в его начале, i, j могут быть вообще любыми – естественно это не совсем так, у них есть свои ограничения, но мы их не учитываем. Наша же цель проверить *assert*$(i = j)$. В данном случае он будет выполняться всегда и солвер вернут $UNSAT$ на следующий path constraint:

$$
\begin{array}{ll}
i_1 = 0 & \wedge \\
j_1 = 0 & \wedge \\
\gamma_1 = (data_0[i_1] \neq \ '\backslash n') & \wedge \\
i_3 = i_2 + 1 & \wedge \\
j_3 = i_3 & \wedge \\
i_4 = \gamma_1 \ ? \ i_3 : i_1 & \wedge \\
j_4 = \gamma_1 \ ? \ j_3 : j_1 & \wedge \\
data_0[i_4] = \ '\backslash n' & \wedge \\
i_4 \neq j_4 &
\end{array}
$$

Заметьте, что i_2 и j_2 – действительно любые. При таком подходе мы получаем такое поведение – если солвер вернет $UNSAT$, то действительно assert никогда не выполнится, но при этом солвер может вернуть SAT и набор переменных, которые не достигаются, но стали возможными из-за того, что мы сделали программу более неточной.