Анализ Алиасов

Обзорная презентация

Савченко Валерий

20 марта 2014 г.

Содержание:

Что такое алиас?

Определение

Алиасом будем называть ситауцию, в которой разные выражения представляют один и тот же участок памяти.

Что такое алиас?

Определение

Алиасом будем называть ситауцию, в которой разные выражения представляют один и тот же участок памяти.

Пример

```
A *p, *q;
p = new A();
q = p;
```

Что такое алиас?

Определение

Алиасом будем называть ситауцию, в которой разные выражения представляют один и тот же участок памяти.

Пример

```
A *p, *q;
p = new A();
q = p;
```

Здесь $*p$ и $*q$ - алиас

Откуда может взяться алиас?

Откуда может взяться алиас?

- Указатели

Откуда может взяться алиас?

- Указатели
- Передача параметров по ссылке

Откуда может взяться алиас?

- Указатели
- Передача параметров по ссылке
- Индексация массивов

Откуда может взяться алиас?

- Указатели
- Передача параметров по ссылке
- Индексация массивов
- Полиморфные структуры данных (типа **union в** C**)**

А что может быть алиасом?

А что может быть алиасом?

Статически аллоцированные объекты

```
void foo() {
  int i, j;
  . . .
}
```

А что может быть алиасом?

Статически аллоцированные объекты

```
void foo() {
  int i, j;
  . . .
}
```

Могут ли i и j быть алиасом?

А что может быть алиасом?

Статически аллоцированные объекты

```
void foo() {
  int i, j;
  . . .
}
```

Могут ли i и j быть алиасом?

Динамически аллоцированные объекты

```
void bar() {
  . . .
  p = new TreeNode();
  p->left = new TreeNode();
  p->right = new TreeNode();
  . . .
}
```

А что может быть алиасом?

Статически аллоцированные объекты

```
void foo() {
  int i, j;
  . . .
}
```

Могут ли i и j быть алиасом?

Динамически аллоцированные объекты

```
void bar() {
  . . .
  p = new TreeNode();
  p->left = new TreeNode();
  p->right = new TreeNode();
  . . .
}
```

А p, $p \to left$ и $p \to right$?

Чем это может навредить?

Чем это может навредить?

CSE

```
*p = a + b;
c = a + b;
```

Чем это может навредить?

CSE

```
*p = a + b;
c = a + b;
```

А если $*p$ алиас для a или b?

Чем это может навредить?

CSE

```
*p = a + b;
c = a + b;
```

А если $*p$ алиас для a или b?

LICM

```
while (--i >= 0) {
  c[i] = *p + a;
}
```

Чем это может навредить?

CSE

```
*p = a + b;
c = a + b;
```

А если $*p$ алиас для a или b?

LICM

```
while (--i >= 0) {
  c[i] = *p + a;
}
```

А если $\exists i : *p$ алиас для $c[i]$?

Чем это может навредить?

CSE

```
*p = a + b;
c = a + b;
```

А если $*p$ алиас для a или b?

LICM

```
while (--i >= 0) {
  c[i] = *p + a;
}
```

А если $\exists i : *p$ алиас для $c[i]$?

И множество других примеров...

Какие бывают алиас анализы

Какие бывают алиас анализы

- Межпроцедурные и внутрипроцедурные

Какие бывают алиас анализы

- Межпроцедурные и внутрипроцедурные
- Потоково-чувствительные и -нечувствительные

Какие бывают алиас анализы

- Межпроцедурные и внутрипроцедурные
- Потоково-чувствительные и -нечувствительные
- Контекстно-чувствительные и -нечувствительные

Какие бывают алиас анализы

- Межпроцедурные и внутрипроцедурные
- Потоково-чувствительные и -нечувствительные
- Контекстно-чувствительные и -нечувствительные
- Поле-чувствительные и -нечувствительные

Какие бывают алиас анализы

- Межпроцедурные и внутрипроцедурные
- Потоково-чувствительные и -нечувствительные
- Контекстно-чувствительные и -нечувствительные
- Поле-чувствительные и -нечувствительные
- Учитывающие тип или нет

Чем они могут еще различаться?

Чем они могут еще различаться?

- Модели памяти

Чем они могут еще различаться?

- Модели памяти
- Необходимость полного кода программы

Чем они могут еще различаться?

- Модели памяти
- Необходимость полного кода программы
- Представление алиасов

Чем они могут еще различаться?

- Модели памяти
- Необходимость полного кода программы
- Представление алиасов
- Представление агрегатов

Некоторые обозначения

Определение

Пусть p - указатель, i - номер вызова функции, содержащей p, j - номер определения p. Тогда через

$$p_{ij} \to \{a, ...\}$$

будем обозначать то, что при i-ом вызове функции и j-ом определении p он может указывать на $\{a, ...\}$

Потоково-чувствительный и контекстно-чувствительный

```
int** foo(int **p, **q) {
  int **x;
  x = p;
  . . .
  x = q;
  return x;
}

int main() {
  int **a, *b, *d, *f, c, e;
  a = foo(&b, &f);
  *a = &c;
  a = foo(&d, &g);
  *a = &e;
}
```

Потоково-чувствительный и контекстно-чувствительный

```
int** foo(int **p, **q) {
  int **x;
  x = p;
  . . .
  x = q;
  return x;
}

int main() {
  int **a, *b, *d, *f, c, e;
  a = foo(&b, &f);
  *a = &c;
  a = foo(&d, &g);
  *a = &e;
}
```

$$p_{11} \to \{b\}$$
$$q_{11} \to \{f\}$$
$$p_{21} \to \{d\}$$
$$q_{21} \to \{g\}$$
$$x_{11} \to \{b\}$$
$$x_{12} \to \{f\}$$
$$x_{21} \to \{d\}$$
$$x_{22} \to \{g\}$$
$$a_{11} \to \{f\}$$
$$a_{12} \to \{g\}$$
$$f_{11} \to \{c\}$$
$$g_{11} \to \{e\}$$

Потоково-чувствительный и контекстно-нечувствительный

```
int** foo(int **p, **q) {
  int **x;
  x = p;
  . . .
  x = q;
  return x;
}

int main() {
  int **a, *b, *d, *f, c, e;
  a = foo(&b, &f);
  *a = &c;
  a = foo(&d, &g);
  *a = &e;
}
```

Потоково-чувствительный и контекстно-нечувствительный

```
int** foo(int **p, **q) {
  int **x;
  x = p;
  . . .
  x = q;
  return x;
}

int main() {
  int **a, *b, *d, *f, c, e;
  a = foo(&b, &f);
  *a = &c;
  a = foo(&d, &g);
  *a = &e;
}
```

$$p\_1 \to \{b, d\}$$
$$q\_1 \to \{f, g\}$$
$$x\_1 \to \{b, d\}$$
$$x\_2 \to \{f, g\}$$
$$a\_1 \to \{f, g\}$$
$$a\_2 \to \{f, g\}$$
$$f\_1 \to \{c\}$$
$$g\_1 \to \{c\}$$
$$f\_2 \to \{c, e\}$$
$$g\_2 \to \{c, e\}$$

Потоково-нечувствительный и контекстно-чувствительный

```
int** foo(int **p, **q) {
  int **x;
  x = p;
  . . .
  x = q;
  return x;
}

int main() {
  int **a, *b, *d, *f, c, e;
  a = foo(&b, &f);
  *a = &c;
  a = foo(&d, &g);
  *a = &e;
}
```

Потоково-нечувствительный и контекстно-чувствительный

```
int** foo(int **p, **q) {
  int **x;
  x = p;
  . . .
  x = q;
  return x;
}

int main() {
  int **a, *b, *d, *f, c, e;
  a = foo(&b, &f);
  *a = &c;
  a = foo(&d, &g);
  *a = &e;
}
```

$$
\begin{aligned}
p_{1\_} &\to \{b\} \\
q_{1\_} &\to \{f\} \\
p_{2\_} &\to \{d\} \\
q_{2\_} &\to \{g\} \\
x_{1\_} &\to \{b, f\} \\
x_{2\_} &\to \{d, g\} \\
a_{1\_} &\to \{b, f, d, g\} \\
b_{1\_} &\to \{c, e\} \\
f_{1\_} &\to \{c, e\} \\
d_{1\_} &\to \{c, e\} \\
g_{1\_} &\to \{c, e\}
\end{aligned}
$$

Потоково-нечувствительный и контекстно-нечувствительный

```
int** foo(int **p, **q) {
  int **x;
  x = p;
  . . .
  x = q;
  return x;
}

int main() {
  int **a, *b, *d, *f, c, e;
  a = foo(&b, &f);
  *a = &c;
  a = foo(&d, &g);
  *a = &e;
}
```

Потоково-нечувствительный и контекстно-нечувствительный

```
int** foo(int **p, **q) {
  int **x;
  x = p;
  . . .
  x = q;
  return x;
}

int main() {
  int **a, *b, *d, *f, c, e;
  a = foo(&b, &f);
  *a = &c;
  a = foo(&d, &g);
  *a = &e;
}
```

$$
\begin{aligned}
p\_\_ &\to \{b, d\} \\
q\_\_ &\to \{f, g\} \\
x\_\_ &\to \{b, f, d, g\} \\
a\_\_ &\to \{b, f, d, g\} \\
b\_\_ &\to \{c, e\} \\
f\_\_ &\to \{c, e\} \\
d\_\_ &\to \{c, e\} \\
g\_\_ &\to \{c, e\}
\end{aligned}
$$

Потоково-нечувствительный и контекстно-нечувствительный

Самые известные и используемые алгоритмы анализа алиасов относятся как раз к этой группе.

Потоково-нечувствительный и контекстно-нечувствительный

Самые известные и используемые алгоритмы анализа алиасов относятся как раз к этой группе.

- Алгоритм Андерсена

Потоково-нечувствительный и контекстно-нечувствительный

Самые известные и используемые алгоритмы анализа алиасов относятся как раз к этой группе.

- Алгоритм Андерсена
- Алгоритм Стинсгарда

Потоково-нечувствительный и контекстно-нечувствительный

Самые известные и используемые алгоритмы анализа алиасов относятся как раз к этой группе.

- Алгоритм Андерсена
- Алгоритм Стинсгарда

Замечание

Большинство разработанных позднее методов являются либо модификациями, либо некими гибридами этих двух.

Алгоритм Андерсена (94')

Разработан Ларсом Уле Андерсеном и представлен в соответствующей работе в 1994-ом году.

Алгоритм Андерсена (94')

Разработан Ларсом Уле Андерсеном и представлен в соответствующей работе в 1994-ом году.

- Потоково- и контекстно-нечувствительный

Алгоритм Андерсена (94')

Разработан Ларсом Уле Андерсеном и представлен в соответствующей работе в 1994-ом году.

- Потоково- и контекстно-нечувствительный
- Среди своего класса анализов обеспечивает наилучшую точность

Основные свойства

Использует “**указывает-на**” **множества** (points-to sets) в качестве более компактного результата анализа.

Основные свойства

Использует “**указывает-на**” **множества** (points-to sets) в качестве более компактного результата анализа.

Пример

Пусть P - множество указателей, V - множество переменных.
Отметим, что $P \subseteq V$.

Решение ранее:

$$\{\forall p \in P, \forall v \in V :< p, v, d_{p,v} >\}$$

где $d_{p,v} \in \{0, 1, \frac{1}{2}\}$

Решение сейчас:

$$\{\forall p \in P : p \rightarrow V_p\}$$

где $V_p \subseteq V$

Основные свойства

Иногда в анализируемой программе совершенно ничего нельзя сказать о том, куда указывает указатель.

Основные свойства

Иногда в анализируемой программе совершенно ничего нельзя сказать о том, куда указывает указатель.

Пример

Указатель $argv$

Основные свойства

Иногда в анализируемой программе совершенно ничего нельзя сказать о том, куда указывает указатель.

Пример

Указатель $argv$

Специально для таких случаев вводится абстрактная локация $Unknown$.

Определение

$Unknown \in V$, причем, если $p \to V_p$ и $Unknown \in V_p$, то p может указывать на **все** доступные в runtime объекты.

Основные свойства

Если $q \to Unknown$ и $*q = \&x$, то **все** указатели могут указывать на x. Или даже хуже, если $*q = r$ и $r \to Unknown$, то $\forall p \in P \Rightarrow p \to Unknown$ (т.е. **каждый** указатель может указывать на **все** объекты).

Основные свойства

Если $q \to Unknown$ и $*q = \&x$, то **все** указатели могут указывать на x. Или даже хуже, если $*q = r$ и $r \to Unknown$, то $\forall p \in P \Rightarrow p \to Unknown$ (т.е. **каждый** указатель может указывать на **все** объекты).

Вывод

Если в $lvalue$ имеется разыменование указателя $p \in P : p \to Unknown$, прекращаем анализ.

Основные свойства

Массивы считаются *агрегатами*, т.е. различные элементы массива считаются *неотличимыми*.

Пояснение

Для анализа $\forall i \in \mathbb{Z}_+ \Rightarrow a[i] \equiv *a$

Основные свойства

Массивы считаются *агрегатами*, т.е. различные элементы массива считаются *неотличимыми*.

Пояснение

Для анализа $\forall i \in \mathbb{Z}_+ \Rightarrow a[i] \equiv *a$

Одноименные поля различных объектов считаются *агрегатами*.

Пояснение

Пусть

```
struct Q { int x; };
```

(через $S_Q \subseteq V$ будем обозначать множество элементов типа Q), тогда для анализа $\forall s, t \in S_Q \Rightarrow s.x \equiv t.x$

Рассматривать алгоритм будем упрощенно и на примерах.

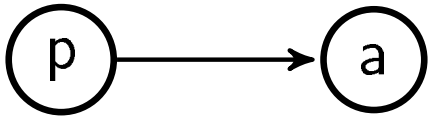
Рассматривать алгоритм будем упрощенно и на примерах.

Возможны следующие шесть основных операций:

- `p = &a;`
- `p = q;`
- `p = *r;`
- `*p = &a;`
- `*p = q;`
- `*p = *r;`

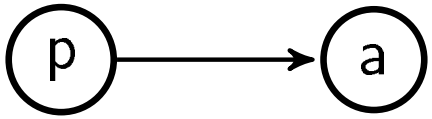
1. `p = &a;`

Добавляем ребро от p к a, показывая, что $a \in V_p$



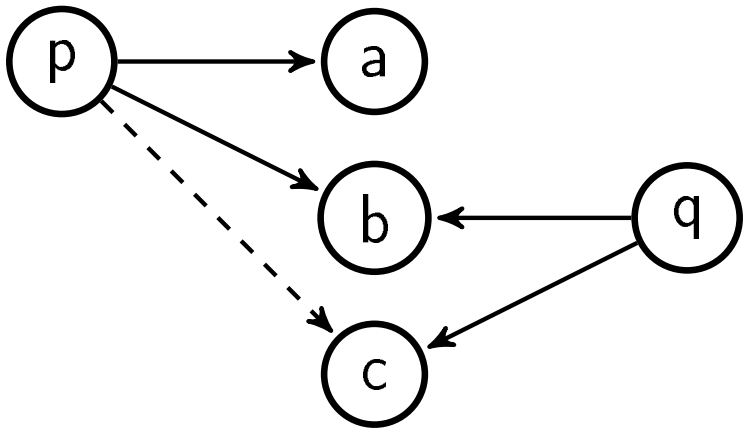
1. `p = &a;`

Добавляем ребро от p к a, показывая, что $a \in V_p$



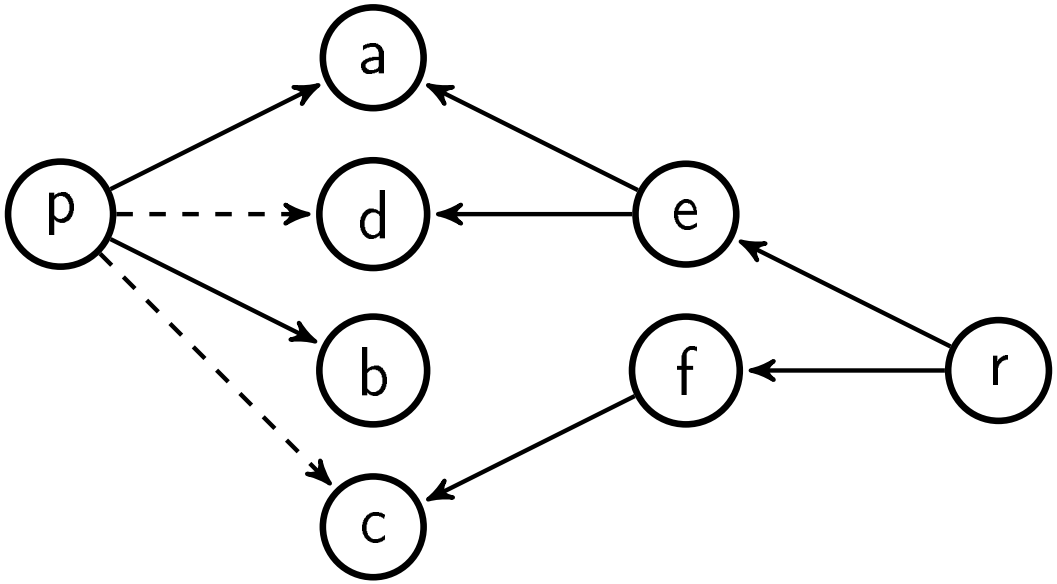
2. `p = q;`

Добавляем ребра от p ко всем вершинам, куда есть ребра из q. Это значит также, что если позже будут добавлены ребра из q, аналогичные ребра должны быть добавлены из p. Т.е. должно сохраняться свойство, что $V_q \subseteq V_p$.



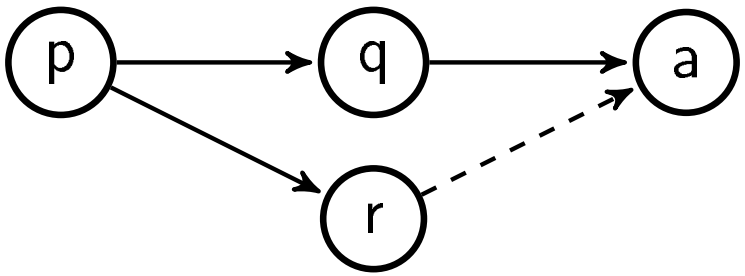
3. `p = *r;`

Пусть $r \to V_r$ и $T = \bigcup_{t \in V_r} V_t$. Тогда добавляем ребро из p к каждому элементу из T. Аналогично предыдущему случаю при изменении V_r или T необходимо добавить соответствующие ребра. Т.е. должно сохраняться свойство, что $\forall t \in V_r \Rightarrow V_t \subseteq V_p$.



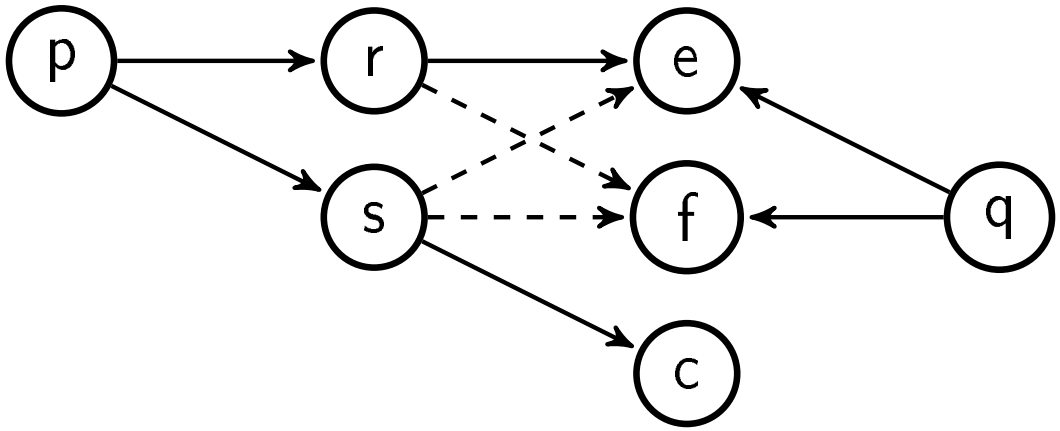
4. `*p = &a;`

Из каждой вершины, куда есть ребра из p добавляем ребро к a. Опять таки, при добавлении новых вершин в V_p из них необходимо добавить ребра к a. Т.е. должно сохраняться свойство, что $\forall t \in V_p \Rightarrow a \in V_t$.



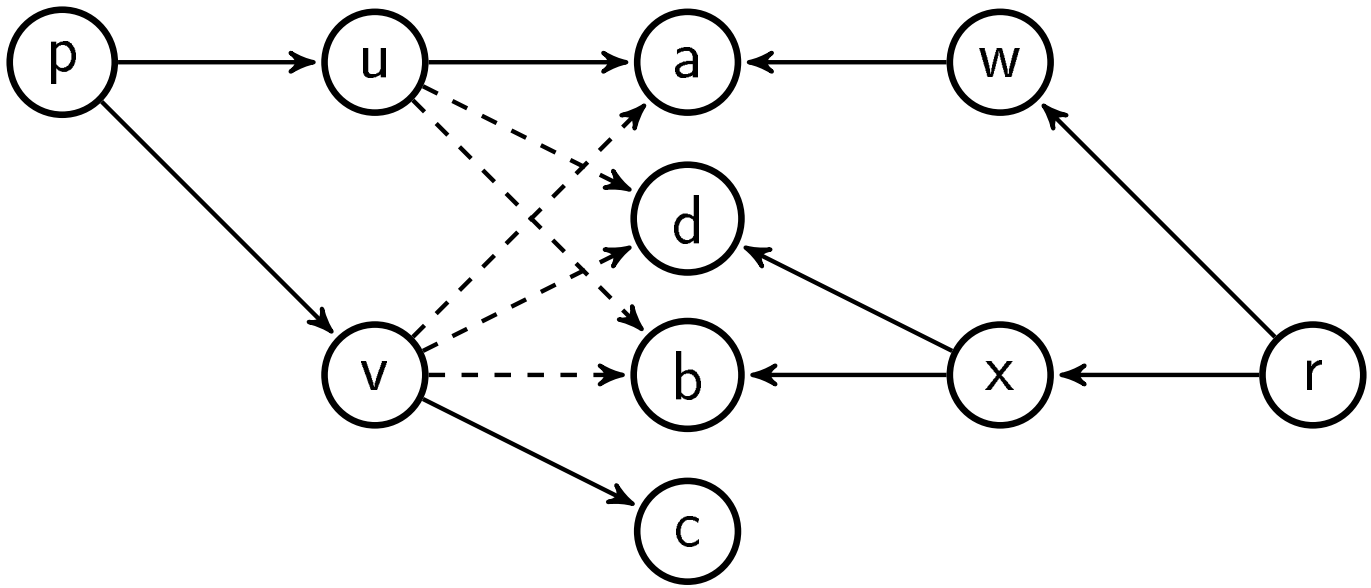
5. `*p = q;`

Из каждой вершины из V_p добавляем ребро к каждой вершине из V_q. При добавлении новых вершин в V_p или в V_q соответствующие новые ребра должны быть добавлены. Т.е. должно сохраняться свойство, что $\forall t \in V_p \Rightarrow V_q \subseteq V_t$.



6. `*p = *r;`

Пусть $T = \bigcup_{t \in V_r} V_t$. Тогда необходимо добавить ребро из каждой вершины из V_p в каждую вершину из T. При добавлении новых вершин в V_p или T соответствующие ребра должны быть добавлены. Т.е. должно сохраняться свойство, что $\forall s \in V_p, \forall t \in V_r \Rightarrow V_t \subseteq V_s$.



Результаты анализа

Опыт показывает, что алгоритм Андерсена предоставляет полезную информацию сильно превосходящую ‘address-taken’подход.

Результаты анализа

Опыт показывает, что алгоритм Андерсена предоставляет полезную информацию сильно превосходящую ‘address-taken’подход.

Интересный факт

Опыт также показывает, что добавление потоковой и контекстной чувствительности лишь незначительно улучшает результаты анализа на распространенных бенчмарках.

Ложка дегтя

Производительность алгоритма Андерсена в первоначальном его виде (итеративный алгоритм разрешения ограничений) оставляет желать лучшего, если говорить о больших программах. Время исполнения алгоритма $O(n^3)$, где n - число вершин в графе (n можно оценить сверху, как $|V|$).

Ложка дегтя

Производительность алгоритма Андерсена в первоначальном его виде (итеративный алгоритм разрешения ограничений) оставляет желать лучшего, если говорить о больших программах. Время исполнения алгоритма $O(n^3)$, где n - число вершин в графе (n можно оценить сверху, как $|V|$).

Несмотря на значительные успехи в ускорении его работы на реальных приложениях в последние годы, асимптотика времени исполнения остается той же.

Алгоритм Стинсгарда (96')

Разработан Бьярне Стинсгардом и представлен в соответствующей работе в 1996-ом году.

Алгоритм Стинсгарда (96')

Разработан Бьярне Стинсгардом и представлен в соответствующей работе в 1996-ом году.

- Потоково- и контекстно-нечувствительный

Алгоритм Стинсгарда (96')

Разработан Бьярне Стинсгардом и представлен в соответствующей работе в 1996-ом году.

- Потоково- и контекстно-нечувствительный
- Работает практически за линейное время ($O(n * \alpha(n, n))$, где $\alpha(n, n)$ - обратная функция Аккермана)

Основная идея

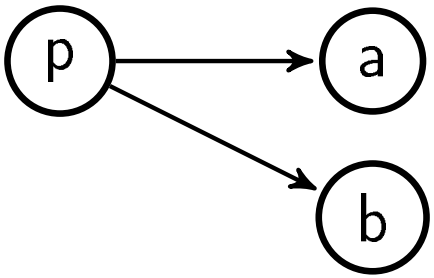
Вершины $a, b \in V$ будут представлены одной вершиной в графе, если $\exists p \in P : a, b \in V_p$.

Основная идея

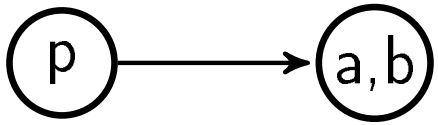
Вершины $a, b \in V$ будут представлены одной вершиной в графе, если $\exists p \in P : a, b \in V_p$.

Тогда в сравнении с алгоритмом Андерсена получаем:

Андерсен:



Стинсгард:

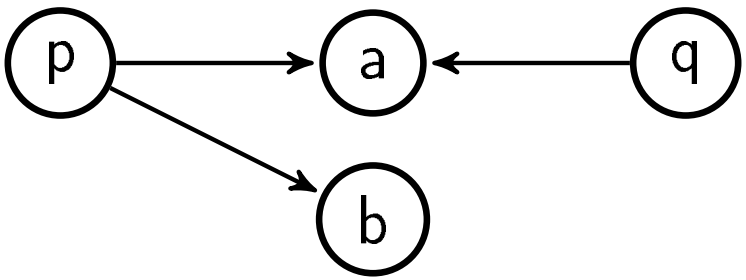


Очевидным является тогда факт того, что результаты подхода Стинсгарда уступают по точности результатам алгоритма Андерсена.

Очевидным является тогда факт того, что результаты подхода Стинсгарда уступают по точности результатам алгоритма Андерсена.

Пример:

Андерсен:



Стинсгард:

Разница с алгоритмом Андерсена на практике

- Горвиц и Шапиро протестировали 61 программу на C с размерами от 300 до 24 300 строк кода

Разница с алгоритмом Андерсена на практике

- Горвиц и Шапиро протестировали 61 программу на C с размерами от 300 до 24 300 строк кода
- Стинсгард менее точен - в среднем размер множеств в 4 раза больше; в худшем случае в 15 раз

Разница с алгоритмом Андерсена на практике

- Горвиц и Шапиро протестировали 61 программу на C с размерами от 300 до 24 300 строк кода
- Стинсгард менее точен - в среднем размер множеств в 4 раза больше; в худшем случае в 15 раз
- Андерсен медленнее - в среднем в полтора раза медленнее; в худшем случае в 31 раз

Разница с алгоритмом Андерсена на практике

- Горвиц и Шапиро протестировали 61 программу на C с размерами от 300 до 24 300 строк кода
- Стинсгард менее точен - в среднем размер множеств в 4 раза больше; в худшем случае в 15 раз
- Андерсен медленнее - в среднем в полтора раза медленнее; в худшем случае в 31 раз

Вывод

Необходимо использовать Андерсена для небольших программ, а Стинсгарда для больших.

Подход Горвиц-Шапиро (97')

Разработан Сюзан Горвиц и Марком Шапиро и представлен в соответствующей работе в 1997-ом году.

Подход представляет собой способ получить промежуточные звенья между подходами Андерсена и Стинсгарда.

Количество исходящих ребер в графе будем называть *степенью* вершины.

Основная идея

Можно ограничить степень вершин в графе неким k $(1 \leq k \leq n)$. При достижении ограничения все последующие вершины объединяются с вершинами, к которым уже есть ребра.

Количество исходящих ребер в графе будем называть *степенью* вершины.

Основная идея

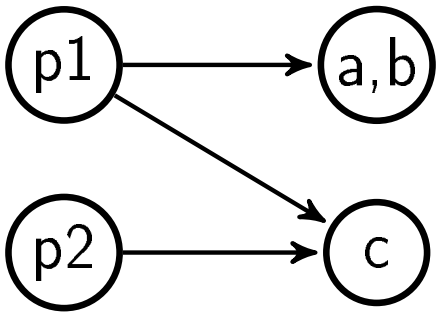
Можно ограничить степень вершин в графе неким k $(1 \leq k \leq n)$. При достижении ограничения все последующие вершины объединяются с вершинами, к которым уже есть ребра.

Тогда при $k = 1$ получаем подход Стинсгарда.
При $k = n$ получаем подход Андерсена.
При этом итоговая сложность алгоритма будет составлять $O(k^2 n)$.

Пример

```
p1 = &a;
p1 = &b;
p1 = &c;
p2 = &c;
```

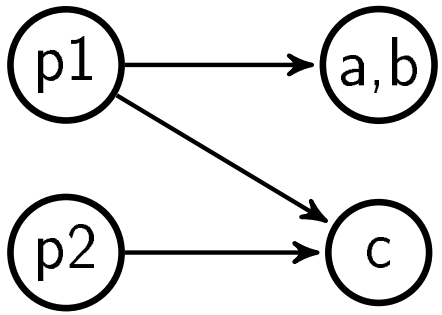
Пусть $k = 2$ и a, b соответствуют первой категории, а c второй. Тогда



Пример

```
p1 = &a;
p1 = &b;
p1 = &c;
p2 = &c;
```

Пусть $k = 2$ и a, b соответствуют первой категории, а c второй. Тогда

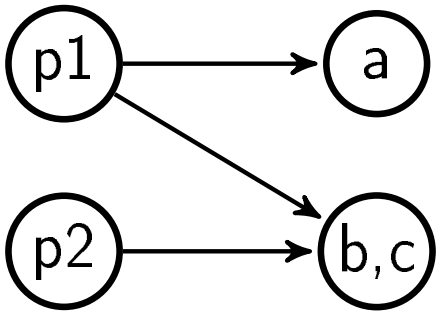


В данном случае точность анализа соответствует алгоритму Андерсена.

Пример

Однако...

Если же мы отнесем a к первой категории, а b, c ко второй, то получим



В данном случае мы потеряли в точности, т.к. результат говорит, что $b \in V_{p2}$.

Модификация подхода

Но, как видно из предыдущего примера, с двумя различными выборами категорий можно получить более точный ответ.

Модификация подхода

Но, как видно из предыдущего примера, с двумя различными выборами категорий можно получить более точный ответ.

Идея

Запускать анализ *несколько* раз.

Модификация подхода

Но, как видно из предыдущего примера, с двумя различными выборами категорий можно получить более точный ответ.

Идея

Запускать анализ *несколько* раз.

Пусть $p \to V_p^i$, где i - номер запуска анализа. Тогда, учитывая, что каждый из анализов на выходе имеет консервативные результаты, можно утверждать, что $p \to \bigcap_i V_p^i$ также является консервативным результатом.

Как и сколько?

Главные вопросы, оставшиеся в данном подходе:

- Как выбирать категории в различных запусках?
- Сколько запусков стоит делать?

Как и сколько?

Главные вопросы, оставшиеся в данном подходе:

- Как выбирать категории в различных запусках?
- Сколько запусков стоит делать?

Требование

Необходимо так подобрать категории и количество запусков, чтобы каждая пара переменных хотя бы раз попала в различные категории.

Для данной цели можно перенумеровать каждую переменную по основанию k. Тогда каждой переменной будет сопоставлено m-значное число. Именно m будет количеством запусков, а i-ый разряд будет категорией данной переменной в i-ом запуске.

Для данной цели можно перенумеровать каждую переменную по основанию k. Тогда каждой переменной будет сопоставлено m-значное число. Именно m будет количеством запусков, а i-ый разряд будет категорией данной переменной в i-ом запуске.

Пример

Пусть имеется 4 переменные a, b, c и d и $k = 2$. Пронумеруем переменные:

$$a^{00}, b^{01}, c^{10}, d^{11}$$

Для данной цели можно перенумеровать каждую переменную по основанию k. Тогда каждой переменной будет сопоставлено m-значное число. Именно m будет количеством запусков, а i-ый разряд будет категорией данной переменной в i-ом запуске.

Пример

Пусть имеется 4 переменные a, b, c и d и $k = 2$. Пронумеруем переменные:

$$a^{00}, b^{01}, c^{10}, d^{11}$$

При таком подходе имеем $m = \lfloor \log_k(n-1) \rfloor + 1$ и, соответственно, время выполнения $O(\log_k(n) k^2 n)$.

Результаты

На 25 тестах, при использовании 3 категорий, результирующие множества, полученные методом Горвиц-Шапиро, в среднем в 2.67 раз больше, чем множества Андерсена (для Стинсгарда этот множитель равен 4.75).

Подход Горвиц-Шапиро медленнее Стинсгарда, но от 7 до 25 раз быстрее Андерсена.

- Использование потоково- и контекстно-нечувствительных алгоритмов

- Использование потоково- и контекстно-нечувствительных алгоритмов
- Разработка более гибкого анализа, чем представленные ранее (поле-чувствительность, разделение агрегатов, моделирование кучи)

- Использование потоково- и контекстно-нечувствительных алгоритмов
- Разработка более гибкого анализа, чем представленные ранее (поле-чувствительность, разделение агрегатов, моделирование кучи)
- Использование алгоритма Андерсена, вернее алгоритмов, берущих от него начало (Andersen-style algorithms) или как их еще называют алгоритмы, основанные на включении (inclusion based)

- Использование потоково- и контекстно-нечувствительных алгоритмов
- Разработка более гибкого анализа, чем представленные ранее (поле-чувствительность, разделение агрегатов, моделирование кучи)
- Использование алгоритма Андерсена, вернее алгоритмов, берущих от него начало (Andersen-style algorithms) или как их еще называют алгоритмы, основанные на включении (inclusion based)
- Анализ формы (shape analysis) и разделяющая логика (separation logic)

Граф ограничений

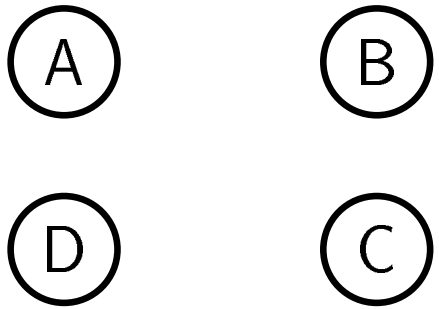
Для построения результирующих множеств все современные алгоритмы данного типа используют граф ограничений.

Определение

Графом ограничений будем называть $G =< M, E >$, такой что $M = P$ и $E = \{(p_1, p_2) : p_1, p_2 \in P \wedge V_{p_1} \subseteq V_{p_2}\}$
При этом каждая вершина p маркируется множеством V_p.

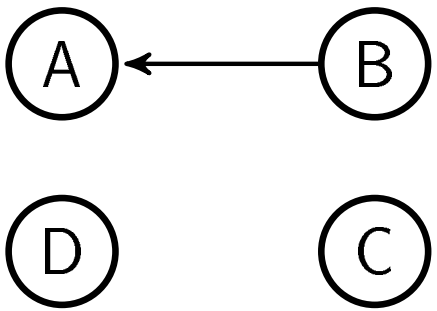
Пример: points-to граф

```
B = &A;
A = &C;
 D = A;
*D = B;
A = *D;
```



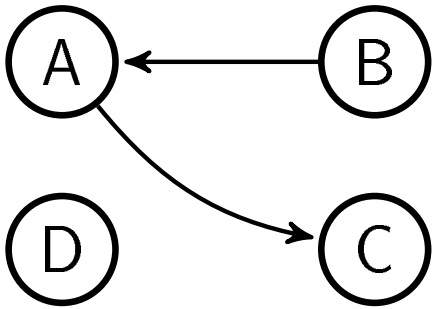
Пример: points-to граф

```
B = &A;
A = &C;
 D = A;
*D = B;
A = *D;
```



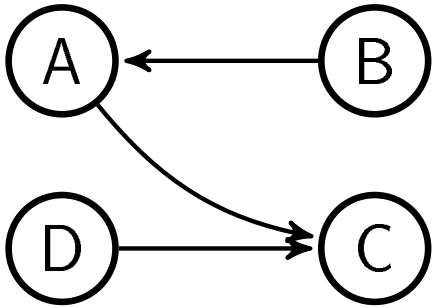
Пример: points-to граф

```
B = &A;
A = &C;
D = A;
*D = B;
A = *D;
```



Пример: points-to граф

B = &A;
A = &C;
D = A;
*D = B;
A = *D;

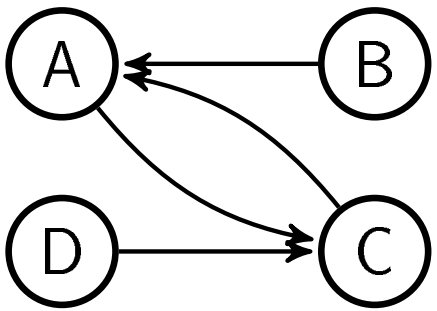


Ограничения:

1. $V_A \subseteq V_D$

Пример: points-to граф

```
B = &A;
A = &C;
D = A;
*D = B;
A = *D;
```

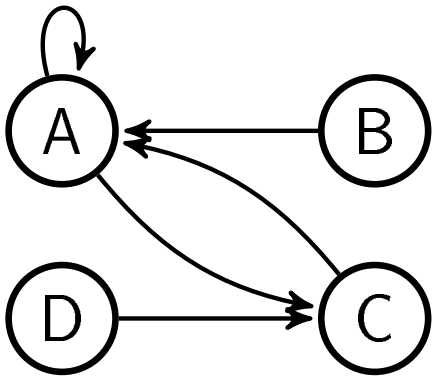


Ограничения:

1. $V_A \subseteq V_D$
2. $\forall T \in V_D \Rightarrow V_B \subseteq V_T$

Пример: points-to граф

B = &A;
A = &C;
D = A;
*D = B;
A = *D;

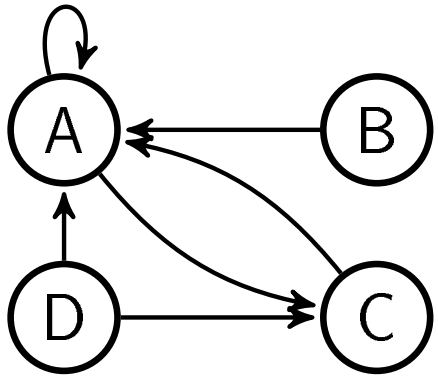


Ограничения:

1. $V_A \subseteq V_D$
2. $\forall T \in V_D \Rightarrow V_B \subseteq V_T$
3. $\forall T \in V_D \Rightarrow V_T \subseteq V_A$

Пример: points-to граф

B = &A;
A = &C;
D = A;
*D = B;
A = *D;

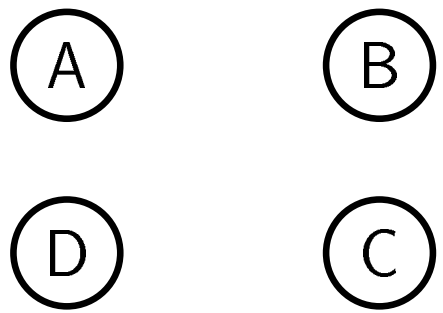


Ограничения:

1. $V_A \subseteq V_D$
2. $\forall T \in V_D \Rightarrow V_B \subseteq V_T$
3. $\forall T \in V_D \Rightarrow V_T \subseteq V_A$

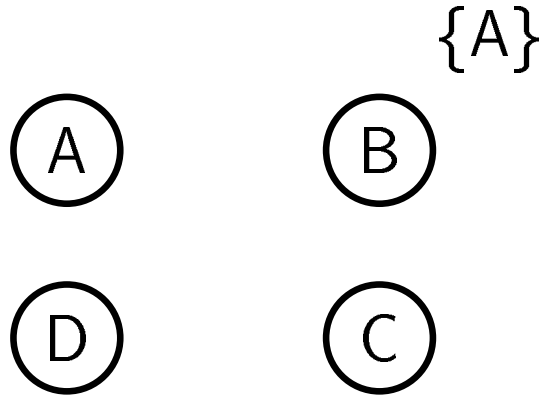
Пример: граф ограничений

```
B = &A;
A = &C;
 D = A;
*D = B;
A = *D;
```



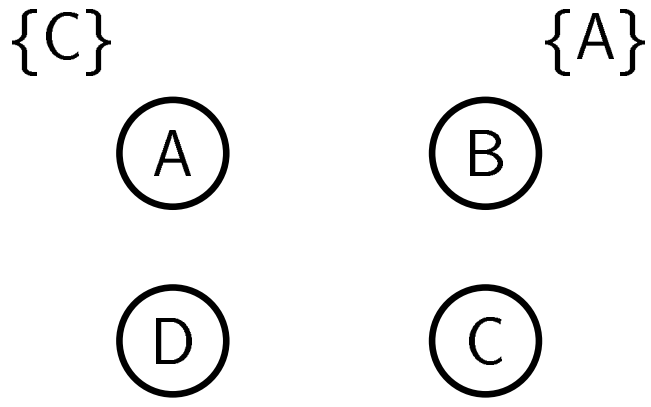
Пример: граф ограничений

```
B = &A;
A = &C;
 D = A;
*D = B;
A = *D;
```



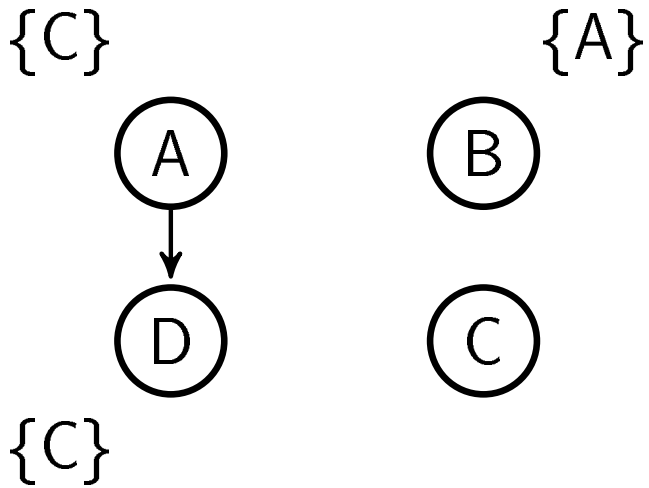
Пример: граф ограничений

```
B = &A;
A = &C;
D = A;
*D = B;
A = *D;
```



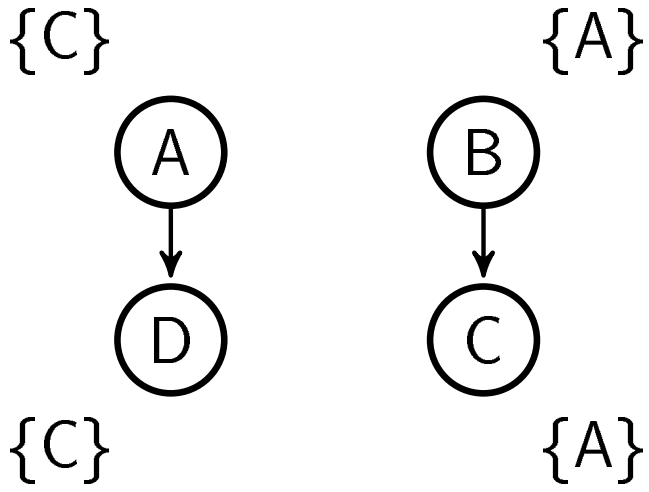
Пример: граф ограничений

```
B = &A;
A = &C;
D = A;
*D = B;
A = *D;
```



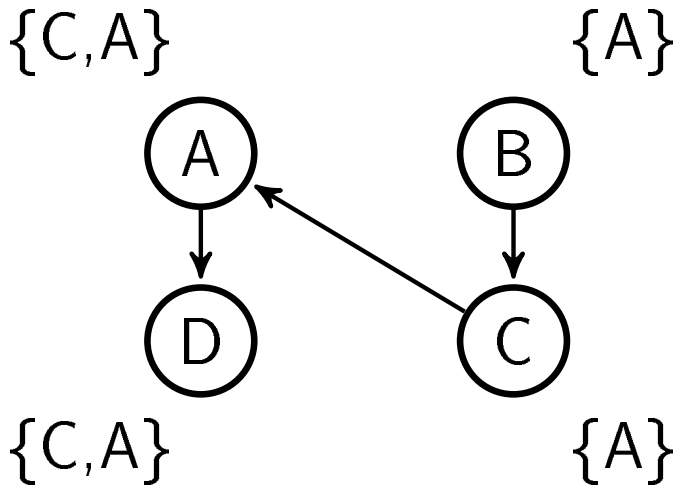
Пример: граф ограничений

```
B = &A;
A = &C;
 D = A;
*D = B;
A = *D;
```



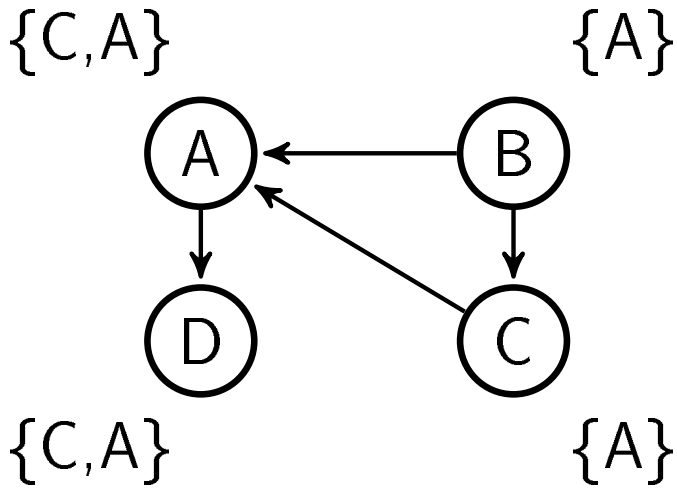
Пример: граф ограничений

```
B = &A;
A = &C;
 D = A;
*D = B;
A = *D;
```



Пример: граф ограничений

```
B = &A;
A = &C;
 D = A;
*D = B;
A = *D;
```



Задачи разрешения алиасов на графе ограничений и points-to графе практически идентичны, так в чем смысл был выбирать именно его?

Задачи разрешения алиасов на графе ограничений и points-to графе практически идентичны, так в чем смысл был выбирать именно его?

Интересное свойство

Пусть в графе ограничений имеется цикл длины n и p_1 принадлежит этому циклу. Тогда $\exists p_2, \ldots, p_n \in P : (p_1, p_2), \ldots, (p_{n-1}, p_n), (p_n, p_1) \in E$. Из определения графа ограничений получаем, что $V_{p_1} \subseteq V_{p_2} \subseteq \cdots \subseteq V_{p_n} \subseteq V_{p_1}$, откуда очевидно, что $V_{p_1} = V_{p_2} = \cdots = V_{p_n}$.

Задачи разрешения алиасов на графе ограничений и points-to графе практически идентичны, так в чем смысл был выбирать именно его?

Интересное свойство

Пусть в графе ограничений имеется цикл длины n и p_1 принадлежит этому циклу. Тогда $\exists p_2, \ldots, p_n \in P : (p_1, p_2), \ldots, (p_{n-1}, p_n), (p_n, p_1) \in E$. Из определения графа ограничений получаем, что $V_{p_1} \subseteq V_{p_2} \subseteq \cdots \subseteq V_{p_n} \subseteq V_{p_1}$, откуда очевидно, что $V_{p_1} = V_{p_2} = \cdots = V_{p_n}$.

Вывод

Без потери точности анализа все указатели, входящие в цикл, можно считать одной вершиной в графе ограничений.

Алгоритм Хайнтце-Тардьё (01‘)

Предложен Невином Хайнтце и Оливером Тардьё в 2001 году.

Является Andersen-style алгоритмом и первым среди такого рода алгоритмов, способный анализировать большие проекты на практике (за разумное время).

Основная идея

Построить граф ограничений, при добавлении новых ребер и распространении points-to множеств попутно сворачивать циклы в одну вершину.

Основная идея

Построить граф ограничений, при добавлении новых ребер и распространении points-to множеств попутно сворачивать циклы в одну вершину.

Основные моменты:

- При построении графа:

Основная идея

Построить граф ограничений, при добавлении новых ребер и распространении points-to множеств попутно сворачивать циклы в одну вершину.

Основные моменты:

- При построении графа:
  1. Каждой вершине p сопоставляется множество $B_p = \{q : \texttt{p = \&q}\}$

Основная идея

Построить граф ограничений, при добавлении новых ребер и распространении points-to множеств попутно сворачивать циклы в одну вершину.

Основные моменты:

- При построении графа:
  1. Каждой вершине p сопоставляется множество $B_p = \{q : \texttt{p = \&q}\}$
  2. Присваивания вида `*x = y` и `x = *y` называются *сложными* и заносятся в отдельный список $\mathcal{C}$

Основная идея

Построить граф ограничений, при добавлении новых ребер и распространении points-to множеств попутно сворачивать циклы в одну вершину.

Основные моменты:

- При построении графа:
  1. Каждой вершине p сопоставляется множество $B_p = \{q : \texttt{p = \&q}\}$
  2. Присваивания вида `*x = y` и `x = *y` называются *сложными* и заносятся в отдельный список $\mathcal{C}$
- Для каждого ограничения вида `*x = y` *находим возможные адресаты* $\{z\}$ для x и проводим ребро из y в каждый z

Основная идея

Построить граф ограничений, при добавлении новых ребер и распространении points-to множеств попутно сворачивать циклы в одну вершину.

Основные моменты:

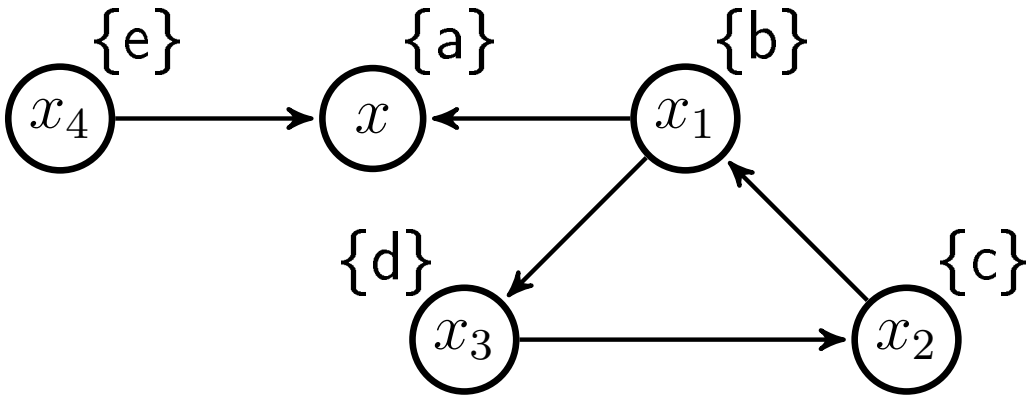
- При построении графа:
  1. Каждой вершине p сопоставляется множество $B_p = \{q : \texttt{p = \&q}\}$
  2. Присваивания вида `*x = y` и `x = *y` называются *сложными* и заносятся в отдельный список $\mathcal{C}$
- Для каждого ограничения вида `*x = y` *находим возможные адресаты* $\{z\}$ для x и проводим ребро из y в каждый z
- Для каждого ограничения вида `x = *y` *находим возможные адресаты* $\{z\}$ для y и проводим ребро из каждого z в $*y$, и ребро из $*y$ в x

Нахождение возможных адресатов

От начальной вершины x проходим в глубину по всем вершинам, из которых имеются ребра в x. При этом собираем множество $\bigcup_{p \in path} B_p$, где $path$ - множество всех пройденных вершин. При проходе в глубину посещенные вершины помечаются, при встрече помеченной вершины циклическая часть графа сливается в одну вершину.

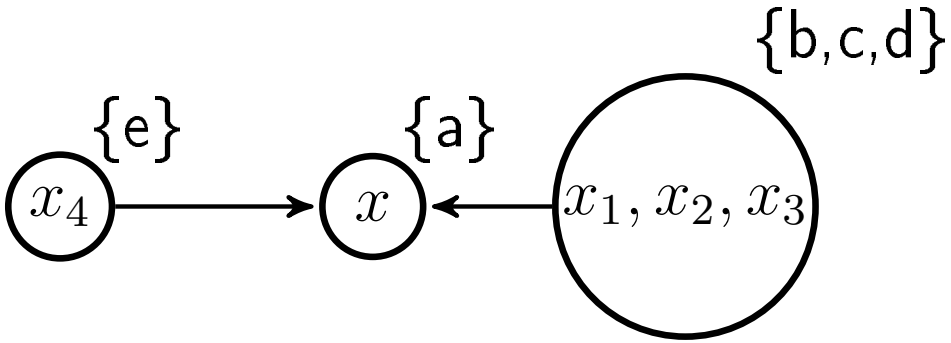
Нахождение возможных адресатов

От начальной вершины x проходим в глубину по всем вершинам, из которых имеются ребра в x. При этом собираем множество $\bigcup_{p \in path} B_p$, где $path$ - множество всех пройденных вершин. При проходе в глубину посещенные вершины помечаются, при встрече помеченной вершины циклическая часть графа сливается в одну вершину.



Нахождение возможных адресатов

От начальной вершины x проходим в глубину по всем вершинам, из которых имеются ребра в x. При этом собираем множество $\bigcup_{p \in path} B_p$, где $path$ - множество всех пройденных вершин. При проходе в глубину посещенные вершины помечаются, при встрече помеченной вершины циклическая часть графа сливается в одну вершину.



Алгоритм Пирса-Келли-Хэнкина (04')

Предложен Дэвидом Пирсом, Полом Келли и Крисом Хэнкином в 2004 году.

Является первым применимым к реальным проектам потоково-чувствительным Andersen-style анализом.

Для упрощения используются другие обозначения для ограничений:

- $V_p \supseteq V_q \equiv p \supseteq q$ (`p = q`)

Для упрощения используются другие обозначения для ограничений:

- $V_p \supseteq V_q \equiv p \supseteq q$ (`p = q`)
- $q \in V_p \equiv p \supseteq \{q\}$ (`p = &q`)

Для упрощения используются другие обозначения для ограничений:

- $V_p \supseteq V_q \equiv p \supseteq q$ (`p = q`)
- $q \in V_p \equiv p \supseteq \{q\}$ (`p = &q`)
- $\forall t \in V_q \Rightarrow V_p \supseteq V_t \equiv p \supseteq *q$ (`p = *q`)

Для упрощения используются другие обозначения для ограничений:

- $V_p \supseteq V_q \equiv p \supseteq q$ (`p = q`)
- $q \in V_p \equiv p \supseteq \{q\}$ (`p = &q`)
- $\forall t \in V_q \Rightarrow V_p \supseteq V_t \equiv p \supseteq *q$ (`p = *q`)
- $\forall t \in V_p \Rightarrow V_t \supseteq V_q \equiv *p \supseteq q$ (`*p = q`)

Для упрощения используются другие обозначения для ограничений:

- $V_p \supseteq V_q \equiv p \supseteq q$ (`p = q`)
- $q \in V_p \equiv p \supseteq \{q\}$ (`p = &q`)
- $\forall t \in V_q \Rightarrow V_p \supseteq V_t \equiv p \supseteq *q$ (`p = *q`)
- $\forall t \in V_p \Rightarrow V_t \supseteq V_q \equiv *p \supseteq q$ (`*p = q`)
- $\forall t \in V_p \Rightarrow q \in V_t \equiv *p \supseteq \{q\}$ (`*p = &q`)

Для упрощения используются другие обозначения для ограничений:

- $V_p \supseteq V_q \equiv p \supseteq q$ (`p = q`)
- $q \in V_p \equiv p \supseteq \{q\}$ (`p = &q`)
- $\forall t \in V_q \Rightarrow V_p \supseteq V_t \equiv p \supseteq *q$ (`p = *q`)
- $\forall t \in V_p \Rightarrow V_t \supseteq V_q \equiv *p \supseteq q$ (`*p = q`)
- $\forall t \in V_p \Rightarrow q \in V_t \equiv *p \supseteq \{q\}$ (`*p = &q`)

Ограничения, содержащие $*$, называются *сложными*.

Также в соответствующей работе были сформулированы формальные *правила* разрешения ограничений.

Также в соответствующей работе были сформулированы формальные *правила* разрешения ограничений.

Правила разрешения:

- Транзитивность: $\dfrac{\tau_1 \supseteq \{\tau_2\}\ \tau_3 \supseteq \tau_1}{\tau_3 \supseteq \{\tau_2\}}$

Также в соответствующей работе были сформулированы формальные *правила* разрешения ограничений.

Правила разрешения:

- Транзитивность: $\dfrac{\tau_1 \supseteq \{\tau_2\}\ \tau_3 \supseteq \tau_1}{\tau_3 \supseteq \{\tau_2\}}$
- Разыменование:

Также в соответствующей работе были сформулированы формальные *правила* разрешения ограничений.

Правила разрешения:

- Транзитивность: $\dfrac{\tau_1 \supseteq \{\tau_2\}\ \tau_3 \supseteq \tau_1}{\tau_3 \supseteq \{\tau_2\}}$
- Разыменование:
  1. $\dfrac{\tau_1 \supseteq *\tau_2\ \tau_2 \supseteq \{\tau_3\}}{\tau_1 \supseteq \tau_3}$

Также в соответствующей работе были сформулированы формальные *правила* разрешения ограничений.

Правила разрешения:

- Транзитивность: $\dfrac{\tau_1 \supseteq \{\tau_2\}\ \tau_3 \supseteq \tau_1}{\tau_3 \supseteq \{\tau_2\}}$
- Разыменование:
  1. $\dfrac{\tau_1 \supseteq *\tau_2\ \tau_2 \supseteq \{\tau_3\}}{\tau_1 \supseteq \tau_3}$
  2. $\dfrac{*\tau_1 \supseteq \tau_2\ \tau_1 \supseteq \{\tau_3\}}{\tau_3 \supseteq \tau_2}$

Также в соответствующей работе были сформулированы формальные *правила* разрешения ограничений.

Правила разрешения:

- Транзитивность: $\dfrac{\tau_1 \supseteq \{\tau_2\}\ \tau_3 \supseteq \tau_1}{\tau_3 \supseteq \{\tau_2\}}$
- Разыменование:
  1. $\dfrac{\tau_1 \supseteq *\tau_2\ \tau_2 \supseteq \{\tau_3\}}{\tau_1 \supseteq \tau_3}$
  2. $\dfrac{*\tau_1 \supseteq \tau_2\ \tau_1 \supseteq \{\tau_3\}}{\tau_3 \supseteq \tau_2}$
  3. $\dfrac{*\tau_1 \supseteq \{\tau_2\}\ \tau_1 \supseteq \{\tau_3\}}{\tau_3 \supseteq \{\tau_2\}}$

В работе сделано обобщение утверждения про циклы на *сильно связные компоненты* графа.
Доказательство данного утверждения аналогично предыдущему.

В работе сделано обобщение утверждения про циклы на *сильно связные компоненты* графа.
Доказательство данного утверждения аналогично предыдущему.

Замечание

Так как сложность всех Andersen-style анализов равна $O(n^3)$, а сильно связные компоненты для графа $G =< V, E >$ могут быть найдены за $O(|V| + |E|)$, то разумным решением будет использование данных алгоритмов поиска при решении.

В результате предыдущих замечаний получаем следующую последовательность действий:

- Сворачиваем вершины из сильно связных компонент, полученных алгоритмом Тарьяна (72‘)

В результате предыдущих замечаний получаем следующую последовательность действий:

- Сворачиваем вершины из сильно связных компонент, полученных алгоритмом Тарьяна (72‘)
- Обходим все вершины графа в топологическом порядке, рассматривая все ограничения, связанные с данной вершиной

В результате предыдущих замечаний получаем следующую последовательность действий:

- Сворачиваем вершины из сильно связных компонент, полученных алгоритмом Тарьяна (72‘)
- Обходим все вершины графа в топологическом порядке, рассматривая все ограничения, связанные с данной вершиной
- Разрешаем их с помощью правил разыменования

В результате предыдущих замечаний получаем следующую последовательность действий:

- Сворачиваем вершины из сильно связных компонент, полученных алгоритмом Тарьяна (72‘)
- Обходим все вершины графа в топологическом порядке, рассматривая все ограничения, связанные с данной вершиной
- Разрешаем их с помощью правил разыменования
- Распространяем множество связанное, с этой вершиной на все вершины, куда есть ребра из рассматриваемой (с помощью правила транзитивности)

В результате предыдущих замечаний получаем следующую последовательность действий:

- Сворачиваем вершины из сильно связных компонент, полученных алгоритмом Тарьяна (72‘)
- Обходим все вершины графа в топологическом порядке, рассматривая все ограничения, связанные с данной вершиной
- Разрешаем их с помощью правил разыменования
- Распространяем множество связанное, с этой вершиной на все вершины, куда есть ребра из рассматриваемой (с помощью правила транзитивности)
- Повторяем до тех пор пока изменения вносятся

Алгоритмы ленивого и гибридного обнаружения циклов (07')

Разработаны и представлены Беном Хардкопфом и Келвином Лином в 2007 году.

Соединяют лучшие идеи предыдущих работ и представляют эвристический подход к поиску циклов.

В данном подходе используется набор ограничений и правила разрешения аналогичные подходу Пирса-Келли-Хэнкина, за исключением ограничения вида $*p \supseteq \{q\}$.

В данном подходе используется набор ограничений и правила разрешения аналогичные подходу Пирса-Келли-Хэнкина, за исключением ограничения вида $*p \supseteq \{q\}$.

Осуществляется также не поиск циклов, а поиск сильно связных компонент (несмотря на название), однако, используется не алгоритм Тарьяна, а его более современная модификация - алгоритм Нуутилы (95‘).

В данном подходе используется набор ограничений и правила разрешения аналогичные подходу Пирса-Келли-Хэнкина, за исключением ограничения вида $*p \supseteq \{q\}$.

Осуществляется также не поиск циклов, а поиск сильно связных компонент (несмотря на название), однако, используется не алгоритм Тарьяна, а его более современная модификация - алгоритм Нуутилы (95‘).

Для реализации множеств используются *бинарные диаграммы решений*.

Алгоритм ленивого обнаружения циклов

Поиск циклов является неотъемлимой частью современных Andersen-style алгоритмов, однако, он проводится на каждой итерации алгоритма, что, *скорее всего*, является избыточным $\Rightarrow$ хотелось бы предъявить некоторую эвристику, чтобы решать, когда стоит запускать поиск циклов.

Алгоритм ленивого обнаружения циклов

Поиск циклов является неотъемлимой частью современных Andersen-style алгоритмов, однако, он проводится на каждой итерации алгоритма, что, *скорее всего*, является избыточным $\Rightarrow$ хотелось бы предъявить некоторую эвристику, чтобы решать, когда стоит запускать поиск циклов.

Основная идея

Будем запускать поиск цикла, *только* если встретили две соседних вершины с одинаковыми points-to множествами. Дважды по одному и тому же ребру поиск циклов не инициируется.

Алгоритм гибридного обнаружения циклов

Было бы хорошо проводить вообще лишь единичный поиск циклов, но это невозможно, т.к. большинство ребер еще не присутствует в графе.

Алгоритм гибридного обнаружения циклов

Было бы хорошо проводить вообще лишь единичный поиск циклов, но это невозможно, т.к. большинство ребер еще не присутствует в графе.
Но можно *заранее* посчитать, что при некотором добавлении элементов в points-to множества будут образовываться циклы, что и реализует данный подход.

Алгоритм гибридного обнаружения циклов

Было бы хорошо проводить вообще лишь единичный поиск циклов, но это невозможно, т.к. большинство ребер еще не присутствует в графе.
Но можно *заранее* посчитать, что при некотором добавлении элементов в points-to множества будут образовываться циклы, что и реализует данный подход.

Основная идея

До начала итеративного алгоритма наряду с графом ограничений строится *расширенный граф ограничений*, в который для ограничений вида $*p \supseteq q$ и $p \subseteq *q$ в граф добавляются вершины $*p$ и $*q$ и соответствующие ребра.
На данном графе и проводится поиск сильно связных компонент.

Рассмотрим найденные компоненты связности подробнее.

Рассмотрим найденные компоненты связности подробнее.

Утверждение 1

Т.к. граф состоял из вершин двух видов: p и $*p$, то и найденная компонента тоже может содержать вершины обоих типов.

Рассмотрим найденные компоненты связности подробнее.

Утверждение 1

Т.к. граф состоял из вершин двух видов: p и $*p$, то и найденная компонента тоже может содержать вершины обоих типов.

Пусть SCC - найденная компонента. Из Утверждения 1 имеем, что $SCC = SCC_p \cup SCC_{*p}$, где SCC содержит лишь элементы вида p, а SCC_{*p} - $*p$ (причем $SCC_p \cap SCC_{*p} = \varnothing$).

Рассмотрим найденные компоненты связности подробнее.

Утверждение 1

Т.к. граф состоял из вершин двух видов: p и $*p$, то и найденная компонента тоже может содержать вершины обоих типов.

Пусть SCC - найденная компонента. Из Утверждения 1 имеем, что $SCC = SCC_p \cup SCC_{*p}$, где SCC содержит лишь элементы вида p, а SCC_{*p} - $*p$ (причем $SCC_p \cap SCC_{*p} = \varnothing$).

Утверждение 2

Несмотря на отсутствие полной информации о вершинах входящих в компоненту связности, можно все вершины, входящие в SCC_p, считать одной вершиной в графе ограничений.

Пусть s - вершина, представляющая множество вершин SCC_p, тогда для всех вершин q из SCC_{*p} в некоторый список $\mathcal{L}$ добавим пару (q, s).

Пусть s - вершина, представляющая множество вершин SCC_p, тогда для всех вершин q из SCC_{*p} в некоторый список $\mathcal{L}$ добавим пару (q, s).

Тогда во время работы итеративного алгоритма при рассмотрении некоторой вершины x, в списке $\mathcal{L}$ ищутся пары вида (x, s). И каждая вершина из V_x сливается с s.

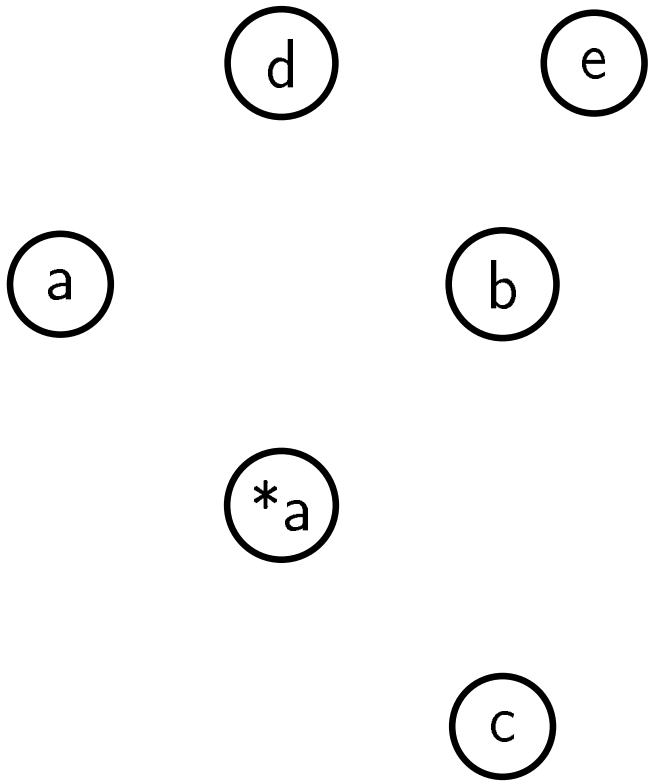
Пусть s - вершина, представляющая множество вершин SCC_p, тогда для всех вершин q из SCC_{*p} в некоторый список $\mathcal{L}$ добавим пару (q, s).

Тогда во время работы итеративного алгоритма при рассмотрении некоторой вершины x, в списке $\mathcal{L}$ ищутся пары вида (x, s). И каждая вершина из V_x сливается с s.

Во время работы итеративного алгоритма используется также и алгоритм ленивого обнаружения циклов.

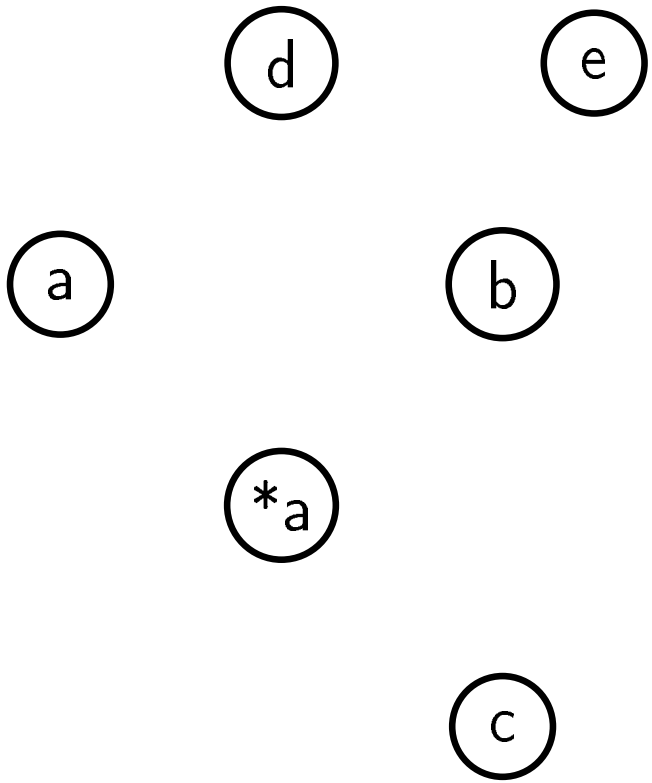
Пример

```
d = &x;
a = &d;
b = *a;
c = b;
e = d;
*a = c;
```



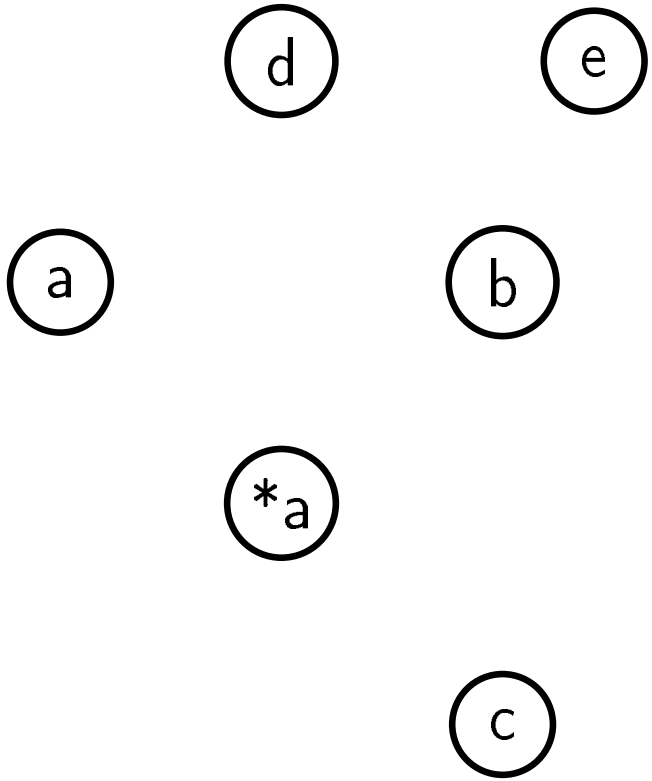
Пример

```
d = &x;
a = &d;
b = *a;
c = b;
e = d;
*a = c;
```



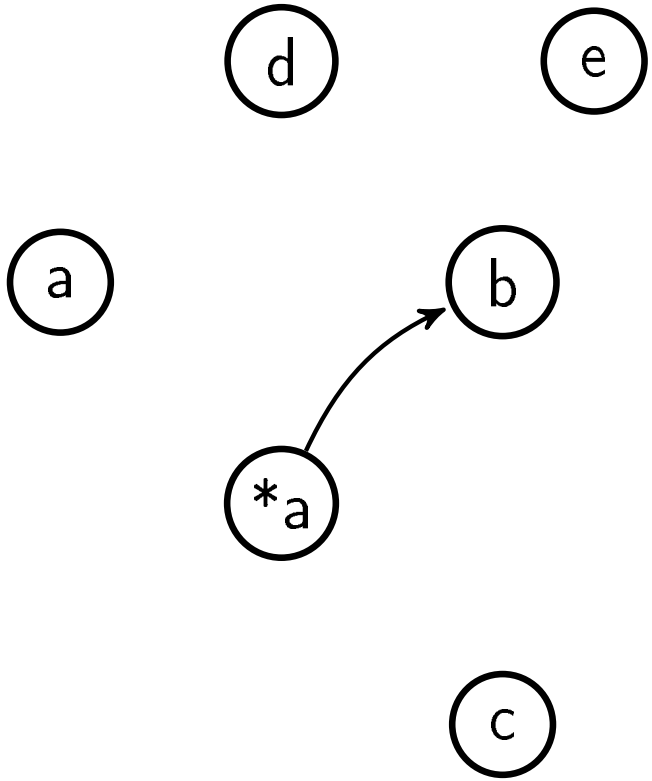
Пример

```
d = &x;
a = &d;
b = *a;
c = b;
e = d;
*a = c;
```



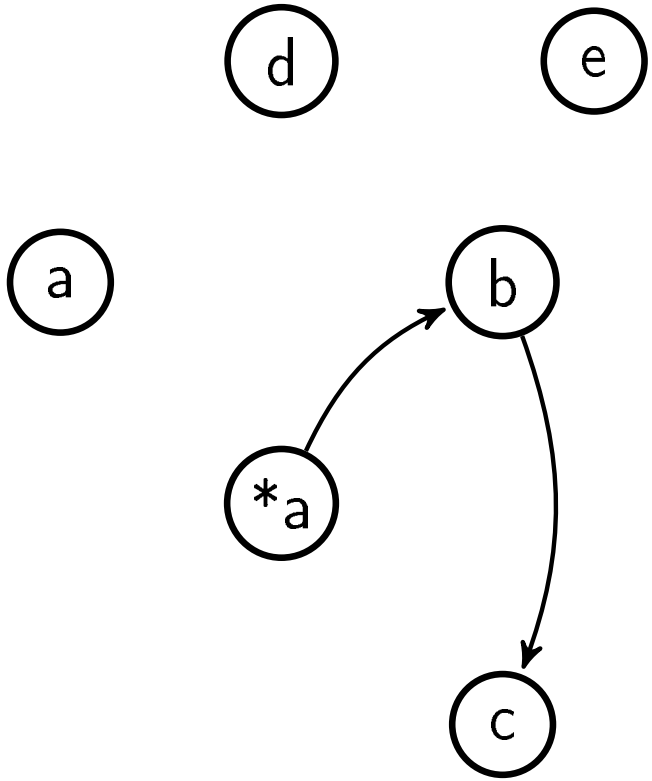
Пример

```
d = &x;
a = &d;
b = *a;
c = b;
e = d;
*a = c;
```



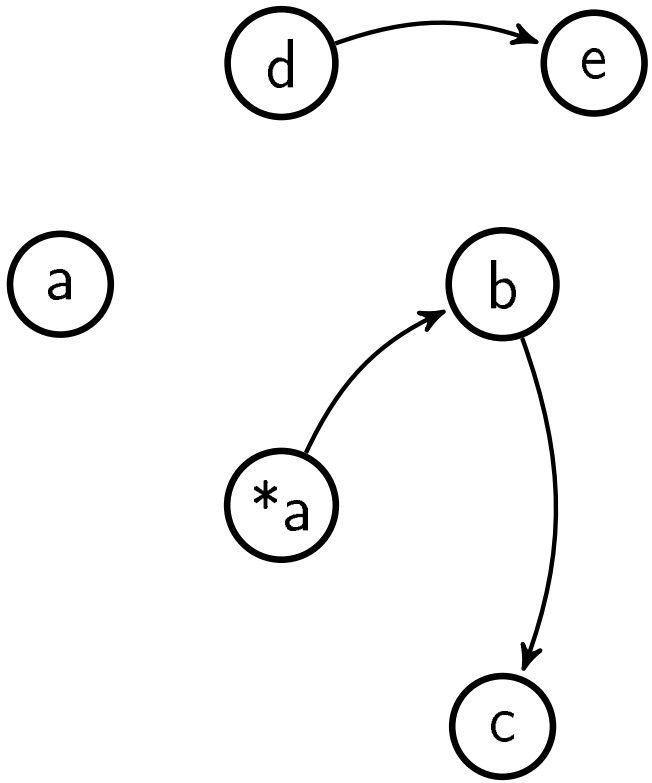
Пример

```
d = &x;
a = &d;
b = *a;
c = b;
e = d;
*a = c;
```



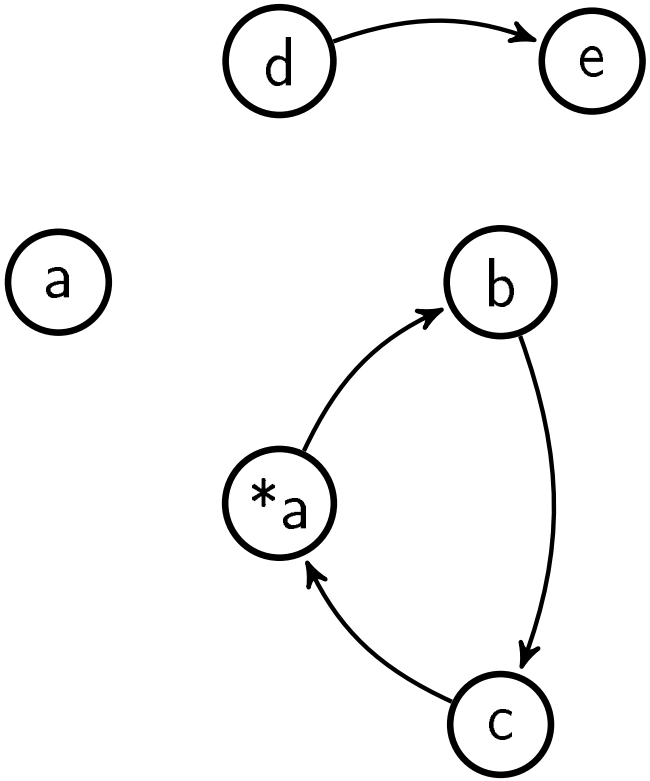
Пример

```
d = &x;
a = &d;
b = *a;
c = b;
e = d;
*a = c;
```



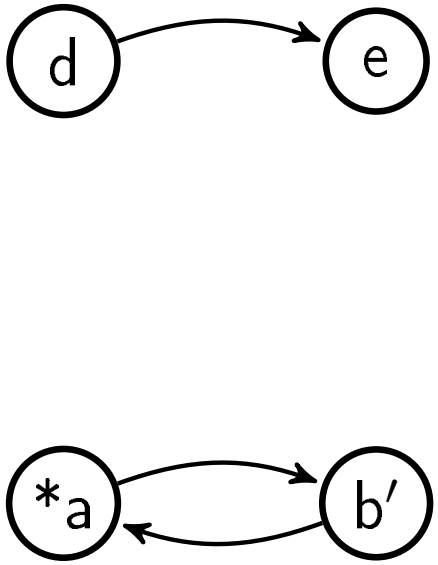
Пример

```
d = &x;
a = &d;
b = *a;
c = b;
e = d;
*a = c;
```



Пример

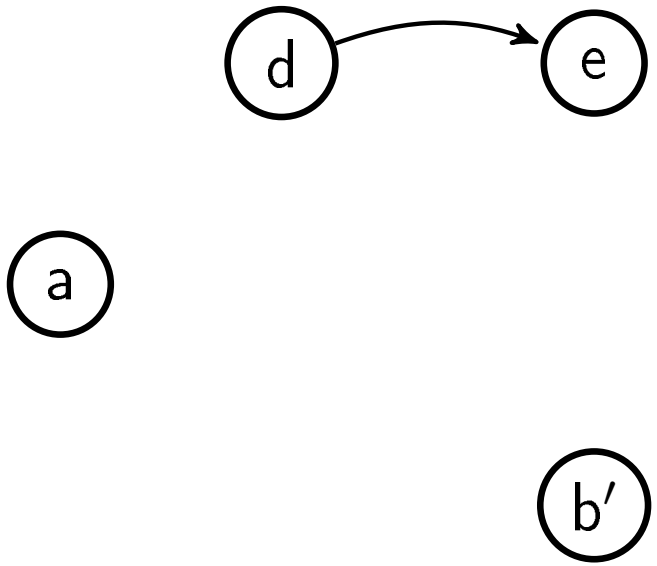
```
d = &x;
a = &d;
b = *a;
c = b;
e = d;
*a = c;
```



$$\mathcal{L} = \{(a, b')\}$$

Пример

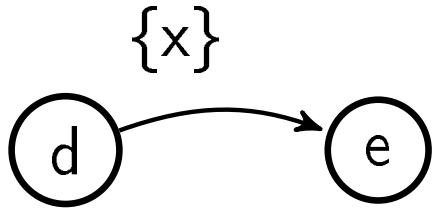
```
d = &x;
a = &d;
b = *a;
c = b;
e = d;
*a = c;
```

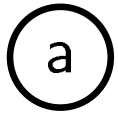


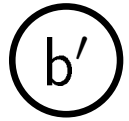
$$\mathcal{L} = \{(a, b')\}$$

Пример

```
d = &x;
a = &d;
b = *a;
c = b;
e = d;
*a = c;
```



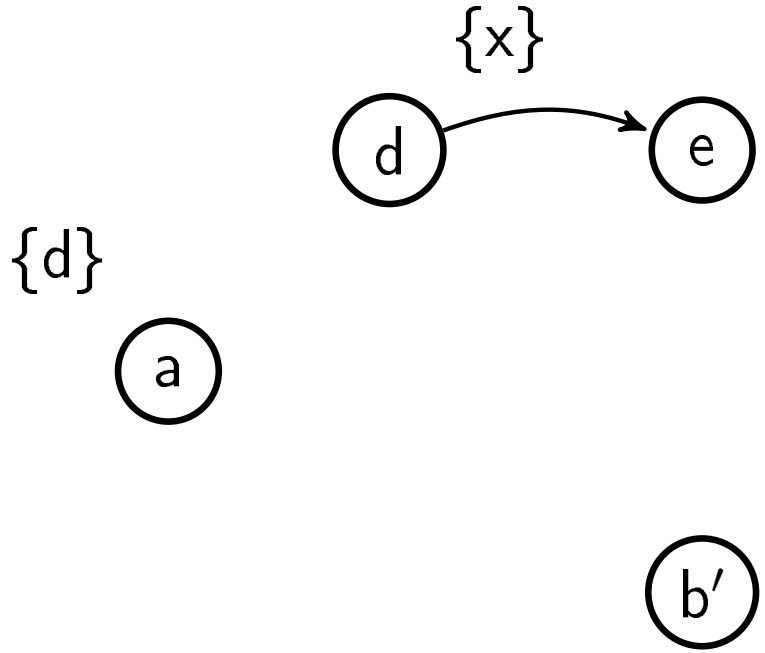




$$\mathcal{L} = \{(a, b')\}$$

Пример

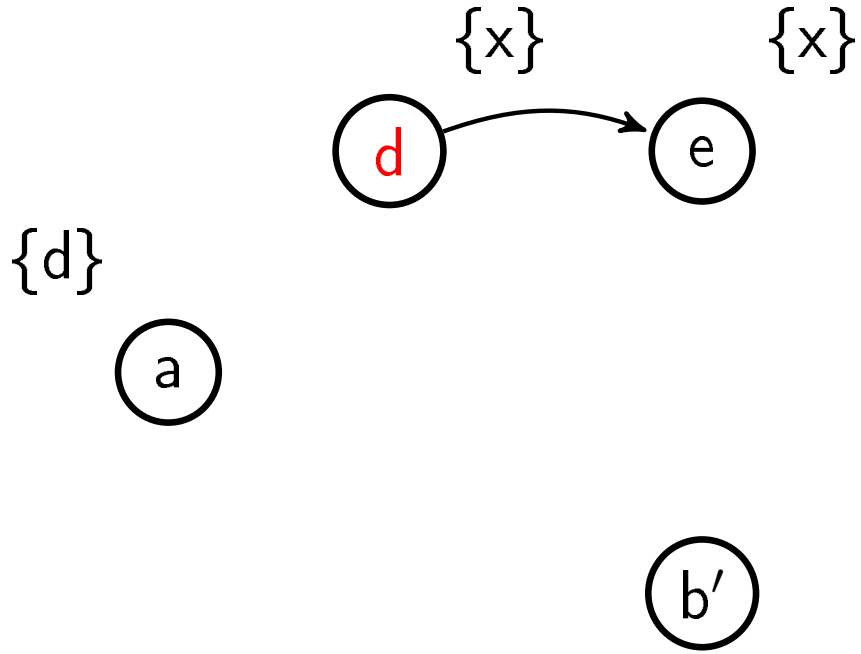
```
d = &x;
a = &d;
b = *a;
c = b;
e = d;
*a = c;
```



$$\mathcal{L} = \{(a, b')\}$$

Пример

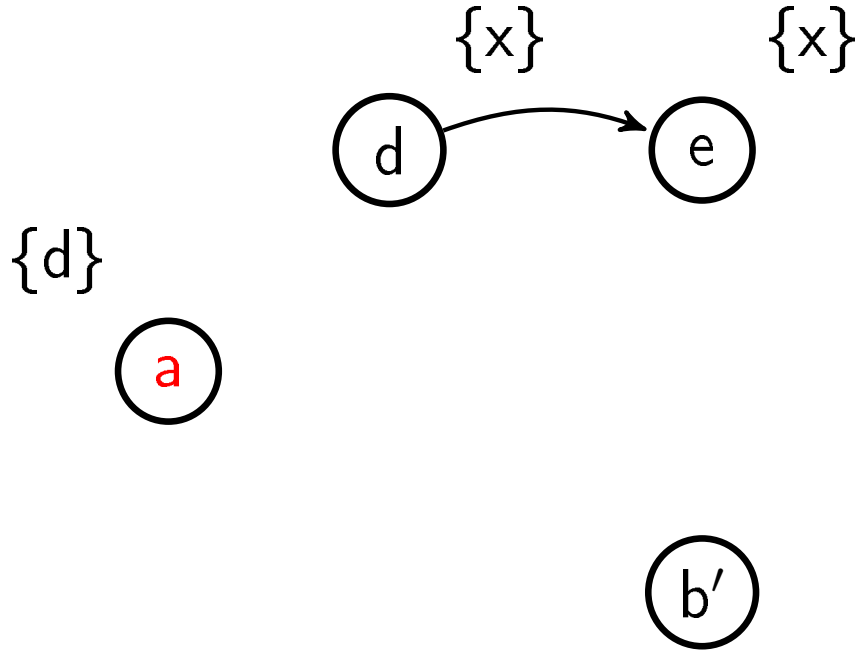
```
d = &x;
a = &d;
b = *a;
c = b;
e = d;
*a = c;
```



$$\mathcal{L} = \{(a, b')\}$$

Пример

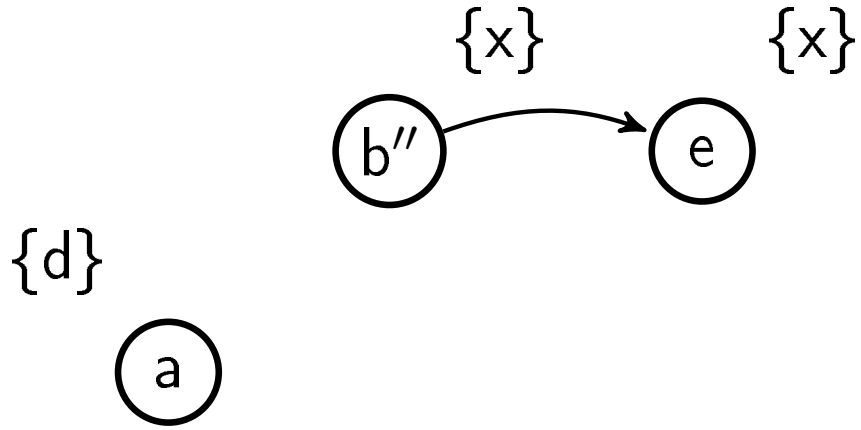
```
d = &x;
a = &d;
b = *a;
c = b;
e = d;
*a = c;
```



$$\mathcal{L} = \{(a, b')\}$$

Пример

```
d = &x;
a = &d;
b = *a;
c = b;
e = d;
*a = c;
```



$$\mathcal{L} = \{(a, b')\}$$

Алгоритмы волнового распространения и распространения в глубину (09')

Предложен Фернандо Перейрой и Даниэлем Бёрлиным в 2009 году.

Отличаются высокой производительностью и совмещением удачных идей предшественников.

Алгоритм волнового распространения

Основная идея

Модифицировать алгоритм Пирса-Келли-Хэнкина с использованием более эффективных используемых алгоритмов и контейнеров данных, а также кэширования промежуточных результатов.

Алгоритм волнового распространения

Основная идея

Модифицировать алгоритм Пирса-Келли-Хэнкина с использованием более эффективных используемых алгоритмов и контейнеров данных, а также кэширования промежуточных результатов.

Данная идея проста, но дала наилучший (в среднем) из до того представленных алгоритмов.

Алгоритм распространения в глубину

Также авторами алгоритмов было отмечено, что несмотря на отставание во многих идеях, алгоритм Хайнтце-Тардьё остается по-прежнему одним из самых производительных.

Алгоритм распространения в глубину

Также авторами алгоритмов было отмечено, что несмотря на отставание во многих идеях, алгоритм Хайнтце-Тардьё остается по-прежнему одним из самых производительных.

Основная идея

Помимо поиска связанных компонент с помощью алгоритма Нуутилы, на этапе распространения значений по ребрам делать это не на глубину 1, а пока возможно, и искать при этом циклы (как в алгоритме Хайнтце-Тардьё).

Сравнение результатов

	WP	DP	HT	LCD	PKH
ex	0.021	0.009	0.020	0.025	0.103
tw	0.022	0.011	0.016	0.061	0.079
pr	0.064	0.032	0.046	0.084	0.338
vt	0.044	0.029	0.028	0.071	0.131
sm	0.148	0.213	0.151	0.269	0.514
gp	0.642	0.397	0.434	0.867	2.064
em	2.407	1.300	2.098	4.094	3.357
pl	1.262	1.326	1.924	4.818	6.683
vm	3.399	2.757	3.926	6.161	21.971
nh	0.277	0.189	0.213	0.441	0.643
gc	1.081	0.880	0.985	3.886	3.84
gs	90.14	244.20	346.24	277.08	301.68
in	131.45	88.24	62.75	90.12	190.42
gd	67.38	103.77	106.10	123.05	283.42
gm	64.69	98.24	102.21	60.38	173.65
wn	1,191.4	2,769.5	2,396.7	1,754.3	4,013.4
lx	1,227.9	605.7	666.61	501.69	1,784.2
Tot	2,783.2	3,916.7	3,709.0	2,817.8	6,565.5

С чего начать?

При начале разработки прежде всего стоит рассмотреть имеющиеся решения и поискать подходящее под необходимые требования.

С чего начать?

При начале разработки прежде всего стоит рассмотреть имеющиеся решения и поискать подходящее под необходимые требования.

Если же разработка неизбежна, то необходимо решить в каком направлении оно будет лучше других решений, а также разработать некоторые *метрики* для определения лучшего решения.

Открытые проблемы

Открытые проблемы

- Масштабируемость

Открытые проблемы

- Масштабируемость
- Улучшение производительности известных алгоритмов

Открытые проблемы

- Масштабируемость
- Улучшение производительности известных алгоритмов
- Улучшение точности известных алгоритмов

Открытые проблемы

- Масштабируемость
- Улучшение производительности известных алгоритмов
- Улучшение точности известных алгоритмов
- Ориентирование на нужды *клиента*

Открытые проблемы

- Масштабируемость
- Улучшение производительности известных алгоритмов
- Улучшение точности известных алгоритмов
- Ориентирование на нужды *клиента*
- Динамические структуры данных в анализируемой программе

Открытые проблемы

- Масштабируемость
- Улучшение производительности известных алгоритмов
- Улучшение точности известных алгоритмов
- Ориентирование на нужды *клиента*
- Динамические структуры данных в анализируемой программе
- Инкрементный анализ

Метрики

Как понять, что один результат *лучше* другого? И что вообще значит *лучше*?

Метрики

Как понять, что один результат *лучше* другого? И что вообще значит *лучше*?

Метрики анализа алиасов:

Метрики

Как понять, что один результат *лучше* другого? И что вообще значит *лучше*?

Метрики анализа алиасов:

- Сравнение с результатами простейшего анализа (предположения в худшем случае)

Метрики

Как понять, что один результат *лучше* другого? И что вообще значит *лучше*?

Метрики анализа алиасов:

- Сравнение с результатами простейшего анализа (предположения в худшем случае)
- Рассчет эффективности не самого анализа алиасов, а анализов-*клиентов*

Метрики

Как понять, что один результат *лучше* другого? И что вообще значит *лучше*?

Метрики анализа алиасов:

- Сравнение с результатами простейшего анализа (предположения в худшем случае)
- Рассчет эффективности не самого анализа алиасов, а анализов-*клиентов*
- Динамические метрики (тестирование, профилирование и т.д.)

Метрики

Как понять, что один результат *лучше* другого? И что вообще значит *лучше*?

Метрики анализа алиасов:

- Сравнение с результатами простейшего анализа (предположения в худшем случае)
- Рассчет эффективности не самого анализа алиасов, а анализов-*клиентов*
- Динамические метрики (тестирование, профилирование и т.д.)

Вывод: все подходы имеют свои недостатки $\Rightarrow$ стоит использовать некоторую их комбинацию, которая более всего подходит рассматриваемой задаче.

Спасибо за внимание!