Este Paper/Articulo presenta dos nuevos algoritmos para realizar análisis de punteros basados en inclusión. *Este análisis es una técnica clave en compiladores y herramientas de análisis estático para determinar qué variables pueden apuntar a qué ubicaciones de memoria durante la ejecución de un programa.*

1. **Método de Propagación por Ondas (Wave Propagation Method):**

* Basado en un enfoque previo de Pearce et al.
* Mejora significativamente el tiempo de ejecución en comparación con el enfoque original.
* Sin embargo, es "hambriento de memoria" (requiere un entorno con abundante memoria).

2. **Método de Propagación Profunda (Deep Propagation Method):**

* Más liviano en cuanto a consumo de memoria.
* Está diseñado para realizar un análisis más eficiente en términos de memoria y simplicidad.

Se ha comparado estos algoritmos con 3 técnicas destacadas:

**Hardekopf-Lin**: conocida por su eficacia.

**Heintze-Tardieu**: otra técnica de referencia en este campo.

**Pearce-Kelly-Hankin**: un enfoque importante en análisis estático.

Los experimentos se llevaron a cabo en 17 benchmarks reconocidos, y los hallazgos fueron:

* **Propagación Profunda (Deep Propagation):**
  + Mejor tiempo de ejecución promedio en todos los benchmarks.
  + Requerimientos de memoria más bajos en términos absolutos.
  + Más rápido para benchmarks con menos de 100,000 líneas de código.
* **Propagación por Ondas (Wave Propagation):**
  + El más rápido en tiempos absolutos para benchmarks grandes, aunque consume mucha memoria.
  + Comparable en velocidad a los algoritmos de análisis de punteros más conocidos en benchmarks extensos.

## ***Introducción***

**Aliasing** ocurre cuando dos variables acceden a las mismas ubicaciones de memoria, ya sea de forma total o parcial.

* Es una característica común en lenguajes imperativos como **C, C++ y Java**, donde es frecuente evitar la copia completa de estructuras de datos grandes al pasar parámetros en llamadas a funciones.
* Ejemplo práctico: en lugar de duplicar un arreglo entero, se pasa una referencia o puntero.

*Aunque es una característica útil, el aliasing tiene un precio:*

* ***Complicaciones para el compilador*:** dificulta el razonamiento sobre el comportamiento del programa.
* ***Impacto en la optimización*:** puede limitar optimizaciones clave como la eliminación de redundancia parcial (*partial redundancy elimination*), que es un enfoque para reducir cálculos repetidos dentro de un programa.

La solución tradicional adoptada por los compiladores para lidiar con este problema es el ***análisis de alias***.

Este análisis es una técnica adoptada por los compiladores para:

1. Identificar qué ubicaciones de memoria **pueden tener alias**.
2. Determinar qué ubicaciones **nunca aliasan**.
3. Reconocer qué ubicaciones **siempre aliasan**.

*Aunque el análisis preciso del alias es un problema* ***NP-completo*** (es decir, muy costoso en términos de tiempo computacional), *incluso análisis imprecisos pueden ofrecer beneficios significativos en la optimización del código*.

* **Optimización agresiva:** Las optimizaciones más avanzadas del compilador suelen requerir un análisis completo del programa (*whole program analysis*), que examina todo el código en lugar de fragmentos aislados.
* **Escalabilidad:** Uno de los mayores desafíos en la última década ha sido lograr que este tipo de análisis sea eficiente para programas de gran tamaño.

RESUMEN

Este trabajo introduce **dos nuevos algoritmos** para realizar análisis de punteros basado en inclusión, siguiendo el estilo de Andersen [1]. *El enfoque de Andersen es un análisis de punteros* ***conservador*** *que infiere todas las posibles relaciones de alias en un programa*.

1. **Método de Propagación por Ondas (Wave Propagation):**
   * Es una evolución del algoritmo presentado por Pearce et al. [13, Fig. 3].
   * Mejoras clave respecto a su predecesor:
     + **Tiempo de ejecución:** Se ha optimizado para ser más rápido en general.
     + **Predecibilidad:** Ofrece un rendimiento más consistente en diversos casos.
     + **Escalabilidad:** Es capaz de manejar programas más grandes de manera eficiente.
2. **Método de Propagación Profunda (Deep Propagation):**
   * Características principales:
     + **Bajo consumo de memoria:** Requiere pocos recursos comparado con otros solucionadores de análisis de punteros.
     + **Tiempo de preprocesamiento reducido:** Minimiza los costos iniciales antes de ejecutar el análisis.
   * **Ventaja:** Resulta especialmente adecuado para programas de tamaño pequeño a mediano (hasta 100,000 líneas de código).

*Ambos algoritmos están diseñados sobre* ***invariantes elegantes****, que son reglas o propiedades que permanecen constantes durante la ejecución del análisis*. Estas invariantes: *- Simplifican el diseño de los algoritmos y - Aseguran que sean competitivos en comparación con los solucionadores más avanzados descritos en la literatura.*

RESUMEN

En la siguiente sección describimos el análisis de puntos con mayor detalle y abordamos algunos trabajos relacionados. En la Sección 3 presentamos el método de propagación de ondas y en la Sección 4 discutimos la técnica de propagación profunda. La Sección 5 describe experimentos que respaldan ambos algoritmos y la Sección 6 concluye este artículo e indica direcciones de investigación futuras.

### **BACKGROUND** (antecedentes(¿))

El análisis de punteros se puede clasificar según su sensibilidad al flujo y al contexto del programa:

1. **Sensibilidad al flujo (flow sensitivity)**:
   * **Análisis insensibles al flujo** [3], [5], [7]**:** Ignoran el orden de las instrucciones en un programa.
     + Ejemplo: Tratan todas las posibles relaciones de alias como si no importara el orden de ejecución.
   * **Análisis sensibles al flujo** [2], [18]**:** Consideran el orden de las instrucciones para obtener resultados más precisos.

**Sensibilidad al contexto (context sensitivity):**

* Los análisis **sensibles al contexto** [17] distinguen entre diferentes contextos de llamada de una función. Por ejemplo, tratan cada invocación de una función de manera separada.
* Los análisis **insensibles al contexto** no hacen distinciones entre los contextos de llamada de una función, lo que reduce la precisión, pero mejora la velocidad.

Los análisis insensibles al flujo y al contexto se dividen a su vez **entre basados en la inclusión** y **basados en la unificación**.

1. **Análisis por inclusión (inclusion-based analysis):**

* *En una asignación como a = b, este enfoque asume que las ubicaciones apuntadas por b son un subconjunto de las ubicaciones apuntadas por a.*
* **Ejemplo práctico:** Si b apunta a {x, y}, entonces a también puede apuntar a {x, y} pero nunca a otras ubicaciones fuera de este conjunto.

2. **Análisis por unificación (unification-based analysis):**

* En lugar de considerar subconjuntos, asume que las ubicaciones apuntadas por a y b son exactamente las mismas. Es más rápido porque sacrifica precisión.
* **Ejemplo representativo:** El algoritmo de Steensgard [15] es el enfoque más conocido en esta categoría.

Aunque los análisis sensibles al flujo y al contexto producen resultados más precisos, para muchos casos prácticos, los análisis insensibles al flujo y al contexto son lo suficientemente precisos, y tienen la ventaja de ser más eficientes.

Por ejemplo:

* **GCC** (GNU Compiler Collection) y **LLVM** son ejemplos de compiladores populares que utilizan análisis **insensibles al flujo** basados en inclusión **y análisis insensibles al contexto**.
* Los algoritmos propuestos en este paper pertenecen a esta misma categoría, lo que los hace altamente relevantes para aplicaciones prácticas.

El análisis de punteros basado en inclusión fue descrito inicialmente en detalle en la disertación de **Andersen [1]**, *que utiliza reglas de tipado para especificar el problema*. Este trabajo seminal ha dado lugar a numerosas investigaciones en dos direcciones principales:

1. **Mejorar la precisión** del análisis de Andersen.
2. **Aumentar la velocidad** de los algoritmos diseñados para resolver conjuntos de restricciones, que es el enfoque adoptado en este artículo.

El análisis de Andersen se basa en un conjunto de restricciones derivadas de:

* Asignaciones entre variables.
* Pasaje de parámetros en el programa que se analiza.

La tabla presentada enumera los **cuatro tipos básicos de restricciones**, que se explican a continuación:

|  |  |  |
| --- | --- | --- |
| **Tipo de declaración** | Nombre | Restriccion |
| a = &b | **Base** | a ⊇ {b} (a apunta a b directamente). |
| a = b | **Simple** | a ⊇ b (a apunta a todo lo que b apunta). |
| a = \*b | **Compleja** | a ⊇ \*b (a apunta a lo que apunta el contenido de b). |
| \*a = b | **Compleja 2** | \*a ⊇ b (lo que apunta a apunta a lo mismo que b). |

**Explicación de las restricciones:**

1. **Restricción base (a = &b):**
   * Si **a** apunta a **b**, se introduce una restricción que indica que el conjunto de ubicaciones apuntadas por **a** incluye a **b**.
2. **Restricción simple (a = b):**
   * Si a se asigna a b, entonces a debe apuntar a todo lo que b apunta.
3. **Restricción compleja 1 (a = \*b):**
   * Si a apunta al contenido al que apunta b (es decir, \*b), entonces se incluye una restricción donde a debe apuntar a lo que apunta el contenido de b.
4. **Restricción compleja 2 (\*a = b):**
   * Si el contenido al que apunta a debe apuntar a b, se incluye una restricción donde lo que \*a apunta ahora incluye a b.

Las restricciones complejas representan **desreferencias de variables** y se interpretan como relaciones más avanzadas entre los conjuntos de punteros de las variables. Hay dos tipos principales:

1. **Restricción a ⊇ \*b** (a = \*b)**:**
   * **Significado:** Si v pertenece al conjunto de punteros de b, entonces el conjunto de punteros de v es un subconjunto del conjunto de punteros de a.
   * **Ejemplo práctico:** Si b apunta a una dirección de memoria que contiene otra variable v, entonces todo lo que v apunta también debe ser considerado como parte de lo que apunta a.
2. **Restricción \*a ⊇ b (\*a = b):**

* **Significado:** Si v pertenece al conjunto de punteros de a, entonces el conjunto de punteros de b es un subconjunto del conjunto de punteros de v.
* **Ejemplo práctico:** Si a apunta a un conjunto de direcciones de memoria, entonces cada dirección debe ser actualizada para incluir a lo que apunta b.

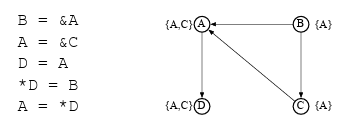
Estas restricciones son clave porque introducen relaciones indirectas entre las variables, necesarias para capturar aliasing en estructuras de datos más complejas como listas enlazadas o árboles.

El problema de análisis de punteros al estilo Andersen se puede formular como un problema de entrada/salida:

* **Entrada:** Un conjunto de restricciones derivadas del código fuente.
  + Estas restricciones pueden ser simples o complejas, como se describió anteriormente.
* **Salida:** Una **asignación conservadora** de variables a sus conjuntos de punteros que satisface todas las restricciones.
  + **"Conservadora"** significa que incluye todas las posibles relaciones de alias, aunque algunas puedan no ocurrir en una ejecución real del programa.

Resolver el problema de análisis de punteros implica calcular el **cierre transitivo** de un grafo de restricciones. Este grafo tiene las siguientes características:

1. **Vértices:**
   * Cada vértice representa una variable del conjunto de restricciones.
2. **Aristas dirigidas:**
   * Hay una arista dirigida de un vértice v a un vértice u si el conjunto de punteros de v es un **subconjunto** del conjunto de punteros de u. La arista dirigida es de v a u.
3. **Cierre transitivo:**
   * Al calcular el cierre transitivo, se garantiza que las relaciones indirectas entre variables (introducidas por desreferencias y asignaciones complejas) también se tengan en cuenta.

g

La figura muestra un **programa simple** y el **grafo de restricciones asociado**, con la solución al problema de puntos-a. Cada línea del programa representa una asignación o relación que introduce una restricción. Estas restricciones son reflejadas en el grafo:

**Programa:**

1. B = &A
   * Restricción base: B ⊇ {A} → El conjunto de punteros de B incluye A.
2. A = &C
   * Restricción base: A ⊇ {C} → El conjunto de punteros de A incluye C.
3. D = A
   * Restricción simple: D ⊇ A → El conjunto de punteros de D incluye todo lo que apunta A.
4. \*D = B
   * Restricción compleja 2: \*D ⊇ B → El contenido apuntado por D incluye lo que apunta B.
5. A = \*D
   * Restricción compleja 1: A ⊇ \*D → El conjunto de punteros de A incluye lo que apunta el contenido de D.

**Grafo de restricciones:**

El grafo asociado tiene:

* **Vértices:** Una variable por nodo (A, B, C, D).
* **Aristas dirigidas:** Cada arista refleja una relación de inclusión entre los conjuntos de punteros. Por ejemplo:
  + De B a A porque B ⊇ {A}.
  + De A a C porque A ⊇ {C}.
  + De D a A porque D ⊇ A.

El análisis de punteros generalmente se resuelve de manera iterativa:

**Propagación:** Las restricciones complejas (\*D = B, A = \*D) añaden nuevas aristas al grafo, lo que fuerza la propagación de los conjuntos de puntos-a entre los nodos.

**Repetición:** Este proceso continúa hasta que no se detecten más cambios en el grafo (es decir, cuando se alcanza un estado estable).

En la figura, los conjuntos de puntos-a asociados a cada nodo representan la solución final del problema.

**Optimización mediante detección de ciclos**

Para escalar el análisis de punteros a programas más grandes, es crucial identificar y manejar **ciclos** en el grafo de restricciones:

* **Importancia de los ciclos:**
  + Todos los nodos dentro de un ciclo comparten el mismo conjunto de puntos-a. Por lo tanto, se pueden **colapsar** en un único nodo, reduciendo la complejidad del grafo.
* **Contribuciones clave:**
  + Fahndrich et al. [3] diseñaron un algoritmo para detectar y eliminar ciclos mientras se procesan restricciones complejas. Desde ahí, se han propuesto muchos algoritmos nuevos.
  + Heintze y Tardieu [7] describen un algoritmo que puede analizar programas C con más de un millón de líneas de código en unos pocos segundos.
  + Este enfoque permitió un avance significativo en la escalabilidad del análisis.

Hardekopf y Lin introdujeron en 2007 dos técnicas significativas para mejorar los algoritmos de análisis de puntos-a:

* **Lazy Cycle Detection (Detección de ciclos perezosa):**  
  Detecta ciclos en el grafo de restricciones solo cuando es necesario, es decir, cuando un ciclo podría afectar el resultado. Esto optimiza el uso de memoria y evita trabajo innecesario.
* **Hybrid Cycle Detection (Detección de ciclos híbrida):**  
  Combina enfoques de detección de ciclos en línea y fuera de línea para lograr un balance entre eficiencia y precisión. Esto resulta en un análisis más escalable.

Además de la detección de ciclos en línea, los análisis de puntos a dependen del preprocesamiento de restricciones para la escalabilidad.

El preprocesamiento es crucial para la escalabilidad de los análisis de puntos-a. Dos métodos importantes son:

* **Off-Line Variable Substitution (Sustitución de variables fuera de línea):**  
  Simplifica las restricciones antes de ejecutar el análisis principal al sustituir variables redundantes por equivalentes más simples. Esto reduce el tamaño del grafo de restricciones.
* **HVN (Harris-Van-Horn):**  
  Un conjunto de algoritmos diseñados para simplificar aún más las restricciones al identificar variables equivalentes y combinarlas en una sola. Esto reduce significativamente la complejidad del grafo.

Ambas técnicas son utilizadas ampliamente en compiladores de producción, ya que simplifican el problema y mejoran la escalabilidad.

Los avances descritos se han integrado en compiladores populares:

* **GCC (GNU Compiler Collection):**  
  Utiliza los algoritmos diseñados por Pearce et al. como núcleo de su solucionador de puntos-a. Además, implementa detección de ciclos fuera de línea y sustitución de variables para optimizar el análisis.
* **LLVM:**  
  Su solucionador de puntos-a fue implementado siguiendo las ideas de Hardekopf y Lin (Lazy Cycle Detection y Hybrid Cycle Detection), lo que lo hace eficiente y escalable.

## **3. Metodo Weve Propagation**

La sección introduce el **método Wave Propagation**, una modificación del algoritmo propuesto por Pearce et al. *Este método se enfoca en optimizar la manera en que se procesan las restricciones en el grafo de puntos-a*.

El método Wave Propagation **separa dos procesos clave** que antes estaban mezclados en el algoritmo original de Pearce:

* **Inserción de nuevas aristas** en el grafo de restricciones.
* **Propagación de conjuntos de puntos-a.**

Esta separación permite tratar el grafo de una manera más eficiente y ordenada, particularmente eliminando ciclos antes de la propagación.

La propagación de la información de puntos-a (el núcleo del algoritmo) **ocurre solo en un grafo de restricciones acíclico**. Esto tiene varias ventajas:

* **Orden topológico:** Al eliminar ciclos, los nodos del grafo se pueden procesar en un orden topológico. Esto significa que cada nodo solo se procesa después de que sus predecesores han sido procesados.
* **Optimización de propagación:** En lugar de propagar todo el conjunto de puntos-a de un nodo a otro, **solo se propagan las diferencias** entre los conjuntos de puntos-a. Esto reduce significativamente el trabajo redundante.

Al final de la propagación, se garantiza la siguiente propiedad:

* El conjunto de puntos-a de un nodo v **incluye los conjuntos de puntos-a de todos los nodos n** que lo preceden en el grafo de restricciones.

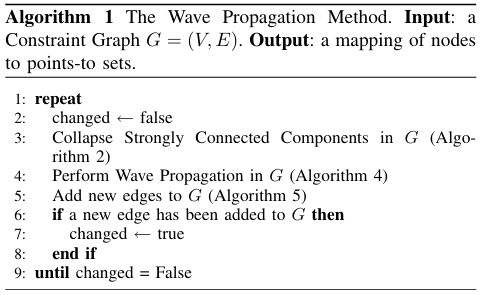
**Este invariante** asegura que la información de puntos-a fluya correctamente a través del grafo.

El algoritmo Wave Propagation opera en **tres fases**, que se repiten hasta que el grafo ya no cambie más:

1. **Colapso de ciclos:** Identificación y eliminación de ciclos en el grafo de restricciones. Los nodos en un ciclo **se combinan en un solo nodo** porque *comparten el mismo conjunto de puntos-a*.
2. **Propagación de puntos-a:** Utilizando el orden topológico del grafo acíclico, **se propaga la información de puntos-a entre nodos**. Solo se propagan los cambios (diferencias entre conjuntos).
3. **Inserción de nuevas aristas:** Si las restricciones complejas generan nuevas relaciones entre nodos, se añaden como aristas al grafo.

Estas tres fases se repiten iterativamente hasta que no se detectan más cambios en el grafo de restricciones.

ALGORITMO 1

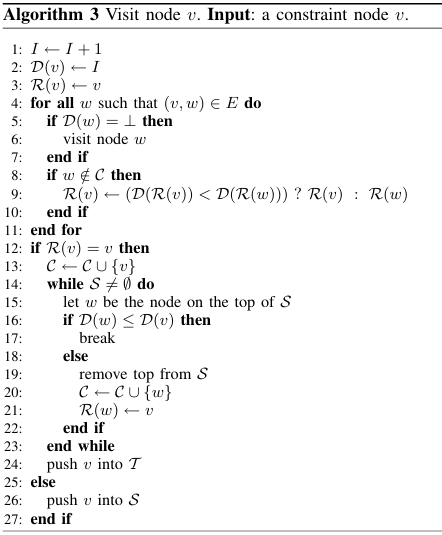
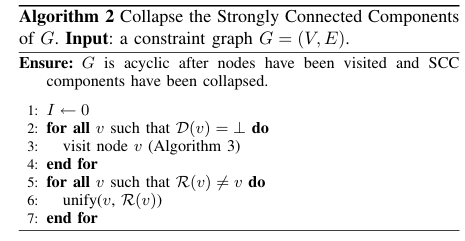


El algoritmo presentado define el flujo principal del método Wave Propagation, dividiendo el proceso en varias fases clave que se repiten hasta que no haya más cambios en el grafo de restricciones.

* **Entrada:** Un grafo de restricciones G=(V,E), donde los nodos representan variables y las aristas representan relaciones de puntos-a derivadas de las restricciones.
* **Salida:** Un mapeo de los nodos del grafo a sus conjuntos de puntos-a, que satisface las restricciones dadas.

**Verificación de cambios (líneas 6-8):**

* Si se agregan nuevas aristas, se establece: changed ← true
* Esto asegura que el algoritmo vuelva a procesar el grafo en la siguiente iteración.

Este fragmento describe dos fases principales: **la identificación y colapso de ciclos** (para encontrar las Componentes Fuertemente Conectadas, SCCs) y **la propagación de puntos-a** (wave propagation).

El pseudocódigo para esta fase se muestra en los algoritmos 2 y 3. Usamos las siguientes estructuras de datos:

**Estructuras de Datos Utilizadas**

1. **D (mapa para el orden de visita):**
   * Representa un mapeo de nodos V a números enteros {1,…,∣V|} ∪ ⊥
   * Esencialmente, almacena el orden en que cada nodo es visitado durante la ejecución del algoritmo de Nuutila.
   * Inicialmente, D(v)=⊥ (es decir, sin visitar).
2. **R - Mapa de V a V (representantes de ciclos):**
   * Mapea cada nodo en un ciclo al "representante” de ese ciclo.
   * Los nodos que están en el mismo ciclo compartirán el mismo representante.
   * Inicialmente, R(v)=v, asumiendo que cada nodo es su propio representante.
3. **C - Subconjunto de V (nodos colapsados):**
   * contiene los nodos que son parte de un componente fuertemente conectado conocido.
   * Inicialmente vacío (C=∅)
4. **S - Pila de V (pila de nodos en un ciclo):**
   * Almacena nodos que podrían pertenecer a un ciclo, pero que aún no han sido insertados en C.
   * Inicialmente vacía.
5. **T - Pila de V (pila para el orden topológico)**
   * Contiene los nodos de V que son representantes de componentes fuertemente conectados.
   * Contiene los representantes de las SCCs en orden topológico.
   * El nodo en la cima de T no tiene predecesores (es decir, no recibe aristas desde ningún nodo). (Osea T mantiene los nodos en orden topologico).
   * Inicialmente vacía.

**Colapso de Ciclos (Identificación de SCCs)**

* En esta fase, los ciclos en el grafo de restricciones se identifican y colapsan.
* Esto asegura que cada ciclo completo se represente como un único nodo en el grafo.
* El algoritmo de Nuutila, una mejora sobre el algoritmo de Tarjan, se utiliza aquí por su eficiencia (tiempo lineal respecto al número de aristas).

**Flujo general:**

* Cada nodo v se visita en un orden determinado (almacenado en D).
* Si v pertenece a un ciclo, todos los nodos de ese ciclo se agrupan bajo un representante común, almacenado en R.
* Los nodos colapsados en C ya no necesitan ser procesados nuevamente.
* Al final de esta fase, T contiene los representantes de todas las SCCs en orden topológico.

**Propagación de Puntos-a (Wave Propagation)**

Después de colapsar los ciclos, realizamos una ronda de *propagación de onda (puntos-a), que para cualquier nodo v, envia el conjunto de puntos-a de v a todos los vecinos.*

*Si el grafo es acíclico y el orden de propagación es el orden topologico de los nodos, entonces:* el orden topológico garantiza que cualquier nodo w al que se pueda llegar desde un nodo v contendrá el conjunto de puntos a los que apuntar del nodo v, ósea (*que los predecesores de un nodo se procesen antes que el nodo mismo*).

Afortunadamente, el orden topológico de los nodos de restricción, almacenados en la pila T, es un subproducto de la técnica de Nuutila (ver línea 24 del Algoritmo 3). La fase de propagación de onda se detalla en el Algoritmo 4. Nuestro algoritmo almacena dos conjuntos de puntos a para cada nodo v.

**Detalles de la Propagación: NOOOO ENTENDIIIII**

1. **Conjuntos de puntos-a:**
   * Cada nodo v tiene dos conjuntos asociados:
     + Pcur(v)): Conjunto actual de puntos-a de v. es el resultado del análisis de punteros para v, de la propagación cuando el algoritmo 1 deja de iterar.
     + Pold(v): Subconjunto que contiene la información de puntos a que se ha enviado desde v desde la última iteración de la propagación de la onda.
   * Realizamos un seguimiento de Pold(v) para evitar volver a enviar todo el conjunto de puntos actuales de v durante cada iteración de nuestro algoritmo.
   * Se mantiene la relación Pold(u) ⊆ Pcur(v) para cualquier arista (u,v), en el grafo de restricciones.

**Fase Completa**

1. **Colapso de Ciclos:**
   * Identifica ciclos en el grafo y los colapsa en un único nodo (representante).
   * Usa las pilas S y T para gestionar los nodos en ciclos y mantener el orden topológico.
2. **Propagación de Puntos-a:**
   * Propaga información en el grafo acíclico, usando diferencias (Δ\DeltaΔ) para evitar trabajo redundante.
   * Aprovecha el orden topológico generado durante el colapso de ciclos.

En la ultima fase del algoritmo propuesto, añadimos nuevas aristas al grafico de restricciones.

* **Se agregan nuevas aristas al grafo de restricciones.**  
  Esto ocurre debido a que se evalúan las **restricciones complejas**, que son aquellas que involucran desreferencias de punteros (como a ⊇ \*b o \*a ⊇ b). Estas restricciones pueden generar relaciones adicionales entre nodos que no se detectaron al principio, cuando se procesaron restricciones más simples.

Este paso se ilustra en el algoritmo 5.

Llevamos un registro de Pcache(c), la última colección de puntos utilizados en la evaluación de la restricción compleja c.

* **¿Qué es Pcache(c)?** Es una estructura que guarda el último conjunto de puntos-to (points-to set) que se usó para evaluar una restricción compleja c.
* **¿Para qué sirve?** Para reducir el número de aristas que deben verificarse o agregarse. Si no hay cambios en los puntos-to sets asociados a c, no se generan nuevas aristas.

Esta optimización reduce la cantidad de aristas que se deben comprobar para su inclusión en G.

* **Cuando se añade una nueva arista (u, v):**
* Se copia el conjunto de puntos-to **Pold(u)** (del nodo origen u) al conjunto de puntos-to actual **Pcur(v)** (del nodo destino v).
* **¿Por qué se hace esto?** Para mantener una propiedad importante del grafo: la información de puntos-to fluye correctamente desde u hacia v. Esto garantiza que la relación entre los nodos refleje todas las dependencias que impone la restricción compleja.

Tomando en cuenta este programa, ilustraremos los algoritmos presentados en este documento.

H = &C E = &G B = C

H = &G H = A C = B

A = &E F = D B = A

D = \*H \*E = F F = &A

* El orden en el que se ejecutan las sentencias no es importante para nosotros, porque nuestro análisis no es sensible al flujo.
* Declaraciones como D = \*H y \*E = F generan **restricciones complejas**, porque involucran desreferencias de punteros. Estas requieren pasos adicionales para determinar cómo afectan los puntos-to sets.

La Figura 2 describe la primera iteración del Algoritmo 1 en ese programa. Solo se muestra el conjunto de puntos a los que apunta cada nodo. Durante la búsqueda de **componentes fuertemente conectados (SCC)** en el grafo de restricciones, se identifica que los nodos B y C forman un ciclo.

* Las instrucciones B = C y C = B generan un ciclo porque ambos nodos se refieren mutuamente.

El algoritmo **colapsa** el ciclo en un único nodo representativo, lo que simplifica el grafo y lo transforma en un grafo **acíclico**.

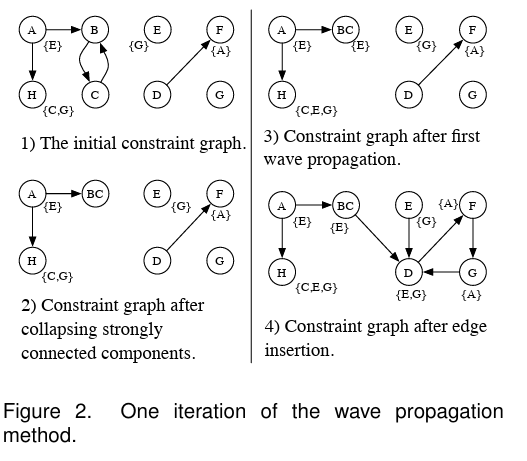
En el siguiente paso, una vez colapsados los ciclos, se realiza una propagación de los conjuntos de puntos-to en el grafo acíclico.

**¿Qué significa esto?**

* Por ejemplo, si H apunta a C y D es igual a \*H, entonces el conjunto de puntos-to de C se propaga a D. Así, D termina apuntando a los mismos elementos que C.

Finalmente, el algoritmo 5 crea nuevas aristas en el grafo de restricciones:

* D = \*H: Esto indica que cualquier cosa a la que H apunte también debe ser parte del conjunto de puntos-to de D.
* \*E = F: Esto implica que cualquier cosa a la que F apunte debe ser parte del conjunto de puntos-to de las direcciones que E contiene.



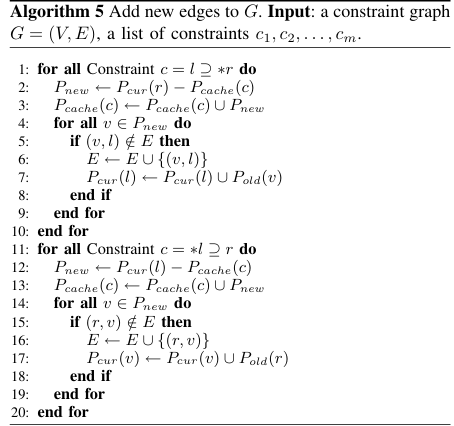
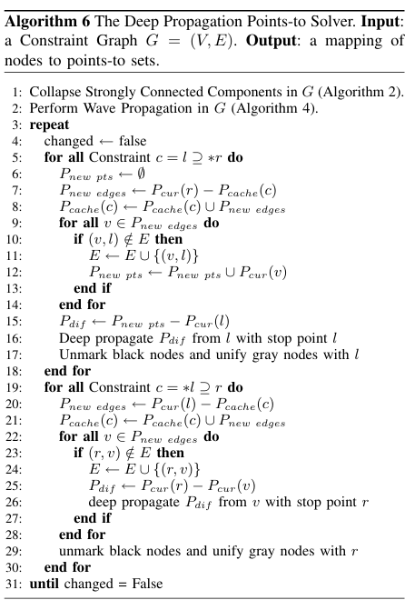
El análisis finalizará en la siguiente iteración, que omitimos en el ejemplo.

Analisis de Complejidad: El colapso de componentes fuertemente conectados es O(V^2) para grafos densos, donde V es el número de nodos en el grafo de restricción. La fase de propagación de onda y la inserción de nuevas aristas son O(V^3)

## **4. Método Deep Propagation**

**4. Deep Propagation: Contexto**

1. **Problema del método anterior (Wave Propagation):**
   * Durante la propagación de ondas, se guarda un caché con información de puntos-a ya procesada para cada nodo y para cada restricción.
   * Este caché consume mucha memoria, especialmente en grafos grandes y densos.
2. **Solución propuesta (Deep Propagation):**
   * El algoritmo garantiza que, si un nodo W es alcanzable desde un nodo V, entonces el conjunto de puntos-a de W contiene el conjunto de puntos-a de V.
   * Esta propiedad se asegura después de:
     1. **Colapso de componentes fuertemente conectados (SCCs).**
     2. **Propagación de ondas inicial.**
     3. Como se muestra en el algoritmo 6.
3. **Problema con la inserción de aristas (Algorithm 5):**
   * Aunque la propiedad mencionada es válida después de la propagación de ondas, deja de serlo cuando se agregan nuevas aristas al grafo (Algoritmo 5).



1. **Deep Propagation corrige esto:**
   * Asegura que la relación de puntos-a entre nodos se mantiene consistente incluso tras la inserción de nuevas aristas.

En el algoritmo 6, calculamos, para cada restricción compleja, el conjunto de nodos que deben propagarse en profundidad a través del grafo de restricciones.

El objetivo del método es:

* Dado un nodo inicial v y un conjunto de diferencias en puntos-a (Pdif), propagar Pdif ​ al conjunto de puntos-a de v y también a todos los nodos que sean alcanzables desde v en el grafo de restricciones.

**Estructura del Algoritmo 6**

El algoritmo se divide en dos partes, dependiendo del tipo de restricción compleja:

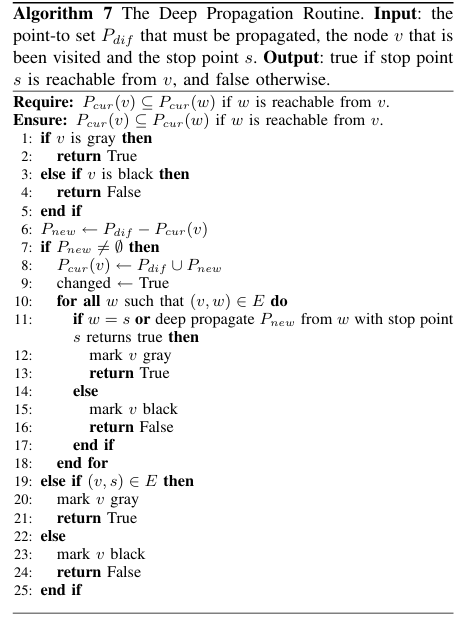
**Parte 1: Restricciones del tipo l ⊇ ∗r (Líneas 5–18)**

* Dada esta restricción, nuestro algoritmo calcula el nuevo conjunto de puntos a los que apuntar Pdif que debe propagarse desde el nodo l.
* Estas restricciones expresan que **el conjunto de puntos-a de l** debe incluir **los elementos apuntados por r**.
* El conjunto de puntos a los que apuntar de cada variable v recientemente añadido al conjunto de puntos a los que apuntar del nodo r contribuye a Pdif.
* Pasos clave:
  1. **Optimización mediante el invariante:**
     + Debido al invariante del algoritmo, los nodos alcanzables desde l ya contienen el conjunto actual de puntos-a de l.
     + Por lo tanto, se elimina Pcur​(l) de Pdif ​ en la línea 15, antes de comenzar la propagación profunda.
  2. **Propagación profunda:**
     + Pdif ​ se añade al conjunto de puntos-a de l y a todos los nodos alcanzables desde l.
     + Se asegura que no se procesen nuevamente nodos o aristas ya incluidas.

**Parte 2: Restricciones del tipo ∗l ⊇ r (Líneas 19–30)**

* Estas restricciones indican que **cada nodo en el conjunto de puntos-a de r** debe ser incluido en **el conjunto de puntos-a de todos los nodos apuntados por l**.
* Pasos clave:
  1. **Cálculo del conjunto Pdif ​:**
     + Se identifican los nodos recientemente añadidos al conjunto de puntos-a de l (línea 20).
     + Estos nodos determinan qué nuevas conexiones deben propagarse desde r.
  2. **Propagación profunda a los nodos de l:**
     + Para cada nodo v recientemente agregado en el conjunto de puntos-a de l:
       - Se asegura que Pcur(r) (conjunto de puntos-a de r) sea incluido en Pcur(v) pero únicamente para los nodos que no estén ya presentes.
     + Se añaden las aristas correspondientes al grafo de restricciones.
  3. **Control de redundancia:**
     + Como en el método de propagación de onda, se utiliza Pcache(c) un caché de puntos-a ya procesados para cada restricción c, para evitar procesar aristas ya añadidas al grafo.

El núcleo del método de propagación profunda (**Deep Propagation**) es un procedimiento recursivo detallado en el **Algorithm 7**, el cual tiene como objetivo garantizar que el conjunto Pdif​ se propague al conjunto de puntos-a de todos los nodos alcanzables desde un nodo v. Este proceso aprovecha optimizaciones derivadas de la invariante del algoritmo, lo que permite reducir la cantidad de nodos visitados y optimizar la propagación. A continuación, explico los elementos principales del algoritmo y su lógica:



El procedimiento recibe tres parámetros clave:

* **v**: El nodo desde el cual se inicia la propagación.
* **Pdif​**: El conjunto de diferencias en puntos-a que debe propagarse.
* **s**: El nodo llamado **stop point (punto de parada)**, que se utiliza para identificar ciclos.

**Objetivo de la Propagación Profunda**

* Asegurar que Pdif​ forme parte del conjunto de puntos-a de v y de todos los nodos alcanzables desde v.
* Detener la propagación en nodos que ya contengan Pdif​, lo cual evita redundancia gracias al **invariante** del algoritmo.

**Optimización del Recorrido**

* **Detención Anticipada del Recorrido:**
  + La propagación no necesita visitar todos los nodos alcanzables desde v. Si se encuentra un nodo que ya contiene Pdif​, el recorrido puede detenerse, debido a nuestro invariante.
  + Esta optimización es posible gracias al **invariante**: garantiza que los nodos alcanzables ya contienen la información actual de v, eliminando redundancias.
* **Reducción del Tamaño de Pdif:**
  + Durante llamadas sucesivas de propagación profunda, el tamaño de Pdif se reduce dinámicamente. Esto se realiza en la **línea 6 del Algoritmo 7**, donde Pdif ​ se actualiza eliminando los elementos ya presentes en Pcur(v).
  + Al calcular esta diferencia "sobre la marcha", se evita el uso de conjuntos adicionales como PoldP ​, lo que simplifica el manejo de memoria y mejora la eficiencia.

**Identificación de Ciclos**

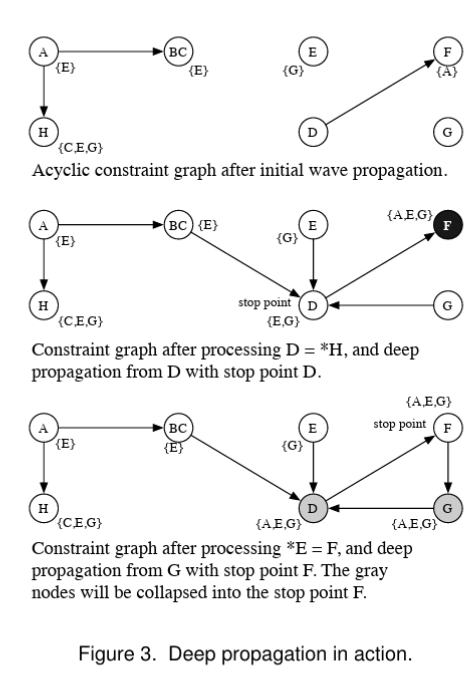
* **Punto de Parada (Stop Point):**
  + Se introduce un nodo llamado **punto de parada** para identificar ciclos en el grafo.
  + Este nodo se establece en las líneas **16 y 26 del Algoritmo 6**, marcando el inicio efectivo de la propagación profunda.
  + Si el punto de parada es alcanzado durante una llamada recursiva, esto indica la **presencia de un ciclo** en el grafo. Esto es fundamental para detectar y manejar dependencias cíclicas.

Las operaciones de conjunto como las que se ejecutan en las operaciones con conjuntos en las líneas 6 y 8 del Algoritmo 7, como calcular la diferencia (Pdif − Pcur(v)) y unir conjuntos, son lineales con respecto al número de nodos del grafo.

Para reducir la cantidad de operaciones de conjunto, se introduce un sistema de **colores.**

* **Gray (gris):** Un nodo v, se marca como gris si existe un camino desde el nodo hasta el **punto de parada**.
* **Black (negro):** Un nodo se marca como negro si no hay camino hacia el punto de parada.
* **Aplicación de operaciones:** Las operaciones de conjuntos solo se realizan en nodos que no están coloreados (sin etiquetas).

Esto ahorra cálculos innecesarios al evitar procesar nodos que ya tienen la información propagada o que no necesitan ser actualizados.



**3. Ejemplo de la Figura 3**

**Primera Iteración:**

* Después de la propagación por ondas inicial, el grafo es acíclico.
* Se procesa la asignación D=∗HD = \*HD=∗H, y la propagación profunda comienza desde D, con D como **punto de parada**.
  + Los nodos alcanzables desde D (en este caso, F) reciben el conjunto de puntos de D.

**Segunda Iteración:**

* Se procesa la asignación \*E = F, y la propagación comienza desde G, con F como **punto de parada**.
  + Los nodos alcanzables desde G reciben el conjunto propagado de G.
  + Los nodos involucrados se colorean según sus caminos hacia el punto de parada.

5. Resultados experimentales

Hemos realizado una serie de experimentos para verificar la eficiencia de los algoritmos propuestos. Hemos realizado pruebas en dos máquinas diferentes.

Comparamos nuestro algoritmo con los tres solucionadores basados ​​en inclusión implementados por Ben Hardekopf y utilizados anteriormente en [5].

Los algoritmos que hemos utilizado, todos ellos disponibles gratuitamente en línea, se enumeran a continuación:

* Detección de ciclos diferido (Lazy Cycle Detection) [5] (LCD): este es el algoritmo más escalable para resolver análisis de punteros-a basados en inclusiones. Básicamente, este algoritmo busca ciclos cuando detecta que el conjunto de punteros-a de un nodo v es igual al conjunto de puntos a de uno de sus nodos sucesores w.

Para reducir búsquedas innecesarias, limita a una búsqueda por arista en el grafo de restricciones.

* Heintze-Tardieu [7] (HT). Este es el probler solucionador escalable de forma masiva.

Propaga conjuntos de puntos únicamente cuando son necesarios (bajo demanda), de manera similar a la búsqueda en profundidad (Depth-First Search), sin embargo, no mantiene el invariante de ese algoritmo. Como resultado, **HT tiene que propagar conjuntos completos de punteros.**

* Pearce-Kelly-Hankin [13] (PKH). La base del solucionador de punteros actual utilizado en GCC. Se basa en un ordenamiento topologico débil del grafo de destino para evitar la búsqueda de ciclos en todo el espacio de nodos.
  + En lugar de realizar búsquedas exhaustivas de ciclos en todo el espacio de nodos del grafo, este algoritmo utiliza una ordenación topológica débil para minimizar estas búsquedas.
  + Una **ordenación topológica débil** organiza los nodos de manera que se prioricen las relaciones directas entre nodos, pero sin requerir un grafo completamente acíclico. Esto es útil para grafos con ciclos, ya que permite identificar y manejar ciclos locales de manera eficiente.
  + Mientras que otros algoritmos, como **Lazy Cycle Detection (LCD)**, buscan ciclos cada vez que encuentran ciertos patrones en los conjuntos points-to, PKH limita estas búsquedas a áreas específicas del grafo.

5.1 Comportamiento asintótico

Para verificar la **estabilidad** y el **comportamiento asintótico** de los algoritmos disponibles, se llevaron a cabo experimentos con **216 grafos de restricciones aleatorios**.

Para generar estos grafos, se generaron restricciones aleatorias basadas en las proporciones promedio observadas en programas reales:

 **14% restricciones base** (por ejemplo, l ⊇ r).

 **49% restricciones simples** (por ejemplo, l ⊇ ∗r o ∗l ⊇ r).

 **25% restricciones complejas tipo 1** (por ejemplo, l ⊇ ∗r con dependencias adicionales).

 **12% restricciones complejas tipo 2** (por ejemplo, ∗l ⊇ r con dependencias adicionales).

Para este experimento en particular, hemos utilizado grafos de restricciones aleatorias, fue difícil, a la **falta de benchmarks** reales que contengan un número de restricciones que aumente progresivamente. Podemos observar que estos grafos son diferentes de los grafos de restricciones que obtendríamos de los programas reales.

En programas reales, ciertos nodos tienden a **acumular o distribuir muchas aristas** (actúan como "hubs").

En contraste, los grafos aleatorios tienen una distribución de aristas más uniforme.

Los grafos reales son más densos que los generados aleatoriamente:

* **Grafos aleatorios**: tras colapsar componentes con el método de propagación profunda, contienen en promedio **0.87 aristas por nodo**.
* **Grafos reales**:
  + Densidad baja: **0.78 aristas por nodo** (como en sendmail).
  + Densidad alta: **139.581 aristas por nodo** (como en wine).

Otra diferencia es el tamaño promedio del conjunto de puntos producidos por los gráficos aleatorios que generan conjuntos de puntos-alcanzados que son **5 a 10 veces más grandes** que los observados en programas reales. A pesar de las diferencias mencionadas, los grafos aleatorios son útiles para obtener una **estimación general del comportamiento asintótico** de los solucionadores.