Universidad de La Habana
Facultad de Matemática y Computación

Resolviendo Errores de Dependencias de Proyectos de Software

Autor: **Daniel Enrique Cordovés Borroto**

Tutor: **MsC. Alejandro Piad Morffis**

Trabajo de Diploma
presentado en opción al título de
Licenciado en Ciencias de la Computación

Noviembre de 2021

Agradecimientos

A mis padres, por su amor y dedicación durante todos estos años, a mi familia por siempre apoyarme, a mis amigos y a mis compañeros de las competencias *ACM-ICPC* de los cuales he aprendido mucho; y a mi tutor el profesor Alejandro Piad por su tiempo y buenas ideas.

Opinión del tutor

El *software* es, sin lugar a dudas, uno de los pilares de la sociedad moderna. Independientemente de nuestra edad, ocupación, o clase social, es prácticamente imposible que pase un día sin que interactuemos con una aplicación de *software*, para tareas tan disímiles como comunicarnos con nuestros seres queridos, comprar la comida de la semana, ponernos al día con las noticias, pedir un taxi, saber si va a llover, e infinidad de otros ejemplos. Una de las características que hace al *software* un tipo cualitativamente distinto de creación humana es la capacidad de reusar, modificar, y mezclar *software* existente para construir aplicaciones nuevas. En términos técnicos, esta idea se concreta en las nociones de encapsulamiento y abstracción, conceptos fundamentales de la ingeniería de *software* que permiten construir aplicaciones duraderas y maleables, capaces de adaptarse continuamente a nuevos requisitos de los usuarios. Pero esta flexibilidad de reusar todo lo existente hace al *software* intrínsecamente más complejo, ya que cualquier aplicación no trivial depende de decenas o centenares de otros componentes de *software*, cada uno de los cuáles a su vez evoluciona y cambia a su propio ritmo, según dictan las necesidades de sus desarrolladores. Por este motivo, mantener actualizadas las dependencias de un proyecto de *software* es una tarea costosa y que tiene un impacto notable en la capacidad del proyecto para suplir las necesidades de los usuarios.

En este contexto se desarrolla la tesis de Daniel Enrique, que propone una visión novedosa a este problema nativo de la ingeniería de *software*. La tesis se cuestiona la siguiente interrogante: ¿será posible determinar automáticamente la versión más actual posible para las dependencias de un proyecto arbitrario, que garanticen que no ocurran errores debido a problemas de compatibilidad? A esta pregunta aparentemente sencilla, Daniel Enrique propone una formulación como un problema de optimización de naturaleza combinatoria, para el que no existen soluciones efectivas exactas. En búsqueda de una solución eficiente, la tesis explora diversos algoritmos de creciente complejidad, desde estrategias simples hasta metaheurísticas poblacionales. Como resultado práctico se obtiene una herramienta que permite actualizar automáticamente las dependencias de un proyecto en el lenguaje Python, siguiendo los estándares del desarrollo de *software* de dicha comunidad. La principal fortaleza de esta investigación, en mi opinión, es la combinación de un análisis teórico de un

problema aparentemente mundano, que resulta ser considerablemente complejo computacionalmente, con el desarrollo de una herramienta que es útil en casos reales y cubre una necesidad para la que no existe, según nuestro conocimiento actual, una solución completamente satisfactoria.

Con esta tesis, considero que Daniel Enrique ha demostrado haber obtenido los conocimientos teóricos y prácticos que esperamos de un graduado en Ciencia de la Computación. Hubo de asimilar un campo nuevo de estudio, y aplicar de forma novedosa técnicas conocidas de la inteligencia artificial en un escenario que no es para nada típico. Además, hizo gala de las habilidades técnicas para implementar un framework extensible y modular que puede ser utilizado por la comunidad, empleando tecnologías y metodologías del estado del arte. Finalmente, fue capaz de resumir en una tesis tanto las ideas teóricas fundamentales, como los detalles técnicos y los resultados experimentales que avalan la efectividad de su propuesta.

Por tales motivos, estimo razonable solicitar al tribunal que acepte la tesis de diploma de Daniel Enrique Cordovés Borroto con la calificación máxima, y de esta manera pase a formar parte, por derecho propio, de nuestro gremio de profesionales en Ciencia de la Computación.

MsC. Alejandro Piad Morffis
Facultad de Matemática y Computación
Universidad de la Habana
Noviembre, 2021

Resumen

La reproducibilidad al actualizar dependencias en proyectos de *software* es un problema fundamental que los desarrolladores enfrentan. Estos deben decidir hasta que punto mantener las versiones de las dependencias del proyecto inmutables o cuándo decidir actualizar alguna; pero de hacer esto último, el *software* debe seguir funcionando. He aquí el problema tratado en esta tesis: ¿cómo hallar para cada dependencia de un proyecto una versión específica, lo más actualizada posible, de tal forma que el *software* siga funcional? Para intentar resolver este problema se propone un *framework* conceptual, una serie de algoritmos y una implementación particular del *framework* llamada *Pydep*, la cual permite la ejecución de estos. Además, se muestran experimentos y comparaciones realizadas con dichos algoritmos y se formaliza la noción de dependencias de *software*.

Abstract

Reproducibility when updating dependencies in software projects is a fundamental problem that developers face. They must decide to what extent to keep the versions of the project dependencies immutable or when to decide to update any; but if they do the latter, the software should continue to work. This is the problem treated in this thesis: how to find a specific version for each dependency of a project, as up-to-date as possible, in such a way that the software remains functional? To try to solve this problem, a conceptual framework is proposed, a series of algorithms and a particular implementation of the framework called *Pydep*, which allows their execution. In addition, experiments and comparisons made with these algorithms are shown and the notion of software dependencies is formalized.

Índice general

Capítulo 1

Introducción

La reproducibilidad en el desarrollo del *software* es uno de los muchos desafíos a los cuáles se enfrentan los programadores. La reproducibilidad se puede definir como la repetibilidad del proceso de establecimiento de un hecho o de las condiciones bajo las cuales se puede observar el mismo hecho [1].

Alcanzar dicha reproducibilidad no es algo sencillo, los proyectos de *software* modernos son complejos; y además, usualmente son realizados en múltiples lenguajes de programación.

Un primer paso para intentar alcanzar dicha reproducibilidad es mantener inmutables las versiones de las dependencias (y subdependencias) del proyecto a lo largo del desarrollo. Pero hay un problema con esto: ¿qué pasa cuando sea crucial actualizar alguna versión de una dependencia?, esto puede ser inevitable por una serie de motivos:

- es necesario añadir una nueva dependencia, la cual tiene subdependencias ya presentes en el proyecto pero necesitan una versión más actualizada.
- la existencia de *bugs* en algunas de las versiones actuales de las dependencias.
- el descubrimiento de alguna vulnerabilidad en algunas de las dependencias usadas, lo cual se resuelve actualizando la misma.

De pasar una de las anteriores, habría que actualizar un subconjunto de las dependencias existentes.

Como es común en el desarrollo de *software*, cada proyecto tiene un conjunto de casos de prueba (*tests*) que se usan para verificar la correctitud (o no) del código actual. Al actualizar las dependencias, es posible que algunos de esos *tests* fallen. En este caso no es nada trivial determinar *a priori* las dependencias actualizadas que introdujeron los fallos.

1.1. Problema a resolver

Esta tesis propone un estudio de algunos algoritmos y propiedades que intentan determinar para cada dependencia (de un proyecto dado) una versión fija, tal que todos los *tests* unitarios pasen y además, la tupla conformada por esas versiones sea la "mejor" (definida por una función de costo) posible. Una formalización se muestra en el apartado 4.1.

El autor realizó varias búsquedas en el estado del arte y no hay ningún artículo que resuelva exactamente este problema; más detalles sobre esto en el capítulo 3. Hay varias razones para esto, en particular la dificultad que presenta el problema planteado, lo cual es analizado en el apartado 4.3.

1.2. Objetivos

Esta tesis tiene como objetivo principal diseñar una estrategia computacional para encontrar la mejor combinación de versiones de dependencias para un proyecto de software a partir de un conjunto de *tests* unitarios; y como objetivos específicos:

- Formalizar el problema propuesto, definirlo como un problema de búsqueda, estudiar sus características y proponer algoritmos de optimización adecuados para su solución.
- Crear un *framework* conceptual que permita la formalización de estos algoritmos.

Además de:

- Presentar una implementación concreta del *framework* conceptual llamada *Pydep*, que permite dado un proyecto, extraer sus dependencias, ejecutar sus *tests* y correr los algoritmos.
- Comparar cada uno de los algoritmos propuestos en términos de complejidad.
- Realizar experimentos con el *framework* y algoritmos propuestos.
- Ofrecer cierta formalización al papel que juegan las dependencias (y sus versiones) dentro de un proyecto de *software*.

1.3. Estructura

La tesis consta de 7 capítulos, el primero es el actual que es la introducción de la misma, luego en el capítulo 2 se presentan algunas definiciones necesarias; le sigue el capítulo 3 en donde se explora el estado del arte de los proyectos similares existentes y de los algoritmos útiles para resolver el problema propuesto. A continuación en el capítulo 4 se brinda una propuesta conceptual de

un *framework* que permita resolver el problema. En el el capítulo 5 se explica en detalle la implementación de *Pydep* y en el capítulo 6 se realizan varios experimentos con los algoritmos implementados. Por último en el capítulo 7 se describen las conclusiones de la tesis.

Capítulo 2

Preliminares

El manejo de dependencias en proyectos de *software* es muy importante. Es uno de los componentes que permite a los desarrolladores mantener a su *software* libre de *bugs*.

Por esto y otras razones mencionadas en la introducción, la creación y estudio de algoritmos para resolver el problema de determinar versiones de dependencias de proyectos tal que todos sus *tests* terminen satisfactoriamente, urge.

Desafortunadamente, en este tema específico se ha investigado pero no tanto como debería, esto se podrá ver en el capítulo 3, donde se exponen varios artículos e ideas que trabajan alrededor de este tema.

A continuación, se presentan algunas definiciones.

2.1. Dependencias

En un proyecto de *software* se puede diferenciar entre dos tipos de dependencias: las dependencias declaradas y las dependencias transitivas.

Dependencias declaradas Las dependencias declaradas son las dependencias de alto nivel (*top level*) de un proyecto, las cuales a su vez pueden tener otras dependencias.

Dependencias transitivas Las dependencias transitivas por otro lado, son la unión de las dependencias definidas anteriormente, con cada una de sus subdependencias, y sus sub-sub-dependencias, etc.

Más claramente, si se define un grafo dirigido G con n nodos donde cada nodo es una dependencia, y una arista del nodo a_i al b_i indica que la dependencia a_i depende de b_i entonces el conjunto de dependencias transitivas equivale a la clausura transitiva [2] de las dependencias declaradas sobre G.

2.2. Versiones de dependencias

Una versión de una dependencia es generalmente un identificador que se le asocia a una dependencia determinada durante el desarrollo. En la medida que se continua el desarrollo de una dependencia, es común que su versión vaya cambiando.

2.2.1. SemVer

Hay muchas formas de especificar estas versiones, una de ellas es *semantic versioning* (versionado semántico, *SemVer*) [3] la cual introduce un conjunto de reglas que sugieren cómo asignar versiones a proyectos para informar a los desarrolladores sobre cambios potencialmente incompatibles [4].

Los desarrolladores que escriben software en función de dichos proyectos pueden utilizar esta información para decidir qué tan "permisivos" pueden ser al aceptar automáticamente nuevas versiones de los mismos. Ser más restrictivos les permite mantener un control total sobre las dependencias de los proyectos, con el riesgo de que estos se queden obsoletos y no puedan beneficiarse de las actualizaciones compatibles con versiones anteriores. Ser más permisivos les permite beneficiarse automáticamente de las actualizaciones que contienen errores o correcciones de seguridad.

Para entender mejor esto de la permisividad se describe como funciona *SemVer*. Básicamente, una versión se compone de tres partes: `MAJOR.MINOR.PATCH`, donde cada una es un número.

De acuerdo con la especificación, el número `MAJOR` debe incrementarse solo si hay cambios de *API* (*Application Programming Interface*) incompatibles. El incremento del número `MINOR` es para nuevas funcionalidades introducidas de manera compatible, y el aumento del número `PATCH` es para correcciones de errores de manera compatible.

También hay etiquetas adicionales para indicar metadatos de *pre-release* y números de construcción (*build numbers*), estas son extensiones de la especificación.

Ejemplos de versiones *SemVer* son:

- 2.5.9
- 4.5.6-rc0
- 7.8.9-alpha+001

Es importante notar que estas reglas, al cumplirlas, pueden funcionar bien, pero de manera general es el desarrollador el que decide usarlas y de él depende su correcta aplicación. Esto quiere decir que aún siguiendo estas reglas pueden existir errores de dependencias en un proyecto dado.

Esto es confirmado por una encuesta reciente [5] de más de 2,000 desarrolladores de diferentes ecosistemas de *software* donde, incluso si el 92 % de los encuestados (para *npm* [6]) afirman que siempre incrementan el dígito más a la izquierda (`MAJOR`) si un cambio puede romper el código, todavía el 70 % de los encuestados declara que descubre que una dependencia cambió porque algo se rompe cuando intentan construir su propio proyecto [4].

2.2.2. CalVer

CalVer [7] es una convención de versiones basada en el calendario de lanzamiento de un proyecto, en lugar de números arbitrarios, a diferencia de *SemVer*.

Una versión *CalVer* puede constar de dos segmentos, el primero denota el año por ejemplo y el segundo puede ser alguna información más de calendario (como el mes) o un número de lanzamiento incremental (*release*). Aunque cada proyecto puede diseñar sus versiones basadas en *CalVer* y aplicarle modificaciones.

Ejemplos de *CalVer* son:

- 2020.1.418-pre
- las versiones del proyecto *Ubuntu* [8]: 4.10, 5.04, 5.10, etc que son `AÑO.MES`.

Esta al igual que *SemVer* tiene el problema de que al final son reglas que se definen y los desarrolladores deben aplicarlas de manera correcta.

Capítulo 3

Estado del Arte

El problema a resolver es un problema difícil, sus desafíos se exponen en detalle en el apartado 4.3. En este capítulo se presentan una serie de proyectos e ideas que trabajan en resolver problemas similares, así como algoritmos específicos ya creados que se pudieran utilizar para el problema.

3.1. Proyectos e ideas similares

A continuación se exponen varios artículos o proyectos que tratan temas similares al planteado aquí, aunque es necesario notar que ninguno trata exactamente este tema, pero que están lo suficientemente cerca de este como para ser útiles.

3.1.1. PyDFix

En [9] realizan un estudio de cómo proyectos de Python especifican versiones de dependencias y cómo su reproducibilidad se impacta por las mismas.

En particular, proponen una herramienta llamada `PyDFix` la cual intenta resolver errores de dependencias, pero a la hora de instalar las dependencias en un proyecto de Python; es decir la herramienta propuesta **no usa los *tests*** del proyecto para verificar que con las versiones de dependencias obtenidas estos *tests* den el resultado correcto, ellos solo analizan el *log* obtenido de la instalación de las dependencias, lo cual difiere con el problema a resolver presentado.

`PyDFix` tienes dos componentes:

- *LogErrorAnalyzer* cuyo propósito es identificar errores relacionados con dependencias a la hora de construir un proyecto.
- *IterativeDependencyResolver* el cual es un algoritmo que proponen para resolver estos errores de dependencia.

`PyDFix` toma como entrada el *log* de compilación actual y el *log* de compilación original (es decir, un *log* en el cual la construcción fue exitosa). `PyDFix` primero identifica los errores de dependencias y los posibles paquetes de dependencias que causan estos errores utilizando *LogErrorAnalyzer*. A esto le sigue la compilación iterativa de un parche (*patch*) que hace que la compilación sea reproducible nuevamente por *IterativeDependencyResolver*. El algoritmo iterativo para construir el parche vuelve a ejecutar la compilación con parches intermedios y analiza los nuevos *logs* de compilación producidos para identificar más errores y especificaciones de versiones de dependencias problemáticas. Este proceso continúa hasta que la compilación se vuelve reproducible o todas las opciones de parche se han probado y no se consideran útiles. Los desafíos claves que aborda `PyDFix` son la identificación de compilaciones no reproducibles relacionadas con dependencias a partir de *logs* de compilación y la selección de dependencias transitivas y de proyectos junto con sus versiones adecuadas para corregir errores de dependencias.

Uno de los algoritmos expuestos en el documento actual se basa en algunas ideas del *IterativeDependencyResolver* expuesto en [9].

3.1.2. DockerizeMe

En [10] proponen `DockerizeMe`, una técnica para inferir las dependencias necesarias para ejecutar un fragmento de código Python sin errores de *imports* de dependencias. `DockerizeMe` comienza con la adquisición de conocimientos *offline* de los recursos y las dependencias de los paquetes populares del índice de paquetes de Python (*PyPI*) [11]. Luego, crea especificaciones de *Docker* [12] (esto es, un *Dockerfile*) mediante un procedimiento de inferencia basado en grafos.

En esta tesis se usan algunas de las ideas expuestas en [10], especialmente las que abordan el tema de obtener las dependencias de un proyecto de Python, para luego instalarlas en un entorno aislado (*Docker*).

3.1.3. Watchman

En [13] hacen un estudio sobre los conflictos de dependencias (DC, *dependency conflicts*) que pueden ocurrir en proyectos de Python; se analizan patrones de manifestación y las estrategias de solución de estos problemas. Con estos hallazgos, diseñaron e implementaron *Watchman*, una técnica para monitorear continuamente los DC para el ecosistema *PyPI*.

La lectura de este artículo brindó información adicional acerca de los desafíos del problema de resolución de versiones de dependencias. Allí se plantean dos preguntas de investigación fundamentales:

- ¿Cómo se manifiestan los problemas de DC en los proyectos de Python? ¿Existen patrones comunes que se puedan aprovechar para el diagnóstico automatizado de estos problemas?

- ¿Cómo solucionan los desarrolladores los problemas de DC en los proyectos de Python? ¿Existen prácticas comunes que se puedan aprovechar para la reparación automatizada de estos problemas?

A la primera pregunta concluyen que los problemas de DC surgen principalmente de conflictos causados por actualizaciones de dependencias remotas o entornos locales.

Antes de exponer su respuesta encontrada a la segunda pregunta, se citan desafíos que encontraron en el mundo de dependencias de Python:

- Primero, la versión de una dependencia instalada para un proyecto de Python puede variar con el tiempo. Hay que notar que para cada dependencia requerida de un proyecto, *pip* (el instalador de paquetes de Python) [14] instalará la última versión de ella que satisfaga la restricción en cuestión. Por lo tanto, cualquier actualización de dependencias en *PyPI* puede afectar la versión de las dependencias instaladas para los proyectos posteriores (es decir, los proyectos que dependen de estas dependencias), causando posibles fallas de compilación.
- En segundo lugar, cuando una dependencia actualiza sus restricciones de versión en otras dependencias, sus proyectos posteriores pueden verse afectados. El impacto se puede propagar aún más a una amplia gama de proyectos.
- En tercer lugar, es difícil para los desarrolladores de Python obtener una idea completa de las dependencias de sus proyectos con información sobre restricciones de versiones.

Para abordar los desafíos y responder la segunda pregunta planteada, los autores diseñaron una técnica, Watchman, que realiza un análisis general desde la perspectiva de todo el ecosistema de *PyPI*, para monitorear continuamente los conflictos de dependencias causados por las actualizaciones de cada biblioteca.

Para cada biblioteca en *PyPI*, Watchman crea un Grafo de Dependencia Completo (*FDG*, Full Dependency Graph), un modelo formal que simula el proceso de instalación de dependencias para proyectos de Python. Los *FDG* se pueden actualizar gradualmente a medida que las bibliotecas evolucionan en *PyPI*. Watchman luego los analiza para detectar y prevenir proactivamente problemas de DC. Dado que los *FDG* registran las dependencias completas de los proyectos de Python con restricciones de versiones, también pueden proporcionar información de diagnóstico útil para ayudar a los desarrolladores a comprender las causas fundamentales de los problemas de DC detectados, lo que facilita su resolución.

3.1.4. Usando tests externos para identificar actualizaciones no compatibles

En [15] realizan un estudio que propone identificar *breaking changes* (cambios no compatibles) al actualizar una dependencia de un proyecto.

Específicamente, los autores proponen una técnica que ejecuta los *tests* de otros proyectos que dependen de una versión específica (de una dependencia o paquete) y utilizan los resultados como indicadores del riesgo de adoptar un paquete recién lanzado.

La técnica ejecuta *tests* de proyectos dependientes antes y después de actualizar una dependencia de una versión anterior a una versión más nueva. A menos que una actualización rompa intencionalmente la compatibilidad con versiones anteriores, los *tests* de la versión anterior deberían continuar funcionando en la versión más reciente. Es decir, los *tests* que pasan la ejecución en la versión anterior pero no la ejecución en la versión más nueva pueden indicar que la versión más nueva ha introducido una falla.

Más claramente, si para una dependencia determinada existiera algún proyecto tal que sus *tests* fallaran al actualizar dicha dependencia (y antes no), el sistema creado por los autores consideraría que dicha dependencia introduciría *breaking changes*.

Con el objetivo de probar su idea, los autores crearon un *dataset* de paquetes publicados en *npm* [6], al cual le hicieron un proceso de filtrado y de extracción de dependencias explícitas. Luego, usaron este mismo *dataset* para encontrar proyectos que dependan de la dependencia a verificar. Crearon un mecanismo para construir el proyecto y correr los *tests* usando *Docker*, un mecanismo similar se presenta en el documento actual.

Para evaluar la técnica propuesta, los autores realizaron un estudio empírico de diez casos en los que una actualización se revirtió debido a una versión que provocó cambios, de estos diez casos, el sistema detectó seis de ellos como positivos, es decir que tenían una dependencia cuya actualización rompía algún proyecto dependiente.

3.2. Algoritmos

En el apartado 4.2 se expone que el problema es de optimización pero que tiene que ser resuelto con metaheurísticas. A continuación se presentan algoritmos de este tipo que pueden ser implementados para intentar darle una solución al problema planteado.

3.2.1. Hill Climbing

Hill Climbing [16] es una técnica de optimización matemática que pertenece a la familia de la búsqueda local. Es un algoritmo iterativo que comienza con

una solución arbitraria a un problema, luego intenta encontrar una mejor solución haciendo un cambio incremental en la misma. Si el cambio produce un mejor óptimo, se realiza otro cambio incremental en la nueva solución y así sucesivamente hasta que no se puedan encontrar más mejoras.

La principal desventaja de este algoritmo es que tiende a quedarse en óptimos locales (a menos que la función de costo sea convexa claro), lo cual no es bueno para el problema tratado en esta tesis.

3.2.2. Simulated Annealing

Simulated Annealing es un método probabilístico propuesto en [17] para buscar el óptimo global de una función de costo que puede tener varios óptimos locales. Funciona emulando el proceso físico donde un sólido es lentamente enfriado de forma tal que su estructura termina "congelada" [18].

El algoritmo se puede utilizar para problemas de optimización computacional muy difíciles donde fallan los algoritmos exactos; aunque suele lograr una solución aproximada al óptimo global.

La inicialización del algoritmo es elegir un estado inicial en el espacio de búsqueda y una cantidad de pasos t. En el paso de iteración se calcula una temperatura T la cual es un número real, se genera una lista de vecinos del estado actual y se selecciona uno de manera aleatoria. Luego usando una función de probabilidad P se decide si moverse a ese nuevo estado o no, en dependencia de lo que devuelva P. El algoritmo termina luego de k pasos y su salida es el mejor estado visto.

También se le pueden hacer una serie de variaciones a dicho algoritmo, por ejemplo realizar reinicios de alguna forma, para prevenir que el algoritmo se pueda "trabar" en un óptimo local.

3.2.3. Particle Swarm Optimization

Particle Swarm Optimization (*PSO*) [19] se le atribuye originalmente a Kennedy y Eberhart [20] y fue pensado inicialmente para simular el comportamiento social; como es la representación del movimiento de organismos en una bandada de pájaros o un banco de peces.

PSO es una metaheurística, ya que hace pocas o ninguna suposición sobre el problema que se está optimizando y puede buscar en espacios muy grandes de soluciones candidatas.

El algoritmo consiste en que se tienen k partículas, cada una empieza en una posición al azar en el espacio de búsqueda y cada una tiene una velocidad. Además para cada partícula se guarda la mejor posición (que optimiza la función de costo) en la que ha estado.

En el paso de iteración, se fija una partícula y se recalcula su velocidad, teniendo en cuenta la posición del óptimo global hasta ese momento y la mejor posición vista por dicha partícula.

Luego se le aplica la nueva velocidad a la posición actual de la partícula, produciendo un desplazamiento de la misma en el espacio de búsqueda; se actualizan la mejor posición vista por la partícula actual y el óptimo global con esta nueva posición.

Después de una cantidad t de iteraciones el algoritmo termina y la salida es el óptimo encontrado.

3.3. Discusión

Como se puede apreciar, no todas las ideas o proyectos mostrados aquí son aplicables al tema de la tesis actual, pero afortunadamente algunos de ellos si, en particular, el algoritmo expuesto en el artículo de *PyDFix*, las ideas expuestas en la investigación sobre la herramienta *DockerizeMe* y las preguntas de investigación realizadas en el desarrollo *Watchman*, entre otras.

De los algoritmos vistos, dos de ellos se implementan en esta tesis, estos son, *Simulated Annealing* y *Particle Swarm Optimization*, debido a que *Hill Climbing* no es muy indicado para intentar resolver el problema planteado.

Capítulo 4

Propuesta conceptual

A continuación, se expone una definición formal del problema a resolver, se clasifica el problema de acuerdo a los algoritmos que puedan ser usados para resolverlo, se muestra una propuesta de algoritmos a usar y se explican los desafíos y dificultades del problema planteado.

4.1. Definición formal del problema

Luego de las definiciones expuestas en el capítulo 2, se presenta una definición más formal del problema a resolver.

Sea:

- P un proyecto de *software*
- T un conjunto de *tests* del proyecto P
- D una tupla, $D_1, D_2, \ldots, D_{|D|}$, que contiene las dependencias transitivas de P
- V una tupla, $V_1, V_2, \ldots, V_{|D|}$, que indica la versión de la dependencia D_i
- $F_D(V, T)$ una función que dado una tupla de versiones V (de las dependencias D) y un conjunto de *tests* T devuelve 1 si todos estos *tests* dan un resultado satisfactorio y 0 sino
- $C_D(V)$ una función que dado una tupla de versiones V (de las dependencias D), devuelve un número real que indica el costo de esas versiones

Se quiere buscar una tupla V' de tamaño $|D|$ tal que V'_i sea la versión asignada a la dependencia D_i, tal que:

- $F_D(V', T) = 1$

- $C_D(V')$ sea máximo.

4.2. Clasificación del problema

El problema planteado es un problema de optimización de un objetivo en un espacio de búsqueda discreto con restricciones que hay que computar (u obtener) ejecutando los *tests*.

El problema cae en la categoría de problema de optimización, pero dado que tiene unas restricciones que hay que computar cada vez, no es posible resolverlo con métodos convencionales de optimización, pero se puede modelar como un problema a resolver usando metaheurísticas.

4.3. Dificultades y desafíos

El problema planteado presenta varios desafíos:

- el número de dependencias transitivas puede ser grande, y cada una de las dependencias a su vez puede tener un gran cantidad de versiones, por lo que se tiene un espacio de búsqueda exponencial.
- $F_D(V,T)$ es una función cuya evaluación es muy costosa, por la cantidad de *tests* que puede contener el proyecto P, por lo que no se pueden realizar muchas evaluaciones de la misma.
- hay que definir una función de costo $C_D(V)$ que devuelva para una tupla de versiones su costo, hay muchas métricas que se pudieran usar, pero no es sencillo determinar cuáles métricas están por encima de otras.
- al fallar un *test*, es difícil diferenciar si es por una de las dependencias que se actualizó o si es por otro tipo de error, en el código del proyecto, o en el código del propio *test*.

Estas dificultades se comprobaron en los experimentos expuestos en el capítulo 6.

4.4. Propuesta

Para intentar resolver el problema planteado en esta tesis, se comparan una serie de algoritmos:

- *Simple Backtrack Algorithm*, ver el apartado 5.2.5.1.
- *Simple Randomized Algorithm*, ver el apartado 5.2.5.2.
- *Simulated Annealing*, ver el apartado 3.2.2 para explicación general y el apartado 5.2.5.3 para una implementación particular.

- *Particle Swarm Optimization*, ver el apartado 3.2.3 para una explicación más general y el apartado 5.2.5.4 para una implementación particular.

Además, se presenta un *framework* en el cual ejecutar estos algoritmos.

4.4.1. Framework

De manera general, cualquier *framework* conceptual que se diseñe, debe ser capaz de dado un proyecto de software, extraer sus dependencias, correr los algoritmos y ejecutar los *tests* del proyecto, quedándose en cada iteración con la mejor solución factible. Idealmente, el proceso de extraer las dependencias y ejecutar los *tests* debe suceder de forma aislada, en un entorno controlado.

Una explicación más particular sobre esto se encuentra en el apartado 5.1.1, donde se muestra una implementación específica de un *framework* con estas características.

Capítulo 5

Implementación

Siguiendo la propuesta, a continuación se presenta una implementación particular del *framework* conceptual expuesto anteriormente.

5.1. Pydep

Pydep es un *framework* que permite dado un proyecto de *software*, instalar sus dependencias, ejecutar sus *tests* y saber el resultado de dichos *tests*; así como correr algoritmos para resolver el problema planteado usando esta información.

Además de esto y por las dificultades planteadas en el apartado 4.3, *Pydep* cuenta con una especificación de "proyectos virtuales", lo cual hace posible correr los algoritmos sobre proyectos creados artificialmente, con una serie de dependencias y versiones ficticias, de forma *offline*.

5.1.1. Resumen general

De manera general, el *framework* realiza las operaciones descritas en la figura 5.1.

En primer lugar, se empieza con un proyecto de software en algún lugar en el sistema de archivos; luego se le extraen las dependencias, usando algún entorno definido en *Pydep*, uno de estos entornos es *Docker*.

A continuación, se selecciona un algoritmo entre los definidos, para resolver los errores de dependencias encontrados. En cada iteración del algoritmo se ejecutan los *tests* del proyecto usando *tests runners*, estas son clases que definen con que comando correr estos *tests* y bajo que entorno hacerlo.

De fallar dichos *tests*, *Pydep* sigue con la siguiente iteración del algoritmo, de

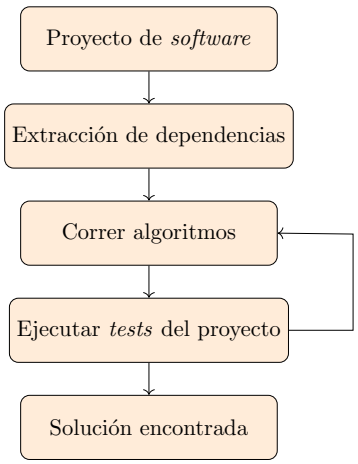

Figura 5.1: *Workflow* general de *Pydep*.

lo contrario, el algoritmo encuentra una solución. Por supuesto, esto se sigue repitiendo un número de iteraciones determinadas, el algoritmo le calcula el costo a cada solución obtenida y se queda con la mejor.

5.1.2. Estructura del *framework*

Pydep es altamente modular, componible y extendible. Fue desarrollado de tal forma que cada uno de sus componentes pueda ser cambiado o extendido fácilmente. Para su desarrollo se usó Python como lenguaje de programación. El *framework* está compuesto por los siguientes módulos fundamentales:

- *algorithms*: contiene la definición de los algoritmos usados por *Pydep*.
- *costs*: allí se especifican las funciones de costo disponibles.
- *deps*: define la clase *Dependency* que se usa para representar cada dependencia.
- *depsmgr*: interfaces para distintos gestores de dependencias, por ejemplo *Pip* [14].
- *opts*: especificación de los optimizadores.
- *tests*: contiene los *tests runners*, que son los encargados de definir como ejecutar los *tests* de un proyecto, ya sea en *Docker* [12] u otro entorno.
- *vercache*: define un caché que mantiene la lista de versiones disponibles para cada dependencia usada.

5.2. Detalles específicos de Pydep

A continuación se explica en detalle cada uno de los pasos mostrados en la figura 5.1.

5.2.1. Obtención del proyecto de software

El proyecto de software debe estar en algún lugar del sistema de archivos para ser cargado. El *framework* recibe el camino absoluto a dicho proyecto.

5.2.2. Extracción de dependencias

Una vez cargado el proyecto, se procede a extraer sus dependencias. *Pydep* al ser tan modular, permite al programador definir clases que se encarguen de hacer esto, para varios lenguajes de programación. Se explica cómo se hace este paso para proyectos hechos en Python.

En primer lugar, es necesario definir una imagen de *Docker* que va a permitir tener el proyecto en un entorno aislado. Se muestra un ejemplo de como puede quedar la definición de dicha imagen:

```
1  FROM python:3.9-slim
2  RUN groupadd pydep && useradd -mg pydep pydep
3  USER pydep
4  ENV VIRTUAL_ENV=/home/pydep/.venv
5  RUN python -m venv $VIRTUAL_ENV
6  ENV PATH=$VIRTUAL_ENV/bin:$PATH
7  RUN pip config set global.disable-pip-version-check true
8  COPY --chown=pydep:pydep . /home/pydep/app/
9  WORKDIR /home/pydep/app
10 ENV PYTHONPATH=/home/pydep/app
11 RUN pip install .[test]
12 CMD pip freeze
```

Figura 5.2: Ejemplo de imagen de *Docker* generada por *Pydep*.

En la línea 1 se define la imagen base, en este caso `python:3.9-slim`, esto es un parámetro que *Pydep* recibe, luego se define un usuario llamado *pydep*, ya que *Docker* por defecto realiza todas las operaciones bajo el usuario *root*.

A continuación, se crea un entorno virtual para *Python*, esto es necesario porque dentro de la imagen base existen otros paquetes de *Python*. Luego en la línea 8, se copia el proyecto hacia `/home/pydep/app/` dentro del contenedor; es importante notar que la construcción de la imagen ocurre en el mismo directorio del proyecto en cuestión.

Por último se define cómo instalar las dependencias del proyecto en la línea 11 y se establece un comando por defecto a la hora de ejecutar la imagen, que es `pip freeze`. Notar que en el momento que la imagen se ejecute ya las dependencias van a estar instaladas y el comando por defecto permite obtenerlas. La salida de este comando es una especificación de dependencia por línea, acorde al *PEP* [1] 508 [22].

```
1  black==21.9b0
2  certifi==2021.10.8
3  cffi==1.15.0
4  charset-normalizer==2.0.7
5  click==8.0.3
6  flake8==3.9.2
7  ghp-import==2.0.2
8  httpx==0.18.2
9  idna==3.3
10 importlib-metadata==4.8.2
11 iniconfig==1.1.1
12 MarkupSafe==2.0.1
13 mccabe==0.6.1
14 mergedeep==1.3.4
15 mkdocs==1.2.3
16 mypy==0.910
```

Figura 5.3: Ejemplo de salida del comando *pip freeze*.

Luego de tener definido el *Dockerfile* [2] se usa el *API* [3] de *Docker* llamada *docker-py* [23] para construir la imagen. Una vez construida, se ejecuta el contenedor obtenido lo cual hace que se ejecute *pip freeze*, se "parsea" su salida y se identifican las dependencias estrictamente declaradas y las dependencias transitivas, comparando la salida del comando con la especificación inicial de las dependencias del proyecto, que puede ser un fichero `setup.py`, `pyproject.toml` o un `requirements.txt`, estos no son más que formas de especificar las dependencias de un proyecto en Python.

Una vez identificadas las dependencias, se consulta el *API* de *PyPI* para hallar las versiones de cada una. Luego de obtenidas las versiones de cada dependencia, se guardan en un caché en el sistema de archivos para su futuro uso.

[1] Python Enhancement Proposals, documentos que definen nuevas funcionalidades para Python [21].

[2] Ficheros que se usan para definir el contenido de una imagen de *Docker*.

[3] Application Programming Interface.

5.2.3. Ejecución de *tests*

Para ejecutar los *tests* para una tupla de versiones fijada se crea una nueva imagen, similar a la definida anteriormente excepto que en vez de tener los dos últimos comandos de la figura 5.2, se usa uno que se encarga de instalar las dependencias con versiones fijas. Un ejemplo se muestra en la figura 5.4.

```
RUN pip install docutils==0.18 flit_core==3.4.0 pytest==6.2.5
↪ pytest_cov==3.0.0 requests==2.26.0 responses==0.15.0
↪ testpath==0.5.0 tomli==1.2.2 tomli_w==0.4.0
```

Figura 5.4: Usando *Pip* para instalar versiones fijas de dependencias.

Luego de instaladas las dependencias, se crea el contenedor a partir de la imagen definida y usando las clases del módulo *tests* se inicia el contenedor con el comando correcto para correr los *tests*. Una de las clases predefinidas es *Pytest* [24].

Por último, se obtiene la salida del comando ejecutado y el código que regresó (el *exit code*), con esto se puede determinar si los *tests* fallaron o fueron exitosos.

5.2.4. Evaluación de soluciones encontradas

Cuando se tiene una tupla de versiones que pasa los *tests*, se le calcula su costo. En esto interviene el módulo *costs*, el cual define funciones de costos.

Por ejemplo, una de las funciones que se define para una tupla de versiones V es la función f:

$$f(x) = \sum_{x \in V} g(x) \tag{5.1}$$

$$g(x) = \sum_{x_i \in x} B^{k-i} \cdot \log x_i \tag{5.2}$$

Donde:

- x es una versión con k $(k \geq 1)$ "componentes" de la forma $x_1.x_2.\cdots.x_k$ (cada x_i separado por un punto); $x = 123.42.1$ y $x = 1.2.3.4$ son ejemplos de versiones.
- B es una constante.

En la ecuación 5.2, la función g le da más prioridad a las componentes más a la izquierda de una versión; la función f es simplemente la suma de g sobre todas las versiones de la tupla V.

Al calcular el costo de una tupla de versiones (o lo que es lo mismo, una solución parcial), se usa el módulo *opts* que define optimizadores para actualizar la solución global. Un optimizador es una clase que define cómo realizar esa actualización.

5.2.5. Algoritmos

Pydep implementa los algoritmos expuestos en el apartado 4.4.

5.2.5.1. Basic Backtracking Algorithm

Este algoritmo es de fuerza bruta, básicamente lo que hace es iterar todo el espacio de búsqueda, para cada solución parcial encontrada, corre los *tests* con esas versiones fijadas, calcula su costo y se actualiza el óptimo global.

5.2.5.2. Basic Randomized Algorithm

Este algoritmo es aleatorio, lo que hace es generar tupla de versiones (soluciones parciales) de forma equiprobable e independiente, corre los *tests* con esas versiones fijadas, calcula su costo y se actualiza el óptimo global.

Este algoritmo y el definido en el apartado 5.2.5.1 son principalmente para probar la correctitud de *Pydep*.

5.2.5.3. Simulated Annealing

En el apartado 3.2.2 se brinda una explicación más general de este algoritmo. A continuación, se ofrece una formalización.

Sea:

- t la cantidad de iteraciones
- g la posición del óptimo
- S el espacio de búsqueda
- *random_state* una función que devuelve un estado aleatorio de S
- *random_neighbour* una función que devuelve un vecino aleatorio del estado actual
- f la función de costo

Un pseudocódigo del algoritmo se muestra en la figura 5.5. Ese pseudocódigo es del algoritmo minimizando, si se quiere maximizar se puede negar la función f.

En este algoritmo se usa una función $P(x, x', T)$ que está definida como:

$$P(x, x', T) = \begin{cases} 1\,, & \text{si } x' < x \\ e^{-\frac{x'-x}{T}}\,, & \text{sino.} \end{cases} \tag{5.3}$$

Donde x es el estado actual, x' es el nuevo estado y T es la temperatura actual.

Intuitivamente, esta función le dice al algoritmo que si el nuevo estado es mejor se mueva allí (con probabilidad 1), sino mientras más malo sea x' con respecto a x más baja será la probabilidad; esto es porque la diferencia $x' - x$ aumenta si x' aumenta, siendo el exponente de e aún más negativo y por lo tanto el valor de la función P mucho menor. Por supuesto, también es posible definir otras funciones para sustituir a P en este algoritmo.

Data: s_0: estado inicial.
Data: p_r: probabilidad de reiniciar.
Result: g: la posición del óptimo.

$s \leftarrow s_0$;
$g \leftarrow s$;
for $steps = [0 \dots t)$ **do**
    $T \leftarrow 2 - \frac{steps+1}{t}$;
    $s_{new} = random\_neighbour(s)$;
    $n_r \sim U(0,1)$
    if s_{new} *no existe* **or** $n_r < p_r$ **then**
        `// reiniciando`
        $s \leftarrow random\_state()$;
        continue
    end
    if s_{new} *no es factible* **then**
        continue
    end
    if $f(s_{new}) < f(g)$ **then**
        $g \leftarrow s_{new}$;
    end
    $r_i \sim U(0,1)$;
    if $P(f(s), f(s_{new}), T) \geq r_i$ **then**
        $s \leftarrow s_{new}$;
    end
end

Figura 5.5: Pseudocódigo de *Simulated Annealing*.

5.2.5.4. Particle Swarm Optimization

En el apartado 3.2.3 se brinda una explicación más general de este algoritmo. A continuación, se ofrece una formalización.

Sea:

- k la cantidad de partículas
- n la cantidad de dependencias del proyecto
- $x_i \in \Re^n$ la posición de la i-ésima partícula (cada componente representa la versión de una dependencia)
- $v_i \in \Re^n$ la velocidad de la i-ésima partícula
- p_i la mejor posición vista por la partícula i
- g la mejor posición vista por una partícula
- S el espacio de búsqueda
- S_{lo} el límite inferior de S
- S_{up} el límite superior de S

Un pseudocódigo del algoritmo, el cual maximiza la función de costo, se muestra en la figura 5.6.

En el algoritmo, para convertir una solución de espacio continuo a discreto se usa una función $c(x)$ que recibe un $x \in \Re$ y devuelve una tupla de índices a versiones (las versiones están guardadas en una lista, para cada dependencia):

$$c(x) = \Big(round(x_{(1)}), round(x_{(2)}), \ldots, round(x_{(n)})\Big) \tag{5.4}$$

En la ecuación 5.4 *round* es una función que toma un real y devuelve el entero más cercano a él.

Data: $w, \phi_p, \phi_g \in \Re$: parámetros del algoritmo.
Result: g: la posición del óptimo.

for *cada partícula* $i = 1 \dots k$ **do**
    $x_i \sim U(S_{lo}, S_{up})$;
    $v_i \sim U(-(S_{up} - S_{lo}), S_{up} - S_{lo})$;
    $p_i \leftarrow x_i$;
    $g \leftarrow \arg\max f(x_i)$
end
$steps \leftarrow 0$;
while $steps < t$ **do**
    for *cada partícula* $i = 1 \dots k$ **do**
        for *cada componente* $d = 1 \dots n$ **do**
            $r_p \sim U(0,1)$;
            $r_g \sim U(0,1)$;
            `// recalculando la velocidad`
            $v_{i,d} \leftarrow w \cdot v_{i,d} + \phi_p \cdot r_p \cdot (p_{i,d} - x_{i,d}) + \phi_g \cdot r_g \cdot (g_d - x_{i,d})$;
        end
        $x_i \leftarrow x_i + v_i$;
        if $c(x_i)$ *no es factible* **then** `// los tests fallaron`
            `// cambiar la dirección del movimiento`
            $v_i \sim U(-(S_{up} - S_{lo}), S_{up} - S_{lo})$;
            continue
        end
        if $f(x_i) > f(p_i)$ **then**
            $p_i \leftarrow x_i$;
            if $f(p_i) > f(g)$ **then**
                $g \leftarrow p_i$;
            end
        end
    end
    $steps \leftarrow steps + 1$
end

Figura 5.6: Pseudocódigo de *Particle Swarm Optimization*.

Capítulo 6

Experimentos

Se han realizado algunos experimentos para probar el funcionamiento del *framework*. Para realizar los experimentos, se creó una especificación para crear "proyectos virtuales" que permite ejecutar los algoritmos de manera *offline*.

La especificación permite establecer un grupo de dependencias, cada una de las dependencias tiene una lista de versiones disponibles, y unas restricciones que debe cumplir, por ejemplo, que las versiones a elegir estén en determinado intervalo. Además que permite especificar bajo que condiciones se pasan los *tests*.

La idea es poder generar varias entradas que conformen con esta especificación para correr los algoritmos sobre ellas. Para almacenar estas entradas en disco, se utiliza el lenguaje *TOML* (*Tom's Obvious Minimal Language*) [25], el cual es un formato de ficheros de configuración minimal.

En la figura 6.1 se tienen tres dependencias con nombres: `A`, `B` y `C` que cuentan con una lista de versiones, unos especificadores (o restricciones) y la versión inicial con las que el "proyecto virtual" empieza. También se puede ver allí la definición de unos *tests*, cada uno define un intervalo de validez para cada dependencia en cada una de las llaves `true_when` (término usado en *TOML*).

Para los experimentos realizados fue necesario crear un generador de entradas, el cual genera varias entradas de forma aleatoria. Luego *Pydep* recibe estas entradas y se corren los algoritmos contra estas.

Para los experimentos mostrados se generaron 15 proyectos virtuales, cada uno con a lo más 20 dependencias y exactamente 10 *tests*.

En la tabla 6.1 se muestra el porcentaje de los proyectos que se le encontraron al menos una solución *vs* una cantidad fija de iteraciones para correr cada algoritmo. Aquí se puede ver como en general el porciento es creciente en cada

```toml
[dependencies]

[dependencies.A]
versions = [ "1.2", "1.5", "2", "1", "5.3", "2.4" ]
specifier = ">=1.2.5,<=6.3.2"
iniver = "2"

[dependencies.B]
versions = [ "1.2", "1.2.9", "3.1", "5.3", "2.71" ]
specifier = ">=1.2.9"
iniver = "5.3"

[dependencies.C]
versions = [ "1.546.3", "2.4.7.3.7", "901.123.4357.7" ]
specifier = "!=2.5.35.4"
iniver = "2.4.7.3.7"

[[tests]]

[[tests.true_when]]
A = [ "1.9", "4.0.0.0" ]
B = [ "2.70", "3.1" ]
C = [ "1", "1" ]

[[tests.true_when]]
A = [ "1.9", "2.0.0.0" ]
C = [ "2", "2" ]

[[tests.true_when]]
A = [ "1.9", "3.3.2" ]
B = [ "2.70", "3.1" ]
C = [ "1", "2" ]
```

Figura 6.1: Ejemplo de definición de un “proyecto virtual”.

Algos. / Iters.	Backtracking	Randomized	SimAnn	PSO
15	0.00	33.33	13.33	40.00
40	6.67	33.33	40.00	66.67
80	6.67	33.33	26.67	66.67
150	13.33	46.67	33.33	60.00
500	20.00	46.67	40.00	73.33
700	20.00	60.00	33.33	80.00
1000	20.00	60.00	40.00	80.00
3000	26.67	66.67	46.67	80.00
5000	26.67	66.67	46.67	80.00

Tabla 6.1: Porciento de proyectos con solución encontrada.

Algos. / Iters.	Backtracking	Randomized	SimAnn	PSO
15	0	4.2081e+09	4.3922e+09	6.0299e+09
40	5.6555e+09	6.0978e+09	3.4444e+09	7.8552e+09
80	5.6604e+09	6.1943e+09	4.7304e+09	6.7114e+09
150	6.5083e+09	5.8498e+09	4.4057e+09	9.1399e+09
500	7.7915e+09	7.9126e+09	7.1814e+09	9.2522e+09
700	7.7915e+09	6.9651e+09	7.6851e+09	9.7902e+09
1000	7.7915e+09	8.1362e+09	7.3671e+09	8.4205e+09
3000	6.3135e+09	7.4022e+09	9.1965e+09	1.0054e+10
5000	6.3135e+09	7.9480e+09	8.2353e+09	1.0818e+10

Tabla 6.2: Promedio de la puntuación obtenida de todos los proyectos.

columna, y que el algoritmo que mejor resultados tiene es el de *Particle Swarm Optimization*.

En la tabla 6.2 se muestra el promedio de la puntuación alcanzada por el algoritmo en cuestión en los proyectos *vs* una cantidad fija de iteraciones para correr cada algoritmo. Aquí se puede ver como en general el promedio es creciente en cada columna, y que el algoritmo que mejor resultados tiene (con diferencia) es el de *Particle Swarm Optimization*. Este resultado era esperado, debido a que *Simulated Annealing* es una metaheurística basada en ideas de *local search* y *PSO* es una metaheurística poblacional.

En la tabla 6.3 se muestran las complejidades temporales de cada algoritmo por iteración.

Algoritmos	Backtracking	Randomized	SimAnn	PSO
Compl.	$O(T)$	$O(T)$	$O(D+T)$	$O(P \cdot (D+T))$

Tabla 6.3: Complejidades de cada algoritmo por iteración.

Donde T es la cantidad de *tests*, D la cantidad de dependencias y P la cantidad de partículas.

Pero lo que realmente hace difícil el problema presentado es la gran cantidad de versiones que puede tener cada dependencia en un proyecto real. Hay que notar que todos estos algoritmos corren con una cantidad de iteraciones fijas, y dependiendo del proyecto a probar, es probable que haya que ponerle varias iteraciones para encontrar al menos una solución, esto se pudo comprobar experimentalmente.

Para tener una idea más clara se muestra un desglose de la cantidad de versiones para las dependencias de dos proyectos reales hechos en Python.

En la tabla 6.4 se muestran las versiones declaradas de *fastapi* [26], en total son 33 y su producto es $9448375619307240000 > 10^{18}$, esto último es el tamaño del espacio de búsqueda para este proyecto.

En la tabla 6.5 [27] se muestran las versiones declaradas de *flit*, en total son 9 y su producto es $190634083200 > 10^{11}$, esto último es el tamaño del espacio de búsqueda para este proyecto.

Todas estas dependencias son necesarias para correr los *tests* del proyecto determinado.

Dependencias	Versiones
anyio	7
autoflake	1
black	1
databases	9
email-validator	3
flake8	5
Flask	6
httpx	17
isort	37
itsdangerous	3
Jinja2	6
mdx-include	1
mkdocs	5
mkdocs-markdownextradata-plugin	6
mkdocs-material	19
mypy	1
orjson	23
passlib	3
peewee	9
pydantic	3
pytest	2
pytest-cov	3
python-jose	1
python-multipart	1
PyYAML	3
requests	4
SQLAlchemy	35
starlette	1
typer-cli	1
types-orjson	1
types-ujson	1
ujson	5
uvicorn	11

Tabla 6.4: Número de versiones de cada una de las dependencias declaradas de fastapi.

Dependencias	Versiones
docutils	11
flit_core	3
pytest	114
pytest-cov	23
requests	51
responses	36
testpath	10
tomli	20
tomli_w	6

Tabla 6.5: Número de versiones de cada una de las dependencias declaradas de flit.

Capítulo 7

Conclusiones

Como conclusiones de la tesis se pueden nombrar las siguientes:

- Se expuso una definición conceptual de un *framework* capaz de resolver el problema propuesto.
- Se realizó una implementación de dicho *framework*, la cual es *Pydep*.
- Se propusieron 4 metaheurísticas para resolver el problema.
- *Pydep* permitió correr los algoritmos propuestos y realizar comparaciones y experimentos entre ellos.
- Se ofreció una cierta formalización al papel que juegan las dependencias (y sus versiones) dentro de un proyecto de *software*.

7.1. Contribuciones

Como contribución principal, esta tesis brinda un *framework* conceptual junto con su implementación (*Pydep*), el cual se puede ver en *GitHub* aquí.

7.2. Recomendaciones

Se recomienda:

- Realizar la implementación de otros algoritmos sobre *Pydep*, el cual lo permite de manera simple, ya que es un *framework* componible y extendible.
- Probar *Pydep* en un mayor número de proyectos reales.

Referencias

[1] Bente CD Anda, Dag IK Sjøberg, and Audris Mockus. Variability and reproducibility in software engineering: A study of four companies that developed the same system. *IEEE Transactions on Software Engineering*, 35(3):407–429, 2008. (Citado en la página 1).

[2] Paul Purdom. A transitive closure algorithm. *BIT Numerical Mathematics*, 10(1):76–94, 1970. (Citado en la página 4).

[3] Semantic versioning. https://semver.org/. (Accessed on 10/19/2021). (Citado en la página 5).

[4] Alexandre Decan and Tom Mens. What do package dependencies tell us about semantic versioning? *IEEE Transactions on Software Engineering*, 2019. (Citado en las páginas 5 y 6).

[5] How ecosystem cultures differ (survey results). http://breakingapis.org/survey/. (Accessed on 11/19/2021). (Citado en la página 6).

[6] npm, the node package manager. https://www.npmjs.com/. (Accessed on 10/23/2021). (Citado en las páginas 6 y 10).

[7] Calendar versioning. https://calver.org/. (Accessed on 11/19/2021). (Citado en la página 6).

[8] Enterprise open source and linux. https://ubuntu.com/. (Accessed on 11/19/2021). (Citado en la página 6).

[9] Suchita Mukherjee, Abigail Almanza, and Cindy Rubio-González. Fixing dependency errors for python build reproducibility. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 439–451, 2021. (Citado en las páginas 7 y 8).

[10] Eric Horton and Chris Parnin. Dockerizeme: Automatic inference of environment dependencies for python code snippets. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 328–338. IEEE, 2019. (Citado en la página 8).

[11] Pypi, the python package index. https://pypi.org/. (Accessed on 10/21/2021). (Citado en la página 8).

[12] Docker. https://www.docker.com/. (Accessed on 10/21/2021). (Citado en las páginas 8 y 17).

[13] Ying Wang, Ming Wen, Yepang Liu, Yibo Wang, Zhenming Li, Chao Wang, Hai Yu, Shing-Chi Cheung, Chang Xu, and Zhiliang Zhu. Watchman: Monitoring dependency conflicts for python library ecosystem. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 125–135, 2020. (Citado en la página 8).

[14] Pip, package installer for python. https://pypi.org/project/pip/. (Accessed on 10/21/2021). (Citado en las páginas 9 y 17).

[15] Suhaib Mujahid, Rabe Abdalkareem, Emad Shihab, and Shane McIntosh. Using others' tests to identify breaking updates. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 466–476, 2020. (Citado en la página 10).

[16] Bart Selman and Carla P Gomes. Hill-climbing search. *Encyclopedia of cognitive science*, 81:82, 2006. (Citado en la página 10).

[17] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983. (Citado en la página 11).

[18] Dimitris Bertsimas and John Tsitsiklis. Simulated annealing. *Statistical science*, 8(1):10–15, 1993. (Citado en la página 11).

[19] Maurice Clerc. Beyond standard particle swarm optimisation. In *Innovations and Developments of Swarm Intelligence Applications*, pages 1–19. IGI Global, 2012. (Citado en la página 11).

[20] James Kennedy and Russell Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95-international conference on neural networks*, volume 4, pages 1942–1948. IEEE, 1995. (Citado en la página 11).

[21] Pep 1 – pep purpose and guidelines. https://www.python.org/dev/peps/pep-0001/. (Accessed on 11/13/2021). (Citado en la página 19).

[22] Pep 508 – dependency specification for python software packages. https://www.python.org/dev/peps/pep-0508/. (Accessed on 11/13/2021). (Citado en la página 19).

[23] docker/docker-py: A python library for the docker engine api. https://github.com/docker/docker-py. (Accessed on 11/14/2021). (Citado en la página 19).

[24] pytest: helps you write better programs. https://docs.pytest.org/en/6.2.x/. (Accessed on 11/14/2021). (Citado en la página 20).

[25] Toml: Tom's obvious minimal language. https://toml.io/en/. (Accessed on 11/18/2021). (Citado en la página 25).

[26] Fastapi framework. https://github.com/tiangolo/fastapi. (Accessed on 11/24/2021). (Citado en la página 28).

[27] flit: Simplified packaging of python modules. https://github.com/takluyver/flit. (Accessed on 11/24/2021). (Citado en la página 28).