Verificación de algoritmos de vuelta atrás en Dafny
Verification of backtracking algorithms in Dafny

Trabajo de Fin de Grado
Curso 2024–2025

Autor
Salma El Kasouari Qessouri

Directores
Clara María Segura Díaz
Jorge Blázquez Saborido

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Calificación: 10

Verificación de algoritmos de vuelta atrás en Dafny
Verification of backtracking algorithms in Dafny

Trabajo de Fin de Grado en Ingeniería Informática

Autor
Salma El Kasouari Qessouri

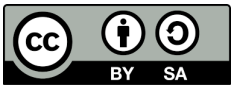
Directores
Clara María Segura Díaz
Jorge Blázquez Saborido

Convocatoria: *Junio 2025*
Calificación: 10

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

18 de junio de 2025

"Un día sin reír es un día perdido"
— Charles Chaplin



Licencia de uso

Dedicatoria

A mis padres, por darme todo lo que necesito, por su apoyo incondicional y por estar siempre ahí, incluso en los momentos más difíciles.

A mis hermanas, por ser no solo mi familia, sino también unas amigas maravillosas, divertidas y siempre dispuestas a sacarme una sonrisa.

A mis sobrinos, por su ternura y por alegrar mis días con su sola presencia.

Agradecimientos

A mis tutores, Clara Segura y Jorge Blázquez, por el tiempo dedicado y su valiosa orientación durante todo este proceso. Gracias por compartir su conocimiento y por guiarme con paciencia en cada etapa del trabajo. También agradecer a Clara, por su excelente labor docente en la asignatura Métodos Algorítmicos en Resolución de Problemas.

A Miguel Isabel, un gran profesor que apareció en el momento oportuno, cuando me encontraba un poco desorientada al inicio de la carrera.

A Isabel Pita, por despertar en mí el interés por la algoritmia, y por su dedicación y excelente labor docente, que siempre me ha inspirado.

A Pablo Gordillo y a Albert Rubio, por ofrecer clases de Programación con Restricciones que fueron no solo interesantes, sino también divertidas, haciéndome disfrutar del aprendizaje.

Resumen

Verificación de algoritmos de vuelta atrás en Dafny

La verificación formal es una técnica que permite demostrar, mediante métodos matemáticos, que un programa cumple su especificación (el comportamiento esperado del programa). A diferencia de las pruebas tradicionales, no se basa en casos de prueba concretos, sino en razonamientos lógicos que garantizan la corrección del código en todos los casos posibles. En este trabajo desarrollaremos una metodología para especificar y verificar en Dafny algoritmos de vuelta atrás. Dicha metodología se aplica a dos ejemplos representativos de los problemas resueltos mediante este método algorítmico.

En este trabajo hemos usado la herramienta Dafny, un lenguaje de programación diseñado para la especificación formal y la verificación asistida de programas.

Se presentarán los elementos esenciales de la metodología utilizando como ejemplo el problema de la mochila. En dicho problema el espacio de búsqueda está estructurado como un árbol binario, lo que se traduce en invocar recursivamente al propio algoritmo dos veces a lo sumo (una cantidad fija). A continuación, extenderemos la metodología para especificar y verificar problemas en los que el espacio de búsqueda está estructurado como un árbol n-ario, lo cual implica un bucle de invocaciones recursivas donde el número de iteraciones depende de una de las dimensiones del problema. Concretamente, verificaremos el problema de asignar tareas a funcionarios. Completaremos la metodología introduciendo podas de optimalidad para reducir el espacio de búsqueda y proporcionaremos varias podas en ambos ejemplos de aplicación. Finalmente, se proporcionarán conclusiones y posibles líneas del trabajo futuro.

Palabras clave

Dafny, verificación, vuelta atrás, especificación, poda.

Abstract

Verification of backtracking algorithms in Dafny

Formal verification is a technique that allows us to demonstrate, through mathematical methods, that a program meets its specification (the expected behavior of the program). Unlike traditional testing, it is not based on specific test cases, but on logical reasoning that guarantees the correctness of the code in all possible cases. In this work, we will develop a methodology to specify and verify backtracking algorithms in Dafny. This methodology is applied to two representative examples of problems solved by this algorithmic method.

In this work we have used Dafny tool, a programming language designed for formal specification and assisted program verification.

We will present the essential elements of the methodology using the knapsack problem as an example. In this problem, the search space is structured as a binary tree, which translates into recursively invoking the algorithm at most twice (a fixed amount). Then we will extend the methodology to specify and verify problems in which the search space is structured as an n-ary tree, which involves a loop of recursive invocations where the number of iterations depends on one of the problem's dimensions. Specifically, we will verify the assigning tasks to employees problem. We will complete the methodology by introducing optimal bounds to reduce the search space and we will provide several bounds in both application examples. Finally, we will present conclusions and possible lines for future work.

Keywords

Dafny, verification, backtracking, specification, bound.

Índice

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

"Puede que no tenga victorias asombrosas, pero puedo sorprenderte con las derrotas de las que salí vivo."
— Antón Chéjov

1.1. Motivación

En el ámbito de la informática, los métodos formales surgieron como enfoques analíticos que permiten verificar el desarrollo de sistemas utilizando la lógica y las matemáticas (15). La principal motivación de su uso es aumentar la fiabilidad y robustez de los sistemas. En este campo, una rama fundamental es la **verificación formal**, que consiste en demostrar matemáticamente que un sistema cumple con su especificación. A diferencia de las pruebas tradicionales, no se basa en casos de prueba concretos, sino en razonamientos lógicos que garantizan la corrección del código en todos los casos posibles. Para ello, primero se modela el sistema mediante varios tipos de estructuras, y se razona sobre el modelo para obtener una prueba de corrección. Estas pruebas pueden ser asistidas por herramientas específicas, como es el caso de **Dafny** (8), un lenguaje y verificador automático que permite escribir código junto con sus especificaciones formales, y comprobar su corrección.

Actualmente, la creciente complejidad de los algoritmos ha incrementado la relevancia de la verificación formal, especialmente en la industria del hardware, donde los errores suelen tener un mayor impacto económico. Por otro lado, en el ámbito del software, la criptografía es especialmente relevante debido a su impacto tanto en la seguridad de las personas como en los aspectos económicos. En hardware, la dificultad de simular todas las posibles interacciones entre componentes favorece el uso de métodos de prueba automatizados, lo que facilita su verificación formal (3).

Este trabajo se enmarca dentro de una línea de investigación centrada en la especificación y verificación de estructuras de datos y esquemas algorítmicos. Entre los trabajos previos más representativos en esta área, tanto de fin de grado como de fin de máster, se encuentran los siguientes:

- *Verificación de estructuras de datos enlazadas en Dafny*, de Jorge Blázquez

Saborido (13).

- *Verificación de estructuras de datos arborescentes con iteradores en Dafny*, de Jorge Blázquez Saborido (14).
- *Verificación de algoritmos voraces en Dafny*, de Paula Pastor Pérez. (11)
- *Verificación formal de algoritmos de programación dinámica en Dafny*, de Daniel Trujillo Viedma (16).
- *Verificación de algoritmos sobre segmentos de un vector utilizando módulos abstractos en Dafny*, de Pablo Martín Viñuelas (17).

El presente trabajo se enfoca en la aplicación de técnicas de verificación formal a algoritmos que siguen el esquema algorítmico de la **vuelta atrás** (backtracking). Se ha elegido este método algorítmico para dar continuidad a la línea de investigación existente por tratarse de un área aún pendiente de abordar. Además, se puede extender la verificación formal a otros esquemas algorítmicos relacionados, como la ramificación y poda (*branch and bound*) junto con las colas de prioridad (*priority queues*).

El esquema de vuelta atrás plantea retos interesantes debido no solo a su relevancia dentro del diseño algorítmico, sino también a su riqueza estructural, que plantea retos interesantes en términos de modelado y verificación.

1.2. Objetivos

El principal objetivo de este trabajo es aplicar técnicas de verificación formal para demostrar la corrección de algoritmos que utilizan el método algorítmico de vuelta atrás. Para ello, se emplea la herramienta Dafny, un lenguaje de programación que permite especificar, implementar y verificar programas.

Para alcanzar este objetivo general, se plantean los siguientes objetivos específicos:

- Comprender y aplicar la metodología de separación entre especificación e implementación en Dafny.
- Desarrollar una metodología de verificación que sea aplicable de manera sistemática a otros problemas resueltos con el mismo método.
- Modelar correctamente estructuras y comportamientos del problema utilizando tipos algebraicos y clases, permitiendo una verificación modular y comprensible.
- Verificar la corrección del algoritmo de vuelta atrás sobre distintos problemas, tanto aquellos cuyo espacio de búsqueda es un árbol binario como aquellos en los que es un árbol n-ario.
- Identificar las ventajas y limitaciones de Dafny en procesos de verificación, proponiendo mejoras o alternativas en caso necesario.

Estos objetivos pretenden no solo demostrar la validez del algoritmo en los casos considerados, sino también mostrar que es posible extender la metodología a otros algoritmos más complejos como los de ramificación y poda, abriendo la puerta a futuros trabajos en esta línea.

Se puede acceder al código de este proyecto a través de este enlace: `https://github.com/SalmaElKasouari/TFG-Salma`. Este se distribuye bajo los términos de la Licencia MIT[1], lo que permite su libre uso, modificación y distribución, siempre que se mantenga el aviso de copyright.

1.3. Plan de trabajo

El desarrollo de este trabajo se ha estructurado en varias fases a lo largo del curso académico, guiadas por una metodología exploratoria basada en reuniones periódicas con los tutores, debido a la naturaleza investigadora del proyecto. A continuación, se detalla el plan de trabajo seguido para alcanzar los objetivos propuestos:

1. **Familiarización con la herramienta Dafny (Verano 2024)**. Durante el verano, dediqué tiempo a entender el funcionamiento de Dafny, no solo su sintaxis, sino su funcionamiento: cómo combina especificación, implementación y verificación formal.

2. **Verificación del problema de la mochila (Septiembre 2024 – Enero 2025)**. En esta fase, mis tutores y yo decidimos elegir el problema de la mochila como primer caso de estudio, debido a que el espacio de búsqueda de soluciones es un árbol binario, por lo que el número de llamadas recursivas al algoritmo de vuelta atrás es fijo y pequeño. Las actividades principales fueron:

   - Modelado de los elementos esenciales del problema y definición de propiedades a verificar. Se optó por utilizar tipos de datos algebraicos para la especificación y clases para la implementación.
   - Especificación e implementación de la llamada inicial, incluyendo la correcta inicialización de todos sus componentes.
   - Verificación formal de la llamada inicial y, posteriormente, del método de vuelta atrás, lo que requirió la introducción de pruebas auxiliares para demostrar propiedades no triviales.
   - Verificación de una de las cotas aplicables al problema.

3. **Verificación del problema de los funcionarios (Febrero 2025 – Abril 2025)**. El objetivo de esta etapa fue aplicar la misma metodología a un problema más complejo, en el que el espacio de búsqueda es el árbol n-ario por lo que la implementación requiere de un bucle de llamadas recursivas, donde el número de iteraciones depende de la entrada del problema.

[1] Una copia completa de la licencia puede encontrarse en el repositorio o en `https://opensource.org/licenses/MIT`.

- Se comprobó que la metodología era válida también para este tipo de problemas, aunque surgieron nuevas dificultades, especialmente en la verificación del bucle de llamadas recursivas.
- Se verificaron tres cotas distintas asociadas a este problema.

4. **Documentación del trabajo (desde mediados de abril de 2025)**. Finalmente, a partir de abril se comenzó con la redacción y estructuración de esta memoria, recogiendo todo el proceso de desarrollo y los resultados obtenidos.

A continuación, se describe brevemente el contenido de cada capítulo de esta memoria:

- Capítulo 2: Dafny. Se presenta la herramienta Dafny, su filosofía de trabajo y las principales características utilizadas en este trabajo.
- Capítulo 3: Vuelta atrás. Se describe el funcionamiento del método algorítmico de vuelta atrás.
- Capítulo 4: Metodología utilizada. Se detalla cómo se ha aplicado una metodología concreta para modelar, especificar e implementar formalmente el problema de la mochila.
- Capítulo 5: Aplicación de la metodología al problema de los funcionarios. Se explica cómo se ha generalizado la metodología anterior para abordar un problema con una estructura más compleja (n-aria).
- Capítulo 6: Podas de optimalidad. Se explica cómo se han verificado formalmente varias cotas en ambos problemas.
- Capítulo 7: Conclusiones y trabajo futuro. Se recogen las principales conclusiones del trabajo, valoraciones personales, dificultades encontradas y posibles líneas de trabajo a desarrollar en el futuro.

Introduction

Motivation

In the field of computer science, formal methods emerged as analytical approaches that allow for system development verification using logic and mathematics (15). The main motivation behind their use is to increase the reliability and robustness of systems. Within this field, a fundamental branch is formal verification, which consists of mathematically proving that a system meets its specification. Unlike traditional testing, it does not rely on specific test cases, but on logical reasoning that guarantees the correctness of the code in all possible scenarios. To achieve this, the system is first modelled using various types of structures, and then the model is analysed to obtain a correctness proof. These proofs can be assisted by specialised tools, such as Dafny (8), a programming language and automatic verifier that allows code to be written alongside its formal specifications and checks its correctness (3).

Currently, the increasing complexity of algorithms has made formal verification more relevant, especially in the hardware industry, where errors tend to have a greater economic impact. On the software side, cryptography is particularly significant due to its impact on both personal security and economic aspects. In hardware, the difficulty of simulating all possible interactions between components favours the use of automated testing methods, which facilitates formal verification (3).

This work falls within a research line focused on the specification and verification of data structures and algorithmic schemes. Some of the most representative previous works in this area, both at the undergraduate and master's level, include:

- *Verification of linked data structures in Dafny*, by Jorge Blázquez Saborido (13).
- *Verication of tree-like data structures with iterators in Dafny*, by Jorge Blázquez Saborido (14).
- *Verification of greedy algorithms in Dafny*, by Paula Pastor Pérez (11).
- *Formal verification of dynamic programming algorithms with Dafny*, by Daniel Trujillo Viedma (16).
- *Verification of algorithms on array slices using abstract modules in Dafny*, by Pablo Martín Viñuelas (17).

This project focuses on the application of formal verification techniques to algorithms that follow the backtracking algorithmic scheme. This method was chosen both to continue the existing research line and because it remains an area yet to be thoroughly explored. Moreover, formal verification could be extended to other related algorithmic schemes, such as branch and bound, along with priority queues.

Backtracking presents interesting challenges not only due to its relevance in algorithm design but also because of its structural richness, which introduces non-trivial modelling and verification difficulties.

Objectives

The main objective of this project is to apply formal verification techniques to prove the correctness of algorithms that use the backtracking algorithmic method. For this, the Dafny tool is used—a programming language that allows for the specification, implementation, and verification of programs.

To achieve this general objective, the following specific goals are set:

- Understand and apply the methodology of separating specification and implementation in Dafny.
- Develop a verification methodology that can be systematically applied to other problems solved using the same approach.
- Properly model structures and behaviours of the problem using algebraic data types and classes, enabling modular and understandable verification.
- Verify the correctness of the backtracking algorithm for various problems, including those where the search space is a binary tree and those where it is an n-ary tree.
- Identify the strengths and limitations of Dafny in verification processes, proposing improvements or alternatives when necessary.

These goals aim not only to demonstrate the validity of the algorithm in the considered cases but also to show that the methodology can be extended to more complex algorithms, such as branch and bound, paving the way for future work in this direction. The code for this project is available at: `https://github.com/SalmaElKasouari/TFGVA-Salma`. This is distributed under the terms of the MIT License[2], which allows free use, modification, and distribution, provided that the copyright notice is retained.

[2] A full copy of the license can be found in the repository or at `https://opensource.org/licenses/MIT`.

Work plan

The development of this project was structured into several phases throughout the academic year, following an exploratory methodology guided by periodic meetings with the supervisors due to the research-oriented nature of the work. Below is a detailed plan of the work carried out to achieve the proposed objectives:

- **Familiarization with the Dafny tool (Summer 2024)**. During the summer, I spent time understanding how Dafny works, not only its syntax but also its operation: how it combines specification, implementation, and formal verification.
- **Verification of the Knapsack Problem (September 2024 - January 2025)**. In this phase, my supervisors and I decided to choose the Knapsack Problem as the first case study, as its solution search space is a binary tree, meaning the number of recursive calls in the backtracking algorithm is fixed and small. The main activities included:
  - Modeling the essential elements of the problem and defining the properties to verify. Algebraic data types were used for specification and classes for implementation.
  - Specification and implementation of the initial call, including the correct initialization of all its components.
  - Formal verification of the initial call and subsequently the backtracking method, which required introducing auxiliary proofs to demonstrate non-trivial properties.
  - Verification of one of the bounds applicable to the problem.
- **Verification of the Employees Problem (February 2025 - April 2025)**. The aim of this stage was to apply the same methodology to a more complex problem, where the search space is an n-ary tree, requiring a loop of recursive calls where the number of iterations depends on the problem input.
  - It was confirmed that the methodology was also valid for this type of problem, although new challenges arose, especially in verifying the loop of recursive calls.
  - Three different bounds associated with this problem were verified.
- **Documentation of the work (from mid-April 2025)**. Finally, starting in April, the writing and structuring of this report began, documenting the entire development process and the results obtained.

The contents of each chapter of the report are briefly described below:

- Chapter 2: Dafny. Presents the Dafny tool, its working philosophy, and the main features used in this project.

- Chapter 3: Backtracking. Describes how the backtracking algorithmic method works.
- Chapter 4: Methodology used. Details how a specific methodology was applied to model, specify, and formally implement the Knapsack Problem.
- Chapter 5: Application of the methodology to the Employees Problem. Explains how the previous methodology was generalised to address a more complex (n-ary) structure.
- Chapter 6: Optimality Pruning. Explains how several bounds were formally verified in both problems.
- Chapter 7: Conclusions and Future Work. Summarises the main conclusions, personal reflections, encountered challenges, and potential directions for future research.

Capítulo 2

Dafny

"Aquellos que pueden imaginar cualquier cosa pueden crear lo imposible."
— Alan Turing

Dafny es un lenguaje de programación que permite la verificación formal de programas. Su uso está especialmente extendido en el ámbito de la enseñanza y proyectos de investigación. Fue creado por Rustan Leino en Microsoft Research (12) como una herramienta para la especificación y verificación formales. A diferencia de lenguajes tradicionales, Dafny integra internamente un verificador automático basado en SMT (Satisfiability Modulo Theories), concretamente Z3, que actúa de forma transparente para el usuario. Esto permite a Dafny analizar estáticamente las especificaciones de un programa mientras se escribe el código. De esta manera, si una postcondición de una función o método no se cumple, Dafny lo toma como un error inmediatamente. Así se pueden detectar errores en la lógica y/o diseño del programa antes de su ejecución, lo que lo hace ideal para el desarrollo de software crítico.

Dafny soporta programación imperativa y funcional, con elementos como clases, tipos de datos algebraicos, métodos, funciones, bucles y estructuras de datos comunes. Además, permite formalizar especificaciones junto al código fuente, facilitando la relación entre la implementación (lo que el programa hace) y la especificación (el comportamiento que se espera del programa). En este capítulo se presentarán con detalle los elementos fundamentales del lenguaje Dafny que han sido utilizados en el desarrollo de este proyecto. Se han utilizado varios ejemplos del libro *Program Proofs* (9), en concreto los ejemplos de las Figuras 2.8, 2.9, 2.10, 2.11, 2.15, 2.16, 2.17, 2.18 y 2.19.

2.1. Tipos en Dafny

Además de los tipos primitivos para números enteros (`int`), números reales (`real`) y valores booleanos (`bool`), Dafny proporciona una variedad de tipos que se pueden utilizar tanto para la especificación como para la implementación. En esta sección haremos una distinción entre los tipos más adecuados para cada contexto.

```
function Sum(s: seq<int>): int
{
  if |s| = 0 then
    0
  else
    s[0] + Sum(s[1..])
}
```

Figura 2.1: Suma de una secuencia mediante recursión

```
datatype ItemData = ItemData(weight: real, value: real) {}
```

Figura 2.2: Tipo de datos algebraico que representa a un ítem

Los tipos inmutables son tipos funcionales cuyos valores no tienen un estado mutable. Permiten razonar más fácilmente sobre el comportamiento abstracto de los programas. Los más utilizados en este proyecto son:

- `seq<T>`: secuencias inmutables de elementos de tipo `T`. Como ejemplo, en la figura 2.1 se muestra una función recursiva que define la suma de los elementos de una secuencia. La función `Sum` recibe un parámetro `s`, una secuencia inmutable de números enteros `seq<int>`, y devuelve un valor de tipo entero (`int`). La lógica de la función se basa en un caso base y en un caso recursivo, por ello se usa la estructura `if-then-else`. El caso base se da cuando la longitud de la secuencia `s` es cero. Para determinar la longitud de un tipo `seq<T>`, se usa el operador de cardinalidad: `|s|`. En esta situación, la función devuelve el valor 0, ya que la suma de los elementos de una secuencia vacía es cero. Por otro lado, el caso recursivo se activa cuando la secuencia contiene uno o más elementos. En este caso, la función calcula la suma tomando el primer elemento de la secuencia, al cual se accede mediante la notación `s[0]`, y lo suma al resultado de una llamada recursiva a la misma función `Sum`, pero esta vez pasando como argumento una nueva secuencia que contiene todos los elementos de `s` desde el segundo elemento hasta el final, lo que se denota como `s[1..]`.

- `datatype`: tipos de datos algebraicos definidos por el usuario utilizados para representar información. La figura 2.2 muestra la declaración de un nuevo tipo de dato llamado ItemData que representa a un ítem con peso y valor reales. La sintaxis para declarar un tipo de dato algebraico en Dafny comienza con la palabra reservada `datatype`, seguida inmediatamente por el nombre que deseamos asignar a nuestro nuevo tipo de dato (en el ejemplo, `ItemData`). A continuación, se utiliza el signo de igual para indicar la definición de este tipo. En este caso, tenemos un constructor que crea instancias de tipo `ItemData` con un peso y un valor reales. Finalmente, se abren y cierran llaves, dentro de las cuales se pueden definir miembros adicionales asociados a este tipo, como funciones o predicados que operan sobre instancias de este tipo.

Los tipos mutables son aquellos cuyos valores tienen un estado que se puede modificar en tiempo de ejecución por estar alojados en la memoria dinámica. Entre ellos se encuentran:

```
var a := new bool[3](i ⇒ false);

var matrix := new int[2,2];
matrix[0,0] := 1;
matrix[0,1] := 2;
matrix[1,0] := 3;
matrix[1,1] := 4;
```

Figura 2.3: Declaraciones de arrays

```
class Item {
    const weight: real
    const value:  real

    constructor(w: real, v: real)
        ensures this.weight = w
        ensures this.value = v
    {
        this.weight := w;
        this.value := v;
    }
}
```

Figura 2.4: Clase que representa a un ítem

- **`array, array2`**: vectores unidimensionales y bidimensionales respectivamente. Como en cualquier otro lenguaje de programación, se utilizan para almacenar una colección de valores. En la figura 2.3 se muestra en primer lugar la declaración de un array booleano unidimensional en la que todas sus componentes se inicializan a `false` mediante la expresión lambda `i ⇒ false` dentro de la construcción `new bool[n]()`. En segundo lugar, se muestra la declaración de un array bidimensional de enteros con la notación `new int[2,2]` y su rellenado componente a componente para representar la siguiente matriz cuadrada de orden 2:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

- **`class`**: permite definir clases con campos y métodos. Se utilizan comúnmente para modelar objetos con estado mutable. La figura 2.4 representa a un ítem con peso y valor constantes. La definición de la clase Item sigue una estructura similar a la de otros lenguajes orientados a objetos, conteniendo atributos (campos) y un constructor. En Dafny, los atributos se declaran especificando primero su nombre, seguido de dos puntos (`:`) y luego su tipo. En este ejemplo, se utiliza la palabra clave `const` para indicar que los atributos `weight` y `value` son constantes, lo que significa que su valor se asigna durante la creación del objeto y no puede modificarse posteriormente. En caso de que se quiera permitir modificarlos, se debería utilizar la palabra clave `var` en lugar de `const`. El constructor `constructor(w: real, v: real)` es el encargado de inicializar estos campos. Las cláusulas `ensures this.weight == w` y `ensures this.value == v` son postcondiciones que verifican que los campos se inicialicen correctamente con los valores proporcionados (`w` y `v`).

Los tipos algebraicos facilitan la verificación modular y permiten razonar sobre el comportamiento del programa sin necesidad de ejecutar el código.

2.2. Métodos, funciones y predicados

En Dafny existen los métodos y las funciones, bloques de código que llevan a cabo tareas específicas. No obstante, existen diferencias entre ellos, especialmente en cómo se utilizan en la implementación y especificación.

- **Funciones (`function`)**: están diseñadas para ser puras y deterministas, es decir, no modifican el estado de la memoria y para las mismas entradas, siempre producen la misma salida. Se puede asemejar al concepto de función matemática. Debido a estas características, las funciones son muy útiles para la especificación. Dafny también permite declarar funciones como `ghost`, lo que significa que solo existen en tiempo de verificación y no se incluyen en el código compilado. Estas funciones son ideales para capturar propiedades auxiliares, definiciones intermedias o condiciones lógicas que ayudan a guiar la verificación, sin afectar la ejecución del programa. La sintaxis de una función en Dafny comienza con la palabra reservada `function`, seguida del nombre de la función, una lista de parámetros entre paréntesis (donde cada parámetro se declara con la notación `nombre: tipo`), el tipo de retorno precedido por dos puntos `: tipo`, y finalmente el cuerpo de la función entre llaves. La figura 2.5 muestra una función de ejemplo que calcula el cuadrado de un número. Esta función recibe un entero `n` y devuelve su cuadrado (`n*n`).

```
1  function Square(n: int): int
2  {
3      n * n
4  }
```

Figura 2.5: Función `Square`

- **Métodos (`method`)**: a diferencia de las funciones, estos pueden modificar el estado de la memoria y pueden realizar operaciones de entrada/salida. Los métodos son la forma que establece Dafny para implementar programas imperativos. La figura 2.6 muestra un método que devuelve el valor máximo entre dos números, `a` y `b`.

Tanto las funciones como los métodos pueden ser `ghost`. Esto es que solo existen para el fin de la verificación formal, pero no se incluyen en el código compilado. Es útil para escribir especificaciones complejas que ayudan a probar la corrección del código ejecutable. Para definirlos como `ghost`, se añade este modificador delante de la declaración de la función o el método.

Por otro lado, existen los **predicados (`predicate`)**, funciones booleanas que permiten definir propiedades. Se utilizan principalmente para establecer condiciones y

```
method Max(a: int, b : int) returns (c: int)
{
  if a < b {
    return b;
  }
  else {
    return a;
  }
}
```

Figura 2.6: Método `Max`

propiedades en las especificaciones. Un ejemplo simple se muestra en la figura 2.7, donde se verifica si un número cumple la propiedad de ser par.

```
predicate IsEven(n: int)
{
  n % 2 = 0
}
```

Figura 2.7: Predicado `IsEven`

2.3. Especificación

Los métodos y las funciones en Dafny pueden tener una especificación asociada que define su comportamiento esperado. Esta especificación se formaliza principalmente a través de las precondiciones y las postcondiciones. El conjunto compuesto por la cabecera de un método, sus precondiciones y sus postcondiciones se denomina terna de Hoare (5, 6). Las precondiciones y postcondiciones se formalizan sintácticamente mediante las cláusulas `requires` y `ensures`, respectivamente. Las precondiciones son las condiciones que se deben cumplir antes de ejecutar un método o función. Las postcondiciones son condiciones que se deben garantizar al finalizar la ejecución. En la figura 2.8 podemos ver la función recursiva `Fib`, que calcula el n-ésimo número de la sucesión de *Fibonacci*. La precondición especifica que el argumento `n` debe ser no negativo (`0 <= n`). La postcondición asegura que el valor devuelto por la función `Fib(n)` siempre será mayor o igual a cero (`0 <= Fib(n)`).

```
function Fib(n: int): int
  requires 0 ≤ n
  ensures 0 ≤ Fib(n)
{
  if n < 2 then n else Fib(n − 2) + Fib(n − 1)
}
```

Figura 2.8: Función `Fib`

Además de las precondiciones y postcondiciones, Dafny ofrece otras cláusulas de especificación:

- **`modifies`**: se utiliza para especificar qué objetos pueden ser modificados por el método. Si se intenta modificar un objeto que no esté incluido en la cláusula `modifies`, Dafny lo reporta como un error de verificación.

- `invariant`: se utiliza en los bucles para declarar propiedades que se mantienen constantes durante cada iteración del bucle. El verificador de Dafny comprueba que el invariante se cumple al entrar en el bucle y que se mantiene verdadero después de cada iteración. Profundizaremos en este aspecto más adelante.

- `decreases`: se utiliza principalmente en métodos y funciones recursivas para asegurar la terminación de la recursión. Establece una **función de cota** positiva decreciente, un valor que decrece en cada llamada recursiva. Un ejemplo de función recursiva es `Fib` vista en la figura 2.8, donde la función de cota es evidentemente `n`, ya que este valor disminuye en cada llamada recursiva. Esto se indicaría mediante la cláusula `decreases n`. En muchos casos, Dafny es capaz de inferir la cota, como sucede en este ejemplo.

Un caso especialmente interesante ocurre cuando se definen funciones mutuamente recursivas, es decir, funciones que se llaman entre sí. Veamos un ejemplo. En primer lugar, vemos que la función `ExpLess1` definida en la figura 2.9 evidentemente termina, pues el parámetro `n` disminuye en cada llamada.

```
function ExpLess1(n: nat): nat
  decreases n
{
  if n = 0 then 0
  else 2 * ExpLess1(n - 1) + 1
}
```

Figura 2.9: Función `ExpLess1`

Ahora, supongamos que queremos dividir el cálculo de $2^n - 1$ con otra función `ExpLess2`, tal y como se oberva en la figura 2.10.

```
function ExpLess1(n: nat): nat {
  if n = 0 then 0
  else ExpLess2(n) + 1
}

function ExpLess2(n: nat): nat
  requires n > 0
{
  2 * ExpLess1(n - 1)
}
```

Figura 2.10: Función `ExpLess1` que llama a `ExpLess2`

Esta refactorización no cambia el resultado de `ExpLess1`, pero genera una **recursión mutua**: `ExpLess1` llama a `ExpLess2` con el mismo `n`, mientras que `ExpLess2` llama a `ExpLess1` con `n - 1`. Luego, la dificultad está en probar que esta recursión mutua termina. Si ambas utilizan `decreases n`, Dafny acepta la llamada de `Expless2` a `Expless1`, pero no la de `Expless1` a `Expless2`, porque en este caso `n` no disminuye.

Para manejar esta situación, Dafny permite usar tuplas lexicográficas en las cláusulas `decreases`. La clave es ordenar explícitamente a una función como «menor»

que la otra, aunque los parámetros sean iguales. En la figura 2.11 vemos que la función de cota de cada función es una tupla con dos componentes.

```
function ExpLess1(n: nat): nat
  decreases n, 1
{
  if n = 0 then 0
  else ExpLess2(n) + 1
}

function ExpLess2(n: nat): nat
  requires n > 0
  decreases n, 0
{
  2 * ExpLess1(n - 1)
}
```

Figura 2.11: Función `ExpLess1` y `ExpLess2` con tuplas lexicográficas

Para `ExpLess1` tenemos la tupla `(n, 1)`, y para `ExpLess2` tenemos `(n, 0)`. Esto asegura la terminación porque:

- `ExpLess1(n)` llama a `ExpLess2(n)` → `(n,1) > (n,0)`, entonces la cota decrece.
- `ExpLess2(n)` llama a `ExpLess1(n-1)` → `(n,0) > (n-1,1)`, entonces la cota decrece.

Aunque usar constantes como 1 y 0 pueda parecer arbitrario, lo importante es que definan un orden válido.

2.4. Instrucciones `assert`

En Dafny, los `asserts` son instrucciones que permiten comprobar propiedades en un punto específico del programa. Cuando Dafny verifica el programa, comprueba que estas propiedades se cumplen. Los `asserts` se utilizan principalmente para asegurar invariantes o condiciones que deben mantenerse durante la ejecución del programa. Un ejemplo sencillo puede verse en la figura 2.12. El primer aserto se verifica trivialmente, pues comprueba que la variable de `b` sea el cuadrado de `a` justo después de la asignación `b := a * a`. El segundo aserto también se verifica de manera trivial, ya que Dafny puede inferir que el cuadrado de un número nunca es negativo. Sin embargo, un aserto como `assert b > 0` no se verificaría correctamente, pues esta propiedad es falsa, no se cumple en todos los casos: si `a` es igual a cero, entonces `b` también lo será. No obstante, si la función incluyese una precondición como `requires a != 0`, o `requires a > 0`, Dafny sí podría verificar ese aserto, dado que bajo esa condición el cuadrado siempre será estrictamente positivo.

```
method Square(a: int) returns(b: int)
{
    b := a * a;
    assert b = a * a;
    assert b ≥ 0;
}
```

Figura 2.12: Ejemplo de uso de `assert`

Con esto podemos concluir que la verificación de un `assert` depende tanto de la lógica de la implementación como de las precondiciones del método y también de información de contexto proveniente del camino que conduce a ese punto del programa, por ejemplo, condiciones de un `if` o de un bucle, etc. En muchas ocasiones, aunque sepamos que una propiedad es verdadera, Dafny no es capaz de comprobarla automáticamente, por lo que será necesario utilizar herramientas adicionales de verificación, como lemas o afirmaciones intermedias, que detallaremos más adelante.

2.5. Bloques `calc`

Dafny dispone de una sentencia `calc` que permite realizar cálculos matemáticos, proporcionando al verificador una secuencia de pasos lógicos que se deben cumplir. Un bloque `calc` tiene la siguiente estructura:

```
calc {
  expresión inicial;
  operador relacional
  // Justificación del paso
  expresión intermedia;
  operador relacional
  // Justificación del paso
  expresión intermedia;
  ...
  operador relacional
  // Justificación del paso
  expresión final;
}
```

El operador relacional indica la relación entre la expresión anterior y la siguiente. Dafny utiliza la transitividad de estas relaciones para concluir la relación entre la expresión inicial y la final.

Se puede especificar un operador por defecto para todo el bloque `calc` colocándolo después de la palabra clave `calc`, como en `calc <= {...}`. Si se omite el operador, se asumirá la relación de igualdad por defecto.

Las justificaciones para cada paso pueden proporcionarse entre llaves mediante bloques calc anidados, lo que permite demostrar igualdades de subexpresiones. Estos bloques ayudan a Dafny a verificar cada paso. En la figura 2.13, tomada del manual online de Dafny, *The Dafny Reference Manual* (7), podemos ver la demostración de la identidad algebraica $(x + y)^2 = x^2 + 2xy + y^2$ utilizando un bloque `calc ==`.

```
1 calc = {
2     (x + y) * (x + y);
3     x * (x + y) + y * (x + y);
4     x * x + x * y + y * x + y * y;
5     x * x + x * y + x * y + y * y;
6     x * x + 2 * x * y + y * y;
7   }
```

Figura 2.13: Ejemplo de `calc`

2.6. Estados anteriores: expresión `old` y uso de `label`

Dafny permite hacer referencia a estados anteriores del programa con `old`. Esto es útil en métodos que modifican parámetros, en los que es necesario comparar el estado antes y después de una modificación. Por ejemplo, un método que multiplica por dos cada una de las componentes de un array tendría como postcondición:

```
1 ensures ∀ i | 0 ≤ i < a.Length • a[i] = old(a[i]) * 2
```

La postcondición establece que para cada índice `i` del array `a`, el valor del elemento en esa posición al finalizar el método (`a[i]`) debe ser igual al doble del valor que ese mismo elemento tenía al inicio de la ejecución del método (`old(a[i]) * 2`).

El uso de `label` también permite etiquetar un punto en el código para luego referirse a los valores o las propiedades que se cumplen en ese punto del programa. En la figura 2.14 se muestra cómo el campo `p.x` se asigna primero a 1, luego se pone una etiqueta con `label L:` y por último se cambia `p.x` a 2. El valor actual de `p.x` es 2, esto se verifica con el `assert` de la línea 12. Usando `old@L(p.x)` (línea 13), se verifica que en el punto etiquetado `L` (antes de cambiar `p.x` a 2), `p.x` valía 1. Esto permite razonar sobre el valor de variables en momentos específicos del programa.

```
1  class Point {
2    var x : int
3    var y : int
4  }
5
6  method LabelExample(p : Point)
7  modifies p
8  {
9    p.x := 1;
10   label L:
11   p.x := 2;
12   assert p.x = 2;
13   assert old@L(p.x) = 1;
14 }
```

Figura 2.14: Ejemplo del uso de `label`

2.7. Lemas

Para verificar la corrección de un programa, tendemos a escribir las demostraciones manualmente. Sin embargo, cuando un programa es largo o cuando se necesita la misma demostración en diversas partes del código, lo más adecuado es encapsular la demostración en un lema y darle un nombre descriptivo. Estas propiedades se definen con la palabra reservada `lemma`. Los lemas son implícitamente métodos `ghost`, lo que significa que se usan exclusivamente con fines de verificación y no forman parte del código ejecutable. Los lemas no afectan el estado del programa ni crean nuevos objetos. Su capacidad para ser utilizados en expresiones los convierte en una herramienta valiosa para guiar al verificador y superar sus limitaciones deductivas.

Además, el uso de lemas ayuda a que el código sea más limpio y facilita la verificación al dividir una tarea en partes manejables.

Un lema consta de dos partes principales: la declaración y la demostración. Consideraremos el ejemplo de la figura 2.15. Dada la función `More`, ¿cómo comprobamos que siempre devuelve un valor mayor al parámetro de entrada?

El razonamiento para demostrar que `More(x) >= x` se basa en la distinción de casos, similar a la definición de la función. Si `x <= 0`, la demostración es directa, ya que la función devuelve 1, que es claramente mayor o igual que 0. Sin embargo, en el caso de `x > 0`, podemos razonar sobre el número de llamadas recursivas que se generan hasta que se aplica el caso base. La función añade 3 al resultado de cada llamada recursiva; de esta manera, si contásemos el número de llamadas recursivas que se ejecutan, seguramente podamos convencernos o encontrar plausible que efectivamente `More(x)` devuelve algo mayor que `x`.

Para demostrar que la función `More(x)` siempre devuelve un valor superior a su argumento, definiremos un lema. Primero, lo declararemos asignándole un nombre e indicando la propiedad que establecerá. Lo denominaremos `Increasing` y tomará un argumento `x`, asegurando que `x` es menor que `More(x)`, tal como se observa en la figura 2.15.

```
1 function More(x: int): int {
2     if x ≤ 0 then 1 else More(x−2) + 3
3 }
4
5 lemma Increasing(x: int)
6     ensures x < More(x)
```

Figura 2.15: Declaración del lema `Increasing`

Tras la declaración del lema, procederemos a su demostración. Esto implica desarrollar el cuerpo del lema, presentando el razonamiento y las justificaciones necesarias hasta verificar que la postcondición del lema se cumple. Al abrir las llaves del cuerpo del lema, no se aprecian errores y la postcondición se verifica automáticamente. Esto se debe a que el verificador aplica una inducción que resulta suficiente para demostrar este lema sencillo. Sin embargo, si quisiéramos prescindir de este automatismo de Dafny, declarando el lema con la cabecera `lemma { :induction false } Increasing(x: int)`, observaríamos que la postcondición no se cumple de forma automática.

En la figura 2.16 vemos la demostración del lema, esta se divide en casos como el razonamiento visto anteriormente.

- El primer caso (`x <= 0`) es trivial y se verifica automáticamente.
- El caso donde `x > 0` no se verifica de inmediato, pues hay que elaborar un poco más la demostración. En esta rama, sabemos que la función `More(x)` se define como `More(x-2) + 3`. Por lo tanto, técnicamente necesitamos demostrar que `x < More(x-2) + 3`, o lo que es lo mismo: `x - 3 < More(x-2)`. El aspecto de esta última desigualdad es similar a la postcondición de nuestro lema (`x < More(x)`). Esto permite llamar a este mismo lema con el argumento `x-2`.

```
lemma {:induction false} Increasing(x: int)
    ensures x < More(x)
{
    if x ≤ 0 {}
    else {
        Increasing(x−2);
    }
}
```

Figura 2.16: Demostración del lema `Increasing`

Esto completa la demostración, ya que el verificador es capaz de comprobar la terminación del lema debido a su naturaleza recursiva. Esto se debe a que Dafny infiere automáticamente la cláusula **`decreases x`**, garantizando que el argumento de la llamada recursiva disminuye en cada paso.

2.8. Instrucciones `assert ... by {}`

En Dafny, cuando una afirmación del tipo `assert X;` no se verifica automáticamente, conviene invocar lemas o insertar otras pruebas que ayuden a demostrarla debido a que pueden guiar al verificador de Dafny para encontrar la demostración de manera más eficiente. Un ejemplo típico es el de la figura 2.17.

```
assert SOMETHING_HELPING_TO_PROVE_LEMMA_PRECONDITION;
LEMMA();
assert X;
...
lemma ()
  requires LEMMA_PRECONDITION
  ensures X { ... }
```

Figura 2.17: Ejemplo de `assert`

Sin embargo, podemos encontrarnos con una desventaja importante, y es que Dafny sigue teniendo en cuenta todas las pruebas intermedias y postcondiciones de los lemas para afirmaciones posteriores. Esto puede ralentizar mucho la verificación, porque el verificador tiene que manejar más información de la necesaria. Para evitar este problema, se utilizan bloques **`assert ... by {}`** que sirven para encapsular demostraciones de manera local. Las pruebas utilizadas dentro del bloque (como lemas o afirmaciones intermedias) solo existen en ese ámbito, una vez que se cierra la llave, desaparecen y no están disponibles para el resto del código:

```
assert X by {
    assert SOMETHING_HELPING_TO_PROVE_LEMMA_PRECONDITION;
    LEMMA();
}
```

Figura 2.18: Uso de `assert ... by {}`

De este modo, para el verificador de Dafny, la figura 2.18 contiene un ejemplo de `assert ... by {}`, que es equivalente a una única instrucción `assert`. Es decir, solo el aserto de `X` queda visible para el verificador. Las pruebas auxiliares internas y las llamadas a lemas solo tienen efecto dentro del bloque.

2.9. Invariantes del bucle

Al verificar programas con bucles, Dafny introduce los invariantes de un bucle con la cláusula `invariant`. Un invariante de bucle es una propiedad que se declara para un bucle y que debe cumplirse justo antes de entrar al bucle y mantenerse cierta al finalizar cada iteración.

La figura 2.19 presenta un método en Dafny llamado `SquareRoot` que calcula la raíz cuadrada de un número natural `N`. El cálculo se basa en un bucle cuya ejecución continúa mientras el valor de `s` (el cuadrado de `r + 1`) sea menor o igual que el número de entrada `N`. Esto significa que la raíz cuadrada actual `r` aún podría ser menor que la raíz cuadrada entera de `N`. Para verificar la corrección del bucle, se define los siguientes invariantes:

- `invariant r * r <= N`: en cada iteración establece que el cuadrado del valor actual de `r` siempre debe ser menor o igual que `N`. Esto asegura que `r` nunca sobrepasa la raíz cuadrada entera de `N`.
- `invariant s == (r + 1) * (r + 1)`: establece la relación entre `s` y `r`. En cada iteración se garantiza que `s` siempre es igual al cuadrado de `r + 1`.

Inicialmente, en la entrada del bucle tenemos `r = 0` y `s = 1`. El primer invariante se convierte en `0 * 0 <= N`, que es verdadero para cualquier número natural `N`. Por otro lado, el segundo invariante se convierte en `1 == (0 + 1) * (0 + 1)`, que es verdadero. Además, el verificador de Dafny intentará probar que si los invariantes se cumplen al inicio de una iteración, también se cumplen al final. Las actualizaciones de `s` y `r` están diseñadas para mantener estas propiedades.

```
1  method SquareRoot(N: nat) returns (r: nat)
2      ensures r * r ≤ n < (r + 1) * (r + 1)
3  {
4      r := 0;
5      var s := 1;
6      while s ≤ N
7          invariant r * r ≤ N
8          invariant s = (r + 1) * (r + 1)
9      {
10         s := s + 2*r + 3;
11         r := r + 1;
12     }
13 }
```

Figura 2.19: Método `SquareRoot`

2.10. Demostraciones universales

Las demostraciones universales (*universal introduction*, en inglés) permiten ejecutar un bloque de código de forma simultánea para todos los valores que cumplen una condición dentro de un dominio. Su sintaxis básica es: `forall x | P(x) ensures Q(x) { I(x); }`, donde:

- `P(x)` es la condición que define el dominio de valores sobre los cuales se itera.

- `I(x)` es el bloque de instrucciones que se ejecuta para cada `x` que cumple `P(x)`.
- `Q(x)` es la propiedad que debe garantizarse para cada `x`.

Este tipo de instrucción se utiliza mucho para demostrar propiedades lógicas de formal general. Por ejemplo, si se quiere aplicar un mismo lema a todos los elementos que cumplen cierta propiedad, se puede usar `forall` para invocar ese lema en todos los casos relevantes. De esta manera, si ese lema garantiza una propiedad `Q(x)`, entonces la instrucción `forall` establece que `Q(x)` se cumple para todos los `x` que satisfacen `P(x)`. En la figura 2.20 vemos que se quiere verificar una propiedad sobre los números del 1 al 9, donde Dafny puede verificarlo automáticamente.

```
1 forall x | 0 < x ≤ 10
2 ensures x * x > 0
3 {
4
5 }
```

Figura 2.20: Ejemplo de demostración universal

2.11. Eliminación existencial (Skolemization)

Dafny también permite la eliminación existencial mediante la sintaxis: `var x :| P(x);`. Esta instrucción significa introducir un valor `x` tal que cumple la condición `P(x)`. Si no existe tal valor, el verificador reportará un error, pues se trata de una suposición lógica que debe estar justificada por las condiciones anteriores del programa. Ese tipo de construcción existencial en Dafny se usa cuando se necesita suponer que existe un valor (un testigo) que satisface una condición para usarlo en una demostración.

Un ejemplo de esta instrucción lo podemos observar en la figura 2.21. Aquí, la precondición del método supone la existencia de un cero (`x`) dentro del array `a`, que se puede usar después como si realmente lo hubiéramos encontrado.

```
1 method ThereIsZero(a: array<int>)
2     requires ∃ i | 0 ≤ i < a.Length • a[i] = 0
3 {
4     var i :| 0 ≤ i < a.Length ∧ a[i] = 0;
5     assert a[i] = 0;
6     ...
7 }
```

Figura 2.21: Ejemplo de eliminación existencial

Capítulo 3

Vuelta atrás

"Preferiría enseñar antes que hacer cualquier otra cosa en el mundo, y cuanto más enseño, más me gusta."
— D. H. Lehmer

En este capítulo se introduce el método algorítmico de vuelta atrás, conocido en inglés como *Backtracking*. Veremos que es una técnica fundamental en la resolución de problemas de satisfacción de restricciones y problemas de optimización. Analizaremos su funcionamiento, sus aplicaciones y las estrategias para mejorar su eficiencia.

En el ámbito computacional, existen diversos métodos algorítmicos diseñados para resolver problemas. Entre ellos, el método de vuelta atrás destaca por su capacidad de explorar de manera exhaustiva un espacio de soluciones. Se trata de un algoritmo que encuentra soluciones a problemas que deben satisfacer ciertas restricciones (problemas de satisfacción de restricciones) mediante un enfoque sistemático y exhaustivo de búsqueda. También es útil para problemas en los que interesa determinar la mejor solución, conocidos como problemas de optimización.

El término *backtrack* fue introducido por el matemático estadounidense D. H. Lehmer en la década de 1950 (4). Este concepto hace referencia a retroceder, rectificar o dar marcha atrás, y define la técnica fundamental de este algoritmo.

La vuelta atrás utiliza un enfoque sistemático para explorar todas las posibles soluciones mediante un árbol de búsqueda. En cada etapa el algoritmo explora todas las alternativas posibles descartando aquellas que no cumplen las restricciones del problema. La solución parcial se extiende con cada de uno de los posibles valores en esa etapa, y en caso de cumplir las restricciones del problema y una vez explorado el espacio de soluciones alcanzables desde esa solución parcial, se retrocede (*backtrack*) al estado anterior para poder explorar la siguiente alternativa. Este mecanismo de retroceso se implementa de forma natural mediante el uso de recursión, lo que permite gestionar el recorrido en profundidad en el árbol de decisiones.

Esta exhaustividad del algoritmo, propia de su estrategia de prueba y error, conlleva un coste computacional habitualmente exponencial. No obstante, en muchos casos es posible reducir su coste descartando soluciones que no superan la mejor solución encontrada hasta el momento. Este proceso se conoce como *poda de optimalidad* y consiste en realizar una estimación que nos permite decidir si vale la

pena continuar explorando ciertas ramas del árbol de búsqueda. De esta manera, se evita profundizar en ramas innecesarias, lo que puede reducir significativamente el espacio de búsqueda explorado en algunos casos.

Como podemos observar, este esquema algorítmico es de gran relevancia, especialmente en la resolución de problemas complejos donde el espacio de soluciones es amplio y no hay estrategias de selección local que garanticen óptimos globales. Puesto que se trata de algoritmos relativamente complejos de programar, una verificación formal que garantice su corrección para cualquier posible ejecución es de gran valor.

3.1. Aplicaciones

El esquema algorítmico de vuelta atrás se aplica a una variedad de problemas cuyos espacios de búsqueda son amplios. Estos problemas se pueden clasificar en dos categorías principales: problemas de satisfacción de restricciones y problemas de optimización. A continuación, se detallan sus características y se presentan ejemplos.

3.1.1. Problemas de satisfacción de restricciones

«Los problemas de satisfacción de restricciones, (CSP, por sus siglas en inglés) son problemas cuya solución debe cumplir una serie de restricciones» (2). Esta solución, en términos prácticos, generalmente se representa mediante una n-tupla $(x_0, ..., x_{n-1})$, donde $x_i \in S_i$ indica qué decisión se ha tomado en la etapa i-ésima de entre un conjunto finito de decisiones S_i.

En este caso, el esquema se utiliza para determinar si un problema es satisfactible. Si, tras explorar todo el espacio de búsqueda, no se encuentra ninguna solución, el problema se considera insatisfactible. En caso contrario, el algoritmo puede generar una o múltiples soluciones válidas.

Las restricciones que definen la validez de la solución se pueden clasificar en dos categorías principales (10):

- **Restricciones explícitas:** que definen los conjuntos S_i. Es decir, el conjunto finito de valores que pueden tomar las componentes de la tupla solución. Además, generalmente se establece el tamaño total que debe tener dicha solución.
- **Restricciones implícitas:** definen las relaciones entre componentes de la tupla solución.

Veamos un ejemplo detallado para ilustrar los conceptos de problemas de satisfacción de restricciones: **el problema de los funcionarios**.

1. **Enunciado:** dado un conjunto de n tareas y un grupo de n funcionarios, ¿es posible asignar exactamente una tarea a cada funcionario (y viceversa) de manera que se cumplan ciertas restricciones, como por ejemplo que determinados funcionarios no pueden realizar ciertas tareas?

2. **Entrada:** el problema recibe como entrada dos conjuntos: el de tareas y el de funcionarios. Ambos tienen el mismo número de elementos, n, y un conjunto de restricciones que indican qué tareas pueden ser realizadas por qué funcionarios. Estas restricciones se representan mediante una matriz de valores booleanos, donde las filas corresponden a los funcionarios y las columnas a las tareas, indicando qué tareas está dispuesto a realizar cada funcionario. Podemos representar las asignaciones como números enteros del 0 al $n-1$, definiendo así nuestros conjuntos finitos S_i. La satisfactibilidad depende del número y tipo de restricciones impuestas: si, por ejemplo, cada funcionario puede realizar al menos una tarea distinta, el problema puede ser satisfactible.

3. **Restricciones**

   - **Explícitas:**
     - El tamaño de la solución debe ser n reflejando las asignaciones funcionario-tarea.
     - Tenemos n tareas disponibles, luego $S_i = \{0, ..., n-1\}$.
     - Cada funcionario debe realizar exactamente una tarea.
   - **Implícitas:**
     - No puede haber dos funcionarios realizando la misma tarea.
     - No se pueden violar las restricciones impuestas.

4. **Salida:** La solución se representa como una n-tupla $(x_0, x_1, ..., x_{n-1})$, donde cada x_i indica la tarea asignada al funcionario i (con $x_i \in S_i$). Por ejemplo, para $n = 3$, y siendo la matriz de booleanos de restricciones la siguiente:

$$\begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}$$

   una posible tupla que satisface las restricciones explícitas podría ser:

$$(2, 0, 1)$$

   Esto indica que:

   - El funcionario 0 realiza la tarea 2.
   - El funcionario 1 realiza la tarea 0.
   - El funcionario 2 realiza la tarea 1.

   Sin embargo, (2,1,0) no sería solución porque no satisface las restricciones implícitas del problema. En particular, el funcionario 0 tiene asignado la tarea 2, que no puede realizar, dado que `matriz[0][2] = 0`.

Este ejemplo ilustra cómo se definen y estructuran los problemas de satisfacción de restricciones. Otros ejemplos comunes incluyen el Sudoku, el problema de las 8 reinas, el problema de satisfacción booleana y el problema del coloreado de grafos.

3.1.2. Problemas de optimización

Los problemas de optimización representan una subcategoría importante dentro de los problemas de satisfacción de restricciones (CSP). A diferencia de los CSP generales, donde el objetivo es simplemente encontrar soluciones que cumplan con un conjunto de restricciones, los problemas de optimización buscan la **mejor solución** posible según un criterio específico. Este criterio consiste en maximizar o minimizar una **función objetivo**. En otras palabras, además de satisfacer las restricciones del problema, se busca optimizar un valor determinado, como minimizar un coste o maximizar un beneficio.

Retomando el ejemplo anterior del problema de los funcionarios descrito en la sección 3.1, podríamos ahora querer no solo encontrar una asignación válida de tareas, sino además obtener un beneficio adicional: que dicha asignación minimice el tiempo total invertido en completar todas las tareas. Para ello, se necesita como entrada una matriz de tiempos que refleje el tiempo que tarda cada funcionario en realizar las distintas tareas. Consideraremos la versión del problema en la que todos los funcionarios pueden realizar todas las tareas, por lo que no será necesaria la matriz de booleanos como dato de entrada. De todas formas, esta versión no impide representar restricciones adicionales porque estas se pueden conseguir utilizando un valor infinito (o muy elevado) en la matriz de tiempos para aquellas parejas funcionario-tarea que no sean posibles. De esa forma las soluciones que contuviesen esas combinaciones se descartarían por ser peores que cualquier otra.

Así, el **problema de los funcionarios** se convierte en un problema de optimización. El problema que vamos a tratar en este trabajo consiste en asignar biunívocamente un conjunto de tareas a un grupo de trabajadores de manera que se **minimice el tiempo total** invertido en realizar dichas tareas.

A continuación, se presentan otros ejemplos comunes de problemas de optimización:

- **El problema del viajante (TSP, Travelling Salesman Problem):** Dado un conjunto de ciudades y las distancias entre cada par de ellas, el objetivo es encontrar la **ruta más corta** que permita visitar todas las ciudades exactamente una vez y regresar a la ciudad de origen.

- **El problema de la mochila (Knapsack Problem):** Dado un conjunto de objetos, cada uno con un peso y un valor, y una mochila con una capacidad máxima de peso, el objetivo es seleccionar un subconjunto de objetos (que no pueden fraccionarse) que **maximice el valor total** sin exceder la capacidad de la mochila.

Más adelante, profundizaremos en los ejemplos del problema de la mochila y del problema de los funcionarios para verificar sus correspondientes algoritmos de vuelta atrás.

3.2. Descripción del método

En esta sección veremos cómo funciona el esquema algorítmico de vuelta atrás y analizaremos su implementación, enfocándonos en los problemas de optimización. Como ejemplo, utilizaremos el problema de asignación de tareas a funcionarios.

3.2.1. Búsqueda en el espacio de soluciones

El esquema algorítmico de vuelta atrás realiza una búsqueda exhaustiva en el espacio de soluciones para encontrar una solución que satisfaga todas las restricciones del problema. Este espacio de soluciones abarca todas las posibles combinaciones que cumplen con las restricciones explícitas, es decir, el conjunto de todas las posibles tuplas $(x_0, x_1, ...x_{n-1})$ que podrían resolver el problema. Pero no todas las tuplas son soluciones válidas, solo aquellas que también cumplen con las restricciones implícitas.

El espacio de búsqueda se puede estructurar como un árbol de búsqueda donde:

- Cada **nivel** del árbol corresponde a una componente x_i de la tupla solución.
- Cada **rama** representa una posible decisión, es decir, un valor que puede tomar la componente x_i.

El algoritmo recorre el árbol de búsqueda de manera sistemática, de tal manera que, en cada nivel, se asigna un valor a la componente x_i. De esta manera, cualquier nodo (tupla) del árbol de exploración que satisfaga las restricciones explícitas, es una **solución parcial**. Por otro lado, una **solución completa** habitualmente la podemos encontrar en los nodos hoja (aquellos que no tienen nodos hijos), pero en otras ocasiones se pueden hallar en cualquier otro nodo del árbol dependiendo de las restricciones del problema. Los nodos solución que resuelven el problema serán los correspondientes a las soluciones completas que además satisfagan las restricciones implícitas.

La estructura del árbol de exploración no se representa explícitamente en memoria, sino que se construye de forma implícita mediante recursión. Para habilitar los retrocesos del algoritmo, se realiza un recorrido en profundidad.

Veamos cómo se estructura el espacio de búsqueda en el problema de los funcionarios descrito en la sección 3.1. Dado que la decisión en cada etapa es asignar una tarea a un funcionario, los niveles del árbol representan a los funcionarios y las ramas las posibles tareas que pueden realizar, tal y como se observa en la figura 3.1. Como vemos, este árbol muestra todas las posibles combinaciones de asignación de tareas a funcionarios cuando $n = 3$.

La exploración mediante el recorrido en profundidad se haría de la siguiente manera. Comienza asignando la tarea 0 al funcionario 0, luego, asigna la tarea 0 al funcionario 1 y finalmente, asigna la tarea 0 al funcionario 2. Al no encontrar más soluciones en esa rama, se retrocede un paso (al último funcionario, el 2) y prueba

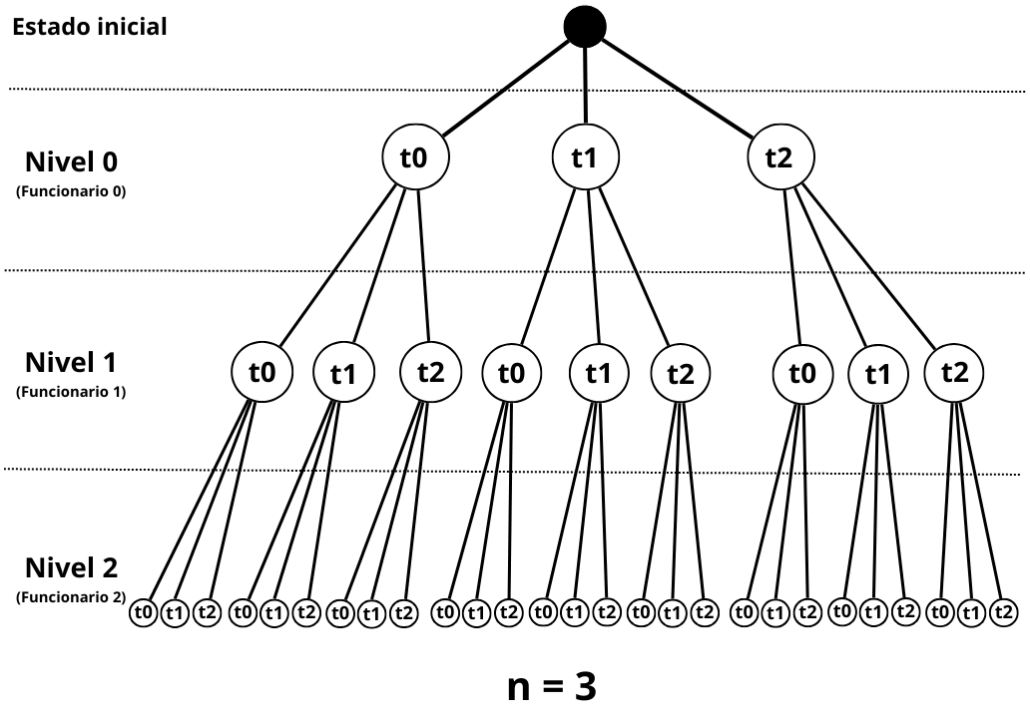


Figura 3.1: Árbol de búsqueda del problema de los funcionarios para $n = 3$

con la siguiente tarea disponible (tarea 1). Un algoritmo de fuerza bruta exploraría todas las combinaciones posibles de asignación, incluso aquellas en las que múltiples funcionarios comparten la misma tarea. Sin embargo, la vuelta atrás evita explorar soluciones inviables, si en algún punto se detecta que dos funcionarios tienen la misma tarea asignada, el algoritmo poda esa rama y retrocede. Profundizaremos en estas técnicas en las siguientes secciones.

3.2.2. Elementos esenciales

El esquema algorítmico de vuelta atrás se adapta y se optimiza a través de varias componentes fundamentales. Estas componentes no solo definen la mecánica del algoritmo, sino que también son cruciales para su eficiencia y aplicabilidad en la resolución de problemas. A continuación, exploraremos cada uno de estos componentes.

- **Función de selección**: es la responsable de elegir la siguiente opción a explorar en cada paso del algoritmo. Esta función define el orden en que se generan las soluciones parciales y, por lo tanto, influye en la eficiencia del algoritmo. En el ejemplo de los funcionarios, la función de selección podría elegir la siguiente tarea para un funcionario en específico (por ejemplo, en orden de identificadores: tarea 0, tarea 1, ..., tarea $n - 1$).

- **Test de factibilidad**: verifica si una solución parcial cumple con las restricciones implícitas del problema. Si la solución parcial no es factible, el método

```
VA(Tupla & sol, int k, Tupla & solMejor) {
    if (esSolucion(sol, k)) {
        casoBase(sol);
    }
    else {
        for i = 0 to n-1 {
            sol[k] = i;
            if (esValida(sol,k)) {
                VA(sol,k+1,solMejor);
            }
        }
    }
}
```

Figura 3.2: Esquema del método de vuelta atrás

retrocede y explora otra opción. En el problema de los funcionarios, la función de factibilidad comprobaría que no haya funcionarios realizando la misma tarea.

- **Test de solución**: verifica si una solución es completa. En el problema de los funcionarios, el test de selección consiste en comprobar que todos los funcionarios tienen una tarea asignada.

3.2.3. Esquema

El método de vuelta atrás, como hemos visto, sigue una exploración sistemática del espacio de soluciones. Para profundizar en su funcionamiento, analizaremos su esquema básico. En la figura 3.2 podemos observar el pseudocódigo detallado que ilustra la lógica del algoritmo.

La función `VA` implementa el método de vuelta atrás, resolviendo el problema de manera recursiva. Recibe como parámetro un índice `k` que representa la etapa actual de la búsqueda, correspondiente al nivel del árbol de exploración. Este índice se inicializa en 0 y va aumentando progresivamente para realizar el recorrido en profundidad. Esta función también recibe un vector `sol` que refleja la tupla solución, donde cada una de sus posiciones representa una etapa; así, en cada paso del algoritmo, se asigna `sol[k]`.

La ramificación del árbol (posibles opciones de asignación para el nivel `k`) se obtiene mediante el bucle `for`. Este bucle genera n ramas, donde n es el número de opciones disponibles para el nivel actual `k`.

La función `esValida` implementa el test de factibilidad. Esta función contiene la lógica necesaria para determinar si una solución parcial es factible hasta el nivel actual `k`, verificando que cumpla con las restricciones implícitas del problema.

La función `esSolucion` comprueba si el vector solución es una solución al problema. Generalmente, esta función comprueba si la etapa `k` es igual al tamaño del vector (`k == sol.size()`), lo cual significa que todas las componentes del vector han sido tratadas. En otros problemas concretos, el vector puede considerarse solución incluso cuando `k < sol.size()`.

```
void FuncionariosVA(vector<int> & sol, int tiempo,
                    vector<int> & solMejor, int tiempoMejor, int k,
                    vector<vector<int>> const & T) {
    if (esSolucion(sol)) {
        casoBase(sol, tiempo, solMejor, tiempoMejor);
    }
    else {
        for (int i = 0; i < m; i++) {
            sol[k] = i;
            tiempo += T[k, i];
            if (esValida(sol,k)) {
                FuncionariosVA(sol, tiempo, solMejor, tiempoMejor,
                               k + 1, T);
            }
            tiempo -= T[k, i];
        }
    }
}

bool esValida(vector<int> const & sol, int k) {
    int i = 0;
    while (i < k) {
        if (sol[i] == sol[k]) {
            return false;
        }
        i++;
    }
    return true;
}

bool esSolucion(vector<string> const & sol, int k) {
    return k == sol.size();
}

void casoBase(vector<int> const & sol, int tiempo,
              vector<int> const & solMejor, int tiempoMejor) {
        if (tiempo < tiempoMejor) {
            solMejor = sol;
            tiempoMejor = tiempo;
        }
}
```

Figura 3.3: Implementación del problema de los funcionarios (optimización)

La función `casoBase` define la acción a realizar cuando se encuentra una solución. En problemas de satisfacción, podemos simplemente imprimir las soluciones, mientras que en problemas de optimización es necesario comparar la solución encontrada con la mejor solución almacenada hasta el momento. De este modo, se optimiza el proceso y el algoritmo finaliza con la mejor solución de todas las posibles del árbol.

Una vez explicado el esquema básico del algoritmo, veamos cómo se aplica a la versión de optimización del problema de los funcionarios. Una implementación en C++ podría ser la que aparece en la figura 3.3.

La función que resuelve el problema es `FuncionariosVA` (líneas 1 a 18), que recibe un vector `sol` y una etapa `k`. Nuestro vector `sol` representa la tupla en la que se construirá la solución del problema. El valor de `k`, que representa el nivel, corresponde a los funcionarios. Inicialmente, con `k = 0`, comenzaríamos con el funcionario 0, y, por ejemplo, para $n = 3$, el algoritmo terminaría cuando `k = 3`, que representaría que ya se han tratado todos los funcionarios. Esta condición se comprueba con la función `esSolucion` (líneas 31 a 33). Como se busca minimizar el tiempo invertido en realizar las tareas, será necesario almacenar la solución actual, la mejor solución encontrada hasta el momento, el tiempo actual que corresponde a la solución actual y el mejor tiempo que corresponde a la mejor solución.

La función `esValida` (líneas 20 a 29), en este caso, debe verificar que el funcionario `k`, que acaba de ser asignado, no tenga la misma tarea que algún otro funcionario, ya que esa es la restricción implícita del problema. Para ello, se recorre el vector solución desde el inicio hasta la posición anterior a `k`, comprobando que ninguno de los funcionarios anteriores haya sido asignado a la misma tarea.

Finalmente, la función `casoBase` (líneas 35 a 41) se invoca cada vez que se encuentra una solución. Este debe verificar si el tiempo actual correspondiente a una solución completa es menor que el mejor tiempo guardado. Si es así, se considerará una nueva mejor solución y deberá reemplazar a la anterior. Esto se consigue comparando el valor de `tiempo` y `tiempoMejor`. En caso de que `tiempo` sea estrictamente menor que `tiempoMejor`, se actualizan las variables correspondientes, de modo que `solMejor` almacena la solución actual `sol` y `tiempoMejor` toma el valor de `tiempo`. De esta forma, en todo momento se conserva la mejor solución encontrada hasta el momento.

3.2.4. Técnica de marcaje

La técnica de marcaje consiste en guardar cierta información que ayuda a decidir en menos tiempo si una solución parcial es válida o no. La información del marcaje se pasa en cada llamada recursiva, por lo tanto, reduce el coste computacional a cambio de utilizar más memoria (10).

Veamos esta técnica en el problema de los funcionarios. En la figura 3.3 hemos observado que el test de factibilidad (reflejado en la función `esValida`) consiste en un recorrido de las componentes anteriores del vector `sol`, lo que hace que tenga un coste lineal. Podemos conseguir que dicha función tenga coste constante mediante la técnica de marcaje. La implementación de esta técnica se traduce en lo siguiente:

- Se utiliza un vector booleano `marcas` de tamaño n (número de tareas), donde `marcas[i]` indica si la tarea i ha sido asignada a un funcionario o no. Este vector se inicializa con todas sus componentes a `false`, indicando que al principio todas las tareas están disponibles y no han sido asignadas a ningún funcionario. Cada posición i de dicho vector es actualizada a `true` cuando la tarea i es asignada a un funcionario.

- Cuando se decide la tarea que va a realizar el funcionario actual `k` (`sol[k] = i`), se llama a la función `esValida`, que verifica el valor de `marcas[i]`. Si `marcas[i]` es

verdadero, significa que la tarea ya había sido asignada, y la solución parcial es inválida.

Como se puede observar en el código de la figura 3.4, se muestra el código con un marcador. Así, la función `esValida` se reduce a una simple verificación con un coste constante.

En lugar de un vector booleano, se puede utilizar un conjunto que almacene las tareas que han sido asignadas. En este caso, la función `esValida` consultaría la pertenencia de la tarea i a dicho conjunto.

3.2.5. Podas de optimalidad

Las podas de optimalidad son técnicas que permiten reducir el espacio de búsqueda explorado por el algoritmo de vuelta atrás, mejorando significativamente su eficiencia. Estas técnicas surgen porque, en muchos problemas de optimización, es posible determinar si una solución parcial no puede conducir a una solución óptima.

Las podas de optimalidad consisten en calcular una estimación del valor de la mejor solución que se puede obtener a partir de una solución parcial dada. Si esta estimación no mejora el coste de la mejor solución encontrada hasta el momento, se puede podar la rama del árbol de búsqueda correspondiente a esta solución parcial. Esta estimación actúa como una cota (superior o inferior) del valor óptimo alcanzable, dependiendo del tipo de problema:

- **Cota inferior:** se calcula en problemas de minimización, si esta es mayor que el valor de la mejor solución almacenada, la rama puede descartarse, ya que no puede conducir a una solución mejor.
- **Cota superior:** se calcula en problemas de maximización, y si es menor que el valor de la mejor solución ya encontrada, se poda la rama.

En el problema de los funcionarios, al ser un problema de minimización, se necesitará una cota inferior. La estimación se lleva a cabo desde k hasta $n-1$, abarcando a los funcionarios sobre los cuales aún no se ha tomado una decisión.

Para la solución parcial $(x_0, ..., x_{k-1})$, el tiempo hasta el momento es $tiempo = \sum_{i=0}^{k-1} T[i, x_i]$ donde T es una matriz de tiempos en la que las filas representan a los funcionarios y las columnas a las tareas.

A continuación se presentan varias cotas ordenadas de la más sencilla (más optimista) a la más compleja que realiza más podas (10):

1. Considerar que el resto de funcionarios no van tardar nada en realizar sus tareas:

$$\boldsymbol{cota = tiempo \ + \ 0}$$

2. Calcular el mínimo global de la matriz T:

$$minT = mín\{\ T[i,j] \mid 0 \leq i < n \ \wedge 0 \leq j < n\ \}$$

y considerar que el resto de funcionarios tardan el mismo tiempo y el mínimo posible:

$$\boldsymbol{cota = tiempo + minT \cdot (n - k - 1)}$$

3. Calcular el mínimo de la submatriz de T que comienza desde la fila k en adelante:

$$minSubmatriz = mín\{\ T[i,j] \mid k \leq i < n\ \wedge 0 \leq j < n\ \}$$

y considerar que el resto de funcionarios tardan el mismo tiempo y el mínimo posible:

$$\boldsymbol{cota = tiempo\ +\ minSubmatriz \cdot (n - k - 1)}$$

Esta cota es similar a la anterior, ya que también estima el tiempo restante asumiendo que los funcionarios que aún no han sido asignados tardarán el mínimo posible. Sin embargo, es más alta porque el mínimo se calcula solo dentro de la submatriz a partir de la fila k, es decir, considera únicamente los funcionarios que aún no han sido asignados. Así, se reduce el conjunto de valores posibles y el mínimo resultante puede ser mayor o igual que el mínimo global de toda la matriz T.

4. Tener calculado el mínimo por cada fila, es decir, para cada funcionario calcular cuánto tarda en realizar el trabajo que realiza más rápidamente:

$$rápido[i] = min\ \{\ T[i,j] \mid 0 \leq j < n\ \}$$

De este modo podemos calcular la cota como sigue:

$$\boldsymbol{cota = \sum_{i=k+1}^{n-1} rapido[i]}$$

Todas estas cotas, excepto la primera, necesitan un precálculo antes de llamar a la función que resuelve el problema.

En la figura 3.5 podemos ver el precálculo y el uso de la cota 2. El precálculo consiste en un bucle que calcula el mínimo global de la matriz T. En la función `FuncionariosVA`, esta cota se compara con el coste de la mejor solución encontrada hasta el momento. Solo si la estimación es mejor (inferior, ya que queremos minimizar) que el tiempo de la mejor solución se hace la llamada recursiva. En caso contrario, se descarta la rama correspondiente del árbol de búsqueda, podando así soluciones parciales no prometedoras.

```
void FuncionariosVA(vector<int> & sol, int tiempo,
                    vector<int> & solMejor, int tiempoMejor, int k,
                    vector<vector<int>> const & T,
                    vector<int> & marcas) {
    if (esSolucion(sol)) {
        casoBase(sol, tiempo, solMejor, tiempoMejor);
    }
    else {
        for (int i = 0; i < m; i++) {
            sol[k] = i;
            tiempo += T[k, i];
            if (esValida(sol,k)) {
                marcas[i] = true;
                FuncionariosVA(sol, tiempo, solMejor, tiempoMejor,
                               k + 1, T, marcas);
                marcas[i] = false;
            }
            tiempo -= T[k, i];
        }
    }
}

void esValida(vector<int> const & sol, int k,
              vector<bool> const & marcas) {
    return !marcas[i];
}

bool esSolucion(vector<string> const & sol, int k) {
    return k == sol.size();
}

void casoBase(vector<int> const & sol, int tiempo,
              vector<int> const & solMejor, int tiempoMejor) {
    if (tiempo < tiempoMejor) {
        solMejor = sol;
        tiempoMejor = tiempo;
    }
}
```

Figura 3.4: Implementación del problema de los funcionarios (optimización) con marcaje

```
int precalculo(vector<vector<int>> const & T){
    int min = T[0,0];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (T[i,j] < min)
                min = T[i,j];
        }
    }
    return min;
}

void FuncionariosVA(vector<int> & sol, int tiempo,
                    vector<int> & solMejor, int tiempoMejor, int k,
                    vector<vector<int>> const & T,
                    vector<int> & marcas) {
    if (esSolucion(sol)) {
        casoBase(sol, tiempo, solMejor, tiempoMejor);
    }
    else {
        for (int i = 0; i < m; i++) {
            sol[k] = i;
            tiempo += T[k, i];
            if (esValida(sol,k)) {
                marcas[i] = true;
                int cota = tiempo + min * (n - k - 1);
                if (cota < tiempoMejor) {
                    FuncionariosVA(sol, tiempo, solMejor,
                                   tiempoMejor, k + 1, T, marcas);
                }
                marcas[i] = false;
            }
            tiempo -= T[k, i];
        }
    }
}
```

Figura 3.5: Implementación del problema de los funcionarios con poda de optimalidad

Capítulo 4

Metodología utilizada

"Mi corazonada es que [P ?= NP] será resuelto por un joven investigador que no esté limitado por demasiada sabiduría convencional sobre cómo abordar el problema."
— Richard Karp

Una vez introducido el método de vuelta atrás y la herramienta Dafny, es momento de abordar el aspecto central de este proyecto: la especificación y la verificación del algoritmo. En este trabajo se han verificado dos problemas de optimización concretos: el problema de la mochila y el problema de los funcionarios. En este capítulo se explicará la metodología utilizada mediante el problema de la mochila y en el siguiente capítulo veremos como se aplica dicha metodología al problema de los funcionarios.

4.1. El problema de la mochila

El problema de la mochila (*Knapsack problem*, en inglés), es un problema clásico en los problemas de optimización. Es uno de los problemas más estudiados y se trata de uno de los 21 problemas NP-completos de Richard Karp (1). Una estrategia común para abordarlo implica el uso del método algorítmico de vuelta atrás.

4.1.1. Enunciado

El problema de la mochila se puede enunciar de la siguiente manera: Dada una mochila de capacidad M, y una serie de ítems con cada uno un peso w_i y un valor v_i, ¿qué ítems debemos meter (sin fraccionar) en la mochila sin sobrepasar el peso de la misma para obtener el máximo valor posible?

Analizando el enunciado, nos daremos cuenta de que se trata de un problema de optimización, ya que el objetivo es encontrar aquella combinación de ítems que maximice el valor total obtenido.

4.1.2. Resolución

Antes de resolver el problema debemos analizar bien los componentes esenciales y ver cómo es el espacio de soluciones, de esta manera sabremos cómo estructurar bien la búsqueda.

- **Entrada**: la entrada esta compuesta por el peso máximo M y una lista de ítems que tienen un peso y un valor cada uno.
- **Restriciones**: este problema cuenta con las siguientes restricciones.
  - **Explícita:** la solución al problema es una n-tupla $(x_0, x_1, ..., x_{n-1})$, siendo n el número de ítems que tenemos y x_i la decisión de seleccionar o no el ítem i para meterlo en la mochila. Por tanto el conjunto de decisiones S_i está compuesto por solo dos opciones: **Sí** o **No**.
  - **Implícita**: la suma de los pesos de los ítems seleccionados no debe superar el peso máximo M:
$$\sum_{i=0}^{n-1} x_i p_i \leq M$$
- **Función objetivo**: al ser un problema de optimización, debemos tener en cuenta cuál es la función objetivo. Esta es la que establece el criterio a maximizar, es decir, la suma de los valores de los ítems seleccionados debe ser la maxima posible:
$$\sum_{i=0}^{n-1} x_i v_i$$
- **Salida**: la salida es una solución que cumple las restricciones explícitas y que tiene un valor mayor que todas las demás soluciones.

Las tablas 4.1 y 4.2 describen la correspondencia entre los ítems del problema y su representación interna mediante índices y valores booleanos.

Ítem	**Índice de la tupla**
Ítem 0	0
Ítem 1	1
...	...
Ítem $n-1$	$n-1$

Tabla 4.1: Relación entre los ítems y los índices asignados

¿El objeto es seleccionado?	**Valor de x_i de la tupla**
Sí	1 (`true`)
No	0 (`false`)

Tabla 4.2: Relación entre las posibles decisiones y los valores de la componente

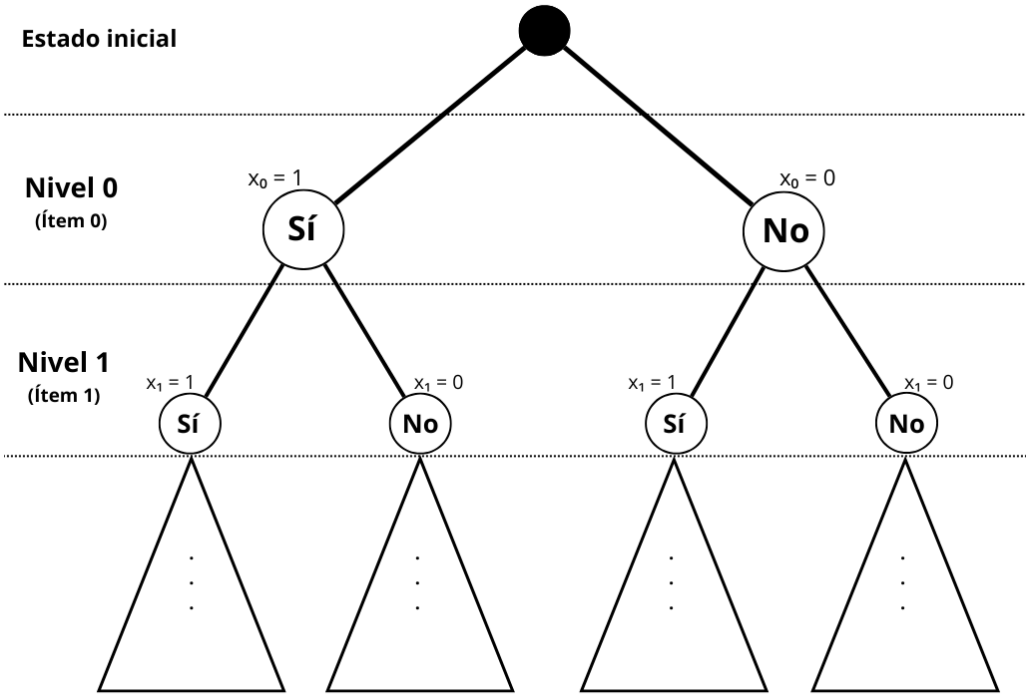


Figura 4.1: Árbol de búsqueda del problema de la mochila

Dado que para cada objeto solo hay dos decisiones posibles (seleccionado o no), el árbol de exploración es binario. En la figura 4.1 se puede observar que cada nivel del árbol representa un ítem y sus ramas corresponden a las dos opciones disponibles.

4.2. Modelización del problema

Para llevar a cabo la verificación del problema de la mochila, es necesario distinguir entre dos niveles fundamentales: la especificación y la implementación. Todos los ficheros se organizan en el directorio `Knapsack` en dos subdirectorios: `Implementation`, que alberga la lógica de implementación, y `Specification`, que contiene todos los elementos necesarios para la especificación. En la implementación se trabaja con objetos cuyo estado cambia durante la ejecución del algoritmo, mientras que la especificación establece propiedades sobre los valores que tienen esos objetos en determinados instantes de la ejecución, como si fuesen fotografías del estado de los mismos. Por ello, para representar los datos de la entrada y la salida del algoritmo utilizaremos **clases** de Dafny, que permiten representar datos mutables; mientras que para representar los valores de dichos objetos en la especificación usaremos **tipos de datos algebraicos** de Dafny, ya que se trata de tipos inmutables. Cada clase y cada tipo de datos algebraicos han sido implementados en un fichero independiente, cuyo nombre corresponde al del componente seguido de la extensión *.dfy*.

Para obtener el valor abstracto, también llamado modelo, de un objeto, cada una de las clases proporciona la función `Model()`. En la figura 4.2 se muestran las

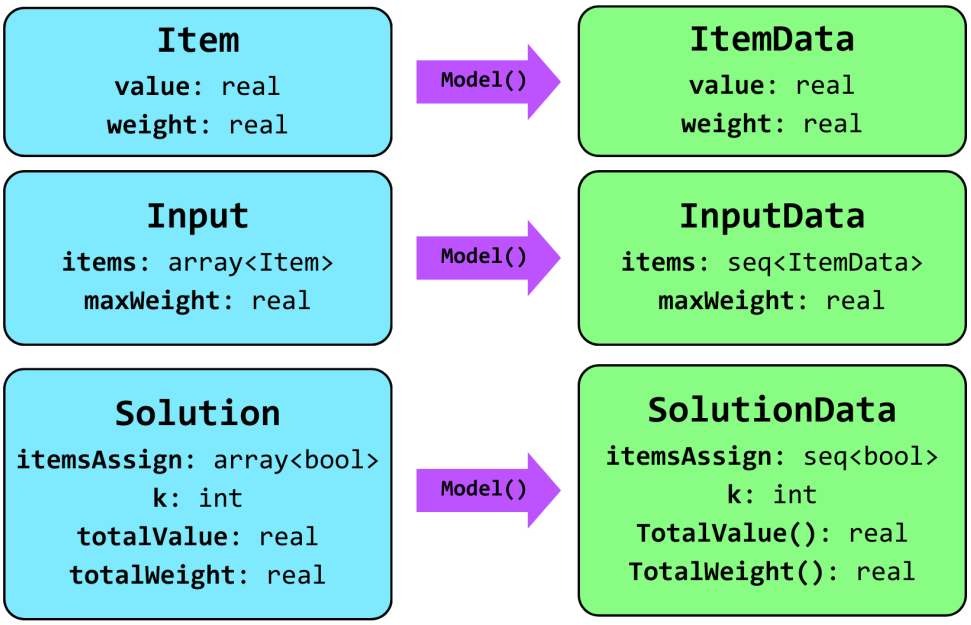


Figura 4.2: Correspondencia entre implementación y su modelo

clases utilizadas para representar la entrada y salida del algoritmo (`Item`, `Input` y `Solution`) y así como los tipos algebraicos correspondientes utilizados en la especificación (`ItemData`, `InputData` y `SolutionData`).

Las clases que definen los tipos de la entrada y salida del algoritmo son:

- **`Item`**: clase que implementa un ítem del problema. Cada ítem cuenta con dos atributos: peso y valor.
- **`Input`**: clase que implementa la entrada del problema. Contiene el peso máximo (`maxWeight`) y un array de tipo `Item` para guardar los ítems con su peso y valor.
- **`Solution`**: clase que implementa una representación formal de las soluciones del problema. Esta clase almacena varios atributos:
  - `itemsAssign`: array de booleanos de tamaño igual al número de ítems, donde cada posición corresponde a un ítem y cuyo valor almacenado indica si el ítem ha sido seleccionado (`true`) o no (`false`).
  - `totalValue`: valor total acumulado de los ítems escogidos (los asignados a `true` en `itemsAssign`).
  - `totalWeight`: peso total acumulado de los ítems escogidos (los asignados a `true` en `itemsAssign`).
  - `k`: etapa del árbol de exploración de la solución.

Los tipos algebraicos que representan los valores de los objetos en un instante de ejecución y que se utilizan para definir propiedades acerca de ellos son:

- **`ItemData`**: tipo de datos que modela formalmente los ítems del problema.

```
ghost function TotalWeight(items: seq<ItemData>): real
    decreases k
    requires Explicit(items)
{
    if k = 0 then
      0.0
    else if itemsAssign[k-1] then
      SolutionData(itemsAssign, k - 1).TotalWeight(items) + items[k-1].weight
    else
      SolutionData(itemsAssign, k - 1).TotalWeight(items)
}
```

Figura 4.3: Definición de la función `TotalWeight`

- `InputData`: tipo de datos que modela formalmente la entrada del problema. Contiene el peso máximo (`maxWeight`) y la secuencia de tipo `ItemData` (`items`) que modela el array contenido en `Input`.

- `SolutionData`: tipo de datos que modela formalmente una solución. Está compuesto por una secuencia de booleanos (`itemsAssign`) y una etapa `k`. Ambos atributos modelan los atributos correspondientes de la clase `Solution`. A diferencia de esta, `SolutionData` no almacena directamente los valores acumulados de la solución (`totalValue` y `totalWeight`), en su lugar proporciona dos funciones `ghost` que calculan dichos valores a partir del contenido de `itemsAssign` y `k`. En la figura 4.3 se puede observar la definición recursiva de la función `TotalWeight`. La función `TotalValue` es similar.

Tanto en las clases como en los tipos algebraicos se definen no solamente sus componentes sino también funciones y predicados `ghost` auxiliares, lemas que demuestran propiedades de dichas funciones y métodos. Estas funciones, al igual que muchas otras en la implementación, delegan parte de su comportamiento a funciones definidas en el modelo. Veamos los predicados que determinan la validez de cada uno de los componentes del problema. Empezando por la clase `Input`, el predicado que comprueba si una entrada del problema es válida, es `Valid` (figura 4.4). Este predicado llama al predicado `Valid` del modelo transfiriendo la lógica al mundo de la especificación (figura 4.5), donde es más sencillo formalizar y razonar sobre el comportamiento de todos los componentes. En este predicado podemos ver que los atributos de una estancia de tipo `Input` deben cumplir lo siguiente:

- Todos los elementos del array `items` deben ser válidos. Cada elemento es de tipo `Item`. Esta clase define su propia función de validez: comprueba que tanto el peso como el valor del ítem sean positivos.

- El peso máximo de la mochila `maxWeight` debe ser positivo.

Por otro lado, es importante entender que la diferencia entre una solución parcial y una solución completa (óptima) radica en el valor del índice `k`. La solución completa debe haber considerado todos los ítems (`k == bs.itemsAssign.Length`). En cambio, una solución parcial puede estar en proceso de construcción, por lo que basta con que `k <= bs.itemsAssign.Length`. Por este motivo, se definen dos predicados de validez en la clase `Solution`: `Partial` y `Valid`. El primero permite que el índice `k`

```
ghost predicate Valid()
    reads this, items, set i | 0 ≤ i < items.Length • items[i]
{
    this.Model().Valid()
}
```

Figura 4.4: Definición del predicado `Valid` de `Input`

```
ghost predicate Valid()
{
    ∧ (forall i | 0 ≤ i < |items| • items[i].Valid())
    ∧ maxWeight ≥ 0.0
}
```

Figura 4.5: Definición del predicado `Valid` de `InputData`

sea menor o igual al número de ítems, mientras que el segundo exige que se hayan procesado todos los elementos.

El predicado `Partial`, mostrado en la figura 4.6, incluye el invariante de representación que indica que:

- La secuencia de valores del array cumple las restricciones explícitas e implícitas.
- Los campos `totalValue` y `totalWeight` contienen respectivamente la suma de los valores y pesos de los ítems que en el array `itemsAssign` están marcados como seleccionados.

Además, `Partial` llama a la función correspondiente del modelo y define las restricciones explícitas e implícitas (figura 4.7). Las restricciones explícitas se definen en el método `Explicit`, en este caso, solo se asegura que el tamaño de la solución coincida con el tamaño del vector `items`, que es el número de ítems disponibles. En `Implicit` tenemos las restricciones implícitas, como vemos, se comprueba que el peso total no sobrepase el peso máximo. La función `Valid` (figura 4.7), que valida soluciones completas, puede entenderse como una versión más restrictiva de `Partial`. Su definición incluye directamente la llamada a `Partial(input)` y añade la condición `k == |itemsAssign|`, que asegura que la solución está completamente construida, es decir, que se hayan procesado todos los ítems.

4.3. Método principal

Una vez presentados los componentes que permiten modelar y resolver el problema de la mochila, en esta sección se detallará la especificación e implementación de su resolución. Además, se explicarán las herramientas empleadas para llevar a cabo la verificación en Dafny.

```
1    ghost predicate Partial (input : Input)
2      reads this, this.itemsAssign, input,
3      input.items, set i | 0 ≤ i < input.items.Length • input.items[i]
4      requires input.Valid()
5
6  {
7      ∧ 0 ≤ this.k ≤ this.itemsAssign.Length
8      ∧ Model().Partial(input.Model())
9      ∧ Model().TotalWeight(input.Model().items) = totalWeight
10     ∧ Model().TotalValue(input.Model().items) = totalValue
11 }
```

Figura 4.6: Definición del predicado `Partial` de `Solution`

```
1  ghost predicate Partial (input: InputData)
2      requires input.Valid()
3  {
4      ∧ Explicit(input.items)
5      ∧ Implicit(input.items, input.maxWeight)
6  }
7
8
9  ghost predicate Explicit (items: seq<ItemData>)
10 {
11     0 ≤ this.k ≤ |items| = |this.itemsAssign|
12 }
13
14
15 ghost predicate Implicit(items: seq<ItemData>, maxWeight : real)
16     requires Explicit(items)
17 {
18     this.TotalWeight(items) ≤ maxWeight
19 }
20
21 ghost predicate Valid(input: InputData)
22     requires input.Valid()
23 {
24     ∧ this.k = |this.itemsAssign|
25     ∧ this.Partial(input)
26 }
```

Figura 4.7: Definición de los predicados de validez de `SolutionData`

El fichero `BT.dfy` contiene la implementación del método algorítmico de vuelta atrás. El método principal, `KnapsackBT`, es el encargado de iniciar la exploración de todas las posibles combinaciones de ítems.

4.3.1. Especificación

En la figura 4.8 se muestra la especificación del método `KnapsackBT`. Este método, recibe tres parámetros: `input`, que representa la entrada del problema; `ps` *(partial solution)*, la solución parcial que se va construyendo durante el proceso de la vuelta atrás y `bs` *(best solution)*, que almacena la mejor solución encontrada hasta el momento. El parámetro `input` permanece constante a lo largo de toda la ejecución del método. Sin embargo, `ps` y `bs` son parámetros de entrada y salida, por lo que pueden modificarse. Para permitir estos cambios, es necesario incluir la cláusula `modifies`, indicando los campos que pueden ser alterados durante la ejecución del método.

```
1  method KnapsackBT(input: Input, ps: Solution, bs: Solution)
2    decreases ps.Bound(),1
3    modifies ps`totalValue, ps`totalWeight, ps`k, ps.itemsAssign
4    modifies bs`totalValue, bs`totalWeight, bs`k, bs.itemsAssign
5
6    requires input.Valid()
7    requires ps.Partial(input)
8    requires bs.Valid(input)
9
10   ensures ps.Partial(input)
11   ensures ps.Model().Equals(old(ps.Model()))
12   ensures ps.k = old(ps.k)
13   ensures ps.totalValue = old(ps.totalValue)
14   ensures ps.totalWeight = old(ps.totalWeight)
15
16   ensures bs.Valid(input)
17   ensures bs.Model().optimalExtension(ps.Model(), input.Model()) ∨
18           bs.Model().Equals(old(bs.Model()))
19   ensures ∀ s : SolutionData | ∧ s.Valid(input.Model())
20                                ∧ s.Extends(ps.Model())
21                                • s.TotalValue(input.Model().items) ≤
22                                  bs.Model().TotalValue(input.Model().items)
23   ensures bs.Model().TotalValue(input.Model().items) ≥
24           old(bs.Model().TotalValue(input.Model().items))
```

Figura 4.8: Especificación del método `KnapsackBT`

Todos estos parámetros deben ser válidos en el momento en el que se llama a `KnapsackBT`. Por ello, se incluyen cláusulas `requires` como **precondiciones** del método (líneas 6 a 8). Estas instrucciones llaman a las funciones que definen la validez de los parámetros: `input.Valid()`, `ps.Partial(input)` y `bs.Valid(input)`.

Al finalizar la ejecución del método `KnapsackBT`, los parámetros deben seguir siendo válidos. Esta consistencia es fundamental para garantizar la corrección del método. Las condiciones que deben cumplirse tras la ejecución del método se expresan como **postcondiciones** y se especifican mediante la cláusula `ensures`. El parámetro `input` al no ser mutable, no puede cambiar durante la ejecución del método, luego no requiere una postcondición explícita: si es válido a la entrada, seguirá siéndolo al finalizar el método. En cambio, los parámetros `ps` y `bs` sí son mutables, por ello, es necesario asegurarse de que sigan siendo válidos al finalizar el método (figura 4.8, líneas 10 y 16).

Además, los cambios realizados sobre `ps` se deshacen (gracias al desmarcaje que se hace en cada llamada), de ahí las postcondiciones establecidas entre las líneas 11 a 14. Concretamente, en la línea 11 se utiliza el predicado `Equals`, este verifica que `ps` y su estado anterior a la llamada (`old(ps.Model())` son iguales. Esto significa que ambos tienen el mismo índice `k` y que la asignación de ítems seleccionados (`itemsAssign`) coincide en todas las posiciones hasta dicho índice.

Otro aspecto fundamental que hay que garantizar es que, además de ser válida, `bs` es la mejor solución posible. Esto lo podemos asegurar mediante las siguientes postcondiciones:

- Pueden darse dos situaciones: o bien `bs` ha sido actualizada y es una extensión óptima de `ps` que mejora a la `bs` inicial, o bien `bs` permanece como la solución de entrada porque no se ha encontrado ninguna mejor, (figura 4.8, líneas 17 y 18). Entendemos por extensión óptima lo que define el predicado `OptimalExtension`,

```
1 ghost function Bound() : int
2     reads this
3 {
4     this.itemsAssign.Length - this.k + 1
5 }
```

Figura 4.9: Definición de la función `Bound` de la clase `Solution`

que verifica que una solución `s` es una extensión de la solución parcial `ps` y, además, garantiza que no existe ninguna otra solución válida que extienda a `ps` con un valor total superior al de `s`. El concepto de extensión está definido en el predicado `Extends`, que asegura que los arrays `itemsAssign` de dos soluciones coinciden hasta el índice `k` de la solución más corta.

- Dada cualquier extensión óptima de `ps`, su valor debe ser menor o igual que la de la mejor solución `bs`, (figura 4.8, líneas 19 a 22).
- El su nuevo valor total de `bs` debe ser mayor o igual al valor anterior, (figura 4.8, líneas 23 a 24).

Dado que el método es recursivo, es necesario especificar una **función de cota** mediante la cláusula `decreases`. En este caso, como el índice `ps.k` va incrementando, la función de cota es la diferencia entre el tamaño de la solución y el valor de dicho índice, como se muestra en la figura 4.9. No obstante, este método llama a otro métodos recursivos (como se observará en su implementación en las siguientes secciones). Por ello, se recurre al uso de las tuplas lexicográficas vistas en la sección 2.3. Así, la cláusula de cota utilizada par `KnapsackBT`, es `decreases ps.Bound(), 1` (figura 4.8, línea 2), mientras que para los otros métodos recursivos invocados por él, tendrán como cota `decreases ps.Bound(), 0`.

4.3.2. Implementación

Una vez especificado el método `KnapsackBT`, pasamos a explicar su implementación. En este contexto, el array `itemsAssign` de `bs` se inicializa con todos sus valores a `false`, ya que es un problema de maximización donde se busca el valor más alto. Debido a que el método aplica recursión, es necesario definir un caso base que garantice su terminación. La recursión termina cuando la solución parcial se convierte en una solución completa al haber procesado todos los ítems. Esto se comprueba mediante su índice k: `ps.k == ps.itemsAssign.Length`. En caso contrario, se entra en el caso recursivo, encargado de la exploración de las ramas del árbol. El método `KnapsackBTTrueBranch` (la rama que sí selecciona el ítem) solo se ejecuta si el ítem que se está procesando cabe en la mochila. En cambio, el método `KnapsackBTFalseBranch` (la rama que no selecciona el ítem) se ejecuta siempre. Profundizaremos en dichos métodos en la siguiente sección.

El método principal `KnapsackBT`, mostrado en la figura 4.10, no implementa explícitamente el caso base ni el caso recursivo. En su lugar, estos se han descompuesto en métodos específicos que encapsulan cada comportamiento. Cuando se cumple `ps.k == ps.itemsAssign.Length`, `KnapsackBT` invoca al método `KnapsackBTBaseCase`, que

```
1  method KnapsackBT(input: Input, ps: Solution, bs: Solution) {
2  if (ps.k = input.items.Length) {
3      KnapsackBTBaseCase(input, ps, bs);
4    }
5    else {
6      if (ps.totalWeight + input.items[ps.k].weight ≤ input.maxWeight) {
7        KnapsackBTTrueBranch(input, ps, bs);
8      }
9      else {
10       InvalidExtensionsFromInvalidPs(ps, input);
11     }
12     label L:
13     ghost var oldbs := bs.Model();
14     assert oldbs.OptimalExtension(
15                     SolutionData(ps.Model().itemsAssign[ps.k:=true],ps.k+1),
16                     input.Model()
17                   )
18            ∨ oldbs.Equals(old(bs.Model()));
19     KnapsackBTFalseBranch(input, ps, bs);
20     assert bs.Model().OptimalExtension(
21                       SolutionData(ps.Model().itemsAssign[ps.k:=false], ps.k+1),
22                       input.Model()
23                     )
24          ∨ bs.Model().Equals(oldbs);
25     if bs.Model().OptimalExtension(
26                     SolutionData(ps.Model().itemsAssign[ps.k:=false], ps.k+1),
27                     input.Model()
28                   )
29     {
30       assert ∀ s : SolutionData | ∧ s.Valid(input.Model())
31                                   ∧ s.Extends(ps.Model())
32                                   • s.TotalValue(input.Model().items) ≤
33                                   bs.Model().TotalValue(input.Model().items);
34     }
35     else if oldbs.Equals(old(bs.Model())) {
36       assert ∀ s : SolutionData | ∧ s.Valid(input.Model())
37                                   ∧ s.Extends(ps.Model())
38                                   • s.TotalValue(input.Model().items) ≤
39                                   bs.Model().TotalValue(input.Model().items);
40     }
41     else {
42       assert bs.Model().Equals(oldbs);
43       assert old@L(ps.Model()).Equals(ps.Model());
44       assert old@L(SolutionData(ps.Model().itemsAssign[ps.k := true], ps.k+1)).
             Equals(SolutionData(ps.Model().itemsAssign[ps.k := true], ps.k+1));
45       assert old(ps.totalWeight + input.items[ps.k].weight ≤ input.maxWeight);
46       bs.Model().EqualsOptimalExtensionFromEquals(
47                   old@L(SolutionData(ps.Model().itemsAssign[ps.k := true],ps.k+1)),
48                   SolutionData(ps.Model().itemsAssign[ps.k:=true],ps.k+1),
49                   input.Model()
50                 );
51       assert oldbs.OptimalExtension(
52                       SolutionData(ps.Model().itemsAssign[ps.k:=true], ps.k+1),
53                       input.Model()
54                     );
55       assert forall s : SolutionData | ∧ s.Valid(input.Model())
56                                        ∧ s.Extends(ps.Model())
57                                        • s.TotalValue(input.Model().items) ≤
58                                        bs.Model().TotalValue(input.Model().items);
59     }
60   }
61 }
```

Figura 4.10: Implementación del método `KnapsackBT`

maneja el caso base (línea 3). En cambio, cuando `ps.k < ps.itemsAssign.Length`, nos encontramos en el caso recursivo, correspondiente a la exploración de las ramas del árbol implementadas en métodos independientes (líneas 7 y 19).

Todos estos métodos comparten las mismas precondiciones y postcondiciones que `KnapsackBT`, con la única diferencia en la condición relacionada con el índice `ps.k`. En el caso base, la precondición es `ps.k == ps.itemsAssign.Length`, lo que indica que se han procesado todos los ítems. Por otra parte, las dos ramas que constituyen el caso recursivo utilizan la precondición `ps.k < ps.itemsAssign.Length`, permitiendo que aún queden elementos por considerar. Como puede observarse, las precondiciones de cada método están alineadas con las condiciones que provocan su llamada dentro de `KnapsackBT`.

Esta división en métodos facilita la verificación del algoritmo, ya que permite abordar cada caso de manera aislada, haciendo que el proceso de razonamiento y prueba se vuelva más claro y manejable. En las próximas secciones veremos la verificación de cada uno de ellos.

Continuando con la estructura del método `KnapsackBT`, su verificación se fundamenta en las postcondiciones. Si la rama `true` no se ejecuta porque el ítem excede la capacidad de la mochila (línea 9), esto implica que en esa rama no se generarán soluciones válidas capaces de superar el valor actual de `bs`. Dado que Dafny no puede verificar esta propiedad por sí solo, se invoca el lema `InvalidExtensionsFromInvalidPs` (línea 10), cuya especificación establece que si una solución parcial extendida con el valor `true` no es válida, entonces ninguna de sus extensiones tampoco será válida. En la figura 4.11, se muestra la demostración de este lema. La idea para demostrar este lema reside en lo siguiente: si una solución `s` extiende a `ps`, tendrá, como mínimo, el mismo peso que `ps`. Dado que `ps` ya excedía el peso máximo permitido, la solución `s` también violará esta restricción y, por lo tanto, no será válida. La demostración del lema la podemos conseguir utilizando otros lemas:

- `GreaterOrEqualValueWeightFromExtends`: establece que si una solución `s1` extiende a otra solución `s2`, entonces `s1` tiene como mínimo el peso y el valor de `s1`.
- `AddTrueMaintainsSumConsistency`: establece una propiedad composicional. Dada una solución `s1` que se extiende añadiendo un elemento a `true`, generando una nueva solución `s2`, la suma de los pesos y los valores de `s2` se actualiza de manera consistente al incluir el peso y el valor del nuevo elemento.

Ambos lemas son correctos gracias a un lema común: `EqualValueWeightFromEquals`, cuya demostración puede consultarse en la figura 4.12. Este lema establece que si dos soluciones (`this` y `s`) son idénticas (igualdad de campos), entonces tienen las mismas sumas de pesos y valores. Esto es porque el contenido de `itemsAssign` de cada solución es igual y los cálculos acumulados de pesos y valores son idénticos. Se demuestra por inducción sobre `s.k`.

Después de la ejecución de las ramas (siguiendo el orden de la rama verdadera y, a continuación, la rama falsa), se comprueba que la solución parcial se restaure correctamente, garantizando que los valores de peso y valor coincidan con el estado previo a la llamada. Además, en este punto se presentan tres posibles escenarios para demostrar la optimalidad de `bs` (figura 4.10, líneas 20 a 59):

- La solución óptima ha sido encontrada en la rama `false`. Esto se verifica de manera trivial a partir de las postcondiciones del método (figura 4.10, líneas 25 a 34).

```
1  lemma InvalidExtensionsFromInvalidPs(ps: Solution, input: Input)
2    requires input.Valid()
3    requires 0 ≤ ps.k < ps.itemsAssign.Length
4    requires ps.itemsAssign.Length = input.items.Length
5    requires ps.totalWeight + input.items[ps.k].weight > input.maxWeight
6    requires ps.Partial(input)
7    ensures ∀ s : SolutionData | ∧ |s.itemsAssign| =
8                                    |SolutionData(
9                                        ps.Model().itemsAssign[ps.k:=true],
10                                       ps.k+1
11                                   ).itemsAssign|
12                                 ∧ s.k ≤ |s.itemsAssign|
13                                 ∧ ps.k + 1 ≤ s.k
14                                 ∧ s.Extends(
15                                         SolutionData(
16                                            ps.Model().itemsAssign[ps.k:=true],
17                                            ps.k+1
18                                         )
19                                    )
20                                 • ¬s.Valid(input.Model())
21 {
22   forall s : ∧ SolutionData |
23              ∧ |s.itemsAssign| =
24                |(SolutionData(ps.Model().itemsAssign[ps.k:=true],
25                ps.k+1)).itemsAssign|
26              ∧ s.k ≤ |s.itemsAssign|
27              ∧ ps.k + 1 ≤ s.k
28              ∧ s.Extends(SolutionData(ps.Model().itemsAssign[ps.k:=true], ps.k+1))
29   ensures ¬s.Valid(input.Model())
30   {
31     assert s.TotalWeight(input.Model().items) > input.maxWeight by {
32       SolutionData(ps.Model().itemsAssign[ps.k := true], ps.k+1).
             GreaterOrEqualValueWeightFromExtends(s, input.Model());
33       SolutionData.AddTrueMaintainsSumConsistency(
34                      ps.Model(),
35                      SolutionData(ps.Model().itemsAssign[ps.k := true], ps.k+1),
36                      input.Model()
37                    );
38     }
39   }
40 }
```

Figura 4.11: Lema `InvalidExtensionsFromInvalidPs`

- La mejor solución encontrada hasta el momento no ha mejorado en ninguna de las ramas y, por lo tanto, sigue siendo la antigua (la que entró en el método). También se verifica trivialmente (figura 4.10, líneas 35 a 40).

- La solución óptima ha sido encontrada en la rama `true`, la primera en ejecutarse. Esto se debe a que la `bs` que entró en `KnapsackFalseBranch` no ha mejorado, y por lo tanto, debe ser idéntica a la que se obtuvo de la rama `true`. Este caso, Dafny lo verifica con ayuda del lema `EqualsoptimalExtensionFromEquals`, el cual establece que, dadas dos soluciones parciales iguales (con campos idénticos), si `bs` es una extensión óptima para una de ellas, entonces también lo es para la otra. Aplicamos este lema utilizando la versión de `ps` previa a la entrada en la rama `false` y la versión de `ps` posterior usando una etiqueta a su salida, confirmando así la igualdad. De nuevo, este lema se verifica gracias al lema de la figura 4.12, (figura 4.10, líneas 41 a 59).

```
lemma EqualValueWeightFromEquals(s : SolutionData, input : InputData)
    decreases k
    requires input.Valid()
    requires |input.items| = |this.itemsAssign| = |s.itemsAssign|
    requires this.k ≤ |this.itemsAssign|
    requires s.k ≤ |s.itemsAssign|
    requires this.Equals(s)
    ensures this.TotalValue(input.items) = s.TotalValue(input.items)
    ensures this.TotalWeight(input.items) = s.TotalWeight(input.items)
{
    if k = 0 {

    }
    else {
      SolutionData(itemsAssign, k - 1).
        EqualValueWeightFromEquals(
            SolutionData(s.itemsAssign, s.k - 1),
            input
        );
    }
}
```

Figura 4.12: Lema `EqualValueWeightFromEquals`

4.4. Llamada inicial

El método `ComputeSolution`, implementado en el fichero `Knapsack.dfy`, es el encargado de resolver el problema de la mochila, lo podemos ver en la figura 4.13. Recibe como parámetro la entrada del problema (`input`) que debe ser válida, y devuelve una solución (`bs`) que debe ser válida y óptima. El propósito de este método es preparar todos los componentes necesarios para la ejecución del método algorítmico de la vuelta atrás (`KnapsackBT`). Para ello, primero se inicializan las dos estructuras clave:

- La solución parcial (`ps`) que representa el estado actual de la mochila. Inicialmente, todos los ítems están sin seleccionar (`ps.itemsAssign` a `false`), el peso y el valor son nulos y el índice `ps.k` es inicializado a 0, indicando que aún no se ha asignado ningún ítem.
- La mejor solución (`bs`) también comienza con todos los ítems no seleccionados, peso y valor nulos, pero su índice `bs.k` inicializado al número de ítems, ya que debe ser una solución completa.

Antes de hacer la llamada inicial de `KnapsackBT`, se verifica que `bs` sea válida, ya que esta es una de las precondiciones necesarias de dicho método. Dado que `bs` se inicializa con todos los ítems asignados a `false`, su peso total es cero; luego cumple con las restricciones del problema y, por lo tanto, es válida. Para verificar esto en Dafny, hace falta un lema auxiliar: `SumOfFalsesEqualsZero`. Este lema garantiza que si no se ha seleccionado ningún ítem (es decir, todos están asignados a `false`), entonces la suma total de pesos es igual a cero, lo que implica que la solución cumple con la restricción de capacidad de la mochila.

A continuación, se realiza la llamada inicial a `KnapsackBT`, pasando como parámetros la entrada (`input`), la solución parcial (`ps`) y la mejor solución encontrada hasta el momento (`bs`). Una vez finalizada la llamada, las dos postcondiciones del método `ComputeSolution` se verifican trivialmente gracias a las postcondiciones del método recursivo `KnapsackBT`.

```
1  method ComputeSolution(input: Input) returns (bs: Solution)
2    requires input.Valid()
3    ensures bs.Valid(input)
4    ensures bs.Optimal(input)
5  {
6    var n := input.items.Length;
7
8    var ps_itemsAssign := new bool[n](i ⇒ false);
9    var ps_totalValue := 0.0;
10   var ps_totalWeight := 0.0;
11   var ps_k := 0;
12   var ps := new Solution(ps_itemsAssign, ps_totalValue, ps_totalWeight, ps_k);
13
14   var bs_itemsAssign := new bool[n](i ⇒ false);
15   var bs_totalValue := 0.0;
16   var bs_totalWeight := 0.0;
17   var bs_k := n;
18   bs := new Solution(bs_itemsAssign, bs_totalValue, bs_totalWeight, bs_k);
19
20   assert bs.Valid(input) by {
21     bs.Model().SumOfFalsesEqualsZero(input.Model());
22   }
23
24   KnapsackBT(input, ps, bs);
25 }
```

Figura 4.13: Implementación del método `ComputeSolution`

4.5. Método del caso base

El método del caso base es `KnapsackBTBaseCase`. Como se ha comentado anteriormente, el caso base se ejecuta cuando todos los ítems han sido procesados, es decir, cuando la solución parcial `ps` es completa. Dado que su especificación es similar a la del método principal `KnapsackBT`, el análisis se centrará en la implementación y en su proceso de verificación. En la figura 4.14 podemos ver la implementación. En un problema de optimización, el objetivo del caso base es determinar si la solución completa actual (`ps`) supera a la mejor solución almacenada (`bs`). Esto se traduce en comparar los campos de los valores totales de dichas soluciones: `ps.totalValue > bs.totalValue`. El método distingue dos casos: cuando esta comparación es verdadera y cuando es falsa.

Cuando `ps.totalValue > bs.totalValue` se cumple, significa que hemos encontrado una solución mejor; luego es necesario actualizar `bs`. Esto se logra copiando los valores de `ps` a `bs` usando el método `Copy` de la clase `Solution` (figura 4.15).

Sin embargo, Dafny no puede verificar automáticamente que `bs` es óptima y válida después de la actualización con `bs.Copy(ps)`. Para resolver esta limitación, es necesario demostrar que cualquier solución `s` que extienda a `ps` tiene un valor menor o igual a la `bs` actualizada (líneas 4 a 15). Para ello, se utiliza la propiedad transitiva de la igualdad mediante un bloque `calc` y algunos lemas auxiliares.

El punto de partida es que Dafny reconoce que cualquier solución `s` válida (completa) que extienda a `ps` es idéntica a `ps`. Esto se deriva de la definición de `Extends`, que establece que una solución es extensión de otra si son iguales hasta el índice `k` de la solución más pequeña. Como `ps` es completa en el caso base, tanto `s` como

```
1  method KnapsackBTBaseCase(input: Input, ps: Solution, bs: Solution) {
2      if (ps.totalValue > bs.totalValue) {
3          bs.Copy(ps);
4          forall s : SolutionData | ∧ s.Valid(input.Model())
5                                    ∧ s.Extends(ps.Model())
6          ensures s.TotalValue(input.Model().items) ≤
7              bs.Model().TotalValue(input.Model().items)
8          {
9            assert s.Equals(ps.Model());
10           calc {
11             s.TotalValue(input.Model().items);
12             {s.EqualValueWeightFromEquals(ps.Model(), input.Model());}
13             ps.Model().TotalValue(input.Model().items);
14             {bs.CopyModel(ps, input);}
15             bs.Model().TotalValue(input.Model().items);
16           }
17         }
18     }
19     else {
20         forall s : SolutionData | ∧ s.Valid(input.Model())
21                                   ∧ s.Extends(ps.Model())
22         ensures s.TotalValue(input.Model().items) ≤
23                 bs.Model().TotalValue(input.Model().items)
24         {
25           assert s.Equals(ps.Model());
26           s.EqualValueWeightFromEquals(ps.Model(), input.Model());
27           assert s.TotalValue(input.Model().items) =
28           ps.Model().TotalValue(input.Model().items);
29         }
30     }
31 }
```

Figura 4.14: Implementación del método `KnapsackBTBaseCase`

`ps` comparten el mismo `k` y, por lo tanto, son idénticas. Dentro del bloque `calc`, el primer paso es establecer que el valor total de `s` es igual al valor total de `ps`. Esta igualdad es evidente dada la identidad de `s` y `ps`, pero de nuevo, Dafny requiere una justificación explícita. Esta justificación se proporciona mediante el lema `EqualValueWeightFromEquals`, descrito previamente en la figura 4.12. A continuación, se demuestra que el valor total de `ps` es igual al valor total de `bs`, para concluir que `bs` es tanto óptima como válida. Esta última igualdad también requiere la aplicación de un lema específico: `CopyModel`. Su demostración se ilustra en la figura 4.16 y garantiza que, si una solución `s` es válida para una entrada, y `this` tiene el mismo modelo, peso acumulado y valor acumulado que `s`, entonces `this` también será válida para la misma entrada.

Por otra parte, cuando `ps.totalValue <= bs.totalValue`, `bs` no se actualiza y mantiene su valor previo. Para que Dafny verifique la optimalidad de `bs` en este escenario, debemos probar que ninguna extensión de `ps` supera a `bs`, (líneas 20 a 29). Esto se deduce del hecho de que, si cualquier solución `s` que es extensión de `ps` es igual a `ps`, y estamos en el caso donde `ps.totalValue <= bs.totalValue`, entonces tenemos lo siguiente:

```
s.TotalValue(input.Model().items) = ps.totalValue <= bs.totalValue
```

Luego, por transitividad, `bs` es óptima. Una vez más, el lema `EqualValueWeightFromEquals` nos ayuda a asegurar la igualdad entre `ps` y cualquier extensión suya.

```
method Copy(s : Solution)
    modifies this`totalValue, this`totalWeight, this`k, this.itemsAssign
    requires this ≠ s
    requires this.itemsAssign.Length = s.itemsAssign.Length
    ensures this.k = s.k
    ensures this.totalValue = s.totalValue
    ensures this.totalWeight = s.totalWeight
    ensures this.itemsAssign = old(this.itemsAssign)
    ensures ∀ i | 0 ≤ i < this.itemsAssign.Length • this.itemsAssign[i] =
                                                    s.itemsAssign[i]
    ensures this.Model() = s.Model()
{
    for i := 0 to s.itemsAssign.Length
      invariant ∀ j | 0 ≤ j < i • this.itemsAssign[j] = s.itemsAssign[j];
    {
      this.itemsAssign[i] := s.itemsAssign[i];
    }
    this.totalValue := s.totalValue;
    this.totalWeight := s.totalWeight;
    this.k := s.k;
}
```

Figura 4.15: Implementación del método `Copy` de `Solution`

```
lemma CopyModel (s : Solution, input : Input)
    requires input.Valid()
    requires s.Valid(input)
    requires s.Model() = this.Model()
    requires s.totalWeight = this.totalWeight
    requires s.totalValue = this.totalValue
    ensures this.Valid(input)
{}
```

Figura 4.16: Lema `CopyModel`

4.6. Métodos del caso recursivo

4.6.1. Método de la rama que selecciona el ítem

El método encargado de manejar la rama que decide seleccionar el ítem es `KnapsackBTTrueBranch`. Su especificación es similar a la de `KnapsackBT`, con la excepción de que incluye dos precondiciones adicionales:

- `ps.k < ps.itemsAssign.Length`
- `ps.totalWeight + input.items[ps.k].weight <= input.maxWeight`

La primera precondición asegura que el método se ejecute únicamente en el caso recursivo, donde aún quedan ítems por procesar, tal como se mencionó anteriormente. La segunda precondición garantiza que el método solo se ejecute si el ítem `ps.k` cabe en la mochila. En la figura 4.17 se muestra la implementación y verificación del método `KnapsackBTTrueBranch`. Se asigna la posición actual (`ps.k`) a `true` en `ps.itemsAssign`, lo que significa que el objeto se selecciona. Se actualizan el peso (`ps.totalWeight`) y el valor total (`ps.totalValue`) de la solución parcial, se avanza a la siguiente posición (`ps.k := ps.k + 1`) y se invoca recursivamente al método `KnapsackBT` para continuar con la exploración. Después de la llamada se restauran los valores de cada uno de

```
1  method KnapsackBTTrueBranch(input: Input, ps: Solution, bs: Solution) {
2      assert ps.totalWeight + input.items[ps.k].weight ≤ input.maxWeight;
3      ghost var oldps := ps.Model();
4      ghost var oldtotalWeight := ps.totalWeight;
5      ghost var oldtotalValue := ps.totalValue;
6
7      ps.itemsAssign[ps.k] := true;
8      ps.totalWeight := ps.totalWeight + input.items[ps.k].weight;
9      ps.totalValue := ps.totalValue + input.items[ps.k].value;
10     ps.k := ps.k + 1;
11
12     PartialConsistency(ps, oldps, input, oldtotalWeight, oldtotalValue);
13
14     KnapsackBT(input, ps, bs);
15
16     label L:
17
18     ps.k := ps.k − 1;
19     ps.totalWeight := ps.totalWeight − input.items[ps.k].weight;
20     ps.totalValue := ps.totalValue − input.items[ps.k].value;
21
22     assert SolutionData(ps.Model().itemsAssign[ps.k := true], ps.k + 1) =
23            old@L(ps.Model());
24 }
```

Figura 4.17: Implementación de `KnapsackBTTrueBranch`

los campos que fueron modificados.

La verificación del método requiere por tanto comprobar que se cumplen las precondiciones de la llamada recursiva y que las postcondiciones de la misma implican las postcondiciones del método. Casi todas las precondiciones de la llamada recursiva se cumplen trivialmente, pero es necesario demostrar que la validez de la solución parcial `ps` se mantiene tras modificar sus campos (garantizar que la condición `ps.Partial(input)` se mantenga verdadera). Esta comprobación es fundamental para que se cumpla la precondición de la llamada recursiva. Para ello, usamos el lema `PartialConsistency` ilustrado en la figura 4.18. Este lema establece que, si extendemos una solución parcial (`oldps`) incorporando un nuevo ítem, obteniendo una nueva solución parcial (`ps`), entonces `ps` mantiene las propiedades de consistencia parcial definidas por el predicado `Partial`.

La verificación de este lema se realiza mediante cálculos formales que demuestran la consistencia del valor y peso de `ps` con su versión antigua antes de verse modificada (`oldps`).

1. Primer cálculo (figura 4.18, líneas 27 a 34): para asegurar que el peso total de `ps` sea la suma del peso de `oldps` y el peso del nuevo ítem, se utiliza el lema `AddTrueMaintainsSumConsistency`. Adicionalmente, se emplea el lema `InputDataItems` para confirmar que el peso y valor de un ítem coinciden con el peso y valor de su modelo. Finalmente, se verifica que el peso total no exceda el peso máximo permitido.

```
1  lemma PartialConsistency(
2      ps: Solution,
3      oldps: SolutionData,
4      input: Input,
5      oldtotalWeight: real,
6      oldtotalValue: real
7  )
8    requires input.Valid()
9    requires 1 ≤ ps.k ≤ ps.itemsAssign.Length
10   requires 0 ≤ oldps.k ≤ |oldps.itemsAssign|
11   requires ps.k = oldps.k + 1
12   requires ps.itemsAssign.Length = |oldps.itemsAssign| = input.items.Length
13   requires oldps.itemsAssign[..oldps.k] + [true] = ps.itemsAssign[..ps.k]
14   requires oldps.Partial(input.Model())
15   requires oldtotalWeight = oldps.TotalWeight(input.Model().items)
16   requires oldtotalValue = oldps.TotalValue(input.Model().items)
17   requires oldps.TotalWeight(input.Model().items) + input.items[ps.k - 1].weight ≤
         input.maxWeight
18   requires oldtotalWeight = ps.totalWeight - input.items[oldps.k].weight
19   requires oldtotalValue = ps.totalValue - input.items[oldps.k].value
20   ensures ps.Partial(input)
21 {
22   assert oldps.Partial(input.Model());
23   assert oldtotalWeight = oldps.TotalWeight(input.Model().items);
24   assert oldps.TotalWeight(input.Model().items) + input.items[ps.k - 1].weight ≤
25          input.maxWeight;
26
27   calc {
28     ps.Model().TotalWeight(input.Model().items);
29     {SolutionData.AddTrueMaintainsSumConsistency(oldps,ps.Model(),input.Model());}
30     oldps.TotalWeight(input.Model().items) + input.Model().items[ps.k - 1].weight;
31     {input.InputDataItems(ps.k - 1);}
32     oldps.TotalWeight(input.Model().items) + input.items[ps.k - 1].weight;
33     ≤ input.maxWeight;
34   }
35
36   calc {
37     ps.totalWeight;
38     oldtotalWeight + input.items[ps.k - 1].weight;
39     oldps.TotalWeight(input.Model().items) + input.items[ps.k - 1].weight;
40     {input.InputDataItems(ps.k - 1);
41     SolutionData.AddTrueMaintainsSumConsistency(oldps, ps.Model(), input.Model());}
42     ps.Model().TotalWeight(input.Model().items);
43   }
44
45   calc {
46     ps.totalValue;
47     oldtotalValue + input.items[ps.k - 1].value;
48     oldps.TotalValue(input.Model().items) + input.items[ps.k - 1].value;
49     {input.InputDataItems(ps.k - 1);
50     SolutionData.AddTrueMaintainsSumConsistency(oldps, ps.Model(), input.Model());}
51     ps.Model().TotalValue(input.Model().items);
52   }
53
54   assert ps.Partial(input);
55 }
```

Figura 4.18: Lema `PartialConsistency`

2. Segundo cálculo (figura 4.18, líneas 36 a 43): se parte de `ps.totalWeight` y se reescribe como la suma de `oldtotalWeight` y el peso del nuevo ítem. Utilizando los lemas `InputDataItems` y `AddTrueMaintainsSumConsistency`, se valida la transición de `oldps` a `ps`. Se concluye que la suma se puede reescribir como `ps.Model().TotalWeight(input.Model().items)`.

3. Tercer cálculo (figura 4.18, líneas 45 a 52): este cálculo es análogo al segundo,

pero se aplica al valor total en lugar del peso.

Después de realizar la llamada recursiva a `KnapsackBT`, se restaura el estado de `ps` para dejarlo exactamente como estaba antes de seleccionar el ítem en la posición `ps.k` (se decrementa `ps.k` y se resta el peso y el valor del ítem añadido previamente). Sin embargo, la siguiente postcondición no se verifica de manera automática:

```
1   ensures  bs.Model().OptimalExtension(
2                          SolutionData(ps.Model().itemsAssign[ps.k:=true], ps.k + 1),
3                          input.Model()
4                        )
5           ∨ bs.Model().Equals(old(bs.Model()));
```

Para justificar esta postcondición, se añade una demostración adicional: un `assert` (figura 4.17, líneas 22 y 23) que comprueba que la restauración se ha realizado de forma precisa. Para ello, se emplea la etiqueta `L` para capturar el estado de `ps` justo antes de la llamada recursiva (marcado como `old@L`), y luego se compara con el estado de la solución al finalizar la recursión, una vez que sus valores han sido restaurados. Esto permite comprobar que el estado de la solución parcial se restaura correctamente después del retroceso.

4.6.2. Método de la rama que no selecciona el ítem

El método encargado de manejar la rama que decide no seleccionar el ítem, es `KnapsackBFalseBranch`. En la figura 4.19 podemos ver su implementación. El propósito de esta rama es asignar el valor `false` a la componente `ps.k` del array `itemsAssign`. Esta asignación indica que el ítem correspondiente no se incluirá en la mochila, lo que implica que no contribuirá ni al peso acumulado (`ps.totalWeight`) ni al valor acumulado (`ps.totalValue`) de la solución parcial. Se avanza a la siguiente posición (`ps.k := ps.k + 1`) y se invoca recursivamente al método principal `KnapsackBT` para continuar con la exploración. Una vez finalizada la llamada recursiva, se restaura `ps.k` a su valor original (`ps.k := ps.k - 1`) para volver al estado previo.

Como en el caso de la otra rama, hay que asegurar la validez de la solución parcial para garantizar las precondiciones de la llamada recursiva. Una posible causa de invalidez de `ps` sería que su peso acumulado excediera el peso máximo de la mochila. Sin embargo, en esta rama, se asigna `false` al ítem, lo que impide el incremento de `ps.totalWeight`. A pesar de esto, Dafny no infiere automáticamente que `ps` siga siendo válida. Por lo tanto, se requiere un lema explícito, `AddFalsePreservesWeightValue`, para demostrar que la asignación de `false` a un ítem en la solución no altera el peso acumulado total y, por ende, mantiene la validez de `ps`. Este lema establece que, dada una solución `s1` que se extiende asignando a `false`, generando una nueva solución `s2`, las sumas de los pesos y los valores siguen siendo las mismas y no se ven alteradas (ya que no sumaría el peso/valor del ítem como se ve en la definición de `TotalWeight` y `TotalValue` de la figura 4.3).

Después de realizar la llamada recursiva, se restauran los valores de `ps` (se decrementa `ps.k`). Al igual que en la otra rama, se utiliza la etiqueta `L` para comprobar que el estado de `ps` tras la restauración coincide con el original antes de la recursión.

```
method KnapsackBTFalseBranch(input: Input, ps: Solution, bs: Solution) {
    ghost var oldps := ps.Model();
    ps.itemsAssign[ps.k] := false;
    ps.k := ps.k + 1;

    assert ps.Partial(input) by {
    SolutionData.AddFalsePreservesWeightValue(oldps, ps.Model(), input.Model());
    input.InputDataItems(ps.k-1);
    }

    KnapsackBT(input, ps, bs);

    label L:

    ps.k := ps.k - 1;

    assert SolutionData(ps.Model().itemsAssign[ps.k := false], ps.k + 1) =
           old@L(ps.Model());

    assert bs.Model().OptimalExtension(
                        SolutionData(ps.Model().itemsAssign[ps.k:=false], ps.k+1),
                        input.Model()
                      )
    ∨ bs.Model().Equals(old(bs.Model()));

    assert ∀ s : SolutionData | ∧ s.Valid(input.Model())
                                ∧ s.Extends(
                                        SolutionData(
                                            ps.Model().itemsAssign[ps.k:=false],
                                            ps.k+1
                                        )
                                    )
                                • s.TotalValue(input.Model().items) ≤
                                bs.Model().TotalValue(input.Model().items);
}
```

Figura 4.19: Implementación de `KnapsackBTFalseBranch`

Capítulo 5

Aplicación de la metodología al problema de los funcionarios

"Si lo intentas, a menudo estarás solo, y a veces asustado, pero ningún precio es demasiado alto por el privilegio de ser uno mismo."
— Friedrich Nietzsche

En este capítulo, nos adentraremos en la verificación del problema de los funcionarios, tal como se describió en la sección 3.2. Aplicaremos la misma metodología presentada en el capítulo anterior a la hora de modelizar los datos del problema y de implementar el algoritmo. La principal diferencia de este problema con el anterior es que en el caso recursivo no tenemos una cantidad constante de llamadas recursivas, sino un bucle de llamadas, lo que requiere establecer un invariante para poder verificar el método.

5.1. Modelización del problema

Al igual que en la verificación del problema de la mochila, para abordar el problema de los funcionarios se requiere una clara distinción entre la implementación y la especificación. En la implementación, nos enfocamos en la creación de clases y, para la especificación, utilizamos tipos algebraicos que modelan de manera abstracta los objetos del mundo real.

Todos los ficheros los podemos encontrar en los subdirectorios `Implementation` y `Specification` del directorio `Employees`. Cada clase o tipo algebraico que forma parte de la representación formal del problema de los funcionarios se implementa en un archivo independiente, siguiendo la convención de nombrar el archivo con el nombre del componente y la extensión *.dfy*.

Al igual que sucede en el problema de la mochila, para obtener el modelo formal de un objeto de implementación, cada una de las clases mencionadas anteriormente proporciona la función `Model()`, la cual permite transformar una instancia mutable (objeto real) a su correspondiente versión inmutable (modelo). En la figura 5.1 se presenta un esquema ilustrativo del uso de `Model()`.

Las clases que definen los tipos de la entrada y salida del algoritmo son:

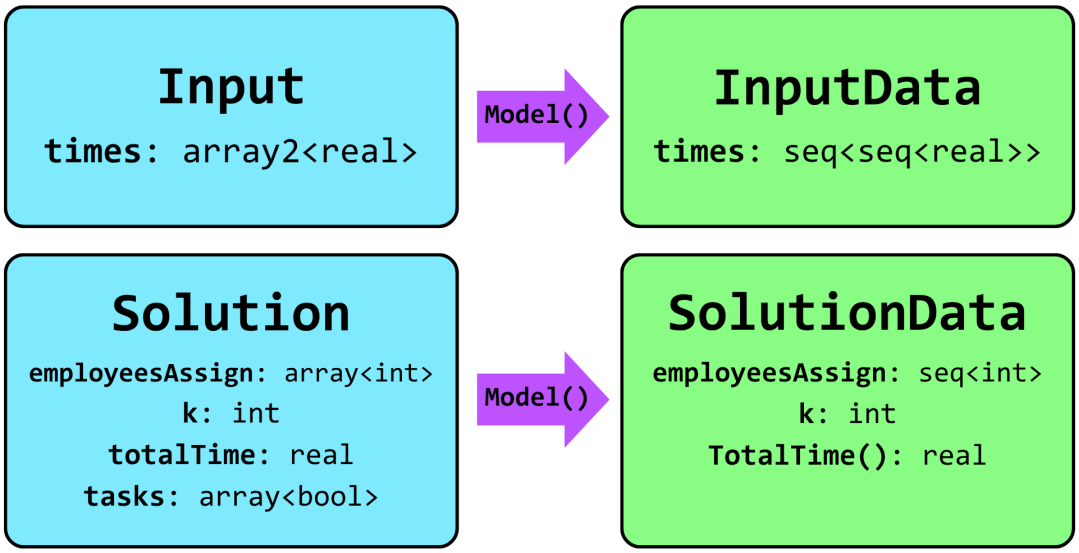


Figura 5.1: Correspondencia entre implementación y su modelo

- `Input`: clase que implementa una representación de la entrada del problema. Contiene la matriz de tiempos, un `array2` (matriz) que contiene el tiempo que tarda cada funcionario en realizar cada tarea. Las filas corresponden a los funcionarios y las columnas a las tareas.
- `Solution`: clase que implementa una representación de la solución del problema. Esta clase almacena varios atributos:
  - `employeesAssign`: array de enteros de tamaño número de funcionarios, donde cada posición corresponde a un funcionario y cuyo valor almacenado indica la tarea que va a realizar.
  - `totalTime`: tiempo total que tardan los funcionarios ya asignados en hacer sus correspondientes tareas.
  - `k`: etapa del árbol de exploración de la solución. Denota el número de funcionarios asignados de `employeesAssign`.
  - `tasks`: array booleano para la técnica de marcaje. Si `tasks[i]=true`, la tarea i ya ha sido asignada a un funcionario. Si `tasks[i]=false`, indica que la tarea i aún no ha sido asignada y, por lo tanto, está disponible.

Los tipos algebraicos que representan los valores de los objetos en un instante de ejecución y que se utilizan para propiedades acerca de ellos son:

- `InputData`: tipo de dato que modela los valores de la entrada del problema. Contiene una secuencia de secuencias que reflejan el modelo del `array2` de `Input`.
- `SolutionData`: tipo de dato que modela formalmente una solución. Está compuesto por una secuencia de enteros (`employeesAssign`) y una etapa `k`. Ambos

```
ghost function TotalTime(times : seq<seq<real>>) : real
    decreases k
    requires Explicit(times)
{
    if k = 0 then
      0.0
    else
      SolutionData(employeesAssign, k - 1).TotalTime(times) +
      times[k - 1][employeesAssign[k - 1]]
}
```

Figura 5.2: Definición de la función `TotalTime` de `SolutionData`

atributos modelan los atributos correspondientes de la clase `Solution`. A diferencia de esta, `SolutionData` no almacena directamente el tiempo acumulado de la solución (`totalTime`), en su lugar proporciona una función que permite calcular dicho valor a partir del contenido de `itemsAssign` y `k`. En la figura 5.2 se puede observar dicha función.

Tanto en las clases como en los tipos algebraicos se definen funciones, predicados `ghost` auxiliares y lemas que demuestran propiedades de dichas funciones y métodos. Las funciones y predicados delegan parte de su comportamiento a funciones definidas en el modelo. La clase `Input` tiene un predicado `Valid` cuya definición se muestra en la figura 5.3. Se exige que la matriz sea cuadrada y a continuación, se llama a la función `Valid` del modelo definida en la figura 5.4, que facilita la inspección de los componentes de la matriz:

- El modelo de input consiste en una secuencia de secuencias que también debe ser cuadrada coincidiendo con la matriz de la implementación.
- Todos los elementos del array `times` deben ser positivos, pues cada elemento `times[i][j]` es un número real que representa el tiempo que tarda el funcionario `i` en realizar la tarea `j`.

En la clase `Solution`, el predicado `Partial`, mostrado en la figura 5.5, incluye el invariante de representación que indica que:

- La secuencia de valores del array `employeesAssign` cumple las restricciones explícitas e implícitas.
- El campo `totalTime` contiene la suma de los tiempos que los funcionarios dedican a las tareas que les han sido asignadas, según lo indicado en el array `employeeAssign`.
- El marcador `tasks` refleja la disponibilidad de las tareas según las asignaciones funcionario-tarea del array `employeeAssign`.

Además, `Partial` llama a la función correspondiente del modelo y define las restricciones explícitas e implícitas como se observa en la figura 5.6. Como vemos, la función `Partial` es igual que la del problema de la mochila, lo que varía son las

```
ghost predicate Valid()
    reads this, times
{
    ∧ times.Length0 = times.Length1 > 0
    ∧ this.Model().Valid()
}
```

Figura 5.3: Definición del predicado `Valid` de `Input`

```
ghost predicate Valid() {
    ∧ (∀ i | 0 ≤ i < |times| • |times[i]| = |times|)
    ∧ ∀ i, j | 0 ≤ i < |times| ∧ 0 ≤ j < |times[i]| • times[i][j] > 0.0
}
```

Figura 5.4: Definición del predicado `Valid` de `InputData`

definiciones internas de las funciones `Explicit` e `Implicit`. Esta flexibilidad permite adaptar la función a diferentes problemas, ajustando las restricciones específicas según sea necesario.

En el problema de los funcionarios, el método `Explicit` verifica que el tamaño de la solución (número de funcionarios) coincida con las filas de la matriz `times`. También comprueba que esta matriz (representada en forma de secuencias) sea cuadrada y que los valores de la solución sean tareas válidas, es decir, números del 0 al tamaño de la solución - 1 (el número de funcionarios coincide con el número de tareas). El método `Implicit` asegura que no haya tareas repetidas, es decir, que todos los valores del array sean distintos entre sí.

La función `Valid` (figura 5.6), que valida soluciones completas, es exactamente igual que en el problema de la mochila, una versión más restrictiva de `Partial` debido a la condición `k == |itemsAssign|`.

5.2. Método principal

Tras establecer los elementos clave que permiten modelar y abordar el problema de los funcionarios, esta sección se enfocará en detallar la especificación e implementación de su resolución. Además, se expondrán las herramientas utilizadas para llevar a cabo la verificación formal en Dafny.

El fichero `BT.dfy` contiene la implementación del algoritmo de vuelta atrás. El método principal, `EmployeesBT`, es el encargado de iniciar la exploración de todas las posibles combinaciones de asignación de tareas a funcionarios. Este algoritmo respeta las restricciones del problema y busca aquellas combinaciones que minimicen el tiempo total invertido en realizar las tareas.

5.2.1. Especificación

Podemos ver toda la especificación del problema mostrada en la figura 5.7. El método `EmployeesBT` recibe los mismos parámetros que `KnapsackBT`: `input`, que representa

```
1  ghost predicate Partial (input : Input)
2      reads this, this.employeesAssign, input, input.times, tasks
3      requires input.Valid()
4  {
5      ∧ 0 ≤ this.k ≤ this.employeesAssign.Length
6      ∧ Model().Partial(input.Model())
7      ∧ Model().TotalTime(input.Model().times) = totalTime
8      ∧ this.employeesAssign.Length = tasks.Length
9      ∧ (∀ i | 0 ≤ i < this.employeesAssign.Length
10         • tasks[i] = (i in this.Model().employeesAssign[0..this.k]))
11 }
```

Figura 5.5: Definición del predicado `Partial` de `Solution`

```
1  ghost predicate Partial (input: InputData)
2      requires input.Valid()
3  {
4      ∧ Explicit(input.times)
5      ∧ Implicit(input.times)
6  }
7
8  ghost predicate Explicit(times: seq<seq<real>>)
9  {
10     ∧ 0 ≤ k ≤ |employeesAssign| = |times|
11     ∧ (∀ i | 0 ≤ i < |times| • |times[i]| = |times|)
12     ∧ (∀ i | 0 ≤ i < this.k • 0 ≤ employeesAssign[i] < |employeesAssign|)
13 }
14
15 ghost predicate Implicit(times: seq<seq<real>>)
16     requires Explicit(times)
17 {
18     ∧ (∀ i,j | 0 ≤ i < this.k ∧ 0 ≤ j < this.k ∧ i ≠ j • employeesAssign[i] ≠
           employeesAssign[j])
19 }
20
21 ghost predicate Valid(input: InputData)
22     requires input.Valid()
23 {
24     ∧ k = |itemsAssign|
25     ∧ Partial(input)
26 }
```

Figura 5.6: Definición de los predicados `Partial` y `Valid` de `SolutionData`

la entrada del problema; `ps` *(partial solution)*, la solución parcial que se va construyendo durante el proceso de la vuelta atrás, y `bs` *(best solution)*, que almacena la mejor solución encontrada hasta el momento.

Como ya sabemos, `ps` y `bs` son parámetros de entrada y salida que pueden modificarse, por ello se incluye en la cláusula `modifies`. Todos estos parámetros deben ser válidos en el momento en el que se llama a `EmployeesBT`. Las funciones que definen la validez de los parámetros son: `input.Valid()`, `ps.Partial(input)` y `bs.Valid(input)`.

Recordemos que la solución completa (`bs`) debe haber considerado todos los ítems (`k == bs.itemsAssign.Length`), mientras que la solución parcial (`ps`) puede estar en proceso de construcción, por lo que basta con que `k <= bs.itemsAssign.Length`.

Al finalizar la ejecución del método `EmployeesBT`, como en `KnapsackBT`, los parámetros deben seguir siendo válidos y `bs` tiene que ser la solución óptima. Por tanto, `EmployeesBT` tiene las mismas postcondiciones que `KnapsackBT` (figura 5.7, líneas 6 a 8). Además, los cambios realizados sobre `ps` se deshacen gracias al desmarcaje de

```
1  method EmployeesBT(input: Input, ps: Solution, bs: Solution)
2    decreases ps.Bound(),1
3    modifies ps`totalTime, ps`k, ps.employeesAssign, ps.tasks
4    modifies bs`totalTime, bs`k, bs.employeesAssign, bs.tasks
5
6    requires input.Valid()
7    requires ps.Partial(input)
8    requires bs.Valid(input)
9
10   ensures ps.Partial(input)
11   ensures ps.Model().Equals(old(ps.Model()))
12   ensures ps.k = old (ps.k)
13   ensures ps.totalTime = old(ps.totalTime)
14
15   ensures bs.Valid(input)
16
17   ensures bs.Model().OptimalExtension(ps.Model(), input.Model()) ∨
18           bs.Model().Equals(old(bs.Model()))
19
20   ensures ∀ s : SolutionData | ∧ s.Valid(input.Model())
21                                ∧ s.Extends(ps.Model())
22                                • s.TotalTime(input.Model().times) ≥
23                                  bs.Model().TotalTime(input.Model().times)
24
25   ensures bs.Model().TotalTime(input.Model().times) ≤
26           old(bs.Model().TotalTime(input.Model().times))
```

Figura 5.7: Especificación del método `EmployeesBT`

cada llamada, de ahí las postcondiciones establecidas entre las líneas 11 a 13. Otro aspecto fundamental que hay que garantizar es que, además de ser válida, `bs` es la mejor solución posible. Al igual que el problema de la mochila, lo podemos asegurar mediante las siguientes postcondiciones:

- Pueden darse dos situaciones: o bien `bs` ha sido actualizada y es una extensión óptima de `ps` que mejora a la `bs` inicial, o bien `bs` permanece como la solución de entrada porque no se ha encontrado ninguna mejor (figura 5.7, líneas 17 y 18).
- Cualquier extensión óptima de `ps`, su tiempo debe ser mayor o igual que la mejor solución `bs` (figura 5.7, líneas 20 a 23).
- Si `bs` cambia, su nuevo tiempo total debe ser menor o igual al valor anterior. (figura 5.7, líneas 25 y 26).

Dado que el método es mutuamente recursivo, (llama a otros métodos que lo llaman a él), es necesario especificar una **función de cota** mediante la cláusula `decreases` usando las tuplas lexicográficas explicadas en la sección 2.3. Al igual que en el problema de la mochila, el índice `ps.k` va incrementando, luego la función de cota utilizada es `decreases ps.Bound(),1`, siendo `Bound()` la función definida en la figura 5.8. Por otro lado, los métodos que son invocados por `EmployeesBT` tendrán `decreases ps.Bound(),0` como función de cota.

5.2.2. Implementación

```
1 ghost function Bound() : int
2     reads this
3 {
4     this.employeesAssign.Length - this.k + 1
5 }
```

Figura 5.8: Definición de la función `Bound` de la clase `Solution`

Una vez especificado el método `EmployeesBT`, pasamos a explicar su implementación. La división en métodos de este algoritmo es similar a la de `KnapsackBT`. Cuando ya se han considerado todos los elementos, se aplica `EmployeesBTBaseCase`. En los demás casos, se generan ramas utilizando el bucle `while`, una por cada tarea disponible (`t` es el índice del bucle). Cada rama es manejada por el método `EmployeesBTRecursiveCase`. Ambos métodos comparten las mismas precondiciones y postcondiciones que `EmployeesBT`, con la única diferencia en la condición relacionada con el índice `ps.k`. La precondición del caso base es `ps.k == ps.employeesAssign.Length`, lo que indica que se han tratado todos los funcionarios. Por otra parte, el caso recursivo utiliza la precondición `ps.k < ps.itemsAssign.Length`, permitiendo que aún queden elementos por considerar.

La figura 5.9 muestra la implementación y verificación del algoritmo `EmployeesBT`. Un bucle de llamadas recursivas permite asignar cada posible tarea al funcionario en curso `ps.k`. El invariante del bucle consta de:

- aquellas propiedades que garantizan que las precondiciones de las llamadas recursivas se cumplen (líneas 8 a 12).
- las siguientes tres propiedades, que garantizan que al finalizar el bucle de llamadas recursivas `bs` cumple las postcondiciones del método relacionadas con la optimalidad:
  - `bs` no ha cambiado o es una extensión óptima de una de las ramas anteriores (líneas 14 a 16):

```
1     invariant bs.Model().Equals(old(bs.Model())) ∨
2               ExistsBranchIsOptimalExtension(bs.Model(), ps.Model(),
3                                              input.Model(), t)
```

  - `bs` es mejor que todas las soluciones generadas por las ramas anteriores que ya fueron exploradas (líneas 18 y 19):

```
1     invariant ForallBranchesIsOptimalExtension(bs.Model(), ps.Model(),
2                                                input.Model(), t)
```

  - `bs` es mejor o igual que la mejor encontrada hasta el momento (líneas 21 y 22):

```
1     invariant bs.Model().TotalTime(input.Model().times) ≤
2               old(bs.Model().TotalTime(input.Model().times))
```

El método del caso recursivo solo se ejecuta si la tarea `t` no ha sido asignada (`ps.tasks[t] = false`). En caso contrario (`ps.tasks[t] = true`), se invoca el lema

```
method EmployeesBT(input: Input, ps: Solution, bs: Solution) {
  if (ps.k = input.times.Length0) {
      EmployeesBTBaseCase(input, ps, bs);
  }
  else {
    var t := 0;
    while t < input.times.Length0
      invariant 0 ≤ t ≤ input.times.Length0
      invariant input.Valid()
      invariant ps.Partial(input)
      invariant bs.Valid(input)
      invariant ∀ i | 0 ≤ i < ps.tasks.Length • ps.tasks[i] = old(ps.tasks[i])

      invariant bs.Model().Equals(old(bs.Model())) ∨
                ExistsBranchIsOptimalExtension(bs.Model(), ps.Model(),
                                               input.Model(), t)

      invariant ForallBranchesIsOptimalExtension(bs.Model(), ps.Model(),
                                                 input.Model(), t)

      invariant bs.Model().TotalTime(input.Model().times) ≤
                old(bs.Model().TotalTime(input.Model().times))
    {
      label L:
      if (¬ps.tasks[t]) {
        EmployeesBTRecursiveCase(input, ps, bs, t);
      }
      else {
        InvalidExtensionsFromInvalidPs(ps, input, t);
      }
      assert bs.Model().Equals(old(bs.Model())) ∨
             ExistsBranchIsOptimalExtension(bs.Model(), ps.Model(),
                                            input.Model(), t+1) by {
        ...
      }
      assert ForallBranchesIsOptimalExtension(bs.Model(), ps.Model(),
                                              input.Model(), t+1) by {
        ...
      }
      t := t + 1;
      assert ForallBranchesIsOptimalExtension(bs.Model(), ps.Model(),
      input.Model(), t);
    }
  }
}
```

Figura 5.9: Implementación de `EmployeesBT`

`InvalidExtensionsFromInvalidPs`. Este lema se diferencia del correspondiente en el problema de la mochila porque las condiciones que invalidan una solución no son las mismas. Mientras que en la mochila una solución no es válida si sobrepasa el peso máximo, en este problema la invalidez proviene de asignar una misma tarea a más de un funcionario. En la figura 5.10 podemos ver su definición y demostración. El lema establece que, si partimos de una solución parcial `ps` y la extendemos asignando al siguiente funcionario una tarea `t` que ya había sido asignada previamente (es decir, `ps.tasks[t] = true`), obtenemos una solución (`invalidPs`) que es inválida. Entonces, cualquier solución `s` que extienda a `invalidPs` también será inválida.

La mayoría de los invariantes Dafny los demuestra sin problema. Sin embargo, el invariante que establece que `bs` o bien no ha cambiado, o bien es una extensión óptima de una de las ramas anteriores, no es verificado por Dafny de manera automática.

```
1  lemma InvalidExtensionsFromInvalidPs(ps: Solution, input: Input, t : int)
2    requires input.Valid()
3    requires 0 ≤ ps.k < ps.employeesAssign.Length
4    requires 0 ≤ t < ps.tasks.Length = ps.employeesAssign.Length
5    requires ps.employeesAssign.Length = input.times.Length0
6    requires ps.tasks[t]
7    requires ps.Partial(input)
8    ensures ∀ s : SolutionData | var invalidPs := SolutionData(ps.Model().
         employeesAssign[ps.k := t], ps.k + 1);
9                                       ∧ |s.employeesAssign| = |invalidPs.
                                            employeesAssign|
10                                      ∧ s.k ≤ |s.employeesAssign|
11                                      ∧ invalidPs.k = ps.k + 1 ≤ s.k
12                                      ∧ s.Extends(invalidPs)
13                                      • ¬s.Valid(input.Model())
14 {
15   forall s : SolutionData |
16           var invalidPs := SolutionData(
17                                ps.Model().employeesAssign[ps.k := t],
18                                ps.k + 1
19                           );
20           ∧ |s.employeesAssign| = |invalidPs.employeesAssign|
21           ∧ s.k ≤ |s.employeesAssign|
22           ∧ ps.k + 1 ≤ s.k
23           ∧ s.Extends(invalidPs)
24   ensures ¬s.Valid(input.Model())
25   {
26     var invalidPs := SolutionData(ps.Model().employeesAssign[ps.k := t], ps.k + 1);
27     assert ∃ i | 0 ≤ i < invalidPs.k •
28         ∧ invalidPs.employeesAssign[i] = t
29         ∧ ¬(∀ j | 0 ≤ j < invalidPs.k ∧ i ≠ j • invalidPs.employeesAssign[j] ≠ t)
30     by {
31           assert invalidPs.employeesAssign[ps.k] = t;
32           assert invalidPs.Extends(ps.Model());
33     }
34   }
35 }
```

Figura 5.10: Lema `InvalidExtensionsFromInvalidPs`

Luego, la demostración de la línea 34 de la figura 5.9, es la que mostramos en la figura 5.11. Podemos ver que se distinguen tres casos:

1. **Primer caso:** `bs` no ha cambiado en ninguna de las ramas y es la que entró desde el principio en el método (línea 7).

2. **Segundo caso:** `bs` ha cambiado como resultado de la llamada recursiva que trata la tarea `t`, convirtiéndose así en la extensión óptima de `ps` extendida con `t` (líneas 8 a 16).

3. **Tercer caso:** `bs` no ha cambiado después de la llamada recursiva con `t`, por lo que debe ser la extensión óptima correspondiente a una de las ramas anteriores a `t` (líneas 17 a 59).

Estos tres escenarios se ilustran en la figura 5.12. Al separar estos casos, se ha visto que Dafny verifica los dos primeros casos de forma directa. No obstante, el tercer caso requiere un razonamiento más elaborado.

```
method EmployeesBT(input: Input, ps: Solution, bs: Solution) {
...
    assert bs.Model().Equals(old(bs.Model())) ∨
                ExistsBranchIsOptimalExtension(bs.Model(), ps.Model(),
                                               input.Model(), t+1) by {

        if bs.Model().Equals(old(bs.Model())) {}
        else if (¬old@L(bs.Model()).Equals(bs.Model())) {
          assert bs.Model().OptimalExtension(
                                  SolutionData(
                                         ps.Model().employeesAssign[ps.k := t],
                                         ps.k + 1
                                  ),
                                  input.Model()
                           );
        }
        else {
          assert ExistsBranchIsOptimalExtension(bs.Model(), ps.Model(),
                                               input.Model(), t+1)
          by {
            assert ExistsBranchIsOptimalExtension(bs.Model(),ps.Model(),
                                                  input.Model(), t+1)
            by {
                assert ExistsBranchIsOptimalExtension(
                            old@L(bs.Model()),
                            old@L(ps.Model()),
                            input.Model(),
                            t
                       );
                var i :| 0 ≤ i < t
                  ∧ var ext := old@L(SolutionData(
                                       ps.Model().employeesAssign[ps.k:= i],
                                       ps.k + 1
                                    )
                                 );
                  ∧ ext.Partial(input.Model())
                  ∧ old@L(bs.Model()).OptimalExtension(ext, input.Model());

                var ext := old@L(SolutionData(
                                    ps.Model().employeesAssign[ps.k := i],
                                    ps.k + 1
                                 )
                            );

                assert ext.Equals(SolutionData(
                                    ps.Model().employeesAssign[ps.k := i],
                                    ps.k + 1
                                  )
                            );

                assert ExistsBranchIsOptimalExtension(old@L(bs.Model()),ps.Model(),
                                                      input.Model(), t);

                assert ExistsBranchIsOptimalExtension(old@L(bs.Model()),ps.Model(),
                                                      input.Model(), t+1);

                assert ExistsBranchIsOptimalExtension(bs.Model(), ps.Model(),
                                                       input.Model(), t+1);
              }
          }
        }
    }
...
}
```

Figura 5.11: Demostración del invariante existencial de `EmployeesBT`

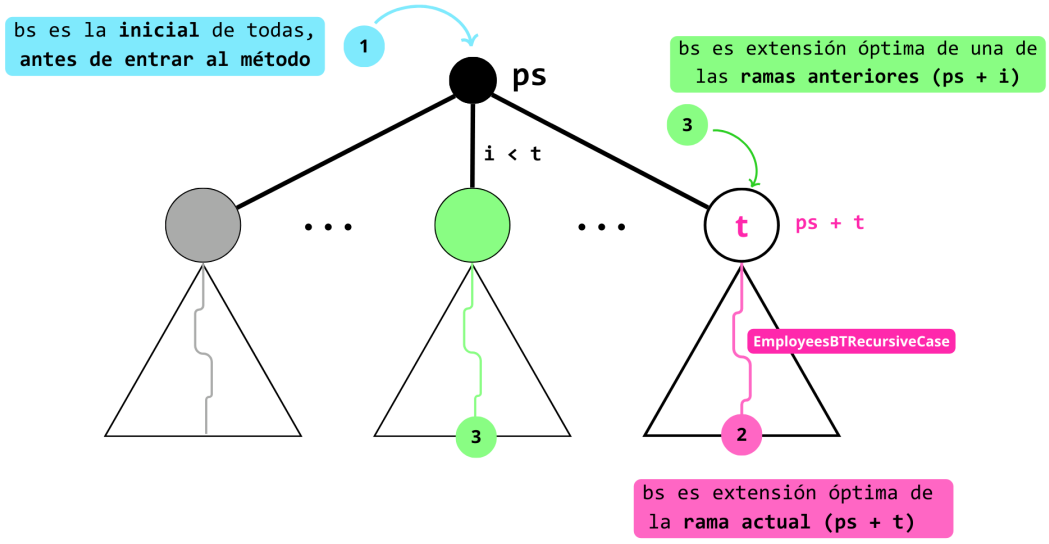


Figura 5.12: Ilustración de los distintos casos del estado de `bs` después del método `EmployeesBTRecursiveCase`

- Se parte de la hipótesis inductiva de que esta propiedad ya se cumplía hasta el índice `t` en el estado anterior (`old@L`), lo cual está garantizado por el invariante (figura 5.11, líneas 24 a 29).
- A continuación, tomamos la tarea `i` en el rango `0 <= i < t` tal que la solución parcial extendida en el punto antes de la llamada recursiva (`ext`) es válida y `bs` es una extensión óptima de dicha rama. Sabemos que existe una por el invariante (figura 5.11, líneas 30 a 43).
- Se concluye que si la versión anterior de `bs` (es decir, `old@L(bs)`) era una extensión óptima de `ps`, es decir, de alguna de las ramas generadas hasta el índice `t`, entonces también lo es hasta el índice `t+1`. Esto es porque si la existencia se cumple en el rango `0..t`, automáticamente se cumple en el rango `0..t+1` (figura 5.11, líneas 51 a 53).
- Finalmente, recordemos que estamos en el caso cuando `bs` no ha cambiado y es igual a `old@L(bs)`. Con esto se puede concluir que `bs` es una extensión óptima de la `ps` actual hasta el índice `t+1` (figura 5.11, líneas 55 y 56).

Por otro lado, para verificar el otro invariante adicional, el que establece que `bs` es una extensión óptima de `ps` en todas las ramas posibles hasta `t+1`, se utiliza una demostración por casos sobre el índice de la rama `i`. La demostración de la línea 38 de la figura 5.9, se muestra en la figura 5.13.

- **Caso** `i < t`: No se requiere verificación adicional, ya que se cumple directamente por el invariante.
- **Caso** `i = t`: Se demuestra explícitamente que cualquier solución válida `s` que extienda esta nueva rama tiene un tiempo total mayor o igual que el de `bs`

```
1  method EmployeesBT(input: Input, ps: Solution, bs: Solution) {
2  ...
3      assert ForallBranchesIsOptimalExtension(bs.Model(), ps.Model(),
4                                              input.Model(), t+1)
5      by {
6          assert ExistsBranchIsOptimalExtension(old@L(bs.Model()), old@L(ps.Model()),
               input.Model(), t);
7              var i :| 0 ≤ i < t ∧
8                       var ext := old@L(SolutionData(
9                                                ps.Model().employeesAssign[ps.k := i],
10                                               ps.k + 1
11                                           )
12                                  );
13                      ∧ ext.Partial(input.Model())
14                      ∧ old@L(bs.Model()).OptimalExtension(ext, input.Model());
15             var ext := old@L(SolutionData(ps.Model().employeesAssign[ps.k := i],
16                                               ps.k + 1));
17             assert ext.Equals(SolutionData(ps.Model().employeesAssign[ps.k := i],
18                                              ps.k + 1));
19             assert ExistsBranchIsOptimalExtension(old@L(bs.Model()), ps.Model(),
20                                                   input.Model(), t);
21             assert ExistsBranchIsOptimalExtension(old@L(bs.Model()), ps.Model(),
22                                                   input.Model(), t+1);
23             assert ExistsBranchIsOptimalExtension(bs.Model(), ps.Model(),
24                                                   input.Model(), t+1);
25     }
26 ...
27 }
```

Figura 5.13: Demostración del invariante universal de `EmployeesBT`

(es decir, `s` no mejora la solución actual, ya que se trata de un problema de minimización). Si `bs` fue actualizada al recorrer la rama `t`, es porque se encontró una solución mejor. Si no fue actualizada, es porque ninguna solución mejoró a la actual.

De este modo, se concluye que `bs` mantiene su optimalidad para todas las extensiones posibles de `ps` hasta el índice `t+1` (línea 23).

5.3. Llamada inicial

El método `ComputeSolution`, implementado en el fichero `Employees.dfy`, es el encargado de resolver el problema de los funcionarios, lo podemos ver en la figura 5.14. Recibe como parámetro la entrada del problema (`input`) que debe ser válida, y devuelve una solución (`bs`) que debe ser válida y óptima. El propósito de este método es preparar todos los componentes necesarios para la ejecución del algoritmo de vuelta atrás (`EmployeesBT`). Para ello, primero se inicializan las dos estructuras clave:

- La solución parcial (`ps`), que representa el estado actual de la asignación funcionario-tarea. Inicialmente, todos los funcionarios no tienen tarea asignada, el tiempo es 0 y el índice `ps.k` es inicializado a 0, indicando que aún no se ha asignado ningún funcionario. El marcador de tareas estará con todos sus componentes a `false`, lo que indica que todas las tareas están disponibles.
- La mejor solución (`bs`), que debe ser una solución completa y válida. Para ello, el array de asignaciones (`employeesAssign`) se inicializa de manera que

```
method ComputeSolution(input: Input) returns (bs: Solution)
  requires input.Valid()
  ensures bs.Valid(input)
  ensures bs.Optimal(input)
{
  var n := input.times.Length0;

  var bs_employeesAssign := new int[n];
  var bs_totalTime := 0.0;
  var bs_tasks := new bool[n](i ⇒ false);
  for i := 0 to n
    invariant input.Valid()
    invariant ∀ j | 0 ≤ j < i • 0 ≤ bs_employeesAssign[j] < n
    invariant ∀ j | 0 ≤ j < i • bs_employeesAssign[j] = j
    invariant ∀ j, k | 0 ≤ j < i ∧ 0 ≤ k < i ∧ j ≠ k • bs_employeesAssign[j] ≠
                                                       bs_employeesAssign[k]
    invariant 0.0 ≤ bs_totalTime =
              SolutionData(bs_employeesAssign[..],i).TotalTime(input.Model().times)
    invariant SolutionData(bs_employeesAssign[..],i).Partial(input.Model())
    invariant (∀ j | 0 ≤ j < i • bs_tasks[j] = (j in bs_employeesAssign[0..i]))
  {
    var s1 := SolutionData(bs_employeesAssign[..], i);
    bs_employeesAssign[i] := i;
    bs_tasks[i] := true;
    bs_totalTime := bs_totalTime + input.times[i,i];

    var s2 := SolutionData(bs_employeesAssign[..], i+1);
    SolutionData.AddTimeMaintainsSumConsistency(s1, s2, input.Model());
  }
  var bs_k := n;
  bs := new Solution(bs_employeesAssign, bs_totalTime, bs_k, bs_tasks);

  assert bs.Valid(input);

  var ps_employeesAssign := new int[n];
  var ps_totalTime := 0.0;
  var ps_k := 0;
  var ps_tasks := new bool[n](i ⇒ false);
  var ps := new Solution(ps_employeesAssign, ps_totalTime, ps_k, ps_tasks);

  EmployeesBT(input, ps, bs);
}
```

Figura 5.14: Implementación del metodo `ComputeSolution`

todos los funcionarios realizan tareas diferentes, por ejemplo, al funcionario i le asignamos la tarea i. En consecuencia, el campo `totalTime` contendrá la suma de los tiempos que cada funcionario dedica a realizar la tarea que le ha sido asignada. El marcador de tareas estará con todas sus componentes a `true`, lo que indica que todas las tareas han sido asignadas.

Antes de hacer la llamada inicial de `EmployeesBT`, se verifica que `bs` sea válida, ya que esta es una de las precondiciones necesarias de dicho método. Esto se consigue mediante los invariantes del bucle que la inicializa.

A continuación, se realiza la llamada inicial a `EmployeesBT`, pasando como parámetros la entrada (`input`), la solución parcial (`ps`) y la mejor solución encontrada hasta el momento (`bs`). Una vez finalizada la llamada, las dos postcondiciones del método `ComputeSolution` se verifican trivialmente gracias a las postcondiciones del método recursivo `EmployeesBT`.

```
method EmployeesBTBaseCase(input: Input, ps: Solution, bs: Solution) {
    if (ps.totalTime < bs.totalTime) {
        bs.Copy(ps);
        forall s : SolutionData | s.Valid(input.Model()) ∧ s.Extends(ps.Model())
          ensures s.TotalTime(input.Model().times) ≥
                  bs.Model().TotalTime(input.Model().times)
        {
          assert s.Equals(ps.Model());
          calc {
            s.TotalTime(input.Model().times);
            {s.EqualTimeFromEquals(ps.Model(), input.Model());}
            ps.Model().TotalTime(input.Model().times);
            {bs.CopyModel(ps, input);}
            bs.Model().TotalTime(input.Model().times);
          }
        }
    }
    else {
        forall s : SolutionData | s.Valid(input.Model()) ∧ s.Extends(ps.Model())
          ensures s.TotalTime(input.Model().times) ≥
                  bs.Model().TotalTime(input.Model().times)
        {
          assert s.Equals(ps.Model());
          s.EqualTimeFromEquals(ps.Model(), input.Model());
          assert s.TotalTime(input.Model().times) =
                 ps.Model().TotalTime(input.Model().times);
        }
    }
}
```

Figura 5.15: Implementación de `EmployeesBTBaseCase`

5.4. Método del caso base

La implementación y verificación de `EmployeesBTBaseCase` es prácticamente idéntica a la de `KnapsackBTBaseCase`. La única diferencia relevante es que la actualización de `bs` se produce cuando `ps.totalTime < bs.totalTime`, ya que en este caso se trata de un problema de minimización. Esta lógica se ilustra en la figura 5.15. Como se puede observar, se emplea un lema análogo al que se utilizó en el problema de la mochila: en lugar de `EqualValueWeightFromEquals`, aquí se usa `EqualTimeFromEquals`.

5.5. Método del caso recursivo

El método `EmployeesBTRecursiveCase` es el encargado de manejar la rama t-ésima, que corresponde a la tarea `t`. Su especificación es similar a la del método principal `EmployeesBT`, con la excepción de que incluye tres precondiciones adicionales:

- `ps.k < ps.employeesAssign`
- `0 <= t < input.times.Length0`
- `!ps.tasks[t]`

La primera precondición asegura que el método se ejecute únicamente en el caso recursivo, donde aún quedan funcionarios. La segunda precondición garantiza que la tarea recibida como parámetro sea válida, pues debe ser una tarea perteneciente

al conjunto de tareas posibles, estas son las que van de 0 a `input.times.Length0`. Y la tercera precondición garantiza que el método solo se ejecute si la tarea `t` no ha sido asignada previamente.

En la figura 5.16 podemos ver su implementación. Se asigna al funcionario actual `ps.k` la tarea de la rama que estamos procesando, la tarea `t`: `ps.employeesAssign[ps.k] = t`. Se actualiza el vector de marcaje poniendo la posición `t` de `ps.tasks` a `true`. Después se actualiza el tiempo total `ps.totalTime` sumándole el tiempo que tarda el funcionario `ps.k` en realizar la tarea `t`: `times[ps.k, t]`. Se avanza a la siguiente etapa y se invoca recursivamente al método `EmployeesBT` para continuar con la exploración. Después de la llamada, se restauran los valores de cada uno de los campos que fueron modificados.

Al igual que en el problema de la mochila, el aspecto clave aquí es asegurar que `ps` se mantiene como una solución parcial válida, es decir, que se cumple `ps.Partial(input)` después de modificar `ps` y antes de la llamada recursiva. Para ello, es necesario verificar dos condiciones fundamentales: por un lado, que el tiempo total acumulado en `ps` coincide con el de su modelo; y por otro, que se respeta la restricción de no asignar una misma tarea a más de un funcionario.

La verificación de que el tiempo total acumulado en `ps` coincide con el de su modelo se realiza mediante un bloque `calc`, que establece la consistencia entre el tiempo actual de `ps` y su estado previo, representado por `oldps` (líneas 12 a 18). Se parte de la expresión `ps.totalTime`, que equivale a `oldTotalTime + input.times[ps.k - 1, t]`, donde `oldTotalTime` representa el tiempo acumulado antes de la modificación. Este valor, a su vez, se sabe que es igual a `oldps.TotalTime(input.Model().times) + input.times[ps.k - 1, t]`. Sin embargo, Dafny no puede verificar directamente que esta expresión sea igual a `ps.Model().TotalTime(input.Model().times)`. Para resolver esto, se introduce un lema análogo al utilizado en el problema de la mochila: `AddTimeMaintainsSumConsistency`.

Para verificar que el marcador de tareas cumple el invariante de representación, es necesario demostrar el cuantificador universal que aparece en la figura 5.16, líneas 22 y 22.

Esta verificación se puede abordar mediante varios asertos, distinguiendo dos casos según el valor de la variable i:

1. **Caso 1:** $i = t$. Este caso es inmediato, ya que la tarea `t` es válida por la precondición del método: `0 <= t <input.times.Length0`.

2. **Caso 2:** $i < t$. Si `ps.tasks[i] == false`, entonces la tarea i no ha sido asignada a ningún funcionario, por lo que el predicado se cumple de forma trivial. Si `ps.tasks[i] == true`, entonces debemos demostrar que la tarea i aparece efectivamente en alguno de los arrays de asignación de empleados en `ps.employeesAssign`. Para ello, se aplican los siguientes razonamientos mediante asertos:

   - En el estado anterior, en `old(ps)`, la tarea i ya estaba marcada como asignada.
   - Se afirma que la tarea i estaba incluida en el modelo de asignaciones anteriores, es decir, que uno de los empleados hasta el índice `ps.k` tenía

asignada la tarea i.

- El modelo antes del cambio (hasta el índice `ps.k`) es igual al modelo actual hasta `ps.k-1`.
- Luego, se deduce que la tarea `i` está en el modelo actual hasta el índice `ps.k-1`, y por tanto, también está en el modelo actual hasta el índice `ps.k`.

```
1  method EmployeesBTRecursiveCase(input: Input, ps: Solution, bs: Solution, t : int)
2  {
3      ghost var oldps := ps.Model();
4      ghost var oldtotalTime := ps.totalTime;method EmployeesBTRecursiveCase(input:
           Input, ps: Solution, bs: Solution, t : int)
5
6      ps.employeesAssign[ps.k] := t;
7      ps.tasks[t] := true;
8      ps.totalTime := ps.totalTime + input.times[ps.k, t];
9      ps.k := ps.k + 1;
10
11     assert ps.Partial(input) by {
12     assert ps.Model().TotalTime(input.Model().times) = ps.totalTime by{
13       calc {
14         ps.totalTime;
15         oldtotalTime + input.times[ps.k - 1, t];
16         oldps.TotalTime(input.Model().times) + input.times[ps.k - 1, t];
17         { SolutionData.AddTimeMaintainsSumConsistency(oldps, ps.Model(), input.
               Model()); }
18         ps.Model().TotalTime(input.Model().times);
19       }
20     }
21
22     forall i | 0 ≤ i < ps.tasks.Length
23     ensures ps.tasks[i] = (i in ps.Model().employeesAssign[0..ps.k])
24     {
25       if i = t {}
26       else {
27         if (ps.tasks[i]) {
28           assert old(ps.tasks[i]);
29           assert i in old(ps.Model().employeesAssign[0..ps.k]);
30           assert old(ps.Model().employeesAssign[0..ps.k]) =
31                  ps.Model().employeesAssign[0..ps.k-1];
32           assert i in ps.Model().employeesAssign[0..ps.k-1];
33           assert i in ps.Model().employeesAssign[0..ps.k];
34         }
35         else {}
36       }
37     }
38
39     EmployeesBT(input, ps, bs);
40
41     label L:
42
43     ps.k := ps.k - 1;
44     ps.totalTime := ps.totalTime - input.times[ps.k, t];
45     ps.tasks[t] := false;
46 }
```

Figura 5.16: Implementación de `EmployeesBTRecursiveCase`

Capítulo 6

Podas de optimalidad

"Genius is patience"
— Isaac Newton

Una vez verificados el problema de la mochila y el problema de los funcionarios, en este capítulo veremos la verificación de sus versiones aplicando la poda de optimalidad. Esta técnica consiste en definir un método `Bound` que calcula la estimación de la mejor solución alcanzable desde una solución parcial y se aplica en los casos recursivos para evitar explorar ramas no prometedoras. A esta cota se la llama cota optimista porque siempre estima el mejor caso posible desde una solución parcial. Esto implica que su valor es siempre mayor o igual al valor real en problemas de maximización, o menor o igual en problemas de minimización. En el problema de la mochila, la cota se invoca en los métodos de sus ramas: `KnapsackBTTrueBranch` y `KnapsackBTFalseBranch`. En el problema de los funcionarios, se utiliza en el método que trata la rama de la tarea t-ésima: `EmployeesBTRecursiveCase`.

6.1. Poda en el problema de la mochila

El problema de la mochila es un problema de **maximización**, pues se busca maximizar el valor total de los objetos seleccionados sin exceder del peso máximo. Por ello, necesitamos una **cota superior** del valor de la mejor solución alcanzable. Entre las estrategias de poda más comunes se encuentran:

- **Incluir en la mochila todos los objetos restantes:** se suman todos los valores de los ítems que aún no han sido considerados, suponiendo que todos caben en la mochila.
- **Estrategia voraz**: seleccionar los ítems en orden decreciente por la relación valor/peso hasta que no quepan más en la mochila. Si el último no cabe completamente, se incluye la fracción del mismo que permite rellenar completamente la mochila.

Verificaremos la primera cota cuya implementación se encuentra en el fichero `BTBound.dfy` ubicado en el directorio `Knapsack`. El método que calcula esta cota se

```
method Bound (ps : Solution, input : Input) returns (bound : real)
  requires input.Valid()
  requires ps.Partial(input)
  ensures ∀ s : SolutionData | ∧ |s.itemsAssign| = |ps.Model().itemsAssign|
                               ∧ s.k = |s.itemsAssign|
                               ∧ ps.k ≤ s.k
                               ∧ s.Extends(ps.Model())
                               ∧ s.Valid(input.Model())
                               • s.TotalValue(input.Model().items) ≤ bound
{
  ghost var ps' := SolutionData(ps.Model().itemsAssign, ps.k);
  assert |ps'.itemsAssign| = |ps.Model().itemsAssign|;
  bound := ps.totalValue;

  assert bound = ps'.TotalValue(input.Model().items);

  for i := ps.k to ps.itemsAssign.Length
    invariant ps.k ≤ ps'.k ≤ |ps'.itemsAssign| = |ps.Model().itemsAssign|
    invariant ps'.Extends(ps.Model())
    invariant ∀ j | ps.k ≤ j < i • ps'.itemsAssign[j]
    invariant i = ps'.k
    invariant bound = ps'.TotalValue(input.Model().items)
  {
    var oldps' := ps';
    ps' := SolutionData(ps'.itemsAssign[ps'.k := true], ps'.k+1);
    bound := bound + input.items[i].value;
    SolutionData.AddTrueMaintainsSumConsistency(oldps', ps', input.Model());
  }
  SolutionData.AllTruesIsUpperBoundForAll(ps.Model(), ps', input.Model());
}
```

Figura 6.1: Implementación de `Bound`

muestra en la figura 6.1. Las precondiciones del método requieren que tanto la entrada como la solución parcial sean válidas. Como postcondición, se garantiza que el valor devuelto por la función (`bound`) es una cota superior del valor total de cualquier extensión válida de `ps`.

Para el cálculo de la cota superior, se crea `ps'`, una `ghost var` de la solución parcial que permite simular extensiones de `ps` sin modificar su estado original. Inicialmente, `bound` toma el valor total actual de `ps`. A continuación, en cada iteración del bucle `for`, `ps'` se actualiza con un nuevo ítem seleccionado (asignando el ítem `i` a `true`) y se suma su valor a `bound`. Es necesario invocar el lema `AddTrueMaintainsSumConsistency` visto en la sección 4.3.2, para asegurar que dicha actualización de `ps'` es correcta y coherente.

Finalmente, el lema `AllTruesIsUpperBoundForAll` permite concluir formalmente que ninguna extensión válida de `ps` puede superar el valor total de esta solución que tiene todos los elementos a `true`, lo que confirma que la cota calculada es efectivamente una cota superior válida. La demostración de dicho lema la podemos ver en la figura 6.2, y se apoya en el lema `EqualValueWeightFromEquals`, ya visto en la sección 4.3.2, y en el lema `AllTruesIsUpperBound`, mostrado en la figura 6.3.

El lema `AllTruesIsUpperBound` puede interpretarse como una versión más general del lema `AllTruesIsUpperBoundForAll`, ya que este último es invocado por el primero con un parámetro adicional que indica una posición específica `i` dentro del array `itemsAssign`. El lema `AllTruesIsUpperBound` establece que dada una solución parcial `ps`, una solución completa `ps'` que extiende a ella con todas las asignaciones a `true`, y otra cualquier solución completa `s` que extiende a `ps`, entonces el valor de `ps'` es cota superior del valor de `s`. La demostración de `AllTruesIsUpperBound` se realiza por inducción sobre la diferencia entre `|ps.itemsAssign|` y el parámetro `i`, recorriendo simultáneamente las asignaciones de `ps'`

```
static lemma AllTruesIsUpperBoundForAll(
    ps : SolutionData,
    ps' : SolutionData,
    input : InputData
)
    requires ps.k ≤ ps'.k = |ps'.itemsAssign| = |ps.itemsAssign| = |input.items|
    requires input.Valid()
    requires ps'.Extends(ps)
    requires ∀ j | ps.k ≤ j < |ps'.itemsAssign| • ps'.itemsAssign[j]
    ensures ∀ s : SolutionData | ∧ |s.itemsAssign| = |ps.itemsAssign|
                                 ∧ s.k = |s.itemsAssign|
                                 ∧ ps.k ≤ s.k
                                 ∧ s.Extends(ps)
                                 • s.TotalValue(input.items) ≤
                                 ps'.TotalValue(input.items)
{
    forall s : SolutionData | ∧ |s.itemsAssign| = |ps.itemsAssign|
                              ∧ s.k = |s.itemsAssign|
                              ∧ ps.k ≤ s.k
                              ∧ s.Extends(ps)
      ensures s.TotalValue(input.items) ≤ ps'.TotalValue(input.items)
    {
      assert SolutionData(s.itemsAssign, ps.k).Equals(ps);
      assert SolutionData(ps'.itemsAssign, ps.k).Equals(ps);
      SolutionData(s.itemsAssign,ps.k).EqualValueWeightFromEquals(
                                          SolutionData(ps'.itemsAssign, ps.k),
                                          input
                                        );
      SolutionData.AllTruesIsUpperBound(ps.k, s, ps, ps', input);
    }
}
```

Figura 6.2: Lema `AllTruesIsUpperBoundForAll`

y `s` desde `ps.k` hasta el final. El objetivo es alcanzar el caso base (cuando `i` llega al tamaño total de la solución) habiendo verificado en cada paso que el valor acumulado de `s` no supera al de `ps'`. Para ello, se distinguen dos casos en el paso inductivo:

- Si en la posición `i` ambas soluciones seleccionan el ítem (es decir, ambas asignaciones son `true`), se continúa la inducción con el siguiente índice.
- Si en la posición `i`, la solución `s` no selecciona el ítem (es decir, su asignación es `false`), mientras que `ps'` sí lo hace (ya que todas sus posiciones están asignadas a `true`), se aplica el lema auxiliar `AddFalsePreservesWeightValue` visto en la sección 4.6.2. Este garantiza que, al omitir un ítem, el valor total no aumenta. En otras palabras, `s` no suma nada en esa posición, mientras que `ps'` sí, lo que implica que `s` ya va quedando por debajo en valor acumulado. A continuación, se continúa con el paso recursivo.

Así, el lema `AllTruesIsUpperBoundForAll` permite concluir que `ps'` representa una cota superior segura para todas las extensiones válidas de `ps`.

Una vez verificado el método `Bound`, solo queda aplicarlo en los métodos correspondientes a las dos ramas del algoritmo. La implementación de esta lógica sería la siguiente:

```
var bound := Bound(ps, input);
if (bound > bs.totalValue) {
    KnapsackBT(input, ps, bs);
}
```

```
static lemma AllTruesIsUpperBound(
    i : int,
    s : SolutionData,
    ps : SolutionData,
    ps' : SolutionData,
    input : InputData
)
    decreases |ps.itemsAssign| - i
    requires input.Valid()
    requires ps.k ≤ ps'.k = |input.items| = |ps'.itemsAssign| = |ps.itemsAssign| =
        |s.itemsAssign|
    requires ps.k ≤ i ≤ |ps.itemsAssign|
    requires ∀ j | ps.k ≤ j < |ps'.itemsAssign| • ps'.itemsAssign[j]
    requires ∧ s.k = |s.itemsAssign| ∧ ps.k ≤ s.k
    requires ps'.Extends(ps)
    requires s.Extends(ps)
    requires SolutionData(s.itemsAssign, i).TotalValue(input.items) ≤
             SolutionData(ps'.itemsAssign, i).TotalValue(input.items)
    ensures s.TotalValue(input.items) ≤ ps'.TotalValue(input.items)
  {
    if i = |ps'.itemsAssign| {
      assert SolutionData(s.itemsAssign, i) = s;
      assert SolutionData(ps'.itemsAssign, i) = ps';
    }
    else {
      if (s.itemsAssign[i] ∧ ps'.itemsAssign[i]) {
        AllTruesIsUpperBound(i + 1, s, ps, ps', input);
      }
      else {
        AddFalsePreservesWeightValue(
            SolutionData(s.itemsAssign, i),
            SolutionData(s.itemsAssign, i+1),
            input
        );
        AllTruesIsUpperBound(i + 1, s, ps, ps', input);
      }
    }
  }
}
```

Figura 6.3: Lema `AllTruesIsUpperBound`

En lugar de invocar directamente `KnapsackBT`, se utiliza primero `Bound` para determinar si merece la pena continuar con la exploración de dicha rama. Esta estrategia permite descartar aquellas soluciones cuya cota superior no mejora la mejor solución encontrada hasta el momento.

6.2. Poda en el problema de los funcionarios

El problema de los funcionarios es un problema de **minimización**, pues se busca minimizar el tiempo total invertido por los funcionarios para llevar a cabo sus tareas. Por ello, necesitamos una **cota inferior** del tiempo total de la mejor solución alcanzable. Como ya adelantamos en la sección 3.2.5, verificaremos las siguientes propuestas de cota:

1. Considerar que el resto de funcionarios no van a tardar nada en realizar sus tareas:

   ```
   bound = ps.totalTime + 0
   ```

2. Calcular el mínimo global de la matriz `times` y considerar que el resto de funcionarios tardan el mismo tiempo y el mínimo posible. Como `ps.k` se incrementa antes de la

```
1  method Bound(ps : Solution, input : Input) returns (bound : real)
2    requires input.Valid()
3    requires ps.Partial(input)
4    ensures ∀ s : SolutionData | ∧ |s.employeesAssign| = |ps.Model().employeesAssign|
5                                ∧ s.k = |s.employeesAssign|
6                                ∧ ps.k ≤ s.k
7                                ∧ s.Extends(ps.Model())
8                                ∧ s.Valid(input.Model())
9                                • s.TotalTime(input.Model().times) ≥ bound
10 {
11   bound := ps.totalTime + 0.0;
12   assert bound = ps.Model().TotalTime(input.Model().times);
13
14   forall s : SolutionData | ∧ |s.employeesAssign| = |ps.Model().employeesAssign|
15                            ∧ s.k = |s.employeesAssign|
16                            ∧ ps.k ≤ s.k
17                            ∧ s.Extends(ps.Model())
18                            ∧ s.Valid(input.Model())
19   ensures s.TotalTime(input.Model().times) ≥ bound
20   {
21       ps.Model().GreaterOrEqualTimeFromExtends(s, input.Model());
22   }
23 }
```

Figura 6.4: Implementación de la primera cota

llamada del cálculo de la cota, la fórmula queda como sigue:

```
bound = ps.totalTime + min * (ps.employeesAssign.Length - ps.k)
```

3. Calcular el mínimo de la submatriz de `times` que comienza desde la fila `k` en adelante y considerar que el resto de funcionarios tardan el mismo tiempo y el mínimo posible:

   ```
   bound = ps.totalTime + submatrixMin[k] * (ps.employeesAssign.Length - ps.k)
   ```

6.2.1. Cota optimista

La primera propuesta la encontramos en el fichero `BTBound.dfy` del directorio `Employees`. Esta propuesta adopta una visión muy optimista, asumiendo que los empleados restantes no tardarán nada en completar sus tareas. En la figura 6.4 podemos ver que, en este caso, el método `Bound` simplemente devuelve el valor actual de `ps` sin realizar ningún cálculo adicional. La verificación del método se reduce a la invocación del lema `GreaterOrEqualTimeFromExtends` (análogo al lema `GreaterOrEqualWeightValueFromExtends` visto en la sección 4.3.2), pues el valor total de cualquier solución `s` debe ser como mínimo el valor de `ps` (`ps.totalTime`). Podemos visualizar esto mediante la ilustración de la figura 6.5. De este modo, `ps.totalTime` es cota inferior, pues el valor de `s` es mayor o igual que la cota.

6.2.2. Cota del mínimo global de la matriz `times`

La segunda propuesta, ubicada en el fichero `BTBound2.dfy` del directorio `Employees`, propone una cota un poco menos optimista que la anterior. En este caso, se estima que todos los funcionarios restantes tardarán en realizar sus tareas el tiempo mínimo, considerando para ello el mínimo global de la matriz `times`. Se debe calcular el mínimo global de dicha matriz como:

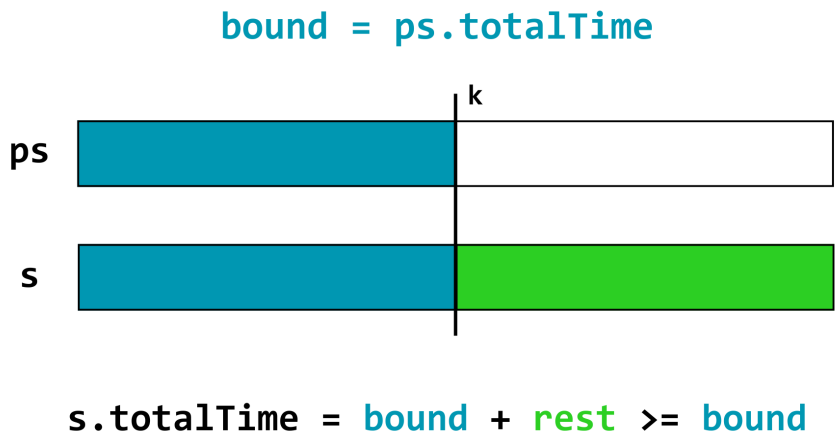


Figura 6.5: Ilustración de la primera cota de los funcionarios

$$minT = min\ \{\ times[i, j] \mid 0 \leq i < n \wedge 0 \leq j < n\ \}$$

Así se asume que cada uno de los funcionarios restantes (es decir, `n - k` funcionarios) realizará sus tareas en ese tiempo mínimo.

En la figura 6.6 se muestra la implementación de esta cota, en la que se define como: `bound = ps.totalTime + (n - k) * min`. Para verificar que es cota inferior de la mejor solución alcanzable, para cada solución `s` que extienda a `ps`, invocamos el lema `AddTimesLowerBound` de la figura 6.7. Este establece que dada una solución parcial `ps`, una solución completa `s` que extiende a `ps` y un mínimo de la submatriz desde la fila `row` en adelante (en este caso desde la fila 0, es decir, es el mínimo global de la matriz), el tiempo total de la solución completa `s` nunca excede del tiempo total de la solución parcial `ps` más el tiempo invertido por los funcionarios restantes de `ps` tardando el tiempo `min`. La idea detrás de la demostración de este lema puede visualizarse fácilmente con el esquema de la figura 6.8.

Al igual que en la cota anterior, sabemos que el tiempo total de `s` será, como mínimo, igual al de `ps.totalTime`. La cota propuesta consiste en `ps.totalTime` más n - k veces el valor mínimo global de la matriz de tiempos. El tiempo total de la solución completa `s` será `ps.totalTime` más una suma de n - k valores adicionales (*rest*), que provienen de acceder a la matriz `times` mediante el funcionario correspondiente y su tarea asignada: `times[ps.k][employeesAssign[ps.k]]`. Como cada uno de estos valores pertenece a la propia matriz `times`, es evidente que no pueden ser menores que el mínimo global `min`. Por lo tanto, el tiempo total de `s` siempre será mayor o igual que la cota propuesta, validando su corrección. La demostración en Dafny del lema `AddTimesLowerBound` se hace por inducción sobre la diferencia de índices entre `s.k` y `ps.k`, es decir, en el número de funcionarios que aún no ha asignado `ps`:

1. **Caso base:** Cuando `ps.k == s.k`, es decir, cuando no hay más empleados por asignar. En este caso, se demuestra que ambas soluciones son equivalentes y, por lo tanto, también lo son sus tiempos totales. Así se cumple la cota trivialmente.

2. **Caso inductivo:** Cuando aún quedan empleados por asignar, se construye una nueva solución parcial `ps'` a partir de `ps` asignando el siguiente empleado según lo

```
1  method Bound(ps : Solution, input : Input, min : real) returns (bound : real)
2    requires input.Valid()
3    requires ps.Partial(input)
4    requires input.IsMin(min, 0)
5    ensures ∀ s : SolutionData | ∧ |s.employeesAssign| = |ps.Model().employeesAssign|
6                                ∧ s.k = |s.employeesAssign|
7                                ∧ ps.k ≤ s.k
8                                ∧ s.Extends(ps.Model())
9                                ∧ s.Valid(input.Model())
10                               • s.TotalTime(input.Model().times) ≥ bound
11 {
12   var rest : real := ps.employeesAssign.Length − ps.k;
13   bound := ps.totalTime + (rest * min);
14
15   forall s : SolutionData | ∧ |s.employeesAssign| = |ps.Model().employeesAssign|
16                             ∧ s.k = |s.employeesAssign|
17                             ∧ ps.k ≤ s.k
18                             ∧ s.Extends(ps.Model())
19                             ∧ s.Valid(input.Model())
20     ensures s.TotalTime(input.Model().times) ≥ bound
21   {
22     SolutionData.AddTimesLowerBound(ps.Model(),s,input.Model(),min, 0);
23   }
24 }
```

Figura 6.6: Implementación de la segunda cota

hace la solución completa `s` (figura 6.7, líneas 29 y 30). Esto significa que tomamos la asignación de `s` en la posición `ps.k` y la incorporamos en `ps'` (`ps`), avanzando un paso más en la construcción de una solución completa. Se utiliza el lema `AddTimeMaintainsSumConsistency` (visto en la sección 5.5) para garantizar que el nuevo tiempo total de `ps'` se obtiene de manera consistente sumando al tiempo total de `ps` el valor correspondiente de la matriz de tiempos. Por definición del mínimo, se cumple que este valor de la matriz es mayor o igual que `min`, entonces:

$$\texttt{ps'.TotalTime} \geq \texttt{ps.TotalTime} + \texttt{min}$$

A partir de aquí, se aplica la hipótesis de inducción sobre `ps'` y `s`. El razonamiento se formaliza mediante un bloque `calc` (líneas 36 a 43), donde se encadenan las desigualdades paso a paso:

$$\begin{aligned} \texttt{s.TotalTime} &\geq \texttt{ps'.TotalTime + (n - ps.k - 1) * min} \\ &\geq \texttt{ps.TotalTime + min + (n - ps.k - 1) * min} \\ &= \texttt{ps.TotalTime + (n - ps.k) * min} \end{aligned}$$

La última igualdad se justifica con un `assert` que confirma la siguiente equivalencia matemática:

$$\texttt{min + (n - ps.k - 1) * min = (n - ps.k) * min}$$

Con esto se concluye que:

$$\texttt{s.TotalTime} \geq \texttt{ps.TotalTime + (n - ps.k) * min}$$

lo que completa la demostración del lema.

```
1  static lemma AddTimesLowerBound(
2      ps : SolutionData,
3      s: SolutionData,
4      input : InputData,
5      min : real,
6      row : int
7  )
8      decreases s.k − ps.k
9      requires input.Valid()
10     requires 0 < |input.times|
11     requires 0 ≤ row < |input.times|
12     requires input.IsMin(min, row)
13     requires |ps.employeesAssign| = |s.employeesAssign|
14     requires ps.Partial(input)
15     requires|s.employeesAssign| = |ps.employeesAssign|
16             ∧ s.k = |s.employeesAssign|
17             ∧ ps.k ≤ s.k
18             ∧ s.Extends(ps)
19             ∧ s.Valid(input)
20     ensures s.TotalTime(input.times) ≥ ps.TotalTime(input.times) + (|ps.
           employeesAssign| − ps.k) * min
21   {
22
23     if (ps.k = s.k) {
24       assert s.Equals(ps);
25       s.EqualTimeFromEquals(ps,input);
26       assert s.TotalTime(input.times) = ps.TotalTime(input.times);
27     }
28     else {
29       var ps' := SolutionData(ps.employeesAssign[ps.k := s.employeesAssign[ps.k]],
30                               ps.k + 1);
31       AddTimeMaintainsSumConsistency(ps,ps',input);
32
33       assert ps'.TotalTime(input.times) =
34              ps.TotalTime(input.times) +
35              input.times[ps.k][s.employeesAssign[ps.k]];
36
37       assert ps'.TotalTime(input.times) ≥ ps.TotalTime(input.times) + min;
38
39       AddTimesLowerBound(ps', s, input, min, row);
40
41       calc {
42        s.TotalTime(input.times);
43        ≥ ps'.TotalTime(input.times) + (|ps.employeesAssign| − ps.k − 1) * min;
44        ≥ ps.TotalTime(input.times) + min + (|ps.employeesAssign| − ps.k − 1) * min;
45        {assert min + (|ps.employeesAssign| − ps.k − 1) * min =
46                (|ps.employeesAssign| − ps.k) * min;}
47        ps.TotalTime(input.times) + (|ps.employeesAssign| − ps.k) * min;
48       }
49     }
50   }
```

Figura 6.7: Lema `AddTimesLowerBound`

A diferencia de la otra cota, esta requiere un precálculo, pues necesitamos tener ya calculado el mínimo de la matriz antes de llamar al método `Bound`. El precálculo se realiza en el fichero `Employees2.BT` cuando se preparan los elementos antes de llamar al método de la vuelta atrás `EmployeesBT`, su implementación la apreciamos en la figura 6.9. El precálculo consiste en dos bucles anidados cuya corrección se consigue mediante varios invariantes:

- **Bucle externo**:
  - En cada iteración i, el mínimo calculado es menor o igual que todos los elementos de las filas anteriores a i (figura 6.9, líneas 8 y 9).

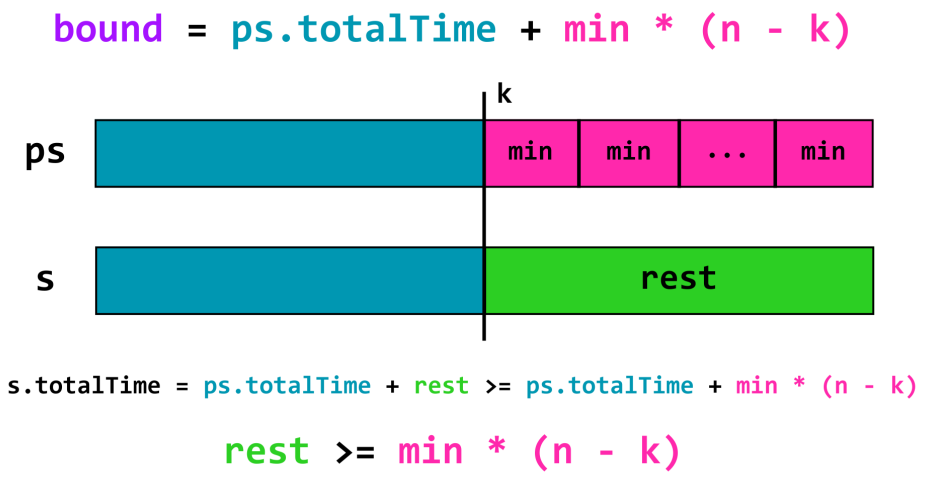


Figura 6.8: Ilustración de la tercera cota de los funcionarios

- En cada iteración i, existe un elemento de la matriz que coincide con el mínimo calculado. (figura 6.9, líneas 10 y 11).

- **Bucle interno**: Se cumplen los mismos invariantes que en el bucle externo y además que el mínimo calculado es menor o igual que los elementos de la fila actual i que van desde 0 a j (línea 16).

```
1  method Precalculation(input : Input) returns (min : real)
2    requires input.Valid()
3    ensures input.IsMin(min, 0)
4  {
5    min := input.times[0, 0];
6
7    for i := 0 to input.times.Length0
8      invariant ∀ f, c | 0 ≤ f < i ∧ 0 ≤ c < input.times.Length1
9                • min ≤ input.times[f, c]
10     invariant ∃ f, c | 0 ≤ f < input.times.Length0 ∧ 0 ≤ c < input.times.Length1
11               • min = input.times[f,c]
12   {
13     for j := 0 to input.times.Length1
14       invariant ∀ f, c | 0 ≤ f < i ∧ 0 ≤ c < input.times.Length1
15                 • min ≤ input.times[f, c]
16       invariant ∀ c | 0 ≤ c < j • min ≤ input.times[i,c]
17       invariant ∃ f, c | 0 ≤ f < input.times.Length0 ∧ 0 ≤ c < input.times.Length1
18                 • min = input.times[f, c]
19     {
20       if input.times[i, j] ≤ min {
21         min := input.times[i, j];
22       }
23     }
24   }
25 }
```

Figura 6.9: Implementación del precálculo de la segunda cota

6.2.3. Cota del mínimo de la submatriz de `times`

```
method Bound(ps : Solution, input : Input, submatrixMin : array<real>) returns (
    bound : real)
  requires input.Valid()
  requires ps.Partial(input)
  requires 0 ≤ ps.k ≤ ps.employeesAssign.Length = submatrixMin.Length = input.times
      .Length0
  requires ∀ i | 0 ≤ i < input.times.Length0 • input.IsMin(submatrixMin[i], i)
  ensures ∀ s : SolutionData | ∧ |s.employeesAssign| = |ps.Model().employeesAssign|
                               ∧ s.k = |s.employeesAssign|
                               ∧ ps.k ≤ s.k
                               ∧ s.Extends(ps.Model())
                               ∧ s.Valid(input.Model())
                               • s.TotalTime(input.Model().times) ≥ bound

{
  var rest : real := ps.employeesAssign.Length − ps.k;
  if (ps.k < ps.employeesAssign.Length)
    {bound := ps.totalTime + (rest * submatrixMin[ps.k]);}
  else
    {bound := ps.totalTime;}

  forall s : SolutionData | ∧ |s.employeesAssign| = |ps.Model().employeesAssign|
                            ∧ s.k = |s.employeesAssign|
                            ∧ ps.k ≤ s.k
                            ∧ s.Extends(ps.Model())
                            ∧ s.Valid(input.Model())
  ensures s.TotalTime(input.Model().times) ≥ bound
  {
    if (ps.k < ps.employeesAssign.Length)
     {SolutionData.AddTimesLowerBound(
                        ps.Model(),
                        s,
                        input.Model(),
                        submatrixMin[ps.k],
                        ps.k
                    );}
    else
     { s.EqualTimeFromEquals(ps.Model(),input.Model());}
  }
}
```

Figura 6.10: Implementación de la tercera cota

La tercera propuesta la encontramos en el fichero `BTBound3.dfy`. Esta es similar a la anterior, pero con una mejora en la precisión de la estimación: en lugar de utilizar el mínimo global de toda la matriz `times`, se considera el mínimo de la submatriz que va desde la fila actual `ps.k` hasta el final. De este modo, se obtiene una cota más ajustada y que poda más en comparación con la anterior. Para ser eficientes, guardaremos los mínimos en un array `submatrixMin`, de tal manera que en cada posición `i` se guarda el mínimo de la submatriz que comienza en la fila `i`. Este es un precálculo que se explicará más adelante.

El método `Bound` mostrado en la figura 6.10 calcula una cota inferior del tiempo total, suponiendo que todos los empleados tardan el tiempo mínimo posible de la submatriz correspondiente de `times`. Si aún quedan funcionarios por asignar, la cota se calcula como `bound := ps.totalTime + (n - k) * submatrixMin[ps.k]`, donde `n` es el número total de empleados. Si ya se han asignado todos los empleados (`ps.k == n`), la cota simplemente es el tiempo acumulado actual. La corrección de esta cota se consigue distinguiendo dos casos:

- Si `ps.k < ps.employeesAssign.Length` (quedan funcionarios por asignar), la verifi-

cación la garantiza el lema `AddTimesLowerBound`, que se utilizó para la segunda cota.

- Si `ps.k == ps.employeesAssign.Length` (no quedan funcionarios por asignar), se invoca el lema `EqualTimeFromEquals`, que asegura la igualdad de tiempos entre `s` y `ps`.

Esta propuesta, por tanto, mantiene una visión optimista pero realista, y mejora la precisión de la cota respecto a propuestas anteriores al tener en cuenta mínimos locales de la matriz en lugar de un mínimo global.

Para poder utilizar esta cota, necesitamos conocer de antemano los mínimos de cada submatriz de la matriz `times`. Para ello, en el fichero `Employees3.dfy` se incluye el método `Precalculation` (figura 6.11), cuya función es calcular y almacenar estos valores en un array `submatrixMin`. Este array tendrá la misma longitud que el número de filas de la matriz `times`, y en cada posición `i` almacenará el mínimo de toda la submatriz que comienza en la fila `i`.

Para que el cálculo sea eficiente, se realiza empezando por la última fila, y en cada paso se compara el mínimo actual de la fila `i` con el mínimo previamente calculado en la fila siguiente, almacenando el menor de ambos. De esta forma, se asegura que `submatrixMin[i]` siempre contiene el valor mínimo entre todos los elementos desde la fila `i` hasta la última. Esta propiedad está establecida por el predicado `input.IsMin(submatrixMin[i], i)`. El invariante del bucle (línea 12) establece que dicha propiedad se cumple para todos los índices de las filas ya procesadas.

```
1  method Precalculation(input : Input) returns (submatrixMin : array<real>)
2    requires input.Valid()
3    ensures submatrixMin.Length = input.times.Length0
4    ensures ∀ i | 0 ≤ i < input.times.Length0 • input.IsMin(submatrixMin[i], i)
5    ensures fresh(submatrixMin)
6  {
7    submatrixMin := new real[input.times.Length0];
8
9    var i := input.times.Length0 − 1;
10   while i ≥ 0
11     invariant −1 ≤ i ≤ input.times.Length0 − 1
12     invariant ∀ f | i < f < input.times.Length0 • input.IsMin(submatrixMin[f],f)
13   {
14     var j := 1; var min :=  input.times[i,0];
15     while j < input.times.Length1
16       invariant 0 ≤ j ≤ input.times.Length1
17       invariant ∀ c | 0 ≤ c < j • min ≤ input.times[i,c]
18       invariant ∃ c | 0 ≤ c < j • min = input.times[i,c]
19       invariant ∀ f | i < f < input.times.Length0 • input.IsMin(submatrixMin[f],f)
20     {
21       if input.times[i, j] ≤ min {
22         min := input.times[i, j];
23       }
24       j := j + 1;
25     }
26     if (i = input.times.Length0 − 1)
27          { submatrixMin[i] := min; }
28     else {
29          if (min < submatrixMin[i+1])
30             {submatrixMin[i] := min;}
31          else
32            {submatrixMin[i] := submatrixMin[i+1];}
33     }
34     i := i − 1;
35   }
36 }
```

Figura 6.11: Implementación del precálculo de la tercera cota

Capítulo 7

Conclusiones y Trabajo Futuro

"No llores porque se terminó, sonríe porque sucedió"
— Gabriel García Márquez

La metodología utilizada en este trabajo ha demostrado ser eficaz en la especificación y verificación de los problemas planteados. En esencia, la metodología se basa en establecer una distinción clara entre el modelo formal de los problemas y su implementación. Se emplearon clases para definir los objetos reales y tipos de datos algebraicos para los correspondientes modelos formales. Una de las características principales de la metodología es establecer una función `Model()` que asigna a cada objeto un valor abstracto con el que razonar más fácilmente a la hora de verificar propiedades. Al centrar la verificación en las operaciones sobre el modelo en vez de sobre la implementación, la verificación resultaba ser más manejable. Esto fue crucial para la verificación de múltiples propiedades sobre distintos componentes del programa.

El problema de la mochila, caracterizado por tener un árbol de exploración binario, fue el primero en ser verificado mediante dicha metodología, la cual permitió validar y desarrollar ciertas propiedades o lemas que eran necesarios. Se trataba de uno de los problemas más manejables, ya que el árbol de búsqueda implicaba, como máximo, dos invocaciones recursivas del propio algoritmo. Al completar la verificación de este primer problema, aplicamos la misma metodología al problema de los funcionarios, cuyo espacio de estados es un poco más complejo, pues es modelado con un árbol de búsqueda n-ario. Este problema requería un razonamiento más elaborado que el anterior, dado que implicaba un bucle de invocaciones recursivas en el que el número de iteraciones depende de la entrada del problema. No obstante, al aplicar la misma metodología, fue posible reutilizar varios lemas y predicados previamente establecidos en el problema de la mochila. También se verificaron varias podas de optimalidad en ambos problemas.

Con este resultado, se concluye que la metodología propuesta se puede aplicar a cualquier algoritmo de vuelta atrás.

Para problemas binarios, la verificación se asemeja al problema de la mochila, donde la estrategia principal radica en la división en dos métodos: uno para la rama `true` y otro para la rama `false`. De este modo, podemos enfocar el razonamiento de la corrección del método considerando, por separado, el resultado de extender la solución asumiendo el valor `true` y el resultado de extenderla asumiendo el valor `false`.

En el contexto de problemas con n ramas, se puede seguir la verificación del problema de los funcionarios. La característica clave reside en los invariantes de bucle que genera las ramas del árbol de búsqueda, donde la mayoría de estos invariantes coinciden precisamente

con las precondiciones necesarias para invocar la función encargada de procesar la tarea i-ésima.

Todos estos enfoques contribuyeron al éxito del trabajo, proporcionando una base sólida para abordar problemas más complejos y garantizando que las soluciones fueran correctas de manera formal.

Este trabajo abre varias vías de investigación y mejora que pueden ser exploradas en el futuro. Algunas propuestas son:

- **Verificación de otras cotas**: en el problema de la mochila, se podría demostrar la corrección de la estrategia voraz mencionada en la sección 6.1. En el problema de los funcionarios, se podría verificar la última cota mencionada en la sección 3.2.5, aquella que estima el tiempo total como la suma de los tiempos que cada funcionario tarda en completar la tarea que realiza con mayor rapidez.
- **Generalización de los módulos de cota**: una extensión útil sería crear un módulo abstracto (`abstract module` en Dafny) de cotas que permita introducir cualquier cota de manera flexible. El módulo proporcionaría la especificación adecuada de lo que sería una cota superior o inferior y el cálculo de dicha cota se implementaría de forma manual. Esta generalización permitiría aplicar diferentes estrategias de cota a un mismo problema sin necesidad de cambiar la implementación y verificación del algoritmo principal de vuelta atrás.
- **Verificación del método de ramificación y poda**: la verificación de algoritmos basados en este método puede apoyarse en este trabajo, ya que sigue un enfoque similar al método de vuelta atrás, pero con una naturaleza iterativa en lugar de recursiva. Se puede aplicar la misma metodología, empleando clases para la implementación y tipos de datos algebraicos para la especificación. Además, será necesario utilizar colas de prioridad que contengan objetos de tipo `Solution` utilizado en este proyecto.

Para concluir esta memoria, quisiera compartir las sensaciones y los desafíos que he encontrado a lo largo del desarrollo de este trabajo.

En primer lugar, siento una profunda gratitud por haber tenido la oportunidad de explorar la verificación formal de un algoritmo como tema de este proyecto, ya que la algoritmia y el razonamiento matemático son las áreas que más disfruto. Uno de los aspectos más gratificantes de este trabajo ha sido la inmersión profunda en el algoritmo de vuelta atrás, un método que siempre ha capturado mi interés dentro del abanico de técnicas algorítmicas estudiadas durante la carrera. Al elegir la temática de este Trabajo de Fin de Grado, mi tutora Clara me orientó hacia la línea de verificación de algoritmos explorada en proyectos anteriores, y fue ella quien me sugirió abordar específicamente la vuelta atrás. Esta propuesta resonó de inmediato conmigo, ya que coincidía con mi algoritmo favorito, lo que añadió una motivación extra al proyecto.

Uno de los principales desafíos al comenzar fue familiarizarme con Dafny, no tanto por su sintaxis, sino por su funcionamiento en particular. Lo especialmente llamativo fue encontrarme con propiedades que, aunque resultaban intuitivamente obvias, requerían una especificación formal y detallada para que Dafny pudiera verificarlas correctamente. Esto me llevaba a reflexionar profundamente sobre cómo comunicar de manera formal y precisa conceptos que a menudo damos por obvios en el razonamiento informal.

Otro desafío significativo encontrado durante el desarrollo de este trabajo ha sido la ralentización ocasional del verificador Dafny al intentar probar ciertas propiedades. En

algunos casos, la complejidad de las demostraciones o la necesidad de proporcionar lemas o guías adicionales para el demostrador automático ha incrementado considerablemente el tiempo en obtener una verificación exitosa.

A pesar de estos desafíos, la experiencia adquirida en el uso de Dafny ha sido fundamental, estableciendo una base sólida para futuros trabajos. Asimismo, la guía constante de mis tutores ha sido de gran valor para desarrollar habilidades de investigación y estructurar el pensamiento analítico.

Conclusions and Future Work

The methodology used in this work has proven effective in specifying and verifying the proposed problems. Essentially, the methodology is based on establishing a clear distinction between the formal model of the problems and their implementation. When defining the types involved to model the input and output data of the problems, classes were used to define real-world objects, and algebraic data types were used for the corresponding formal models. One of the main features of the methodology is the definition of a `Model()` function that assigns each object an abstract value, which facilitates reasoning when verifying properties.

By delegating responsibilities to specific functions within the model, verification became more manageable. This was crucial for verifying multiple properties across different components of the program.

The knapsack problem, characterized by a binary exploration tree, was the first to be verified using this methodology. It allowed for the validation and development of certain necessary properties or lemmas. This was one of the more manageable problems, as the search tree involved, at most, two recursive calls of the algorithm itself. After completing the verification of this initial problem, we applied the same methodology to the civil servants problem, whose state space is somewhat more complex, being modeled with an n-ary search tree. This problem required more elaborate reasoning than the previous one, as it involved a loop of recursive calls where the number of iterations depends on the input size. Nonetheless, by applying the same methodology, we were able to reuse several lemmas and predicates previously established in the knapsack problem. Several optimality prunings were also verified in both problems.

Based on these results, we conclude that the proposed methodology can be applied to any backtracking algorithm. For binary problems, the verification process resembles that of the knapsack problem, where the main strategy lies in dividing the procedure into two methods: one for the `true` branch and one for the `false` branch. This allows us to focus the correctness reasoning of the method by separately considering the result of extending the solution assuming the value `true` and the result of extending it assuming the value `false`.

In the context of problems with n branches, the verification can follow the approach used in the civil servants problem. The key feature lies in the loop invariants that generate the branches of the search tree, where most of these invariants precisely match the preconditions required to invoke the function responsible for processing the i-th branch.

All of these approaches contributed to the success of the project, providing a solid foundation for tackling more complex problems and ensuring that the solutions were formally correct.

This work opens several avenues for further research and improvement that can be explored in the future. Some proposals include:

- **Verification of other bounds**: In the knapsack problem, the correctness of the greedy strategy mentioned in section 6.1 could be demonstrated. In the civil servants problem, the last bound mentioned in section 3.2.5 could be verified—this is the one that estimates the total time as the sum of the times each civil servant takes to complete the task they perform fastest.

- **Generalization of bounding modules**: A useful extension would be to create a general bounding module that allows the flexible introduction of any bound. The module would provide the appropriate specification of what constitutes an upper or lower bound, and the calculation of that bound would be implemented manually. This generalization would allow different bounding strategies to be applied to the same problem without needing to change the type of the precomputed values specifically for each bound.

- **Verification of the branch and bound method**: The verification of this algorithm can build upon this work, as it follows a similar approach to the backtracking algorithm but with an iterative rather than recursive nature. The same methodology can be applied, using classes for the implementation and algebraic data types for the specification. Moreover, it will be necessary to use priority queues containing `Solution`-type objects, as used in this project.

To conclude this thesis, I would like to share the impressions and challenges I encountered during the development of this work.

First and foremost, I feel a deep sense of gratitude for having had the opportunity to explore the formal verification of an algorithm as the topic of this project, since algorithmics and mathematical reasoning are the areas I enjoy the most. One of the most rewarding aspects of this work has been the deep dive into the backtracking algorithm, a method that has always captured my interest among the all algorithmic techniques studied throughout my degree. When choosing the subject of this Final Degree Project, my advisor Clara guided me towards the line of algorithm verification explored in previous projects, and it was she who suggested specifically tackling backtracking. This proposal immediately resonated with me, as it matched my favorite algorithm, which added extra motivation to the project.

One of the main challenges at the outset was getting familiar with Dafny, not so much due to its syntax, but rather because of how it operates. What stood out most was encountering properties that, while intuitively obvious, required a formal and detailed specification so that Dafny could verify them correctly. This led me to think deeply about how to formally and precisely convey concepts that are often taken for granted in informal reasoning.

Another significant challenge during the development of this work was the occasional slowdown of the Dafny verifier when attempting to prove certain properties. In some cases, the complexity of the proofs or the need to provide additional lemmas or guidance to the automatic prover greatly increased the time required to achieve successful verification.

Despite these challenges, the experience gained using Dafny has been fundamental, laying a solid foundation for future work. Likewise, the continuous guidance from my advisors has been invaluable in developing research skills and structuring analytical thinking.

Bibliografía

Y así, del mucho leer y del poco dormir, se le secó el celebro de manera que vino a perder el juicio.

Miguel de Cervantes Saavedra

[1] Colaboradores de Wikipedia. Problema de la mochila, 2024. URL: https://es.wikipedia.org/wiki/Problema_de_la_mochila.

[2] Colaboradores de Wikipedia. Problemas de satisfacción de restricciones, 2024. URL: https://es.wikipedia.org/wiki/Problema_de_satisfacci%C3%B3n_de_restricciones.

[3] Colaboradores de Wikipedia. Verificación formal, 2024. URL: https://es.wikipedia.org/wiki/Verificaci%C3%B3n_formal.

[4] Colaboradores de Wikipedia. Vuelta atrás, 2024. URL: https://es.wikipedia.org/wiki/Vuelta_atr%C3%A1s.

[5] Robert W. Floyd. *Assigning Meanings to Programs*, pages 65–81. Springer Netherlands, Dordrecht, 1993. URL: https://doi.org/10.1007/978-94-011-1793-7_4.

[6] C. A. R. Hoare. *An axiomatic basis for computer programming*, pages 576–580. Association for Computing Machinery New York, NY, United States, 1969. URL: https://dl.acm.org/doi/10.1145/363235.363259.

[7] The Dafny lang community. The Dafny Reference Manual, 2024. URL: https://Dafny.org/Dafny/DafnyRef/DafnyRef.

[8] K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR*, pages 348–370. Springer, 2010.

[9] K.R.M. Leino. *Program Proofs*. MIT Press, 2023.

[10] Narciso Martí Oliet, Yolanda Ortega Mallén, and José A. Verdejo López. *Estructuras de datos y métodos algorítmicos: 213 Ejercicios resueltos*. MIT Press, 2013.

[11] P. Pastor Pérez. Trabajo Fin de Máster: Verificación de algoritmos voraces en Dafny. UCM (*Docta Complutense*), 2022.

[12] Microsoft Research. Dafny: A language and program verifier for functional correctness, 2008. URL: `https://www.microsoft.com/en-us/research/project/Dafny-a-language-and-program-verifier-for-functional-correctness/`.

[13] J. Blázquez Saborido. Trabajo Fin de Grado: Verificación de estructuras de datos enlazadas en Dafny. UCM (*Docta Complutense*), 2021.

[14] J. Blázquez Saborido. Trabajo Fin de Máster: Vericación de estructuras de datos arborescentes con iteradores en Dafny. UCM (*Docta Complutense*), 2022.

[15] Edgar Serna Montoya. Métodos formales e ingeniería de software. *Revista Virtual Universidad Católica del Norte*, 1(30):158–184, jul. 2011.

[16] D. Trujillo Viedma. Trabajo Fin de Máster: Verificación formal de algoritmos de programación dinámica en Dafny. UCM (*Docta Complutense*), 2023.

[17] P. Martín Viñuelas. Trabajo Fin de Grado: Verificación de algoritmos sobre segmentos de un vector utilizando módulos abstractos en Dafny. UCM (*Docta Complutense*), 2024.