Universitat
de les Illes Balears

# TRABAJO DE FIN DE MÁSTER

# EVALUACIÓN DE TÉCNICAS PARA LA BÚSQUEDA DE CONFIGURACIONES EN SISTEMAS EMPOTRADOS DISTRIBUIDOS Y ADAPTATIVOS


**Bartomeu Alcover Borrás**


**Máster Universitario en Sistemas Inteligentes (MUSI)**

**Especialidad: Internet de las cosas e Inteligencia artificial aplicada**

**Centro de Estudios de Posgrado**

**Año Académico 2020-21**

# Evaluación de Técnicas para la Búsqueda de Configuraciones en Sistemas Empotrados Distribuidos Adaptativos

Bartomeu Alcover Borrás
**Tutor:** Alberto Ballesteros Varela y Julián Proenza Arenas
Trabajo de fin de Máster Universitario en Sistemas Inteligentes (MUSI)
Universitat de les Illes Balears
07122 Palma, Illes Balears, Espanya
t.alcover@estudiant.uib.com

***Resumen*—Un Sistema Empotrado Distribuido Adaptativo (SEDA) es capaz de operar de forma autónoma en contextos operacionales cambiantes. Este tipo de sistema típicamente está constituido por múltiples nodos interconectados en los que se ejecutan tareas que cooperan para conseguir un objetivo común. Para conseguir un alto nivel de adaptabilidad, los SEDA pueden alojar, desalojar o, incluso, realojar tareas cuando el contexto operacional cambia. Sin embargo, encontrar una nueva asignación de tareas a los nodos que satisfaga los requisitos del sistema puede ser un proceso largo, dependiendo del número de nodos, tareas y atributos de ejecución de éstas. Además, los requisitos de la búsqueda pueden ser diferentes, dependiendo de diversos factores, como el estado del sistema o el entorno. El objetivo de este trabajo es el de implementar, probar y evaluar varias técnicas de búsqueda para determinar la mayor o menor adecuación de cada una de ellas en diferentes escenarios.**

***Index Terms*—sistema empotrado distribuido, adaptabilidad, backtracking, ramificación y poda, algoritmo voraz, tabu search, SMT solver**

## ABSTRACT

An Adaptive Distributed Embedded System (ADES) is able to operate autonomously in changing operational contexts. This kind of system is typically constructed as a set of multiple interconnected nodes in which tasks execute to achieve some common goal. To achieve a high level of adaptivity ADES can allocate, deallocate or, even, reallocate tasks when the operational context changes. However, finding a new assignment of tasks to nodes that satisfies the system requirements can be a lengthy process, depending on the number of nodes, tasks and the operational attributes of these last ones. Moreover, the search requirements may be different, depending on various factors, such as the state of the system or the environment. The aim of this work is to implement, test and evaluate various search techniques to determine the suitability of each of them in different scenarios.

***Index Terms*—distributed embedded system, adaptivity, backtracking, branch and bound, greedy search, tabu search, SMT solver**

## I. INTRODUCCIÓN

Los Sistemas Empotrados Distribuidos (SEDs) juegan un papel clave en múltiples sectores económicos tales como la aviación, la industria automotriz, la industria ferroviaria, la sanidad, la distribución de energía y las telecomunicaciones.

Un SED es una combinación de hardware y software de cómputo, y tal vez otras partes adicionales, diseñado para realizar una función concreta, en muchos casos formando parte de un sistema o producto mayor [6]. Más concretamente, los SEDs se caracterizan por estar formados por diversos componentes que realizan diferentes acciones de forma coordinada, comunicándose entre sí [25]. La parte hardware se implementa gracias a múltiples *nodos* interconectados por una red de comunicaciones, mientras que la parte software se suele construir en forma de múltiples *tareas*, que se ejecutan en dichos nodos. A lo largo de este trabajo nos referiremos a la asignación de tareas a los nodos, junto con los atributos de ejecución de dichas tareas, como *configuración del sistema* o, simplemente, *configuración*.

Este tipo de sistemas se usa típicamente para interactuar con el mundo físico. Así pues, no es extraño que éstos tengan requisitos de tiempo real. Un sistema tiene restricciones de tiempo real si el correcto funcionamiento de éste no sólo depende de que sea capaz de proporcionar una respuesta correcta, sino también de que lo haga antes de un plazo máximo llamado *deadline*. Por poner algunos ejemplos, la mayoría de sistemas de control son SEDs de tiempo real: control de algunos electrodomésticos, control de frenada de un vehículo, control de procesos industriales o control de experimentos científicos.

Dependiendo de la aplicación, un SED también puede tener requisitos de *garantía de funcionamiento*. Más concretamente, este atributo es necesario cuando un servicio incorrecto o la ausencia de éste puede provocar la pérdida de bienes de gran valor como pueden ser datos, grandes cantidades de dinero o, incluso, vidas humanas. Se puede definir la garantía de funcionamiento como la capacidad del sistema para proporcionar un servicio confiable [16]. Este es un concepto integrador que incluye varios atributos, como son: la disponibilidad (la capacidad del sistema de estar preparado para proporcionar un servicio correcto, incluso tras haber sufrido alguna avería) y la fiabilidad (la capacidad del sistema de proporcionar un servicio correcto de manera ininterrumpida). Algunos ejemplos de SED con requisitos de garantía de funcionamiento son los sistemas

que gobiernan: el servidor de una web importante, vehículos que transportan personas, equipamiento médico y dispositivos espaciales.

Por otro lado, hoy en día ha crecido el interés en usar SEDs en *contextos operacionales* que pueden cambiar de manera impredecible. Tal y como se puede ver en la figura 1, consideraremos que el contexto operacional incluye los *requisitos operacionales* y las *condiciones operacionales* [4]. Por un lado los requisitos operacionales cubren los *requisitos funcionales* (lo que el sistema tiene que hacer) y los *requisitos no funcionales* (las características de funcionamiento, en este caso, el funcionamiento en tiempo real y la garantía de funcionamiento). Por otro lado, las condiciones operacionales son las circunstancias bajo las cuales el sistema tiene que operar. Esto incluye tanto el entorno en el que el sistema opera, así como el propio sistema, que puede cambiar debido a averías en sus subsistemas.

Un SED capaz de operar en contextos operacionales dinámicos tiene que ser *adaptativo*. Es decir, el sistema, de manera autónoma, tiene que ser capaz de cambiar su configuración siempre que el contexto operacional cambie. Esto implica que el sistema tiene que: (1) monitorizar y analizar tanto los requisitos como las condiciones operacionales para determinar cuándo es necesario cambiar de configuración; (2) encontrar una nueva configuración; y (3) aplicar la nueva configuración.

Si consideramos como configuraciones *posibles* aquellas que son compatibles con las condiciones operacionales de una situación concreta, podemos decir que encontrar una nueva configuración consiste en recorrer el espacio de configuraciones posibles hasta encontrar una *configuración correcta*, es decir, una configuración que cumpla con los requisitos funcionales y no funcionales del sistema. Este proceso de búsqueda de la nueva configuración puede requerir una gran cantidad de tiempo dependiendo del número de nodos, tareas y atributos de éstas.

Además, los requisitos de búsqueda pueden ser diferentes dependiendo del momento:

- Si ocurre un cambio repentino en el contexto operacional, el sistema debería intentar restablecer el servicio lo antes posible. Así pues, la búsqueda debería encontrar una configuración correcta en el menor tiempo posible.
- Si el sistema ya está en una configuración correcta, pero una configuración diferente pudiera beneficiar su funcionamiento, valdría la pena dedicar tiempo a encontrar esa configuración mejor. Por ejemplo, en un sistema que esté alimentado por baterías, una configuración que agrupe las tareas en un grupo reducido de nodos permitiría apagar el resto de ellos, lo que reduciría el consumo energético. En este caso, la búsqueda debería encontrar una configuración correcta mejor en el menor tiempo posible, pero con menor urgencia que en el caso anterior.

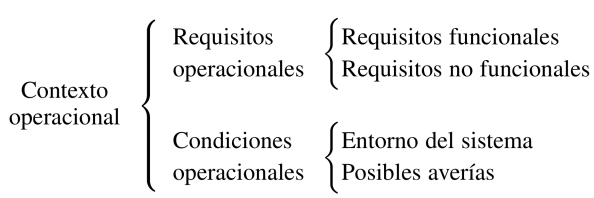


Figura 1. Definición de contexto operacional.

El objetivo de este trabajo es el de implementar, probar y evaluar varias técnicas para la búsqueda de configuraciones en Sistemas Empotrados Distribuidos Adaptativos (SEDAs). En nuestro caso concreto, hemos hecho la gran mayoría de búsquedas de una configuración correcta partiendo desde una configuración vacía, es decir, una situación en la que no hay ninguna tarea asignada a ningún nodo, aunque hemos intentado implementar cada técnica de forma que nos sirva también para buscar una configuración mejor partiendo de una configuración correcta previa. Sin embargo, no siempre ha sido posible: concretamente, la falta de familiaridad con la herramienta tipo *Solver*, que explicaremos más adelante, nos ha impedido implementarla de esta forma, aunque estamos bastante seguros de que sería posible hacerlo.

Es importante decir que el sentido común ya hace previsible el hecho de que no exista una técnica de búsqueda objetivamente mejor en todos los aspectos, dado que la valoración que se haga de los resultados dependerá de muchos factores. Es por ello que nuestro objetivo no es determinar cuál es mejor, sino implementarlas, probarlas en diferentes escenarios, identificar cuáles son las ventajas y desventajas de cada una y ver cuál se adapta mejor a cada situación. Las conclusiones extraídas de este trabajo servirán para dar soporte a decisiones de diseño e implementación del proyecto de investigación DFT4FTT [5], cuyo objetivo es desarrollar una infraestructura para sistemas empotrados distribuidos adaptativos críticos.

Dicho esto, hemos seleccionado un conjunto que incluye las 5 técnicas de búsqueda que listamos a continuación.

- Algoritmo *Backtracking* Básico
- Algoritmo de Ramificación y Poda
- Algoritmo Voraz
- Algoritmo *Tabu Search*
- Herramienta *SMT Solver*

Consideramos que dentro del conjunto seleccionado hay un buen balance entre soluciones clásicas como *Backtracking*, *Ramificación y Poda* y *Algoritmo Voraz* (basado en heurística, es decir, en utilizar algún método para reducir el espacio de búsqueda con el objetivo de agilizar la misma), así como soluciones más avanzadas como *Tabu Search* (basado en metaheurística) y *SMT Solver*. En concreto, estas dos últimas nos parecen especialmente interesantes ya que se usan en la actualidad para resolver problemas similares [17][21][19].

Dentro de este conjunto de 5 técnicas, diferenciaremos entre aquellas técnicas en las que hemos utilizado algún procedimiento externo, como una librería, para implementar la lógica de búsqueda, y aquellas en las que las que hemos implementado nosotros mismos todos los elementos. Concretamente, la única solución en la que hemos utilizado un procedimiento externo (una librería, concretamente) para implementar la lógica de búsqueda ha sido el *Solver*, en todas sus variantes. Es por ello que, de aquí en adelante, nos referiremos a éste en concreto como herramienta, y no como técnica, si bien seguiremos utilizando el término técnica para referirnos al conjunto de soluciones de búsqueda en general. Más adelante, al hablar de cada una de las técnicas y herramientas con más

detalle, explicaremos el grado de implementación que hemos realizado en cada una de ellas.

El resto de este documento está estructurado de la siguiente forma. Primero, en la sección II presentamos las características del modelo de sistema que hemos supuesto, e indicaremos lo que es una configuración posible y una correcta dentro de ese modelo. Después, en la sección III explicamos el criterio que hemos utilizado en las diferentes técnicas de búsqueda para determinar cuándo una configuración es mejor que otra, siempre dependiendo del posible contexto operacional. Este criterio es clave, por ejemplo, en el caso que hemos mencionado anteriormente, en el que el sistema ya está en una configuración correcta, pero queremos que busque una configuración mejor. En la sección IV describimos cada una de las técnicas de búsqueda seleccionadas, explicando su funcionamiento teórico, ventajas y desventajas a priori, y cómo la hemos adaptado a nuestro caso particular. A continuación, en la sección V mostramos los resultados obtenidos de la evaluación cualitativa y cuantitativa que hemos llevado a cabo para cada una de las técnicas. Para terminar, en el apartado VI haremos un resumen del trabajo realizado y los resultados obtenidos, expresando nuestra valoración y pensamientos finales. Además, también haremos un breve comentario sobre los posibles trabajos futuros que se podrían acometer para ampliar y mejorar este estudio.

## II. Modelado del Sistema

El primer paso en la evaluación de las técnicas de búsqueda es definir qué es un sistema y qué es una configuración del mismo. En este trabajo suponemos que nuestro sistema es una versión simplificada de un SEDA. Esto se debe a que los sistemas distribuidos pueden ser muy complejos. Por ejemplo, como hemos comentado en la introducción, es habitual que éstos tengan requisitos de tiempo real y/o de garantía de funcionamiento. Modelar con gran detalle un sistema con estos requisitos no es viable en este trabajo. Por un lado, modelar estas propiedades es bastante complejo. Por otro lado, la cantidad adicional de parámetros del sistema haría que el espacio de configuraciones posibles no fuera manejable, teniendo en cuenta el tiempo y los recursos disponibles para realizar este trabajo. Es por ello que hemos optado por una definición más simple, obviando los requisitos de tiempo real y de garantía de funcionamiento. Sin embargo, aunque no tratemos estos requisitos directamente, en algunos puntos de este documento sí que discutiremos algunos aspectos relacionados con el comportamiento en tiempo real de las técnicas. Por ejemplo, valoraremos la posibilidad de encontrar una configuración correcta en un tiempo limitado. Dicho esto, el sistema que hemos definido está compuesto por:

- Un conjunto de nodos denominado $N$, de cardinal $n$ y definido como $N = \{n_i(Cap_i), i = 1, 2, ...n\}$, cada uno con una capacidad, es decir, una cantidad limitada de recursos para alojar y ejecutar tareas.
- Un conjunto de tareas a ejecutar, denominado *T*, de cardinal *t* y definido como $T = \{t_i(Cost_i), i = 1, 2, ...t\}$, cada una con un coste determinado que consideraremos que se mide en las mismas unidades que la capacidad de cada nodo.

Dentro de un sistema, cada nodo y cada tarea tiene asociado un número identificativo, y cada tarea puede ser asignada únicamente a un nodo, sea cual sea éste, y siempre que el mismo tenga capacidad suficiente para asumir el coste de alojar y ejecutar dicha tarea. Consideraremos cómo una configuración *posible* cualquier correspondencia entre el conjunto *T* y el conjunto *N* que cumpla las condiciones que acabamos de mencionar, sin importar las asignaciones de tareas a nodos, lo cual implica, lógicamente, que el espacio de configuraciones posibles abarca todas las asignaciones de tareas a nodos. Para simplificar el documento, y dado que en la mayor parte del mismo estaremos hablando de configuraciones posibles, nos referiremos a las mismas simplemente como configuraciones. Con esto en mente, definiremos como *configuración correcta* aquella configuración donde:

1. Cada tarea esté asignada a un nodo y sólo a un nodo.
2. La suma de los costes de las tareas asignadas a cada nodo no supere la capacidad de dicho nodo.

Este planteamiento se asemeja al problema de la mochila (o *Knapsack Problem*) [13], un problema de optimización combinatoria que ha inspirado algunas partes de este proyecto. El objetivo de este sistema compuesto de nodos y tareas es proporcionar un servicio al usuario y, para ello, debe ser posible asignar todas las tareas. Además, en algunos casos será deseable que la configuración sea lo *mejor* posible, teniendo en cuenta unos criterios de evaluación (hablaremos de los criterios de evaluación de configuraciones más adelante, en la sección III).

Dicho esto, lo que queremos es poder guardar, definir y utilizar las configuraciones, por lo que crearemos las siguientes clases:

**Clase *Node***: Esta clase se utilizará para definir cada uno de los nodos, y tendrá los siguientes atributos:

- *id*: número identificativo del nodo.
- *tasks*: lista con los números identificativos de las tareas que tiene asignadas ese nodo.
- *total_node_occupation*: cantidad de recursos ya asignados a tareas.

Y los siguientes métodos:

- El constructor, que tiene como parámetro de entrada el número identificativo del nodo.
- *rec_task* y *del_task*: métodos mediante los que se asignará y desasignará una tarea al nodo.

Además, la capacidad de cada nodo, así como el coste de cada tarea y su número identificativo, se guardarán en variables globales, para que puedan ser accedidas en cualquier momento.

**Clase *Config***: Esta clase se utilizará para definir una posible configuración del sistema (que no tiene que ser necesariamente una configuración correcta), y tendrá los siguientes atributos:

- *id*: número identificativo de la configuración.
- *nodes*: lista con los nodos del sistema. Cada elemento de esta lista será un objeto de la clase *Node*.
- *n_tasks_assigned*: cantidad de tareas que se han asignado en total.
- *tasks_assigned*: lista con los números identificativos de todas las tareas asignadas, sin importar a qué nodo hayan

sido asignadas.

Y los siguientes métodos:

- El constructor, que tiene como parámetro de entrada el número identificativo de la configuración y el número de nodos que va a tener el sistema. Al inicializarse, se encargará de inicializar tantos nodos como este último.
- *assign_task* y *delete_task*: métodos mediante los que se asignará y desasignará una tarea determinada a/de un nodo determinado.

## III. Criterio de Evaluación de Configuraciones

Una vez explicado el modelo del sistema sobre el que vamos a trabajar, es necesario hablar sobre el criterio de evaluación de las configuraciones. Si bien en el apartado anterior hemos especificado qué criterios debe cumplir una configuración para que la consideremos como *correcta*, es lógico pensar que hay muchas situaciones en las que existirán varias configuraciones correctas. Es por ello que necesitamos definir un criterio para poder comparar estas configuraciones correctas, y decidir así cuales son mejores o peores; además, esto también servirá a algunas técnicas de búsqueda para guiar la búsqueda. Ese criterio es el *criterio de evaluación de configuraciones*.

Antes de empezar, debemos aclarar que, si bien nosotros hemos escogido un criterio concreto, la adecuación de éste depende de la función que vaya a tener el sistema y no hay uno que vaya a ser siempre mejor que otro. Volviendo al ejemplo del problema de la mochila, el criterio de evaluación de configuraciones no será el mismo si queremos guardar primero los objetos pesados o si por el contrario priorizamos primero los objetos ligeros.

El estudio de los diferentes criterios que se podrían aplicar queda fuera del ámbito de este trabajo. Aplicaremos el criterio descrito más abajo en todas las técnicas con el fin de que los resultados obtenidos en las diferentes técnicas de búsqueda sean comparables. Sin embargo, a nivel de implementación de las diferentes técnicas, hemos intentado aislar la definición del criterio del resto del código todo lo posible, de forma que dicha definición pueda cambiarse fácilmente por otra sin tener que cambiar el conjunto de la implementación en exceso. De todas formas, esto no siempre es posible, y será precisamente uno de los aspectos que consideraremos para valorar cualitativamente cada técnica. Centrándonos ya en el criterio de evaluación de configuraciones que hemos utilizado en este proyecto, se trata del siguiente.

El criterio que hemos escogido considera mejores aquellas configuraciones que utilicen el menor número de nodos y, a su vez, la totalidad o, en su defecto, la mayor cantidad posible de recursos de esos nodos activos. Por ejemplo, en el caso de tener dos nodos, uno con una capacidad igual a 2 y otro con una capacidad igual a 3, y dos tareas con costes iguales a 1, consideraremos que la mejor solución es que el nodo con capacidad igual a 2 tenga asignadas ambas tareas, pues de esa forma utilizamos sólo un nodo, y reducimos al máximo la cantidad de recursos sin utilizar de los nodos activos. De esta forma, favorecemos el hecho de que el sistema pueda recibir nuevas tareas con costes altos, pues la probabilidad de que haya un nodo con la capacidad disponible suficiente para ejecutarlas es mayor que si las tareas estuvieran más repartidas, situación en la que a lo mejor ese nodo tendría una o más tareas ya asignadas, y su capacidad restante no sería suficiente para alojar la nueva tarea. Este criterio, como hemos dicho, no es en todas las circunstancias peor o mejor que otro que, por ejemplo, considere mejores aquellas configuraciones que repartan las tareas entre los nodos lo máximo posible, pero entendemos que es un criterio aplicable a muchas situaciones reales. La elección del criterio de evaluación de configuraciones depende enteramente del problema a resolver, por lo que el contexto en cada caso es primordial en esa elección.

Es necesario decir que, aunque acabamos de definir cómo decidiremos en este trabajo cuál es la *mejor* configuración en cada caso, a lo largo de este proyecto utilizaremos el término *ineficiencia* para medir los resultados. El motivo es que, al calcular los recursos sin asignar de los nodos activos (es decir, los nodos que tienen alguna tarea asignada) el resultado será un valor que consideraremos peor cuanto más grande sea (de hecho, el resultado óptimo sería de 0). Si dijéramos que este valor es la eficiencia de la configuración, resultaría contradictorio, pues estaríamos indicando que cuanto más eficiente sea una configuración, peor será. Es por ello que hemos decidido utilizar el término antónimo, la ineficiencia, para referirnos a este valor a calcular. Dicho esto, calculamos la ineficiencia total ($I_{total}$) con la siguiente fórmula, donde definiremos la ineficiencia parcial ($I_{parcial}$) como la suma de los recursos sin utilizar de todos los nodos activos, y $N_{nodos}$ como el número de nodos activos:

$$I_{total} = \begin{cases} N_{nodos} - 1, & I_{parcial} = 0 \\ I_{parcial} \times N_{nodos}, & I_{parcial} > 0 \end{cases}$$

Recordemos que los dos valores a observar según nuestro criterio son el número de nodos activos y la cantidad de recursos sin utilizar de los nodos que tengan al menos una tarea asignada. Es por ello que multiplicamos ambos valores para calcular la ineficiencia total, a no ser que la suma de los recursos sin utilizar en los nodos activos sea igual a cero. En ese caso, el único valor a observar es el número de nodos activos, valor al que restaremos 1 para evitar errores de comparación. Por ejemplo, si tuviéramos una configuración donde sólo hay una tarea que asignar de coste 0.5, y dos nodos con capacidades de 1 y 0.5, la mejor solución debería ser asignar la tarea al nodo con capacidad 0.5, pero si no restamos 1 como hemos dicho, el valor de ineficiencia sería menor asignando la tarea al nodo con capacidad 1.

Por último, cabe aclarar que, a partir de ahora y con el fin de acortar la escritura, llamaremos solución a cualquier configuración correcta. En cualquier situación en la que una de las técnicas de búsqueda no encuentre una configuración de este tipo, se considerará que no ha sido capaz de encontrar una solución, independientemente de lo cerca o lejos que esté la configuración de lograr dichas exigencias o de la calidad de la misma.

Una vez definido el sistema, el objetivo, y el criterio con el que evaluaremos las configuraciones, pasaremos a explicar las técnicas de búsqueda que hemos utilizado.

## IV. Implementación

En esta sección describimos las técnicas de búsqueda que hemos considerado. Más concretamente, para cada una de ellas haremos una introducción teórica, donde explicaremos en qué consiste y sus parámetros de funcionamiento, para luego detallar cómo hemos implementado o adaptado la técnica a nuestro problema, y terminar con una pequeña explicación de lo que esperamos de ella.

### *IV-A. Búsquedas Basadas en Backtracking*

El algoritmo de *Backtracking* es tan simple como efectivo, y consiste en buscar sistemáticamente cada solución posible de un problema probando todas las opciones posibles. Al tomar cada decisión, el algoritmo va eligiendo una opción sin priorizar unas u otras, hasta que llega un punto en el que o bien se ha encontrado una solución, o bien no se puede avanzar más. Llegado a este punto, si se trata de una solución el algoritmo se detendrá, pero si no es el caso, eliminará la última decisión tomada, cambiando la opción elegida (es decir, vuelve atrás). Un ejemplo práctico de aplicación sería un laberinto con varios caminos posibles [15]. Aplicando este algoritmo, se recorrería uno por uno cada uno de estos caminos, regresando a la última bifurcación si no se encuentra una salida para probar el siguiente camino, hasta encontrar la salida. De esta forma se va generando un árbol de soluciones, parecido al que vemos en la figura 2, y cuyo funcionamiento explicaremos detalladamente más adelante.

En base a esta definición se pueden plantear muchas variantes, como por ejemplo que el algoritmo recorra todo el árbol (es decir, recorra todos los caminos posibles) para encontrar todas las soluciones posibles y poder compararlas. En nuestro caso, hemos decidido utilizar dos algoritmos basados en este principio, los cuales explicaremos a continuación.

A nivel de implementación, y como ya hemos comentado anteriormente, podemos decir que ésta ha sido realizada por nosotros en su totalidad, sin utilizar librerías o elementos externos. Nos hemos apoyado en el concepto de la recursividad para llevar a cabo la implementación de la mayoría de técnicas, facilitando el control del bucle y la propia escritura en gran medida. En las siguientes secciones, donde hablaremos de forma más detallada de cada técnica de búsqueda, añadiremos también un breve comentario sobre las particularidades en la implementación de cada una.

*IV-A1. Backtracking Básico:* Para empezar, tenemos la variante más sencilla de *Backtracking*, que llamaremos *Backtracking* Básico. Al ser la primera y, como su propio nombre indica, la más básica, y con el objetivo de favorecer la claridad en futuras explicaciones, cuando nos refiramos al algoritmo de *Backtracking* estaremos hablando de esta variante, la que va asignando tareas a los nodos ordenadamente hasta encontrar una solución. Para explicarlo mejor, podemos observar de nuevo la figura 2: en esta figura vemos como, en la primera bifurcación (estado 1), si gira a la izquierda se encontrará con otra bifurcación (estado 2), pero ninguna de las direcciones posibles a partir de esta bifurcación lleva a la salida, por lo que elegirá una (estado 3) y al ver que no hay salida volverá atrás (estado 2) para probar la otra (estado 4). Como

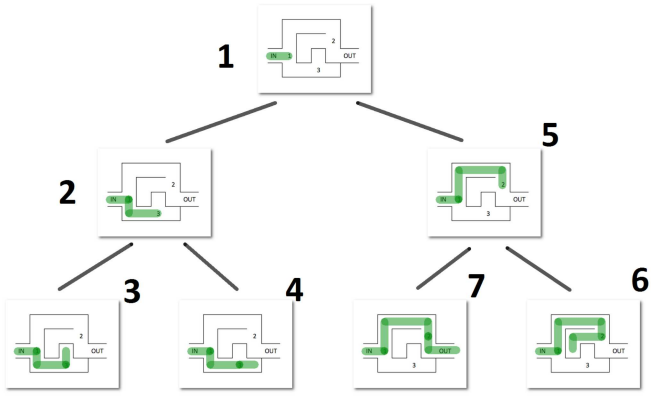


Figura 2. Árbol de soluciones formado por Backtracking [15].

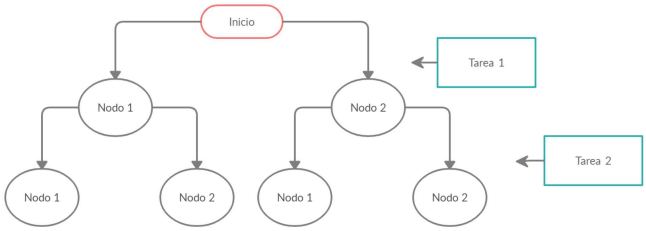


Figura 3. Diagrama del árbol de soluciones formado por Backtracking.

ésta tampoco permite seguir adelante, volverá a la bifurcación anterior (estado 1) y tomará el otro camino, llegando a una nueva bifurcación (estado 5) en la que, finalmente, y siguiendo el mismo proceso, llegará a la salida (estado 7). Es importante aclarar que, como no diferencia entre un camino hacia la salida u otro, no dará prioridad a un camino o solución por encima del resto. Sin embargo, es posible obtener la solución óptima (por ejemplo, la salida más rápida del laberinto) si hacemos que el algoritmo no se detenga hasta haber visitado todas las soluciones, permitiendo compararlas para saber cual es la mejor.

Aplicándolo a nuestro caso particular, las decisiones a tomar serán las asignaciones de tareas a nodos. Por lo tanto, cada nivel del árbol corresponderá a una tarea, y cada rama derivada de la asignación de esa tarea corresponderá a un nodo elegido para alojar y ejecutar dicha tarea. En la figura 3 podemos ver un diagrama que ilustra este proceso con dos nodos y dos tareas.

Como ya hemos dicho, esta variante de Backtracking puede utilizarse tanto para encontrar una solución cualquiera, haciendo que se detenga al encontrar una, como para encontrar la solución óptima, de acuerdo con algún criterio. Con todo esto, este algoritmo tiene, a priori, una ventaja, y es que, si existe alguna solución, tarde o temprano la encontrará, y lo mismo pasa si queremos encontrar la solución óptima. Sin embargo, por otro lado, las características del mismo también nos hacen pensar que los tiempos de computación pueden llegar a ser extremadamente grandes en algunos casos, pues deberá recorrer todo el espacio de posibilidades, lo que nos hace pensar que éste no es la mejor opción para sistemas con muchos nodos y tareas.

En cuanto a la implementación de esta variante, es probable que haya sido la más sencilla de todas, dado que el proceso a realizar en cada iteración del bucle es prácticamente idéntico, y no hay ninguna heurística ni ningún proceso complejo que aplicar, por lo que no hay mucho que comentar en ese sentido.

*IV-A2. Ramificación y Poda:* Como ya hemos comentado, los algoritmos basados en el *Backtracking* generan un árbol de soluciones donde cada rama es una posible solución. Lógicamente, no todas las ramas conducen a una solución, y no todas las ramas conducen a la solución óptima. Es aquí donde entra en juego la segunda variante que trataremos, la de Ramificación y Poda. Aunque su comportamiento general es muy parecido al del *Backtracking* Básico, la diferencia respecto a éste consiste en que, antes de escoger la siguiente asignación, se realiza una serie de comprobaciones para prever de antemano si es posible que esta rama pueda dar lugar a una solución más adelante; si el resultado es que no, se desecha esa rama (se poda) y, por consiguiente, todas sus posibles ramificaciones. De esta descripción se infiere que, al igual que pasaba con la primera variante, ésta también puede utilizarse tanto para encontrar una solución cualquiera como para encontrar la solución óptima.

En la figura 4 podemos observar de forma gráfica el proceso que acabamos de describir. En este ejemplo el objetivo es encontrar un subconjunto del conjunto total que sume 10, y cada nivel consiste en decidir si escoge o no cada número del conjunto total. Empieza por el 1, y no lo elige, por lo que el valor máximo posible de la suma pasa a ser 14. A continuación hace lo mismo con el 2 y el 3, pero, una vez decide no sumar el 4, el valor máximo posible de la suma es menor que 10, por lo que será imposible encontrar una solución siguiendo esa rama. Por ende, decide podarla.

En nuestro caso particular, hemos establecido que una rama será podada si se da alguno de los siguientes supuestos:

- La suma de los recursos sin asignar restantes de los nodos es menor a la suma de los costes de las tareas sin asignar. Por simple lógica, si se da esta situación no es posible encontrar una solución por esta rama.
- El valor del coste más grande entre las tareas sin asignar es superior a los recursos individuales disponibles en cualquiera de los nodos. De ser así, esta tarea no podrá ser asignada a ningún nodo.

La principal ventaja teórica de esta variante es que evita tomar caminos innecesarios, por lo tanto se disminuye el tiempo necesario para encontrar una solución respecto a la variante de *Backtracking* Básico. Una posible pega, sin embargo, es que en problemas de menor tamaño puede que el tiempo necesario para realizar las comprobaciones necesarias para decidir si se poda o no una rama afecte significativamente al rendimiento, tardando más que la variante básica.

Respecto a la implementación de esta variante, lo primero es decir que, como es lógico, tanto éste como los demás códigos de los algoritmos basados en *Backtracking* (como por ejemplo el del algoritmo voraz que veremos en el siguiente apartado) están basados en el de la variante básica de la que hemos hablado antes. En este caso, la principal diferencia respecto a la variante básica es la introducción de una comprobación heurística antes de realizar cada iteración, para comprobar así si se poda o no la rama.

### *IV-B. Algoritmo Voraz*

Se define como algoritmo voraz cualquier algoritmo que sigue una heurística, es decir, una estrategia concreta, para intentar elegir la mejor opción en cada paso para resolver un problema [12]. En ese sentido, se podría considerar que es el siguiente paso después del algoritmo de Ramificación y Poda. Si bien éste último normalmente elimina sólo algunas ramas en cada decisión, el algoritmo voraz directamente ignora todas las ramas salvo una, haciendo un solo camino sin volver atrás, de ahí su nombre.

El funcionamiento es el siguiente. Cada vez que se deba tomar una decisión (por ejemplo, y volviendo al ejemplo del laberinto, cada vez que se deba elegir una dirección en una bifurcación), el algoritmo voraz analizará las posibles ramas para detectar cuál es más probable que desemboque en la solución óptima. Una vez hecho esto, el algoritmo seguirá ese camino, repitiendo el proceso para tomar cada decisión hasta alcanzar el final de la rama. Una vez alcanzado el final, haya encontrado una solución o no, dará el proceso por terminado. En la figura 5 podemos ver un ejemplo de esta técnica con el siguiente problema: determinar el número mínimo de monedas para dar el cambio al pagar en una tienda, siendo la cantidad a dar de 36 céntimos y contando con un número ilimitado de monedas de 20, 10, 5 y 1 céntimo. En este ejemplo podemos ver que el algoritmo elige, en cada paso, la moneda más grande posible, es decir, elige el óptimo local al tomar cada decisión.

Esto ya implica una ventaja respecto a los algoritmos descritos hasta el momento, y es que se trata de un algoritmo que garantiza un tiempo de ejecución menor a los demás. De hecho, este tipo de algoritmos suelen asegurar tiempos de ejecución acotados, y por lo tanto cumplen uno de los requisitos para considerarse aptos para aplicaciones de tiempo real. Por poner algunos ejemplos, en [20] los autores utilizan un algoritmo voraz para detectar en tiempo real correspondencias en imágenes, mientras que en [10] los autores utilizan otro algoritmo voraz para implementar un planificador *online* de tareas. Sin embargo, aunque un algoritmo de este tipo pueda ejecutarse en un tiempo limitado, es posible que la solución encontrada esté lejos de la óptima, e incluso existe la posibilidad de que no encuentre ninguna solución aún cuando ésta sí exista.

Como último apunte respecto a esta técnica del Algoritmo Voraz, la implementación de la misma ha implicado, además de introducir una comprobación previa a cada paso, cambiar las condiciones necesarias para terminar el proceso, dado que, como ya hemos explicado, el Algoritmo Voraz no vuelve hacia atrás, sino que realiza un solo camino hasta el final, encuentre o no una solución.

### *IV-C. Algoritmos basados en Metaheurística - Tabu Search*

Observando las descripciones de los algoritmos anteriores, podemos intuir que, probablemente, existe una relación entre la calidad de los resultados obtenidos y el tiempo de computación: cuanto más tiempo tenga el algoritmo para operar, más

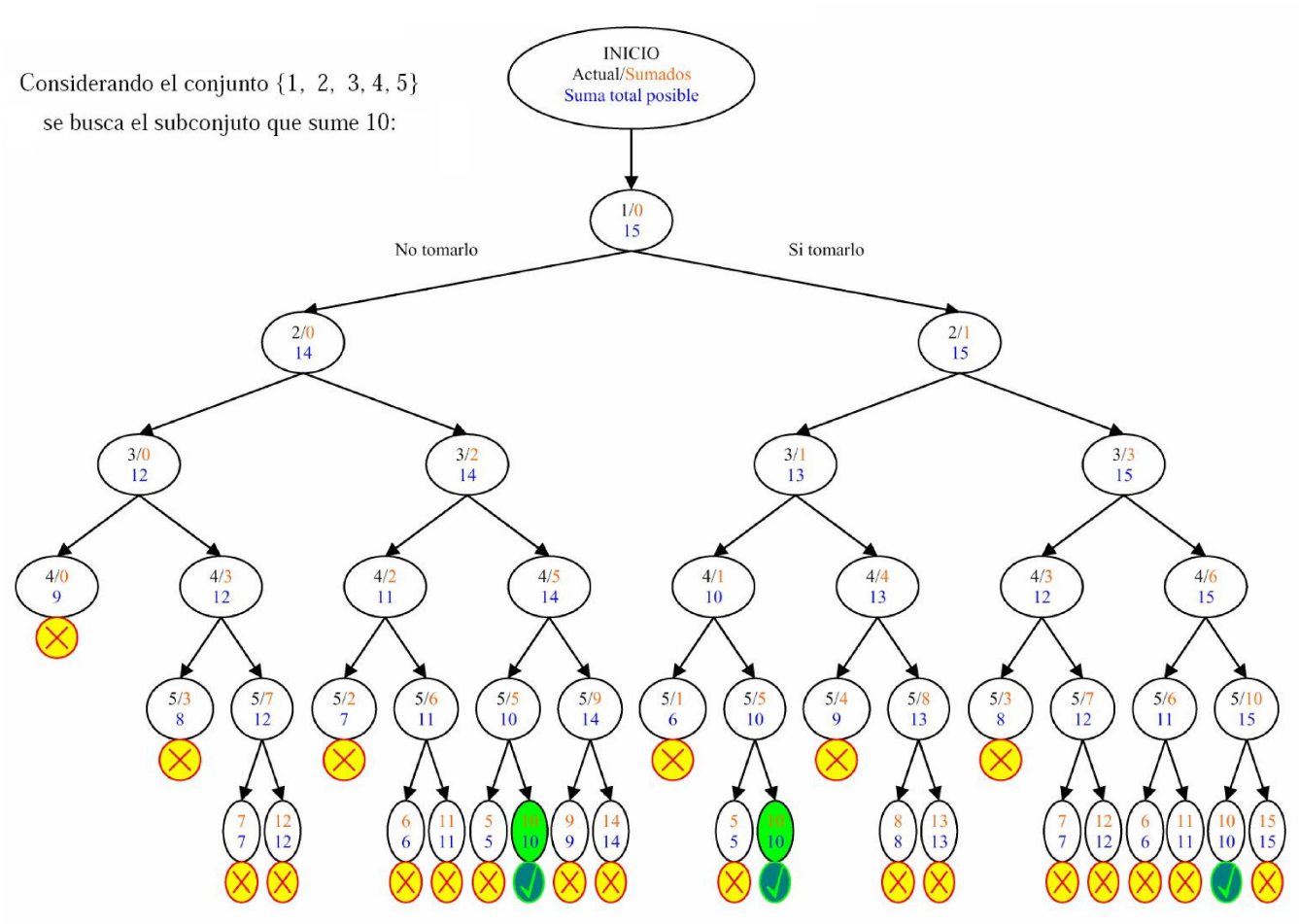


Figura 4. Ejemplo de proceso de Ramificación y Poda [3].

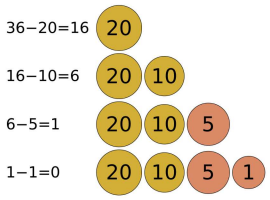


Figura 5. Ejemplo del Algoritmo Voraz [1].

probable es que la solución encontrada sea mejor. El problema es que, cuando el número de nodos y/o tareas aumenta, la cantidad de configuraciones posibles aumenta exponencialmente, reduciendo con ello la probabilidad de encontrar una solución mejor en un tiempo razonable. Es por ello que existe un gran interés en definir algoritmos que proporcionen buenos resultados en el menor tiempo posible, y ese es precisamente el objetivo de los algoritmos basados en metaheurística.

Se denominan como *algoritmos metaheurísticos* aquellos algoritmos diseñados para encontrar soluciones tan buenas como sea posible, que no tienen por qué ser las óptimas, en un tiempo razonable, utilizando procesos con un considerable grado de complejidad [8]. La clave es que estos algoritmos buscan el equilibrio entre el tiempo y el resultado, a diferencia de otros algoritmos, como puede ser el *Backtracking*, que permite encontrar la solución óptima, pero a costa de un tiempo de computación previsiblemente enorme. Dentro de los algoritmos metaheurísticos, nosotros hemos decidido utilizar el algoritmo *Tabu Search*.

Creado durante la segunda mitad de la década de 1980 [14], el algoritmo *Tabu Search* puede definirse como una técnica iterativa que explora un conjunto de soluciones de un problema, pasando de una solución a otra. Definimos como técnica iterativa aquella cuyo comportamiento consiste en la repetición de una rutina, y que habitualmente aumenta el valor de un contador con cada repetición hasta que este contador alcanza un cierto valor, momento en el que da por finalizada la ejecución. En la figura 6 podemos ver la estructura básica de este tipo de técnica.

A continuación, explicaremos el funcionamiento de este algoritmo por pasos:

1. Obtener una solución inicial. Ésta puede obtenerse de diferentes maneras, y nosotros hemos decidido obtenerla mediante alguna de las otras técnicas. Más adelante, en el apartado de evaluación, hablaremos de esta primera solución, y explicaremos qué técnica hemos utilizado nosotros para obtenerla y por qué.
2. Partiendo de la última solución obtenida (si se trata

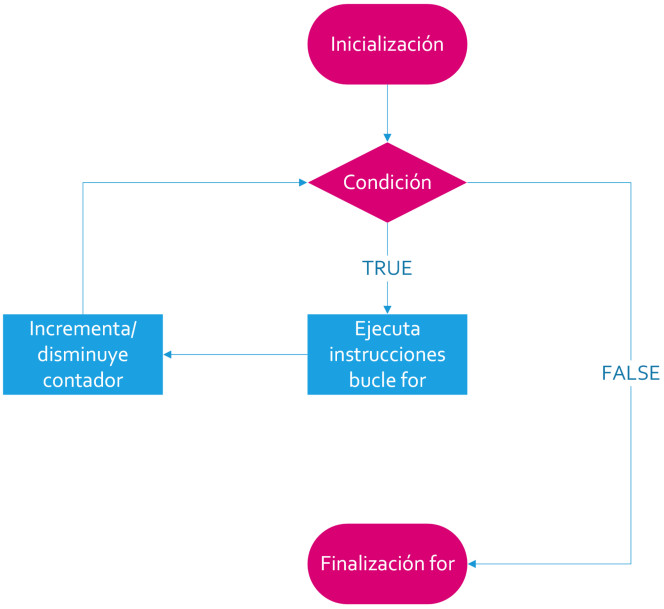


Figura 6. Estructura de una técnica iterativa [22].

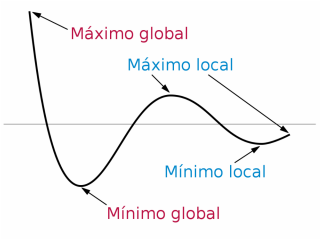


Figura 7. Ejemplo de un mínimo local [2].

del inicio de la ejecución, estaremos hablando de la solución inicial), crear una lista de *movimientos* candidatos, considerando como movimiento una acción o conjunto de acciones que, de producirse, daría lugar a una nueva solución diferente a la actual que llamamos vecino o *neighbour*. Para ejemplificar esta definición, consideremos una vez más el problema del laberinto. Si hemos encontrado ya un camino que lleva a la salida, un movimiento sería volver atrás y encontrar un camino distinto que nos lleve a la misma salida. A esta lista de candidatos se la conoce como vecindario o *neighbourhood*. Este proceso puede cambiar según la aplicación que se le dé a este algoritmo, y más adelante explicaremos cómo lo hemos llevado a cabo nosotros. Daremos más detalles de este paso más abajo.

3. Elegir el mejor candidato. También daremos más detalles de este paso más abajo.
4. Comprobar si se ha cumplido alguna de las condiciones de parada. En nuestro caso, estableceremos un máximo de iteraciones del algoritmo, aunque éste también se detendrá en el caso de que la solución sea óptima (es decir, el valor de ineficiencia sea 0). Si se cumple alguna de las condiciones, el algoritmo termina su ejecución, y devolverá como resultado la mejor solución encontrada durante su ejecución. En caso de no cumplirse ninguna de las condiciones de parada, actualiza las *Tabu Restrictions*, que definiremos más abajo, y regresa al paso 2.

Como hemos introducido anteriormente, en el paso 2 creamos la lista de candidatos (*neighbourhood*). Al tratarse de un proceso de asignación de tareas lo que haremos será, partiendo de la configuración actual, quitar una tarea del nodo al que estuviera asignada para, a continuación, intentar asignarla a algún otro nodo. De esta forma, la lista de candidatos será cualquier cambio de asignación de esa única tarea a otro nodo. Si, por ejemplo, tenemos tres nodos, $d1$, $d2$ y $d3$, y dos tareas, $s1$ y $s2$, asignadas a $d1$, la lista de candidatos contendrá las configuraciones resultantes de asignar $s1$ a $d2$, $s1$ a $d3$, $s2$ a $d2$ y $s2$ a $d3$. Sin embargo, no consideramos el cambio de asignación simultáneo de ambas tareas, dado que sólo cambiamos una asignación por iteración. Si bien se podría modificar el algoritmo para realizar varios cambios de asignación por iteración, no tiene sentido, pues aumentando el número de iteraciones se obtiene el mismo efecto.

En este algoritmo, la elección del mejor candidato (paso 3) es un paso crucial y, como ya hemos dicho, ésta merece una explicación propia. Sin embargo, antes debemos definir dos conceptos propios de este algoritmo que pueden afectar a la elección del mejor candidato, y que sirven principalmente para que el algoritmo no quede atrapado en un mínimo local. Un mínimo local es aquel valor de una función que es menor que sus valores cercanos, pero que no es el menor de todos los valores de la función (ver figura 7). En nuestro caso, hablaríamos de aquella solución que es la mejor dentro del conjunto de soluciones cercanas, pero que no es la mejor solución posible para el problema. Sin los mecanismos que definiremos a continuación, el algoritmo *Tabu Search* puede quedar atrapado fácilmente en mínimos locales, pues volvería una y otra vez al mínimo local, ya que éste da lugar a una mejor solución que todas las soluciones cercanas. Es por ello que este algoritmo utiliza los siguientes mecanismos:

- *Tabu Restrictions*: Éste es el mecanismo central de este algoritmo, y al que debe su nombre. Se trata de una lista en la que se almacenan las últimas asignaciones de tareas a nodos que se hayan eliminado en las últimas iteraciones del algoritmo para dar lugar a soluciones nuevas, y que impide al algoritmo volver a realizar esas asignaciones de la lista (salvo en el caso en que se cumpla el *Aspiration Criteria*, que veremos a continuación) durante un cierto número de iteraciones, para así no volver a las soluciones por las que ya ha pasado. Por ejemplo, si en una iteración del algoritmo eliminamos la asignación de la tarea $s$ al nodo $d1$, y en su lugar asignamos esa misma tarea al nodo $d2$ para obtener una solución distinta, la asignación de la tarea $s$ al nodo $d1$ se guardará en la lista Tabú, y no podrá ser asignada de nuevo mientras esté en dicha lista. De esta forma, aunque se encuentre en un mínimo local, el algoritmo realizará un movimiento para llegar a otras soluciones, aunque sean peores, y no podrá rehacer esa asignación mientras esté incluido en la lista. Hemos implementada esta lista como una cola circular de tamaño $n$, de tal forma que las asignaciones realizadas más de $n$ iteraciones atrás se desasignan automáticamente al

actualizar la lista, proceso en el que se guarda la última asignación Tabú y se deshace la asignación Tabú más antigua. Siguiendo con el ejemplo anterior, la asignación de la tarea T al nodo N2 no podrá deshacerse durante $n$ iteraciones.

- *Aspiration Criteria*: Este criterio permite que el algoritmo ignore la lista de Tabú en algunas situaciones. Concretamente, lo más habitual es que el algoritmo ignore dicha prohibición si de esta forma obtiene una solución mejor que cualquier otra encontrada hasta la fecha. Ese será precisamente nuestro caso particular, pues ya hemos comentado que, para nosotros, la lista Tabú representa asignaciones eliminadas previamente que no se pueden rehacer. Sin embargo, es posible que, pasadas unas cuantas iteraciones, rehacer alguna de esas asignaciones dé lugar a una solución que no haya sido explorada previamente, y que a su vez sea mejor que cualquier otra. Mediante el mecanismo de *Aspiration Criteria* el algoritmo podrá ignorar las asignaciones Tabú, si con ello obtiene una configuración cuya ineficiencia es mejor a cualquier otra encontrada hasta la fecha. De esta forma, el mecanismo de la lista Tabú no impedirá alcanzar una mejor solución.

Una vez definidos estos conceptos, podemos explicar los pasos a seguir para seleccionar el mejor movimiento candidato (paso 3 de la descripción del algoritmo proporcionada al principio de esta sección IV-C). Es importante tener en cuenta que, al hablar de un movimiento candidato, se trata de dos acciones, eliminar una asignación de una tarea a un nodo, y asignar esa misma tarea a otro nodo distinto:

3.1 Evaluar cada candidato dentro de la lista de movimientos candidatos y seleccionar el mejor de dicha lista, sea mejor o no que la solución actual, según nuestro criterio de evaluación. De esta forma, se evita estancarse en un mínimo local.

3.2 Comprobar si el movimiento candidato está dentro de la lista Tabú. En nuestro caso particular, se deberá comprobar si la asignación que hace falta eliminar para llevar a cabo el movimiento está dentro de la lista Tabú. Si es así, pasamos al siguiente paso para determinar si, a pesar de ser una asignación incluida dentro de la lista, debemos tenerla en cuenta; en caso contrario, pasamos al paso 3.4.

3.3 Comprobar si el movimiento satisface el *Aspiration Criteria*. En nuestro caso, comprobar si el movimiento daría lugar a una solución mejor que cualquier otra encontrada hasta la fecha. De ser así, pasamos al siguiente paso; en caso contrario, volvemos al paso 3.1 y evaluamos el siguiente candidato.

3.4 Realizar el movimiento candidato.

Una vez explicado el funcionamiento de este algoritmo, podemos decir que, a priori, debería ser un compromiso entre los algoritmos que buscan una solución cualquiera y los que buscan la solución óptima, pues no busca la mejor solución, sino la mejor que pueda encontrar en un tiempo determinado. Además, aunque se podría pensar que limitando el tiempo de operación de las técnicas *Backtracking* Básico y Ramificación y Poda se obtendría el mismo resultado, *Tabu Search* hace una búsqueda ordenada, lo que aumenta las posibilidades de obtener una mejor solución en el mismo tiempo, aunque no sabemos hasta que punto puede afectar la elección de la solución inicial al resultado. Aun así, podemos esperar que ésta sea el punto intermedio que ya hemos comentado, tanto en términos de tiempo como de calidad de los resultados.

Para terminar la explicación respecto a esta técnica de búsqueda, hay que decir que es sin duda la técnica cuya implementación ha resultado ser más compleja, en parte debido a que su funcionamiento ya es de por sí más complejo que el de las demás, y en parte porque ese mismo funcionamiento es bastante diferente al de los demás. Es por ello que, aunque el algoritmo de *Backtracking* ha sido también la base sobre la que hemos diseñado el resto, ha sido necesario introducir varias funciones por las que va pasando el proceso para realizar los distintos pasos que hemos explicado hace un momento. Siguiendo esta metodología hemos podido separar bastante cada paso, aislando así los errores y facilitando tanto la lectura como la comprensión del código, lo cual es clave, sobre todo al implementar algoritmos de una cierta complejidad.

### *IV-D. Búsqueda Basada en SMT Solver*

Por último, también hemos probado los *Satisfiability Modulo Theories Solvers*, o *SMT Solvers*. Para definir esta técnica, primero debemos explicar brevemente su origen. Dentro de la computación, la lógica proposicional juega un papel básico, y es por ello que desde hace tiempo ha habido un gran interés en crear herramientas que permitan resolver problemas relacionados con dicha lógica en los que es necesario entender las relaciones de causa y efecto entre los distintos elementos que conforman un conjunto de fórmulas lógicas. Como resultado de este interés han surgido varias herramientas que, utilizando fórmulas lógicas como *input*, devuelven un valor booleano indicando si satisfacer la fórmula en cuestión es posible o no. Estas herramientas reciben el nombre de *Boolean Satisfiability Problem Solver* (*SAT Solver*), y suelen ser herramientas automáticas y bastante eficientes. Sin embargo, la mayoría de sistemas suelen ser diseñados y modelados a un nivel más alto que el booleano, por lo que no siempre pueden utilizarse estas herramientas. Con esto en mente, se empezaron a crear nuevas herramientas con la intención de que puedan interpretar fórmulas con un nivel de abstracción mayor, pero manteniendo la eficiencia de los *SAT Solvers*. Estas herramientas son los *SMT Solvers* [7].

Como definición, el término SMT se refiere a los problemas donde se debe determinar si es posible cumplir una serie de fórmulas lógicas de primer orden [11], aquellas que implican relaciones y operaciones entre los objetos que las forman, como por ejemplo $x < y$. Para entender mejor qué son los *SMT Solvers*, supongamos que tenemos una fórmula que queremos verificar, como por ejemplo una fórmula matemática del tipo $x + 3 - z < y$, en la que las variables pueden tomar varios valores distintos. Utilizando esta fórmula como base, esta herramienta nos permite saber rápidamente si existe alguna combinación de valores que hagan cierta dicha fórmula.

Concretamente, nosotros utilizaremos el *Z3 Theorem Prover*, un *solver* desarrollado por Microsoft Research que se

utilizaba inicialmente para detectar vulnerabilidades en los sistemas de seguridad, y que incluso ha ganado premios debido a su eficacia e influencia en el sector de la ciberseguridad. Además, este *solver* es de código abierto, y puede utilizarse en diferentes entornos y con varios lenguajes de programación. Este *solver*, al igual que muchos otros, utiliza también unos comandos propios que el usuario debe introducir para utilizarlo, definidos dentro del lenguaje SMT-LIB, creado en 2003 [18]. Este lenguaje define la estructura léxica, como se escriben las fórmulas lógicas, los comandos, y las diversas lógicas subyacentes del funcionamiento de los mismos. Dado que estudiar los detalles del funcionamiento de los *solvers* no es uno de los objetivos de este trabajo, no entraremos en muchos detalles, pero, en términos prácticos, el uso de un *solver* consiste en definir las fórmulas lógicas que deben cumplirse, para que sea el *solver* el que determine la solución. Por poner un ejemplo sencillo, es posible definir un sistema de ecuaciones con varias variables y que el *solver* analice si existe alguna combinación que satisfaga el sistema, pudiendo definir también el rango de valores que pueden tener las variables si queremos. Si definimos un sistema de ecuaciones con dos ecuaciones como $x < y$ y $x < z$, nos indicará que existe al menos una solución; por el contrario, si definimos las ecuaciones $x < y$ y $x > y$, nos indicará que no hay ninguna solución posible.

En el caso concreto del Z3, éste no sólo permite saber si existe una posible solución para el problema, sino que puede generar una solución del sistema definido que satisfaga las restricciones establecidas, entendiendo como solución los valores a asignar a cada variable para que se cumpla la solución encontrada. Por supuesto, y aplicándolo ya a nuestro caso particular, dentro de este abanico de soluciones habrá algunas mejores y peores, y el *solver*, por defecto, devuelve una solución cualquiera dentro del conjunto de soluciones. Sin embargo, existe la posibilidad de "obligar" al *solver* a que devuelva una solución que no sólo satisfaga las restricciones, sino que optimice uno o más valores. De esta forma, podemos conseguir, por ejemplo, que el *solver* nos devuelva la solución que minimice o maximice un cierto valor. Esta variante del *solver* está incluida dentro de Z3, y recibe el nombre de Vz [9]. Gracias a este optimizador, podemos indicarle al *solver* que devuelva la solución que dé como resultado un menor valor de ineficiencia, consiguiendo así el mejor resultado.

Aún así, y con el fin de explorar el mayor número de opciones posibles, no realizaremos las pruebas sólo con el optimizador, sino que también haremos pruebas con el *solver* sin optimizar, para poder observar los resultados del mismo y compararlos con los demás. Tanto si aplicamos el optimizador como el *solver* simple, las reglas que definiremos serán las mismas. A continuación enumeraremos estas reglas:

1. Cada tarea debe estar asignada a un solo nodo.
2. Cada nodo tendrá una variable asociada, la ocupación, que será el resultado de la suma de los costes de todas las tareas asignadas al mismo.
3. La ocupación de cada nodo debe ser menor o igual a su capacidad.
4. La ineficiencia asociada a cada nodo es la diferencia entre su capacidad y su ocupación.
5. El valor total de ineficiencia es la suma de la ineficiencia de aquellos nodos que tengan alguna tarea asignada, multiplicado por el número de nodos utilizados. En caso de que la suma de las ineficiencias de los nodos utilizados sea 0, el valor de la ineficiencia final será igual al número de nodos utilizados - 1. En definitiva, aplicamos el mismo cálculo de ineficiencia que hemos utilizado con todas las demás técnicas.

Aplicando estas reglas y definiendo las variables que intervendrán en el proceso (número de nodos, número de tareas, costes, etc.) creamos el modelo de sistema en forma de expresiones sobre las que trabajarán los *solvers*.

Esta herramienta es la más novedosa de las tres, y de la que menos podemos predecir los resultados. Nuestra idea es que debe ser especialmente útil para encontrar soluciones óptimas de problemas complejos, pero para comprobar que es así para nuestro caso concreto debemos determinar si su tiempo de computación es menor que el de las otras técnicas de optimización. Además, al ser más imprevisible su comportamiento, no sabemos hasta que punto ciertos factores, como el tipo de variables utilizadas, afectarán a los resultados. En cuanto a la implementación, ya hemos dicho anteriormente que estas (el *SMT Solver* Simple y Optimizador) son las únicas técnicas donde nos hemos valido de elementos externos (librerías en este caso) para implementarlas. En ese sentido, y si bien es cierto que el diseño de las reglas y la programación de las mismas es nuestro, el proceso de búsqueda per se lo lleva a cabo el software de dicha librería, por lo que nosotros no controlamos totalmente este proceso.

## V. Evaluación

Una vez definidas y explicadas las distintas técnicas de búsqueda, pasaremos a evaluarlas. Para ello, dividimos la evaluación realizada en dos partes:

- **Evaluación cualitativa**: en esta sección listaremos y compararemos cualitativamente las diferentes propiedades de las técnicas de búsqueda.
- **Evaluación cuantitativa**: en esta sección mostraremos y discutiremos los datos obtenidos de varios experimentos que hemos llevado a cabo para medir el rendimiento en diferentes escenarios.

### *V-A. Evaluación Cualitativa*

Como hemos dicho antes, en esta primera sección nos interesa estudiar cualitativamente las características y capacidades de las técnicas. Es por ello que hemos seleccionado una serie de aspectos que consideramos relevantes y que valoraremos en todas las técnicas de búsqueda. Esta comparación nos permitirá discernir cuál de estas técnicas es la más adecuada para cada situación (problemas de mayor o menor tamaño, con tiempo limitado o no, etc.). De esta forma conseguiremos tener una idea de cómo respondería cada técnica a una determinada situación.

Respecto a las técnicas que hemos comparado, es importante comentar que, tanto para cada una de las dos variantes de *Backtracking*, como para el *SMT Solver*, incluimos su versión *básica*, es decir, la que devuelve la primera solución que

encuentra, y su versión *optimizador*, es decir, la que devuelve la solución óptima. Por otro lado, hay que recordar que el *Tabu Search* requiere de una solución inicial, y deberemos decidir también qué técnica utilizaremos para obtener esa primera solución. Esto puede ser relevante para algunos de los aspectos valorados.

Dicho esto, los aspectos que valoraremos, siempre con un SI o un NO, son los siguientes:

1. La capacidad de obtener una solución, sin importar el tiempo que tarde. Con este aspecto nos referimos al hecho de saber si una técnica te asegura obtener una solución, sea la que sea y sin importar el tiempo necesario.
2. La capacidad para obtener la solución óptima, sin importar el tiempo que tarde. Esto nos permitirá distinguir aquellas técnicas capaces de encontrar la mejor solución posible, y no sólo una solución cualquiera.
3. La capacidad para proporcionar una mejor solución en función del tiempo de ejecución del que disponga. Esto implica que, aunque se detuviese la ejecución antes de terminar el proceso completo, la técnica es capaz de proporcionar la mejor solución encontrada hasta el momento.
4. La velocidad llevando a cabo la tarea. Es importante aclarar que consideraremos como veloces todas aquellas técnicas que se centran principalmente en encontrar una solución cuanto antes, sin importar la calidad de ésta.
5. La facilidad de integrar nuevas restricciones en sus condiciones de búsqueda. Por ejemplo, restricciones de tiempo real, ejecución durante un tiempo fijo o dependencias entre tareas (por ejemplo, si dos o más tareas tienen que ejecutarse en el mismo nodo).
6. La facilidad de modificar el criterio de evaluación de las soluciones.
7. La capacidad para limitar el tiempo de ejecución.

En la figura 8 podemos ver una tabla con los resultados de nuestra evaluación cualitativa. A continuación detallamos algunos comentarios que es importante mencionar en algunos apartados.

Solución Asegurada: en este aspecto queremos hacer una puntualización respecto al *Tabu Search*. El motivo por el que no hemos valorado si es capaz o no de obtener siempre una solución es que esto depende absolutamente de la técnica que utilice para obtener una solución inicial, y no del propio *Tabu Search*. Por lo tanto, creemos que es un apartado donde no debe ser valorado.

Velocidad: como veremos más adelante, hay casos en que algunas técnicas (especialmente el *Backtracking* Básico y el algoritmo de Ramificación y Poda simple) pueden estar ejecutándose durante períodos muy largos cuando se trata de problemas complejos, es decir, problemas con muchas posibles combinaciones de nodos y tareas, pero donde pocas de esas combinaciones llevan a una solución. Aún así, los hemos considerado como veloces porque en la mayoría de los casos lo son, y se centran en encontrar una solución lo antes posible.

Nuevas restricciones: en este apartado debemos hacer un inciso, y es que, por nuestra experiencia, cuanto más complicado es el código de implementación de la técnica, más complicado es introducir nuevas restricciones, dado que los cambios afectan a más partes del código y no son tan aislables. Esto también implica que hay distintos grados de dificultad, pues, aunque tanto el algoritmo Voraz como el *Tabu Search* tengan la misma valoración, es más difícil añadir una restricción en el segundo que en el primero, pues su funcionamiento es más complejo, y los cambios necesarios para adaptar el funcionamiento son mayores. Sin embargo, en todos estos casos, si bien la dificultad de la que hablamos implica sobretodo un aumento del tiempo necesario para realizar la implementación, no consideramos que sea algo muy complejo, pues normalmente solo significa cambiar algunas partes concretas del código. El caso de los *solvers* es un punto aparte. Al controlarse éstos mediante fórmulas lógicas, añadir nuevas restricciones puede ser o muy fácil, o muy difícil. Por poner un ejemplo, es posible que si queremos añadir una nueva restricción, debamos cambiar muchas otras expresiones que hayamos definido antes para adaptarlas a la nueva restricción, por lo que ya no estamos cambiando puntos concretos, sino buena parte del código. Todo esto hace que la dificultad al añadir nuevas restricciones en estas técnicas no sea cuestión de la cantidad de tiempo que necesite el usuario para hacer los cambios en las restricciones, sino que puede llegar a ser necesario reestructurar o formular de nuevo todo el proceso.

Nuevo criterio de evaluación: en la sección III ya comentamos que trataríamos de aislar la definición del criterio de evaluación todo lo posible en los códigos y, en la mayoría de casos, esto ha sido posible. Lo hemos conseguido encapsulando el código relacionado en una función aislada que devuelve una valoración numérica (la ineficiencia de la configuración). Si bien en algunos casos sería necesario cambiar otros aspectos menores, como podría pasar, por ejemplo, si en lugar de tener un criterio donde las evaluaciones con un valor numérico menor son mejores, tuviéramos uno donde las mejores fueran las de más alto valor numérico; los cambios vitales se pueden llevar a cabo en esa función aislada, sin cambiar demasiado el resto. De nuevo éste no es el caso de los *solvers*, en los que la misma naturaleza de su funcionamiento obliga a cambiar casi toda la estructura de comandos para adaptarlos a un nuevo criterio, hasta el punto que, en algunos casos, prácticamente es necesario definir todos los comandos de nuevo. Eso sí, vale la pena mencionar el hecho de que esta técnica permite introducir restricciones más o menos exigentes, lo que facilita su uso con ciertos criterios donde no se busque maximizar o minimizar sólo un valor.

Tiempo limitable: en este último apartado tenemos que hacer una puntualización, pues tanto el *Backtracking* como el algoritmo de Ramificación y Poda, ambos optimizando y sin optimizar, pueden tener el tiempo limitado si se acota el tiempo de ejecución, de forma que el procedimiento se detenga pasado un cierto tiempo y dé como resultado la mejor solución encontrada hasta la fecha, si es que ha encontrado alguna. De hecho, esta característica se ha aplicado en las pruebas cuantitativas, tal y como veremos más adelante, para evitar tiempos de ejecución prácticamente infinitos. Sin embargo, para valorar los apartados de las pruebas cualitativas no hemos considerado esta posibilidad, dado que al aplicarla anulamos una de las características a priori más valiosas de las técnicas

| | *Backtracking* | | | | Heurística | Metaheurística | *SMT Solver* | |
|---|---|---|---|---|---|---|---|---|
| | Simple | Optimizador | Poda Simple | Poda Optimizador | Algoritmo Voraz | Tabu Search | Simple | Optimizador |
| Solución asegurada | SI | SI | SI | SI | NO | - | SI | SI |
| Solución óptima | NO | SI | NO | SI | NO | SI | NO | SI |
| ↑ Tiempo ↑ Solución | NO | SI | NO | SI | NO | SI | NO | NO |
| Velocidad | SI | NO | SI | NO | SI | NO | NO | NO |
| Nuevas restricciones | SI | SI | SI | SI | SI | SI | NO | NO |
| Nuevo criterio de evaluación | SI | SI | SI | SI | SI | SI | NO | NO |
| Tiempo limitable | NO | NO | NO | NO | SI | NO | NO | NO |

Figura 8. Cuadro con las valoraciones cualitativas.

mencionadas, que es la capacidad de encontrar una solución en caso de que esto sea posible, pues no podemos asegurar que el tiempo de ejecución sea suficiente para llevar a cabo todo el proceso. Por lo tanto, no tendremos en cuenta esta posibilidad al valorar los apartados de esta sección, aunque es una opción a tener en cuenta.

### *V-B. Evaluación Cuantitativa*

En esta segundo parte de la evaluación mostraremos los resultados numéricos de los experimentos que hemos llevado a cabo. En estos experimentos valoramos especialmente dos aspectos: el **tiempo de ejecución** y el **valor de la ineficiencia** de la solución.

Cómo hemos visto en la sección anterior, las técnicas de búsqueda que hemos considerado poseen características muy diferentes entre sí, por lo que no todas ellas serán útiles en las mismas situaciones. Por ejemplo, si lo que queremos es encontrar una solución cuanto antes, no tiene mucho sentido utilizar el algoritmo *Backtracking* en su versión de optimizador. Es por ello que hemos diseñado una serie de campañas de experimentos, cada una enfocada a un problema diferente, y utilizaremos en cada una las técnicas que creamos más adecuadas para ese problema.

Por un lado tendremos las técnicas *rápidas*, que son aquellas que priorizan obtener una solución lo antes posible por encima de la calidad de dicha solución. Dentro de este grupo hemos seleccionado los algoritmos de *Backtracking* Básico, Ramificación y Poda simple, Voraz y *SMT Solver* simple. Por el otro lado tenemos las técnicas de *optimización*, más enfocadas en obtener la mejor solución posible, aunque ello implique un aumento considerable del tiempo de ejecución. Aquí encontramos los algoritmos de *Backtracking*, Ramificación y Poda, y *SMT Solver*, todos en su forma de optimizadores. La única técnica que incluiremos en ambos grupos es el *Tabu Search*, pues es una técnica más intermedia y que no encaja sólo con las características de un único grupo. Aun así, y precisamente porque se trata de una técnica intermedia, analizaremos los resultados obtenidos en apartados separados. Una vez aclarado estos puntos, pasaremos a definir la campaña de experimentos a realizar:

1. *Búsqueda Primera Solución - Escenario Sencillo* (V-B2): Campaña de experimentos destinada a medir el tiempo que tardan las técnicas rápidas en encontrar una solución cualquiera. En esta campaña definimos un sistema con un número fijo de nodos y en cada experimento aumentamos el número de tareas. De esta manera simulamos un escenario en el que el espacio de soluciones va creciendo gradualmente.
2. *Búsqueda Primera Solución - Escenarios Aleatorios* (V-B3): Campaña de experimentos similar a la anterior, con la diferencia de que en cada experimento el número de nodos y tareas, así como sus capacidades y costes, es aleatorio. Nos permite probar el comportamiento de las técnicas rápidas frente a escenarios muy diferentes.
3. *Búsqueda Mejor Solución en Tiempo Limitado* (V-B4): Campaña de experimentos destinada a medir la calidad de las soluciones que las técnicas de optimización y *Tabu Search* son capaces de proporcionar en un tiempo limitado.
4. *Tabu Search - Solución Inicial* (V-B5): Campaña de experimentos destinada a determinar las ventajas e inconvenientes de usar el algoritmo de *Backtracking* o el algoritmo Voraz como técnicas para determinar la solución inicial que necesita el algoritmo *Tabu Search*.
5. *Tabu Search - Iteraciones y Tamaño Lista Tabú* (V-B6): Campaña de experimentos destinada a determinar los efectos que tienen el número de iteraciones y el tamaño de la lista Tabú en el rendimiento del *Tabu Search*.

*V-B1. Montaje de los Experimentos:* En primer lugar, y hablando del lenguaje de programación utilizado, hemos elegido *Python* por varias razones. Por un lado, se trata de un lenguaje de alto nivel que facilita el diseño del tipo de programas que hemos implementado. La sintaxis hace que el código resultante sea fácil de leer, y podemos definir y manejar las estructuras de datos que necesitamos también de manera sencilla. Por otro lado, en *Python* es posible implementar todos los algoritmos y técnicas que queremos probar. Si bien esta no

es una cualidad única de *Python*, este lenguaje nos da muchas facilidades en ese sentido, como hemos podido comprobar, por ejemplo, al abordar la implementación de programas basados en *SMT Solver*, donde solamente ha sido necesaria una librería adicional para poder utilizar dicha herramienta.

También cabe destacar que los algoritmos de *Backtracking*, Ramificación y Poda, y Voraz han sido implementados usando su forma recursiva. Si bien la forma iterativa de estos algoritmos debería proporcionar unos tiempos de ejecución mejores, consideramos que, por el tipo y tamaño del problema, la diferencia no debería ser muy relevante. Además, el uso de la forma recursiva permite que la implementación sea mucho más sencilla, y que el código resultante sea mucho más fácil de leer y modificar. Esto último punto es muy relevante dado que el desarrollo ha implicado múltiples iteraciones en las para encontrar la configuración adecuada para cada algoritmo.

Todas las pruebas se han llevado a cabo en las mismas condiciones, es decir, sobre la misma computadora, sin ningún otro proceso activo salvo la ejecución de la técnica en cuestión. Esto se ha hecho con el fin de preservar lo máximo posible la ecuanimidad del proceso de evaluación. Concretamente, se ha utilizado un PC con un procesador Intel(R) Core(TM) i5-4690K y una memoria RAM de 16 GB.

Además, para evitar que los resultados de los experimentos se vean afectado por la variabilidad en las mediciones que pueden aparecer, por ejemplo, por los procesos internos del sistema operativo, repetiremos cada experimento 3 veces, siendo el resultado final en términos de tiempo de ejecución la media de los tres resultados.

*V-B2. Búsqueda Primera Solución - Escenario Sencillo:* Lo que nos interesa en esta campaña de experimentos es observar la evolución de los resultados a medida que la cantidad de combinaciones posibles aumenta. Es por ello que partiremos del sistema siguiente:

- 110 nodos con capacidades de entre 0 y 10
- De 1 a 525 tareas con costes de entre 0 y 2

El proceso que seguimos consiste en que cada una de las técnicas rápidas (*Backtracking* simple, Ramificación y Poda simple, Voraz y *SMT Solver* simple) lleva a cabo la búsqueda, empezando con una sola tarea, y añadiendo una tarea en cada nuevo experimento, hasta alcanzar las 525. El número de nodos no cambiará, ni tampoco los costes y capacidades que, por supuesto, serán iguales para todas las técnicas. El motivo que nos ha llevado a utilizar tareas con costes bastante menores a las capacidades de los nodos es que, de esta forma, queremos suavizar el aumento del tiempo de ejecución para poder observar mejor su evolución. Los resultados pueden verse en las figuras 9 y 10, donde observamos los valores de tiempo e ineficiencia, respectivamente. Recordemos que, en cuanto mayor sea el valor de la ineficiencia, menor es la calidad de la solución.

A simple vista, hay varias conclusiones que obtenemos de estos gráficos. Hablaremos primero de los tiempos de ejecución, vistos en la figura 9. Para empezar, en cuanto a los tiempos de ejecución el *SMT Solver* simple es el que parece dar peores resultados, pues los tiempos que tarda en obtener una solución son mucho peores que los de las otras técnicas. Debido a la diferencia de magnitudes, y dado que

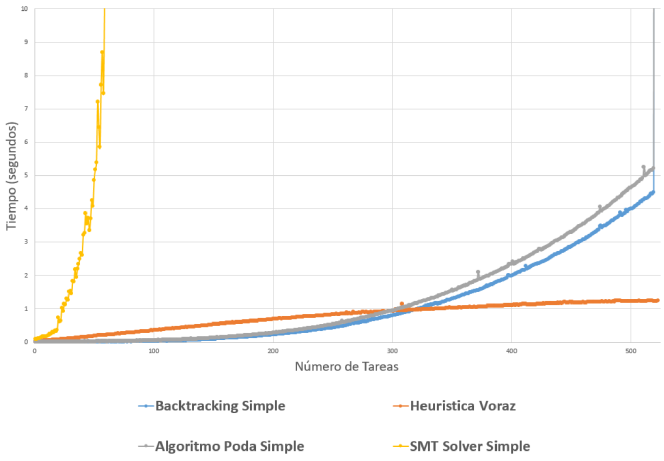


Figura 9. Tiempo para encontrar una solución cualquiera.

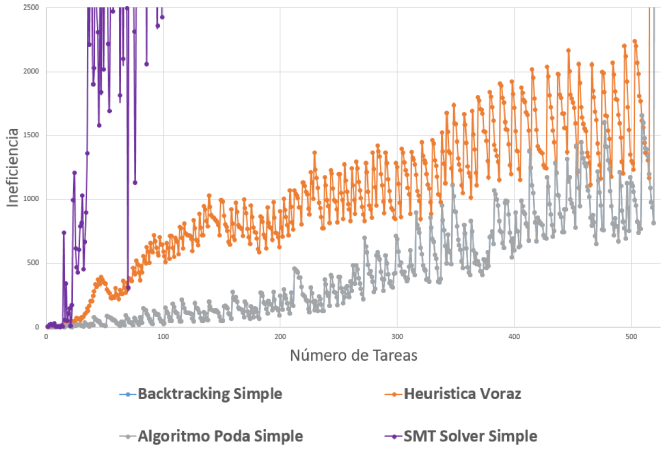


Figura 10. Ineficiencia resultante de las soluciones rápidas.

nos interesa mostrar el crecimiento de los algoritmos basados en *Backtracking* y Voraz, no mostramos los resultados del *SMT Solver* por encima de 10 segundos. Sin embargo, como referencia, éste tarda cerca de un minuto en alcanzar una solución con 100 tareas.

Si observamos los tiempos de ejecución de los algoritmos de *Backtracking* y Ramificación y Poda, sorprende bastante que este último obtenga peores tiempos. Analizándolo más en profundidad hemos determinado que la cantidad de veces que se poda alguna rama es bastante baja. Dadas las condiciones de poda utilizadas, podemos intuir que éstas sólo se cumplirán en situaciones donde haya pocas soluciones y, por ende, muchas ramas que no conducen a una solución. Sin embargo, teniendo en cuenta las condiciones de estas pruebas ese supuesto no se dará la mayoría de los casos, pues los nodos tienen capacidad de sobra para albergar las tareas. Es probable que esta sea la causa de estos malos tiempos, dado que si la poda no tiene efecto, lo único que hará será alargar el tiempo de ejecución.

En cuanto al algoritmo Voraz, la evolución de sus tiempos de ejecución es lineal, mientras que las de *Backtracking* y Ramificación y Poda parecen ser más exponenciales. Esto implica que, en pruebas pequeñas, los tiempos del algoritmo Voraz son ligeramente peores que los de los otros dos algoritmos

mencionados, pero en pruebas más grandes el primero obtiene tiempos mejores que los otros dos.

Refiriéndonos ya a la calidad de los resultados, es decir, a la figura 10, debemos decir que el *SMT Solver* simple es el que peores soluciones proporciona, con una magnitud de las mismas muy superior a los demás. Por otro lado, y como ya esperábamos, el *Backtracking* y el algoritmo de Ramificación y Poda alcanzan exactamente los mismos valores, pues el proceso que realizan es muy similar, y por ende sus tiempos quedan solapados. Donde no parece haber color es con el algoritmo Voraz, pues los resultados aquí indican que esta técnica alcanza peores soluciones que las dos anteriores, aunque no tan malas como el *SMT Solver* simple. Sin embargo, si observamos los resultados en conjunto, es probable que ante problemas con muchos nodos y tareas donde se busca una solución cualquiera, la diferencia en el tiempo de ejecución del algoritmo Voraz frente a los otros dos compense la diferencia en la calidad de las soluciones, pues mientras esta última se mantiene mas o menos constante, la diferencia en los tiempos parece que va en aumento a medida que aumenta el número de nodos y tareas.

Un punto importante relacionado con ambas figuras es que, llegados a cierta cantidad de tareas (cerca de las 520), los resultados de los algoritmos *Backtracking*, Ramificación y Poda y Voraz sufren un cambio importante, con un aumento drástico de la ineficiencia y del tiempo de ejecución. Esto nos lleva a pensar que, o bien a partir de este punto ya no hay solución, o la cantidad de soluciones posibles es reducida y es complicado encontrarlas. Dado que se trata del punto del proceso en el que más tareas por asignar hay, ambas posibilidades tienen sentido y, de hecho, si observamos el valor de ineficiencia obtenido por el algoritmo Voraz, podemos ver que este también se dispara un poco antes, lo que parece indicar que este algoritmo tampoco es capaz de encontrar una solución llegado a este punto, lo que refuerza la teoría comentada. Teóricamente, éste es uno de los puntos en que el *SMT Solver* simple debería ser superior, pues, si hay una solución, éste la encontrará, y debería encontrarla en un tiempo mucho menor que el algoritmo de *Backtracking*. Sin embargo, debido a los enormes tiempos que acarrean estas situaciones, no hemos podido poner a prueba este punto, pues las pruebas tardaban demasiado en ejecutarse, y, en el caso del *SMT Solver*, se intentó tenerlo en marcha hasta que alcanzará una solución para los parámetros de 110 nodos y 522 tareas, pero tras más de 8 horas de ejecución, siguió sin dar resultado.

*V-B3. Búsqueda Primera Solución - Escenarios Aleatorios:* Una vez observados los resultados de la primera campaña de experimentos, pasaremos a realizar una prueba distinta. En este caso intentaremos añadir un punto más de aleatoriedad al experimento. Más concretamente, ejecutamos un total de 50 experimentos donde en cada uno el número de nodos, tareas, capacidades y costes es aleatorio. Esto nos permitirá estudiar el comportamiento de las técnicas rápidas frente a escenarios muy diferentes. Los parámetros variarán de la siguiente manera:

- Entre 10 y 20 nodos, con capacidades variables de entre 0 y 10.
- Entre 1 y 20 tareas, con costes variables de entre 0 y 4.

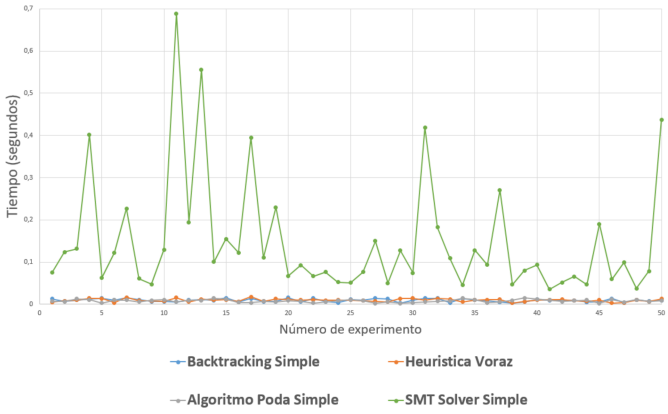


Figura 11. Tiempo para encontrar una solución en problemas aleatorios.

Al elegir estos parámetros, hemos intentado que estos permitan bastante variedad en los posibles escenarios y que, por ejemplo, se den tanto casos en los que hay muchas soluciones fáciles de encontrar como casos en los que haya menos soluciones posibles y, por tanto, sea más difícil encontrar alguna. Sin embargo, también hemos querido asegurarnos en la medida de lo posible de que era posible encontrar alguna solución en todos los experimentos, motivo por el que hemos establecido que siempre habrá como mínimo 10 nodos y que el valor máximo posible de la capacidad de los nodos es superior al valor máximo posible del coste de las tareas.

Los resultados, que podemos ver en las figuras 11 y 12 nos muestran, respectivamente, el tiempo que se ha tardado en realizar cada experimento y el valor de la ineficiencia resultante, y nos permiten sacar algunas conclusiones más. Por ejemplo, se reafirma que los resultados del *SMT Solver* simple son bastante peores que el resto, tanto por los tiempos de ejecución, mayores que los de las demás herramientas, como por los valores de ineficiencia, que son peores que el resto salvo en contadas excepciones. Además, comprobamos que, en términos de tiempo, hay algunos casos en los que el algoritmo de Ramificación y Poda es más rápido que el *Backtracking*, lo cual confirma en cierta medida nuestra teoría de que su efecto se vio reducido en la prueba anterior. Aun así, sigue sin ser una mejora clara respecto a los tiempos del *Backtracking*, lo cual nos lleva a pensar que tal vez deberíamos buscar otras condiciones de poda más efectivas. Ya para terminar con el análisis de estos resultados, además de comprobar que, nuevamente, los resultados en cuanto a ineficiencia son exactamente iguales tanto con el *Backtracking* como con el algoritmo de Poda, también vemos como el algoritmo Voraz da lugar a algunas soluciones cuya ineficiencia es menor que la conseguida por las demás técnicas, lo cual indica que la heurística que hemos seleccionado da buenos resultados. De hecho, y aunque no es algo que consiga exclusivamente el algoritmo Voraz, sino también los otros, hay varios casos en los que la ineficiencia resultante es 0, lo cual nos indica que se ha alcanzado una solución "perfecta"desde el punto de vista de la eficiencia.

*V-B4. Búsqueda Mejor Solución en Tiempo Limitado:* En esta campaña de experimentos mediremos la calidad de las

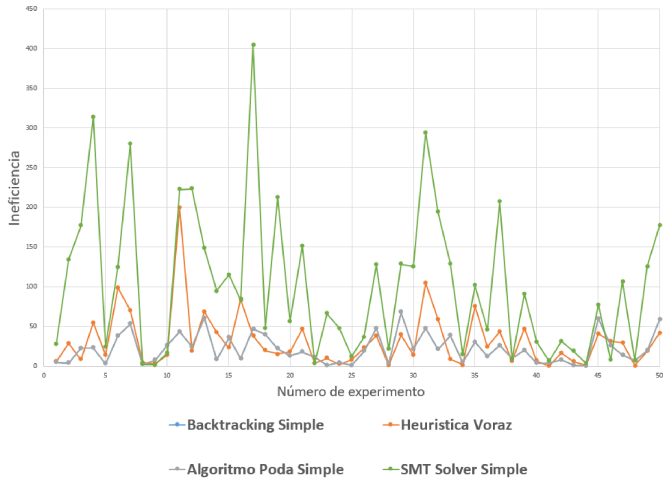


Figura 12. Ineficiencia resultante con problemas aleatorios.

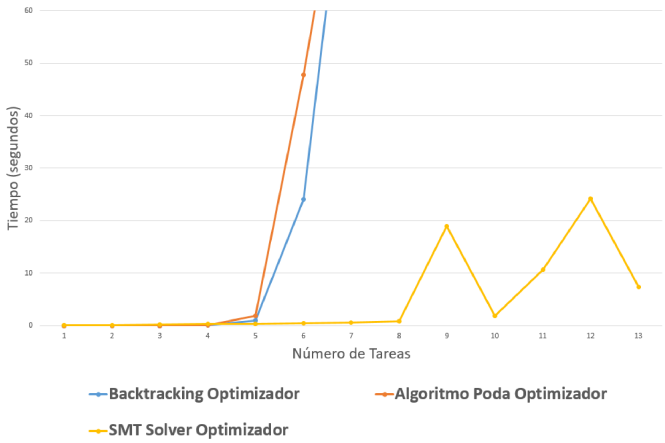


Figura 13. Tiempo para encontrar la mejor solución.

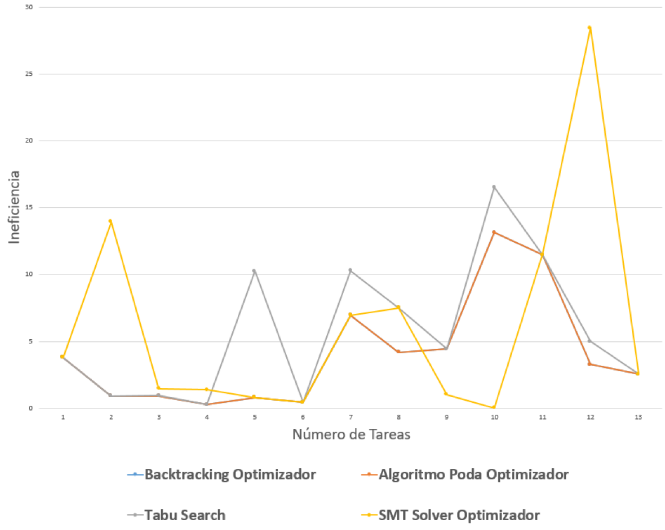


Figura 14. Ineficiencia resultante de las soluciones óptimas con *Tabu Search*.

soluciones proporcionadas por las técnicas de optimización (recordemos, el *SMT Solver* y los algoritmos de Poda y *Backtracking*, todos en sus versiones de optimización) y *Tabu Search* después de un tiempo determinado de ejecución. Es importante aclarar que, si bien la técnica *Tabu Search* normalmente limita su ejecución a un cierto número de iteraciones y no a un límite de tiempo, hemos pensado que es mejor aplicar la misma limitación a todas las técnicas, dado que de lo contrario estaríamos favoreciendo al *Tabu Search* al no limitar su tiempo de ejecución. En este caso, después de varias pruebas, hemos establecido este límite temporal en un minuto, dado que hemos comprobado que la gran mayoría de experimentos que superan ese tiempo de ejecución tienden a superarlo por mucho, es decir, que es bastante probable que superen también los 10 o 20 minutos. Además, cabe destacar que se ha utilizado el algoritmo simple de *Backtracking* (no como optimizador) para encontrar la solución inicial de *Tabu Search*, y el tiempo que necesita para encontrar esa solución inicial esta incluido dentro del tiempo de ejecución total. Igual que en la primera campaña de experimentos el número de nodos es fijo y en cada experimento iremos aumentando el número de tareas a alojar. Concretamente, los parámetros de la campaña son los siguientes:

- 4 nodos con capacidades variables de entre 0 y 10
- Entre 1 y 13 con coste variables de entre 0 y 4

En la figura 13 podemos ver los tiempos de ejecución y en la figura 14 los valores de ineficiencia de las soluciones encontradas. Primero de todo, si nos fijamos en los tiempos de ejecución veremos que *Tabu Search* ha alcanzado el límite de tiempo en todos los casos. En lo que respecta a los algoritmos de *Backtracking* y Ramificación y Poda, los tiempos se disparan a partir de 5 tareas pero, aunque no se muestre aquí, hemos comprobado que, si eliminamos el límite de tiempo, *Tabu Search* mantiene unos tiempos de ejecución bastante estables, mientras que los otros dos algoritmos pueden llegar a tardar horas en terminar la ejecución. Además, el algoritmo de Ramificación y Poda no mejora los tiempos del de *Backtracking*. La explicación seguramente sea la misma una vez más, la condición de podado no afecta realmente en problemas donde hay muchas soluciones posibles, dando lugar a un aumento de tiempo debido a las comprobaciones extra que debe hacer respecto al *Backtracking* Básico. Por otro lado, podemos observar que en el caso del *SMT Solver* optimizador los tiempos son bastante bajos, aunque en algunos casos fluctúa. Es decir, aumentar el espacio de posibilidades no implica un aumento del tiempo de ejecución. En cualquier caso, éstos siempre están por debajo del minuto, es decir, el *SMT Solver* siempre termina su ejecución. Hemos hecho otras pruebas con casos aislados para observar el comportamiento del *SMT Solver*, y hemos llegado a la conclusión de que, si bien sus tiempos también aumentan exponencialmente, la curva de los mismos es bastante menor que la de las otras dos técnicas.

Cabe mencionar que, si bien no se muestran en este gráfico, se hizo la misma prueba añadiendo una tarea más, tanto para los algoritmos como para el *SMT Solver*, y, mientras éste alcanzó una solución en unos 13 minutos, los otros dos algoritmos no habían terminado aun el proceso después de más de una hora de ejecución.

Respecto a la ineficiencia, cabe destacar que los valores de las técnicas basadas en *Backtracking* se corresponden con la solución óptima cuando el tiempo de ejecución es menor de un minuto y con la mejor solución encontrada hasta el momento cuando llega al minuto. En el caso del *SMT Solver* podemos observar que, aunque todos los experimentos terminan en menos de un minutos, algunas veces no devuelve una solución mejor que los competidores. Es decir, el optimizador del *SMT Solver* no devuelve la solución óptima, simplemente una solución buena. Finalmente, *Tabu Search* muestra unos resultados intermedios entre los algoritmos basados en *Backtracking* y el *SMT Solver*. De todas formas, como se mostrará más adelante esto se debe a que el problema es demasiado pequeño, por lo que la fuerza bruta sigue siendo más eficiente. En la siguiente sección mostraremos por qué hemos seleccionado este algoritmo.

*V-B5. Tabu Search - Solución Inicial:* En esta sección explicaremos el proceso que hemos seguido para seleccionar la técnica utilizada para obtener la solución inicial de *Tabu Search*. Para ello, replicaremos las condiciones de la primera campaña de experimentos pero esta vez usando *Tabu Search* con el algoritmo de *Backtracking* simple y el Voraz como técnicas para obtener la solución inicial. Recordemos que *Tabu Search* se detiene tras realizar un cierto número de iteraciones o al encontrar una solución óptima (es decir, con un valor de ineficiencia igual a 0), por lo que en este caso no utilizaremos un límite de tiempo.

Podemos ver las medidas de los tiempos en la figura 15. En esta figura hemos mantenido los resultados del algoritmo de *Backtracking* y Voraz, para así poder tenerlos como referencia. Podemos observar que los tiempos de ejecución de *Tabu Search* con *Backtracking* como técnica para la búsqueda de la solución inicial son sensiblemente mayores que los obtenidos usando el algoritmo Voraz para hallar dicha solución inicial. Esto se debe a que, como se ha mostrado en la sección IV-C, en este problema el *Backtracking* da lugar a mejores soluciones, como se ha mostrado en otro experimento. Así pues, cuando *Tabu Search* parte de la solución proporcionada por el *Backtracking*, la cantidad de soluciones vecinas que mejoren la actual es menor, por lo que tardará más en decidir el siguiente paso, alargando por tanto el tiempo de ejecución.

En la figura 16 se muestran las ineficiencias resultantes en cada uno de los experimentos. En esta figura hemos mantenido todos los demás resultados para poder comprobar cómo *Tabu Search* mejora sus resultados, dependiendo de la técnica para obtener la solución inicial. Como podemos observar, a medida que aumenta el número de tareas, la ineficiencia de los algoritmos de *Backtracking* y Voraz aumentan; por lo tanto, a medida que aumenta el número de tareas también aumenta la ineficiencia de la solución inicial de *Tabu Search*. Además, este aumento también repercute en el tamaño del espacio de configuraciones posibles. Todo esto hace que la diferencia entre la ineficiencia inicial y la ineficiencia óptima aumente cuando aumenta el tamaño del problema. Al ser la distancia entre ambas cada vez mayor, y el espacio de posibilidades más grande, hay cada vez más soluciones vecinas que mejoran la solución actual, pero ello también hace que el ”camino” hacia la solución óptima sea más largo.

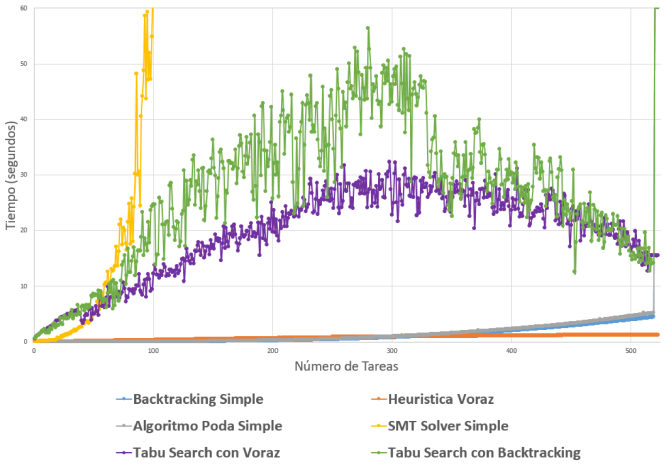


Figura 15. Tiempo de ejecución de *Tabu Search*.

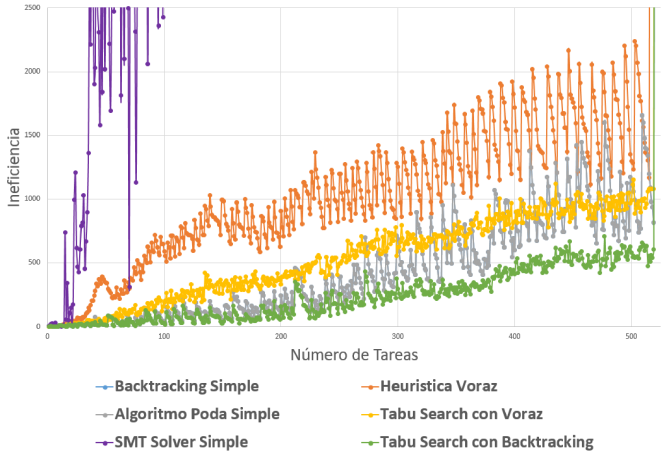


Figura 16. Ineficiencia resultante de *Tabu Search* y las técnicas rápidas.

Para entenderlo mejor, es como si tuviéramos que salir de un laberinto, pero sólo podemos escoger direcciones que lleven hacia la salida, y tenemos un número de movimientos limitado. Si en cada cruce hay muchos caminos que lleven a la salida, como no sabemos cual es más rápido, escogeremos rápido en cada cruce, y terminaremos antes nuestros movimientos. Por contra, si hay pocos caminos que lleven a la salida en cada cruce, tardaremos más en encontrar uno, y el tiempo total será mayor a pesar de haber hecho el mismo número de movimientos. En el caso del *Tabu Search* pasa algo similar. A medida que hay más posibles combinaciones que mejoran la solución actual, el algoritmo tarda cada vez menos en elegir el siguiente movimiento, lo que explica el hecho de que los tiempos de ejecución disminuyan a medida que aumenta el número de tareas, y a partir de un cierto punto.

Para terminar de comentar los resultados de esta campaña de experimentos, es notable el hecho de que las ineficiencias resultantes son mucho peores al utilizar el algoritmo Voraz. Si bien era de esperar un peor resultado tras ver los valores de ineficiencia resultantes utilizando este algoritmo, no nos esperábamos que los resultados finales pudieran llegar a ser peores que los de las técnicas de *Backtracking* y Ramificación

y Poda. Por supuesto, esto podría mejorar con un mayor número de iteraciones, pero eso también implica tiempos más grandes (veremos hasta que punto más adelante). Viendo esto, y sabiendo que los tiempos parece que tienden a igualarse con un número grande de tareas, pensamos que utilizar el algoritmo de *Backtracking* como base para obtener la primera solución es mejor, y por lo tanto será la técnica que utilizaremos como técnica para obtener la solución inicial de *Tabu Search*.

*V-B6. Tabu Search - Iteraciones y Tamaño Lista Tabú:* En esta sección mostraremos la campaña de experimentos que hemos llevado a cabo para determinar los efectos del **número de iteraciones** y el **tamaño de la lista tabú** en el comportamiento de *Tabu Search*. El objetivo es el mismo que en la primera campaña de experimentos, simular un conjunto representativo de los escenarios posibles que nos permita ver como se comporta el *Tabu Search*. Sin embargo, en este caso los resultados obtenidos en casos con muchos nodos y tareas no son tan relevantes, dado que el efecto del número de iteraciones y el tamaño de la lista ya puede verse en escenarios con menos nodos y tareas, por lo que esta vez realizaremos la primera campaña de experimentos con 75 nodos con capacidades de cómo máximo 10, y hasta 190 tareas costes de como máximo 4, y los siguientes parámetros para el *Tabu Search*:

- 20 iteraciones y una lista Tabú con 20 espacios
- 50 iteraciones y una lista Tabú con 20 espacios
- 50 iteraciones y una lista Tabú con 50 espacios
- 80 iteraciones y una lista Tabú con 20 espacios

El motivo para elegir estos parámetros es, ante todo, que nos permitirán observar las diferencias en el rendimiento cambiando sólo el número de iteraciones, sólo el tamaño de la lista, y cambiando ambos. Lógicamente, como en todas las campañas de experimentos, hemos hecho bastantes pruebas más para perfilar los valores que mejor nos permiten ver las diferencias en el rendimiento, pero no mostraremos todas esas pruebas en una misma gráfica para no saturar en exceso la misma.

Lo primero que salta a la vista en la figura 17 es el hecho de que los tiempos de ejecución han quedado muy diferenciados según el número de iteraciones, mientras que el tamaño de la lista tabú parece afectar muy poco al tiempo de ejecución, pues, salvo en casos aislados, los tiempos resultantes con una lista de 20 y una de 50 han quedado prácticamente solapados. Volviendo a las diferencias según el número de iteraciones, como es lógico cuantas menos iteraciones menor es el tiempo de ejecución, además de que parece que esa diferencia se va incrementando hasta un cierto punto, cerca de las 140 tareas, para luego verse reducida cuando el número de tareas pasa de las 150. Por otro lado, en la figura 18 se muestra el valor de la ineficiencia para cada experimento y valores de parámetros. Concretamente, podemos ver diferencias entre los distintos resultados de ineficiencia según los valores de los parámetros. Sin embargo, estas diferencias no son significativas, por lo que no nos queda claro si en problemas con un número alto de combinaciones de nodos y tareas vale la pena el aumento de tiempo que conlleva aumentar el número de iteraciones.

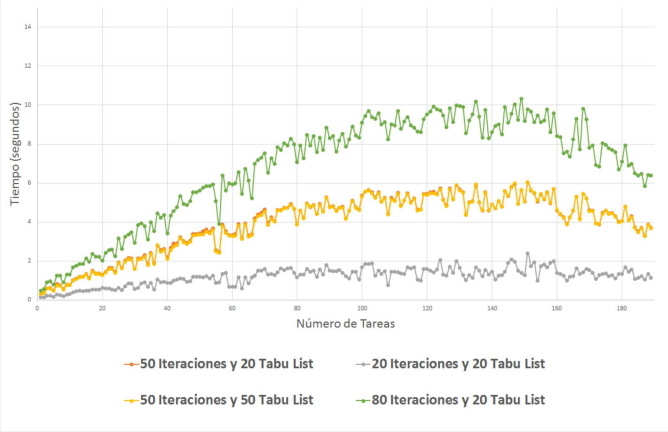


Figura 17. Tiempo de *Tabu Search* con 75 nodos y hasta 190 tareas.

Precisamente con el objetivo de comprobar si con combinaciones de nodos y tareas aún mayores se producen cambios, y por lo tanto aumentar el número de iteraciones es útil, hemos probado con 110 nodos de hasta 10 de capacidad, y hasta 350 tareas con costes de hasta 2. Estos cambios se han hecho para aumentar el número de combinaciones de nodos y tareas posibles.

Los resultados en términos de ineficiencia de esta segunda prueba se muestran en la figura 19, y aquí sí podemos apreciar diferencias significativas, sobre todo al cambiar el número de iteraciones. En problemas grandes, un número de iteraciones pequeño implica un mayor valor y variabilidad de la ineficiencia. Por su parte, el tamaño de la lista tabú parece no afectar demasiado a los resultados y, sólo en ocasiones contadas, un tamaño mayor de lista mejora los resultados. Esto nos indica que el número de iteraciones tiene mucha más relevancia en los resultados que el tamaño de la lista, pues vemos claramente cómo hay bastante diferencia entre los resultados con 20 iteraciones y los demás. Eso sí, viendo que estas diferencias son mucho menores entre las pruebas con 50 y 80 iteraciones, podemos concluir que, a partir de un cierto número de iteraciones, la mejora es cada vez menor. Esto es lógico, pues cuanto mejor es la solución encontrada, menos probable es obtener una mejor.

## VI. Conclusiones

Para terminar esta memoria, expondremos las conclusiones que hemos obtenido tras analizar los resultados de las pruebas realizadas. Aquí haremos un breve repaso sobre los resultados obtenidos para cada técnica y explicaremos cuál creemos que se adapta mejor a cada situación, y por qué.

Empezaremos hablando de los algoritmos de *Backtracking* Básico y de Ramificación y Poda. Para empezar, hay que decir que uno de los puntos a favor de estas técnicas es su simpleza, la cual las hace fáciles de implementar, fáciles de manejar, modificables y adaptables a muchas situaciones. Mediante las pruebas se ha demostrado que, ante problemas donde hay muchas soluciones correctas y no es vital encontrar una solución óptima, pueden resultar muy útiles, pues permiten obtener soluciones rápidamente a base de fuerza bruta. Sin

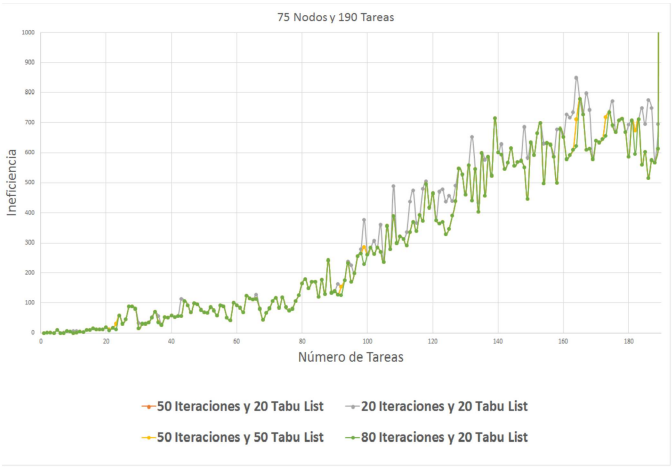


Figura 18. Ineficiencia de *Tabu Search* con 75 nodos y hasta 190 tareas.

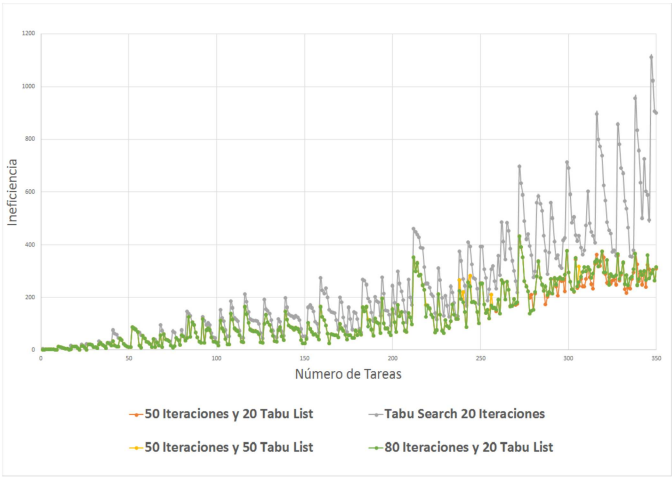


Figura 19. Ineficiencia de *Tabu Search* con 110 nodos y hasta 350 tareas.

embargo, en el momento en el que el conjunto de soluciones correctas no es muy grande, el tiempo de ejecución puede aumentar de forma muy brusca.

En cuanto a sus versiones de búsqueda de solución óptima, es necesario que el número de nodos y tareas no sea muy elevado, dado que en caso contrario los tiempos se disparan. Sin embargo, en casos donde se pueda asegurar que esas cantidades van a ser muy reducidas, seguramente sean la mejor opción.

Para terminar la valoración de estas técnicas, esperábamos un mejor rendimiento del algoritmo de Ramificación y Poda en lo que a tiempos de ejecución se refiere. Ya no es que no mejore demasiado los resultados del *Backtracking*, sino que en la mayoría de los casos los empeora. Como ya hemos comentado, esto se debe a las condiciones de poda que hemos implementado. Un posible trabajo de cara a futuro sería buscar, implementar y evaluar otros criterios para podar, aunque, dadas las características de nuestro problema de asignación de tareas a nodos, no esperamos una mejora muy elevada.

Respecto al algoritmo Voraz, sus resultados han confirmado en cierta parte lo que ya esperábamos. Es el algoritmo más rápido en conjunto, aunque esa velocidad le lleva a obtener soluciones no óptimas. Sin embargo, lo que no esperábamos es que la diferencia en la calidad de los resultados con los algoritmos de *Backtracking* y Ramificación y Poda sería tan grande ni tan constante. Aunque los resultados con las pruebas aleatorias muestran que hay algunos casos en los que esta técnica desemboca en soluciones mejores, es posible que el rendimiento en este apartado pueda mejorarse aplicando una heurística diferente. La gran desventaja de este algoritmo es que no asegura obtener alguna solución, como si hacen los demás. Sin embargo, hay que remarcar que en las pruebas realizadas pocas veces se ha dado que no se llegara a obtener una solución. Sea como sea, consideramos que esta técnica es la más adecuada para situaciones donde la velocidad de ejecución es primordial y hay muchas soluciones correctas, lo que reducirá las probabilidades de que el algoritmo no obtenga ninguna solución.

Sin duda, el *Tabu Search* ha resultado ser la técnica más completa, pues no ha mostrado resultados negativos en ninguna de las pruebas realizadas, y observando los resultados no parece que el valor de sus parámetros internos (número de iteraciones y lista Tabú) puedan afectar de forma imprevisible a su comportamiento. Aun así, hemos identificado dos desventajas. Por una parte, esta técnica depende de otra técnica de búsqueda para obtener la solución inicial. Por otra parte, no es la técnica indicada para resolver problemas en los que sólo se acepte la solución óptima, o haya un límite estricto de tiempo. Sin embargo, creemos que es la técnica más recomendable en un caso general, donde se busque más un cierto equilibrio entre velocidad y calidad del resultado.

Por último tenemos el *SMT Solver*, cuyos resultados han sido algo decepcionantes, sobre todo en el caso del *SMT Solver* simple. Teníamos la esperanza de que ésta fuera una herramienta que, si bien no diera los mejores tiempos, al menos asegurara una solución en un tiempo razonable, cubriendo así la principal carencia del *Backtracking*. Sin embargo, los tiempos de ejecución que hemos obtenido han sido demasiado grandes incluso para compararlos con los demás. El caso de su uso en su versión como optimizador sí deja lugar al optimismo, pues sus tiempos de ejecución son menores respecto a sus competidores pero, por contra, hemos descubierto que no siempre da la mejor solución y sufre de una variabilidad muy grande. Tanto en el caso del *SMT Solver* básico como en el caso del mismo *Solver* en su versión como optimizador, la falta de experiencia con el funcionamiento de la herramienta tal vez ha provocado que no obtengamos mejores resultado. A pesar de todo, creemos que esta herramienta tiene mucho potencial, sobre todo de cara a problemas donde sea necesario encontrar la mejor solución sin importar el tiempo, dado que, en problemas complejos donde las otras herramientas de optimización disparaban sus tiempos de ejecución, la versión como optimizador del *SMT Solver* nos ha permitido obtener la solución óptima en menos tiempo.

Para terminar estas conclusiones, vale la pena recordar cuál era el objetivo de este proyecto: implementar, probar y evaluar varias técnicas de búsqueda para determinar la mayor o menor adecuación de cada una de ellas en diferentes escenarios. Teniendo en cuenta los resultados expuestos, creemos que se ha cumplido este objetivo, dado que hemos podido obtener

conclusiones relevantes sobre las técnicas tratadas que pueden ser de mucha utilidad a la hora de utilizarlas en un caso real.

## VII. Posible Trabajo Futuro

Finalmente, haremos un breve comentario sobre el posible trabajo futuro y las mejoras que se podrían realizar. Para empezar, creemos que el primer punto de expansión obvio son los propios experimentos. Por mucho que hayamos intentado cubrir los aspectos más importantes, es imposible realizar, en el tiempo correspondiente a un trabajo fin de máster, todas las pruebas que uno pudiera concebir, y nosotros mismos nos hemos dejado sin realizar varias de las que habíamos previsto, debido a falta de tiempo o porque hemos dado prioridad a otras. Muchas de éstas ya han sido mencionadas durante el documento, pero otras no, y en ese sentido siempre es posible probar alguna cosa más, aunque ese es un problema que se asume siempre antes de empezar proyectos de este tipo. Concretamente, lo que más nos hubiera gustado habría sido poder probar más a fondo las técnicas de optimización, con pruebas con mayor número de nodos y tareas, para poder observar los tiempos que se manejan en estos casos.

Más allá de realizar más pruebas, creemos que es posible mejorar los resultados obtenidos con algunas técnicas. Concretamente, nos hubiera gustado dedicar aún más tiempo a estudiar el funcionamiento de los *SMT Solver* para poder estar totalmente seguros de que hemos optimizado su rendimiento todo lo posible. Si bien tenemos cierta seguridad en que la diferencia en el rendimiento, de haberla, no sería muy grande, nos hubiera gustado dedicarle ese tiempo, dado que, como hemos dicho, ésta es la técnica con cuyo funcionamiento estábamos menos familiarizados, por lo que creemos que valdría la pena estudiarla aún más, aunque ese estudio podría dar para un proyecto en sí mismo.

Y ya para terminar, así como hemos dicho que siempre es posible realizar más pruebas, pensamos que un punto clave para expandir el trabajo es estudiar de forma parecida otras técnicas que no hemos estudiado aquí, con el fin de comparar también los resultados. Particularmente, y tras ver los buenos resultados obtenidos por el algoritmo *Tabu Search*, creemos que sería interesante estudiar otras técnicas que incluyan metaheurística, como por ejemplo podrían ser el algoritmo *Guided Local Search*, que basa su funcionamiento en aplicar "penalizaciones"para modificar la heurística de búsqueda cuando el algoritmo se queda en un mínimo local [23], o el algoritmo *Fast Local Search*, que realiza un pequeño test previo a las soluciones vecinas para determinar si vale la pena probarlas o no [24].

## Referencias


[1] Algoritmo Voraz en Wikipedia. https://es.wikipedia.org/wiki/Algoritmo_voraz.

[2] Extremos de una función en Wikipedia. https://es.wikipedia.org/wiki/Extremos_de_una_función.

[3] Ramificación y Poda en Wikipedia. https://es.wikipedia.org/wiki/Ramificación_y_poda.

[4] A. Ballesteros, M. Barranco, S. Arguimbau, M. Costa, and J. Proenza. Temporal replication of messages for adaptive systems using a holistic approach. In *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1379–1382, 2019.

[5] A. Ballesteros, J. Proenza, and P. A. Palmer-Rodríguez. Towards a Dynamic Task Allocation Scheme for Highly-Reliable Adaptive Distributed Embedded Systems. *Proc. 22th IEEE Int. Conf. on Emerging Tech. and FactoryAutom. (ETFA)*, September 2017.

[6] M. Barr. *Programming Embedded Systems in C and C++*. O'Reilly Media Inc, 1999.

[7] C. Barrett. *SMT Solvers: Theory and Practice*, September 2008.

[8] L. Bianchi, M. Dorigo, L. M. Gambardella, and W. J. Gutjahr. A survey on metaheuristics for stochastic combinatorial optimization. *Natural Computing*, 8:239–287, September 2008.

[9] N. Bjørner, P. Anh-Dung, and L. Fleckenstein. vZ - An Optimizing SMT Solver. *Tools and Algorithms for the Construction and Analysis of Systems*, 9035:194–199, 2015.

[10] Z. Cheng, H. Zhang, Y. Tan, and A. O. Lim. Greedy scheduling with feedback control for overloaded real-time systems. In *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 934–937, 2015.

[11] L. De Moura and N. Bjørner. Satisfiability Modulo Theories: Introduction and Applications. *Communications of the ACM*, 54:69–77, September 2011.

[12] P. E. Black. Greedy Algorithm definition. *Dictionary of Algorithms and Data Structures*, February 2005.

[13] M. A. G. B. Mathews. On the Partition of Numbers. *Proceedings of the London Mathematical Society*, pages 486–490, 1896.

[14] F. Glover. Tabu Search—-Part i. *ORSA Journal on Computing*, 1:135–206, August 1989.

[15] A. Iacono. Backtracking Explained. https://medium.com/@andreaiacono/backtracking-explained-7450d6ef9e1a.

[16] J.-C. Laprie, A. Avizienis, and B. Randell. Fundamental concepts of dependability. Technical Report 010028, 2001.

[17] E. Maftei, P. Pop, and J. Madsen. Tabu Search-Based Synthesis of Dynamically Reconfigurable Digital Microfluidic Biochips. In *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '09, page 195–204, New York, NY, USA, 2009. Association for Computing Machinery.

[18] S. Ranise and C. Tinelli. The SMT-LIB Format: An Initial Proposal. *Proceedings of the 1st Workshop on Pragmatics of Decision Procedures in Automated Reasoning*, 2003.

[19] R. Serna Oliver and S. Craciunas. SMT-based Task and Network-level Static Schedule Generation for Time-Triggered Networked Systems. 10 2014.

[20] K. Shafique and M. Shah. A non-iterative greedy algorithm for multiframe point correspondence. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27:51–65, January 2005.

[21] W. Steiner. An Evaluation of SMT-Based Schedule Synthesis for Time-Triggered Multi-hop Networks. In *2010 31st IEEE Real-Time Systems Symposium*, pages 375–384, 2010.

[22] R. Sánchez. *Ciencia de datos con R*. 2017.

[23] C. Voudouris and E. Tsang. Guided local search and its application to the traveling salesman problem. *European Journal of Operational Research*, 113:469–499, March 1999.

[24] C. Voudouris, E. Tsang, S. Alsheddy, and A. Alhindi. *Handbook of Heuristics*. Springer International Publishing, 2016.

[25] J. Wu. *Distributed System Design*. 1st edition, 1999.