

Universidad
Rey Juan Carlos

Escuela Técnica Superior
de Ingeniería Informática

Grado en Ingeniería informática

Curso 2022-2023

Trabajo Fin de Grado

**BACKTRACKINGTUTOR: APLICACIÓN PARA LA
ENSEÑANZA TUTORIZADA DE BACTRACKING**

Autor: Irene Casero Gómez
Tutor: Manuel Rubio Sánchez

Agradecimientos

En primer lugar, quería agradecer a mi tutor, Manuel, por acompañarme y ayudarme a lo largo de todo este proyecto.

En segundo lugar, a mis padres, por el apoyo incondicional.

Resumen

El backtracking es una técnica algorítmica para encontrar soluciones a problemas que satisfacen restricciones. Esta técnica se apoya en la recursividad, método para resolver problemas computacionales donde la solución depende de soluciones a instancias más pequeñas del mismo problema, permitiendo que un subprograma se llame a sí mismo. Teniendo estas cosas en cuenta se ha creado BacktrackingTutor, una aplicación para el ámbito de la enseñanza cuya idea de funcionamiento se basa en la resolución de algunos problemas de backtracking y sus versiones a través de la participación activa del usuario. Este irá respondiendo preguntas cuyos aciertos implicarán, cuando sea necesario, la adición de código en la solución hasta crearla íntegra.

El desarrollo de la aplicación ha sido utilizando Python (lenguaje elegido también para el código que va creando el usuario al utilizar la aplicación) y distintas librerías, siendo la principal tkinter, que funciona para la creación y desarrollo de aplicaciones de escritorio.

Palabras clave: BacktrackingTutor, backtracking, recursividad, enseñanza, python, tkinter.

Abstract

Backtracking is an algorithm method for searching solutions of constraint-satisfying problems. Backtracking is supported by recursion, method used for solving computational problems where the solution depends on smaller instances of the same problem; the subprogram can call itself. Keeping this in mind, we have created BacktrackingTutor, an application for the field of education whose working idea is solving some backtracking problems with the users active participation. The user will be answering questions, when the answer is correct, and when necessary, code will be appearing in the solution until it is whole created.

The code development is in Python (also used for the codes inside the application) using different libraries, the main one is tkinter, it works for creation and development of desktop applications.

Palabras clave: BacktrackingTutor, backtracking, recursion, teaching, python, tkinter.

Índice de contenidos

Índice de tablas	XI
Índice de figuras	XIII
1. Introducción y objetivos	1
1.1. Contexto y alcance	1
1.2. Objetivos	1
1.3. Estructura del documento	2
2. Estado del arte	3
2.1. SRec	3
2.2. <i>An innovative Javascript-based framework for teaching backtracking algorithms interactively</i>	4
2.3. <i>Experiencia para la evaluación de Visback, una herramienta para la visualización de algoritmos de backtracking</i>	6
2.4. <i>Teaching the backtracking method using intelligent games</i>	6
3. Descripción informática	9
3.1. Requisitos	9
3.1.1. Requisitos funcionales	10
3.1.2. Requisitos no funcionales	11
3.2. Diseño	18
3.2.1. Diseño de cada ejercicio	18
3.2.2. Diseño de la interfaz e implementación	28
3.2.3. Evaluación de la aplicación	36
4. Conclusiones	39
5. Trabajos futuros	41
Bibliografía	41
Apéndices	45

Índice de tablas

3.1. CU-1 - Seleccionar ejercicio	13
3.2. CU-2 - Resolver ejercicio	13
3.3. CU-3 - Responder pregunta	14
3.4. CU-4 - Regresar al menú	14
3.5. CU-5 - Siguiente pregunta	15
3.6. CU-6 - Pregunta anterior	15
3.7. CU-7 - Descargar código	16
3.8. CU-8 - Finalizar ejercicio	16
3.9. CU-9 - Verificar	17

Índice de figuras

2.1. Imagen pantalla sRec, obtenida en <i>SRec and VAST: Visualizing Software with a Student-Centered Aim</i> [1]	4
2.2. Imagen ejecución nReinas con TAI. Imagen obtenida en <i>An innovative Javascript-based framework for teaching backtracking algorithms interactively</i> [2]	5
2.3. Imagen visback obtenida en <i>Experiencia para la evaluación de Visback, una herramienta para la visualización de algoritmos de backtracking</i> [3]	6
3.1. Diagrama UML	12
3.2. Grafo de dependencias nReinas	19
3.3. Grafo de dependencias Suma de subconjuntos	20
3.4. Grafo de dependencias Mochila 0-1	22
3.5. Grafo de dependencias Sudoku	23
3.6. Grafo de dependencias Laberinto	24
3.7. Grafo de dependencias Salto del caballo	25
3.8. Grafo de dependencias todos los problemas	26
3.9. Grafo de dependencias Suma de subconjuntos	27
3.10. Grafo de dependencias Suma de subconjuntos	27
3.11. Grafo de dependencias todos los problemas	28
3.12. Pantalla 1: Portada	29
3.13. Pantalla 2: Índice	30
3.14. Pantalla 3: Ejercicios - Primera pregunta de un ejercicio.	32
3.15. Pantalla 3: Ejercicios - Pregunta intermedia sin responder dentro de un ejercicio	33
3.16. Pantalla 3: Ejercicios - Acierto de pregunta intermedia dentro de un ejercicio	34
3.17. Pantalla 3: Ejercicios - Última pregunta de ejercicio y botón menú	35
3.18. Pantalla 3: Ejercicios - Descarga código	36

1

Introducción y objetivos

En este capítulo analizaremos el contexto actual de la tecnología en la enseñanza relacionándolo al motivo por el cual se decidió desarrollar este proyecto, también los objetivos y la estructuración del documento.

1.1. Contexto y alcance

Actualmente la tecnología cumple un papel fundamental en el mundo de la enseñanza. Esto viene confirmado con el auge de las TIC (conjunto de tecnologías hardware y software que contribuyen al procesamiento de la información educativa). Teniendo esta información, la idea es crear una aplicación que sirva para facilitar el aprendizaje de un concepto que es complejo y abstracto, el backtracking.

1.2. Objetivos

El objetivo principal es crear una aplicación que sirva como tutor para guiar a estudiantes en la implementación de algoritmos basados en backtracking. Otro objetivo es incorporar los problemas tratados en el libro *Introduction to Recursive Programming* [4] los cuales se imparten en la asignatura de diseño y análisis de algoritmos (DAA) en la URJC para poderse utilizar tanto en las clases teóricas como en las prácticas. También buscamos ayudar al alumno a poder estructurar de cierta manera su pensamiento a la hora de “encarar” la resolución de este tipo

de problemas.

1.3. Estructura del documento

Analizaremos la estructura del documento desde el punto en el que nos encontramos, cabe destacar que anteriormente contamos con un resumen de lo que será y un contexto.

- **Capítulo 2 - Objetivos:** Análisis de objetivos.
- **Capítulo 3 - Análisis de herramientas y métodos actuales:** Se hará un repaso y análisis de las aplicaciones existentes
- **Capítulo 4 - Descripción informática:** Será el de mayor extensión dentro de nuestro escrito. Abordaremos los requisitos, tanto funcionales como no funcionales, diagrama de uso, diseño de la interfaz, diseño de los ejercicios e implementación.
- **Capítulo 5 -Conclusiones:** Se repararán los puntos principales analizando su cumplimiento
- **Capítulo 6 - Posibles trabajos futuros:** Se analizarán posibles mejoras y vías de crecimiento.

2

Estado del arte

En este capítulo haremos un breve recorrido y análisis por los proyectos y documentos ya desarrollados acerca de la enseñanza de backtracking.

2.1. SRec

SRec es una aplicación de software desarrollada por Antonio Pérez Carrasco en 2008 [5] [1] con el fin de crear visualizaciones interactivas sobre programas recursivos implementados en lenguaje Java en los sistemas operativos de Windows y Linux. Es una herramienta muy útil para comprender cómo funcionan una gran cantidad de algoritmos recursivos. Para ello, el usuario debe proporcionar la clase, el método y los parámetros de entrada.

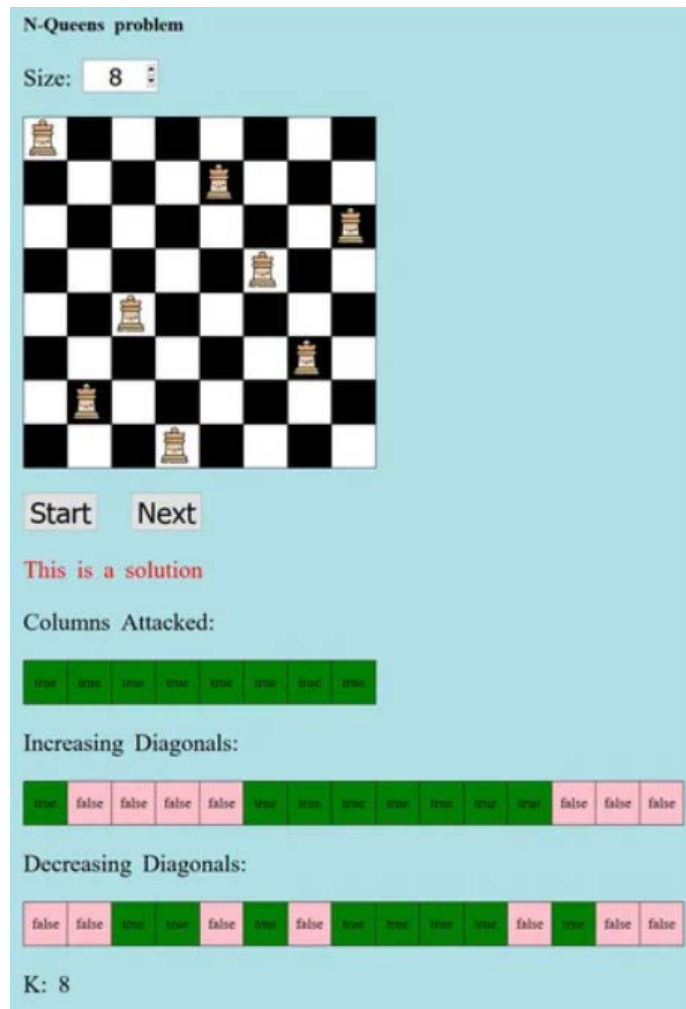


Figura 2.2: Imagen ejecución nReinas con TAI. Imagen obtenida en *An innovative Javascript-based framework for teaching backtracking algorithms interactively* [2]

El usuario solo tiene que pulsar un botón para que se vayan produciendo llamadas, no se le está explicando nada de codificación en ningún momento. Es como otra forma de ejecución por pasos en la que puede ver cómo se comporta o funciona el algoritmo. Los resultados a priori son satisfactorios, pero hay que tener en cuenta que el número de personas a las que se le ha consultado es bastante pequeño y, por tanto, poco significativo.

2.3. *Experiencia para la evaluación de Visback, una herramienta para la visualización de algoritmos de backtracking*

En este artículo [3] nos hablan de Visback, una herramienta que nos permite visualizar gráficamente la traza de la ejecución de los algoritmos que generan las variaciones, permutaciones y combinaciones de una serie de elementos con y sin repetición. Se pueden visualizar los árboles de exploración generados por las distintas llamadas recursivas, mostrando en cada nodo la información que elige el usuario. Es una aplicación que puede ser desarrollada tanto por alumnos como por docentes.

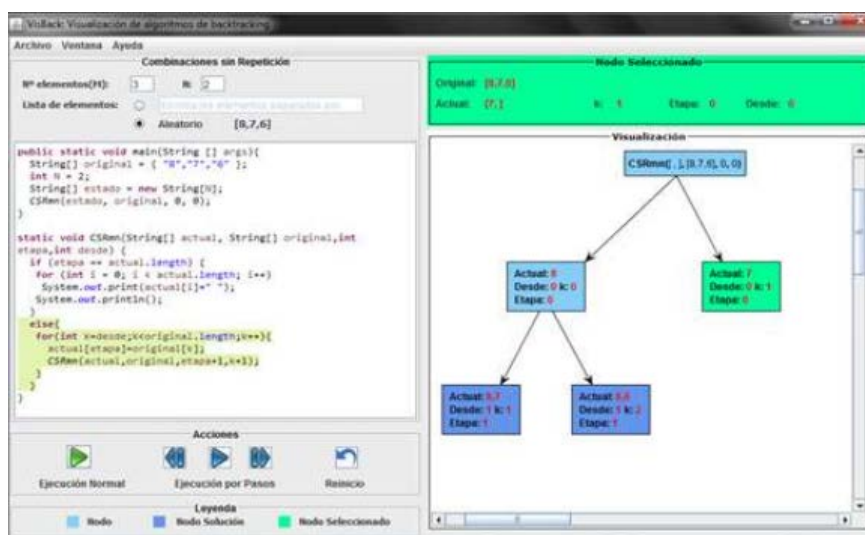


Figura 2.3: Imagen visback obtenida en *Experiencia para la evaluación de Visback, una herramienta para la visualización de algoritmos de backtracking* [3]

Esta aplicación nos ayuda a comprender los árboles de la traza de los ejercicios, pero no cómo implementarlos.

2.4. *Teaching the backtracking method using intelligent games*

En este artículo [6] nos explican que la enseñanza de técnicas de programación siempre ha sido como un desafío, pero se les ha ocurrido una forma innovadora de enseñar la estrategia del backtracking utilizando pseudocódigo del árbol solución de un juego educativo como es, en este caso, el sudoku. Primero explican un poco cómo es la técnica de backtracking y los tipos de soluciones a obtener. A

continuación se centran en el sudoku explicando varios métodos de resolución mediante fórmulas matemáticas.

Puede que sea una técnica innovadora, pero requiere entender muchas fórmulas matemáticas que distan del propósito principal.

3

Descripción informática

En este capítulo se tratarán cuestiones más técnicas como pueden ser los requisitos o requerimientos, el diseño de los ejercicios en base a cómo se han estudiado las diferentes combinaciones de las preguntas y el porqué se han escogido unas y no otras, esto se hará utilizando grafos de dependencia, un algoritmo de backtracking que comprobará la validez de las permutaciones que se generan y un criterio a través de un análisis. También veremos el diseño de la aplicación y la implementación desglosando las distintas pantallas con sus elementos y funcionalidad.

3.1. Requisitos

Antes de entrar en la especificación de requisitos de nuestra aplicación es conveniente dar algunas definiciones de lo que son. Según la RAE (Real Academia de la Lengua Española), un requisito es una definición necesaria para algo. IEEE, Standard Glossary of Software Engineering Terminology, o Glosario de Terminología Estándar de Ingeniería del Software en español, nos ofrece la siguiente definición de lo que es un requisito en el estándar IEEE 830-1998 [7]:

1. Condición o capacidad que necesita un usuario para resolver un problema o lograr un objetivo.
2. Condición o capacidad que tiene que ser alcanzada o poseída por un sistema o componente de un sistema para satisfacer un contrato, estándar, u otro

documento impuesto formalmente.

3. Una representación en forma de documento de una condición o capacidad como las expresadas en 1 ó 2.

3.1.1. Requisitos funcionales

Un requisito funcional es una declaración de cómo debe comportarse un sistema. Define lo que el sistema debe hacer para satisfacer las necesidades o expectativas del usuario. Se debe considerar como características que el usuario detecta.

RF1 - Elección del ejercicio. El sistema debe permitir a los usuarios seleccionar el ejercicio que desean resolver.

RF2 - Información de pantalla. El sistema debe mostrar al usuario información de en qué ejercicio/menú se encuentra en todo momento.

RF3 - Navegación. El sistema debe permitir a los usuarios navegar por cada ejercicio.

RF4 - Enunciado. El sistema debe mostrar el enunciado del ejercicio en todo momento durante la resolución del mismo.

RF5 - Información enunciado. El sistema debe mostrar en todo momento el título del ejercicio a resolver encima del enunciado.

RF6 - Resolución. El sistema debe permitir a los usuarios responder las preguntas tantas veces como le sean necesarias.

RF7 - *Feedback*. El sistema debe proporcionar retroalimentación a los usuarios al responder ciertas preguntas.

RF8 - Código parcial. El sistema debe mostrar el código parcial de los ejercicios cada vez que responda una pregunta de manera correcta y sea necesario.

RF9 - Código nuevo. El sistema debe ser capaz de diferenciar el nuevo código añadido dentro del código general que se está formando. Es decir, cuando se añada nuevo código al ya creado el sistema debe ser capaz de resaltarlo.

RF10 - Código final. El sistema debe ser capaz de generar el código final una vez que se hayan respondido todas las preguntas.

RF11 - Información código. El sistema debe mostrar información en todo momento de qué código es el que se está desarrollando.

RF12 - Desplazamiento. El sistema debe permitir a los usuarios hacer *scroll* en la caja de códigos siempre que sea oportuno y lo considere necesario.

RF13 - Menú. El sistema debe permitir a los usuarios volver al menú de selección y, por tanto, dar por terminada la resolución de ese ejercicio en cualquier momento.

RF14 - Descarga. El sistema debe permitir a los usuarios descargar los códigos una vez se haya finalizado un ejercicio o alguna versión.

RF15 - Aparición botón descarga. El sistema debe hacer aparecer el botón de descarga siempre que se responda correctamente la última pregunta de un ejercicio o versión.

RF16 - Comprobación. El sistema debe hacer que los usuarios confirmen la acción de dejar de resolver un ejercicio y volver al menú, también la de descarga.

RF17 - Siguiente. El sistema debe permitir a los usuarios avanzar a la siguiente pregunta.

RF18 - Aparición botón siguiente. El sistema debe hacer aparecer el botón siguiente siempre que se acierte la pregunta actual.

RF19 - Anterior. El sistema debe permitir a los usuarios retroceder a la pregunta anterior, mostrando esta sin responder.

RF20 - Aparición botón anterior. El sistema debe hacer aparecer el botón anterior a partir de la pregunta número dos de cada ejercicio, esta incluida.

3.1.2. Requisitos no funcionales

Son las restricciones o los requisitos impuestos al sistema. Especifican un atributo de calidad del software. Cómo debe ser un sistema.

Usabilidad:

RNF1 - El tiempo de aprendizaje del uso del sistema debe ser escaso. Un usuario debería poder estar utilizando todas las funciones que se le proporcionan por su cuenta en menos de 5 minutos.

RNF2 - El sistema debe poseer interfaces gráficas bien formadas en las que los elementos se presenten de forma clara y estructurada.

RNF3 - Rendimiento. Todas las operaciones de la aplicación deben poderse realizar en un tiempo ínfimo, menos de un segundo, desde que se selecciona la opción hasta que se termina de ejecutar.

A continuación vamos a desarrollar el diagrama de casos de uso 3.1 y cada caso de uso indicando los actores, su descripción, el escenario, precondition y postcondición.

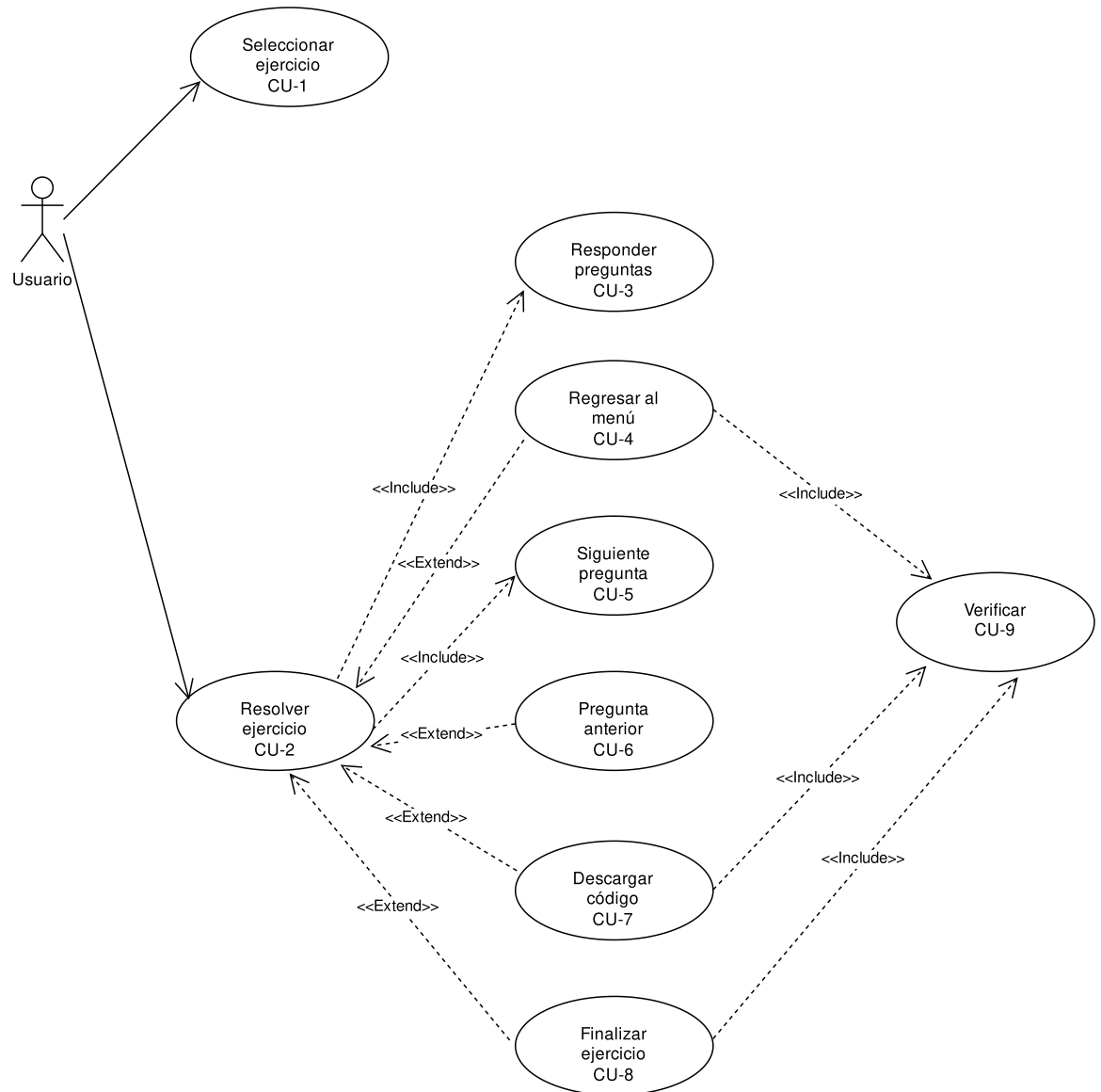


Figura 3.1: Diagrama UML

Tabla 3.1: CU-1 - Seleccionar ejercicio

Identificador	CU-1
Caso de uso	Seleccionar ejercicio.
Actores	Usuario.
Descripción	El usuario selecciona el ejercicio que desea comenzar a resolver.
Escenario	1.El usuario accede a la pantalla índice. 2.El usuario contempla las opciones que tiene. 3.El usuario selecciona el ejercicio que desea resolver.
Precondición	-
Postcondición	El sistema muestra la primera pantalla del ejercicio seleccionado.

Tabla 3.2: CU-2 - Resolver ejercicio

Identificador	CU-2
Caso de uso	Seleccionar ejercicio.
Actores	Usuario.
Descripción	El usuario comienza la resolución del ejercicio.
Escenario	1.El usuario debe estar en la pantalla inicial de alguno de los problemas.
Precondición	Haber seleccionado un ejercicio.
Postcondición	-

Tabla 3.3: CU-3 - Responder pregunta

Identificador	CU-3
Caso de uso	Responder pregunta.
Actores	Usuario.
Descripción	El usuario comienza a responder cuestiones.
Escenario	1.El usuario se encuentra “dentro” de alguno de los ejercicios. 2.El usuario responde las preguntas.
Precondición	Estar resolviendo un ejercicio.
Postcondición	El sistema muestra feedback de la respuesta seleccionada al usuario.

Tabla 3.4: CU-4 - Regresar al menú

Identificador	CU-4
Caso de uso	Regresar al menú.
Actores	Usuario.
Descripción	El usuario selecciona la opción de volver al menú.
Escenario	1.El usuario se encuentra en pantalla de resolución de alguno de los ejercicios. 2.El usuario presiona el botón menú.
Precondición	Estar resolviendo alguno de los ejercicios.
Postcondición	El sistema manda un mensaje de confirmación que el usuario deberá responder.

Tabla 3.5: CU-5 - Siguiete pregunta

Identificador	CU-5
Caso de uso	Siguiete pregunta.
Actores	Usuario.
Descripción	El usuario desea avanzar a la siguiete pregunta.
Escenario	1. El usuario está en pantalla de resolución de alguno de los ejercicios. 2.El usuario pulsa el botón siguiete.
Precondición	1. Estar resolviendo alguno de los ejercicios. 2. Haber acertado la pregunta en la que nos encontramos. 3. Que la pregunta no sea la última del ejercicio.
Postcondición	El sistema avanza a la siguiete pregunta.

Tabla 3.6: CU-6 - Pregunta anterior

Identificador	CU-6
Caso de uso	Pregunta anterior.
Actores	Usuario.
Descripción	El usuario selecciona el ejercicio que desea comenzar a resolver.
Escenario	1.El usuario está en pantalla de resolución de alguno de los ejercicios. 2.El usuario pulsa el botón anterior.
Precondición	1. Estar resolviendo alguno de los ejercicios. 2. No estar en la primera pregunta de alguno de los ejercicios.
Postcondición	El sistema retrocede a la pregunta anterior.

Tabla 3.7: CU-7 - Descargar código

Identificador	CU-7
Caso de uso	Descargar código.
Actores	Usuario.
Descripción	El usuario desea descargar el código creado.
Escenario	1.El usuario está en pantalla de resolución de alguno de los ejercicios. 2.El usuario pulsa el botón de descarga.
Precondición	1.Estar resolviendo alguno de los ejercicios. 2. Haber llegado a la última pregunta de algún ejercicio o versión dentro de un ejercicio.
Postcondición	El sistema manda un mensaje de confirmación que el usuario deberá responder.

Tabla 3.8: CU-8 - Finalizar ejercicio

Identificador	CU-8
Caso de uso	Finalizar ejercicio.
Actores	Usuario.
Descripción	El usuario selecciona que quiere ir a la siguiente pregunta.
Escenario	1.El usuario está en pantalla de resolución de alguno de los ejercicios. 2.El usuario pulsa el botón siguiente.
Precondición	1. Estar resolviendo alguno de los ejercicios. 2. Haber llegado a la última pregunta de un ejercicio.
Postcondición	El sistema manda un mensaje de confirmación que el usuario deberá responder.

Tabla 3.9: CU-9 - Verificar

Identificador	CU-9
Caso de uso	Verificar.
Actores	Usuario.
Descripción	Permite al usuario verificar o cancelar una acción.
Versión 1	
Escenario	1.El usuario confirma la acción.
Precondición	1. Estar resolviendo alguno de los ejercicios. 2. El usuario ha pulsado el botón finalizar ejercicio o el botón menú.
Postcondición	El sistema vuelve al menú de selección de ejercicio.
Versión 2	
Escenario	1.El usuario confirma la acción.
Precondición	1. Estar resolviendo alguno de los ejercicios. 2. El usuario ha pulsado el botón descarga.
Postcondición	El sistema crea el fichero de texto del código que el usuario ha decidido descargar.
Versión 3	
Escenario	1.El usuario rechaza la acción.
Precondición	1. Estar resolviendo alguno de los ejercicios. 2. El usuario ha pulsado el botón finalizar ejercicio o el botón menú.
Postcondición	El sistema se mantiene en la pantalla actual.

3.2. Diseño

En esta sección vamos a ver cómo se ha diseñado cada ejercicio, cómo se ha diseñado la interfaz y cómo se ha ido implementando. Estos dos últimos apartados los juntaremos en uno mismo, para explicar de cada pantalla el diseño y la posterior implementación.

3.2.1. Diseño de cada ejercicio

En este apartado vamos a analizar las decisiones de las preguntas seleccionadas para cada ejercicio y versiones de los mismos que vamos a desarrollar en nuestra aplicación. Los ejercicios son Nreinas, suma de subconjuntos, mochila 0-1, sudoku, laberinto y salto del caballo. Mostraremos el grafo de dependencias de preguntas de cada ejercicio explicando la secuencia elegida, así como el grafo de dependencias general y su análisis.

Antes de empezar pondremos cada una de las preguntas que formarán parte de los ejercicios y les asignaremos una letra para trabajar con mayor comodidad.

A- Se trata de un problema del tipo:

B- ¿Se trata de un problema de optimización?

C- ¿Desea obtener una solución o todas las soluciones?

D- ¿De qué forma se añadirán las restricciones de colocación?

E- ¿Buscamos soluciones con la misma cardinalidad?

F- ¿Cuáles son las variables restantes para optimizar el algoritmo?

G- ¿Es posible añadir alguna condición para dejar de buscar soluciones en caso de no poder encontrar una factible?

H- ¿Desea realizar podas adicionales (descartar soluciones que nunca puedan alcanzar el valor óptimo)?

I- Aparte de la variable de la solución parcial será necesaria. . .

J- ¿Cómo será la solución parcial?

K- Sabiendo que la forma de recorrido del sudoku será por filas (desde la 0 hasta la 8), habremos encontrado una solución cuando. . .

L- ¿Qué pasaría si $S[\text{fila}, \text{columna}]$ es distinto de cero (la celda no es vacía)?

M- Si $S[\text{fila}, \text{columna}]$ es igual a 0 (la celda está vacía), entonces. . .

N- ¿Se puede acelerar el algoritmo utilizando estructuras de datos adicionales?

O- Sabremos que ha finalizado el ejercicio cuando. . .

P- ¿Cuántos candidatos hay?

Q- Cuando obtengamos un candidato habrá que comprobar que sea válido.

Esto es:

R- ¿Es necesario marcar la celda vacía si no se ha encontrado solución?

Z- ¿Qué versión desea desarrollar?

Procedemos al análisis de los ejercicios de forma individual:

Para el **problema de las nReinas** se van a utilizar las siguientes preguntas:

A- Se trata de un problema del tipo:

B- ¿Se trata de un problema de optimización?

C- ¿Desea obtener una o todas las soluciones?

D- ¿De qué forma se añadirán las restricciones de colocación?

Grafo de dependencias de este ejercicio:

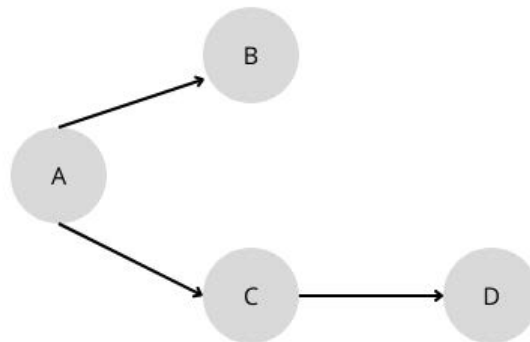


Figura 3.2: Grafo de dependencias nReinas

1. ABCD, opción elegida.
2. ABDC, no válida.
3. ACDB, válida.
4. ACBD, válida.

5. ADBC, no válida.
6. ADCB, no válida.

El motivo por el cual no se han escogido ni 2 ni 3, es que vimos conveniente que ya se haya elegido la versión que se desea desarrollar (una solución o todas las soluciones) para ir desarrollando los códigos de cada una de ellas de una manera más específica. También se ha decidido que el orden de plantearse si era un problema de optimización fuese ese, ya que, si bien en este ejercicio al no serlo no se produce ninguna modificación de código, en algún otro si ocurre y hemos querido aplicar la misma estructura.

Para el **problema de la suma de subconjuntos** se van a utilizar las siguientes preguntas:

Z- ¿Qué versión desea desarrollar?

A- Se trata de un problema del tipo:

C- ¿Desea obtener una solución o todas las soluciones?

E- ¿Buscamos soluciones con la misma cardinalidad?

G- ¿Es posible añadir alguna condición para dejar de buscar soluciones en caso de no poder encontrar una factible?

I- Aparte de la variable de la solución parcial será necesaria. . .

Grafo de dependencias de este ejercicio:

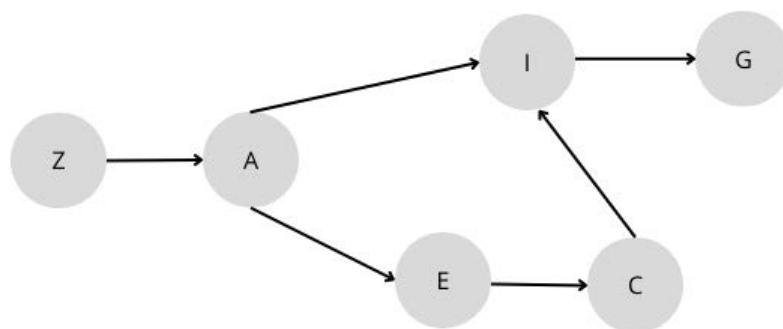


Figura 3.3: Grafo de dependencias Suma de subconjuntos

Al haber 6 nodos en el grafo de dependencias 3.3, el número de permutaciones obtenidas será bastante elevado. Si bien, se han comprobado todas las necesarias

para asegurarnos de que estábamos eligiendo la que mejor se adaptase a como queríamos plantear el problema, comentaremos solo algunas. No serán válidas algunas como ZEACGI, ZACIGE, ZAEGCI. La válida escogida será ZAECIG, siendo EC solo típicas de una de las versiones.

La idea es que se elija primero la versión que se desea desarrollar, ya que tienen diferente esquema de código y, en alguna ocasión, distintas preguntas. La pregunta de cómo será la solución parcial debemos formularla una vez sepamos cómo va a ser la solución a desarrollar y dentro de las versiones también puede haber distintas ramas, como puede ser C, que debemos contemplar antes.

Para el **problema de la mochila 0-1** se van a utilizar las siguientes preguntas:

Z- ¿Qué versión desea desarrollar?

A- Se trata de un problema del tipo:

B- ¿Se trata de un problema de optimización?

E- ¿Buscamos soluciones con la misma cardinalidad?

F- ¿Cuáles son las variables restantes para optimizar el algoritmo?

G- ¿Es posible añadir alguna condición para dejar de buscar soluciones en caso de no poder encontrar una factible?

H- ¿Desea realizar podas adicionales (descartar soluciones que nunca puedan alcanzar el valor óptimo)?

Grafo de dependencias:

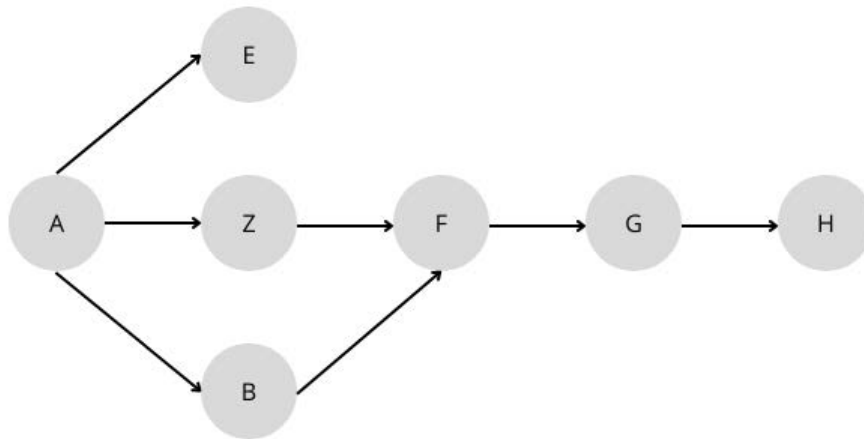


Figura 3.4: Grafo de dependencias Mochila 0-1

Al haber 7 nodos en el grafo de dependencias 3.4, vuelve a ocurrir lo comentado en el problema inmediatamente anterior, así que procedemos de la misma manera. No serán válidas algunas como AZEBGFH, ABEFZGH, AFGHEZB, AEZFBGH, AEZBFHG. Serán válidas algunas como AZBFGHE, ABEZFGH, ABZEFGH, AEZBFGH, siendo esta última la opción elegida. Motivos de elección:

- Primero elegimos la versión que desea desarrollar para aplicar los códigos solución ya basados en cada uno de ellas y no de forma general.
- Para preguntar las variables restantes para la optimización primero hemos de saber que, efectivamente, es de optimización.
- Las demás preguntas tienen ese orden para ir generando el código de manera progresiva desde arriba hacia abajo.

Para el **problema del sudoku** se van a utilizar las siguientes preguntas:

A- Se trata de un problema del tipo:

B- ¿Se trata de un problema de optimización?

J- ¿Cómo será la solución parcial?

K- Sabiendo que la forma de recorrido del sudoku será por filas (desde la 0 hasta la 8), habremos encontrado una solución cuando...

L- ¿Qué pasaría si $S[\text{fila}, \text{columna}]$ es distinto de cero (la celda no es vacía)?

M- Si $S[\text{fila}, \text{columna}]$ es igual a 0 (la celda está vacía), entonces...

N- ¿Se puede acelerar el algoritmo utilizando estructuras de datos adicionales?

Grafo de dependencias:

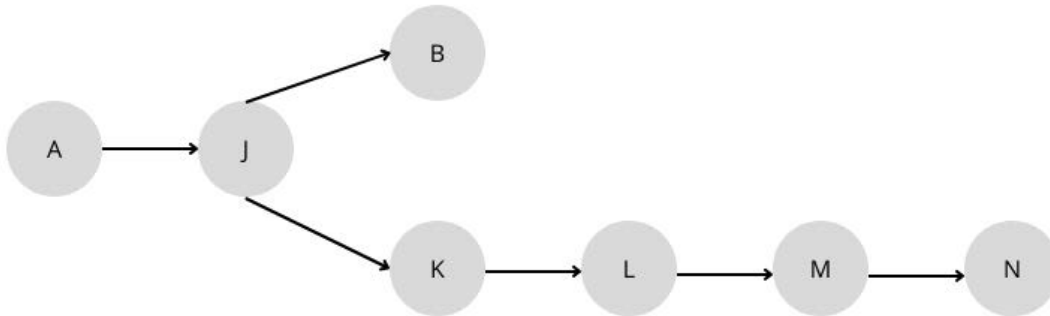


Figura 3.5: Grafo de dependencias Sudoku

Ahora con 7 nodos en el grafo de dependencias 3.5 vuelve a ocurrir lo de los ejercicios anteriores. No serán válidas algunas como ABJKLMN, AKJBLMN, AJKBMLN, AJBKLNM, AJBLKMN. Serán válidas algunas como AJKLBMN, AJKBLMN, AJKLMNB, AJBKLNM, siendo esta última la opción elegida.

Antes de preguntar si es de optimización debemos preguntar cómo será la solución parcial, que sería lo que estaríamos optimizando a medida que se desarrolla la ejecución. Respecto a lo del orden de si es de optimización y, teniendo en cuenta lo que acabamos de explicar, se podría colocar en cualquier posición, ya que no se estaría produciendo ninguna modificación de código. La elección es debido a que habrá problemas en los que el código sí sea modificado tras responder esta pregunta, entonces estamos buscando la misma estructura para ayudar a los alumnos a pensar de la misma forma para que pueda percibir cuando esta pregunta sí tenga un orden más establecido. Una vez aprendan, ellos mismos decidirán cómo proceder.

Para el **problema del laberinto** se van a utilizar las siguientes preguntas:

A- Se trata de un problema del tipo:

B- ¿Se trata de un problema de optimización?

C. ¿Desea obtener una solución o todas las soluciones?

J- ¿Cómo será la solución parcial?

O- Sabremos que ha finalizado el ejercicio cuando...

P- ¿Cuántos candidatos hay?

Q- Cuando obtengamos un candidato habrá que comprobar que sea válido.
Esto es:

R- ¿Es necesario marcar la celda vacía si no se ha encontrado solución?

Grafo de dependencias:

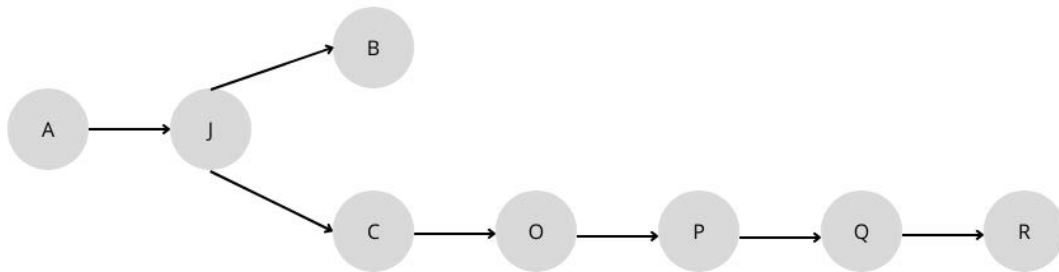


Figura 3.6: Grafo de dependencias Laberinto

Vuelve a tener lugar lo mismo que en los ejercicios anteriores, esta vez contando el grafo de dependencias 3.6 con 8 nodos. No serán válidas algunas como ABJ-COPQR, ACJBOPQR, AJBOCPQR, AJCBPOQR, AJBCOPRQ. Serán válidas algunas como AJCBOPQR, AJCOBPQR, AJCOPQRB, AJBCOPQR, siendo esta última la elegida. Los motivos de elección han sido los siguientes:

- Respecto a la solución parcial los motivos son análogos a los del ejercicio anterior.
- Debemos saber la versión a desarrollar, una o todas las soluciones, antes de continuar con el resto de preguntas, para ir generando el código propio de cada versión.
- Respecto a la pregunta de optimización los motivos son análogos a los del ejercicio anterior.

Para el **problema del salto del caballo** se van a utilizar las siguientes preguntas:

A- Se trata de un problema del tipo:

B- ¿Se trata de un problema de optimización?

J- ¿Cómo será la solución parcial?

O- Sabremos que ha finalizado el ejercicio cuando. . .

P- ¿Cuántos candidatos hay?

Q- Cuando obtengamos un candidato habrá que comprobar que sea válido.

Esto es:

R- ¿Es necesario marcar la celda vacía si no se ha encontrado solución?

Grafo de dependencias:

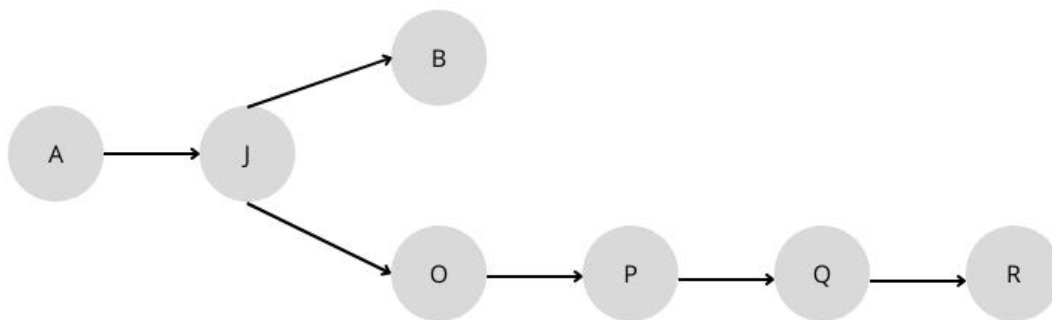


Figura 3.7: Grafo de dependencias Salto del caballo

Pasa lo mismo que en los ejercicios anteriores, 7 nodos en este caso 3.7. No serán válidas algunas como AOJBPQR, ABJOPQR, AJBOQPR, AJOBPRQ. Serán válidas algunas como AJOPQRB, AJOPQBR, AJOBPQR, AJBOPQR, siendo esta última el orden elegido. Los motivos de elección han sido análogos a los del ejercicio anterior.

Una vez terminamos el análisis de cada problema con su grafo de dependencias y su orden elegido, decidimos hacer un grafo de dependencias a nivel general con los elegidos en cada ejercicio individual con el fin de ver cómo se comportaba y por si aparecía algún ciclo. Cada problema se ha puesto de un color; esto es:

nReinas color rojo, suma de subconjuntos color azul, mochila 0-1 color verde, sudoku color naranja, laberinto color negro y salto del caballo color rosa.

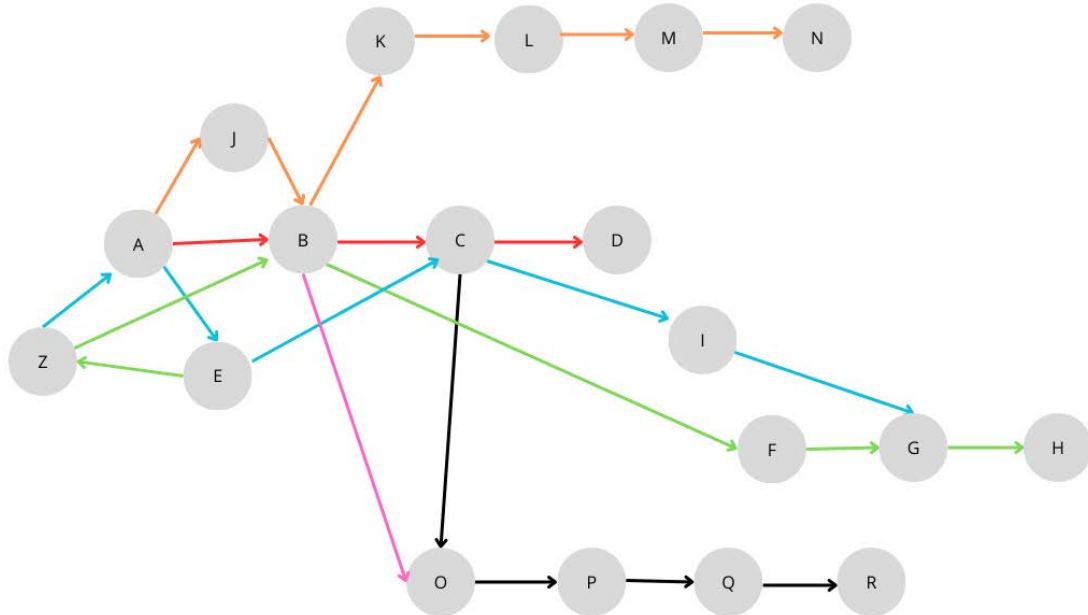


Figura 3.8: Grafo de dependencias todos los problemas

Vemos que tiene lugar un ciclo (ZAE), A necesita Z, E necesita A y Z necesita E, algo que a priori no puede suceder. Esto ocurre por los ejercicios 2, suma de subconjuntos, y 3, mochila 0-1 3.8.

En el ejercicio de la suma de subconjuntos, se tiene como primera pregunta la elección de la versión (Z), mientras que en el ejercicios de la mochila 0-1, esta misma pregunta se realiza más adelante, dando prioridad a otras como el tipo de ejercicio o la cardinalidad. Para solucionarlo se harán los siguientes cambios:

- En el ejercicio de la suma de subconjuntos, se decide preguntar la versión a desarrollar una vez que ya hayamos preguntado el tipo del problema.
- En el ejercicio de la mochila 0-1, se decide suprimir la pregunta de la cardinalidad e incluir esta respuesta en la caja de indicaciones una vez que el usuario haya seleccionado la versión que desea desarrollar.

Por lo que los grafos de dependencias y permutaciones elegidas quedarían de la siguiente manera:

Suma de subconjuntos:

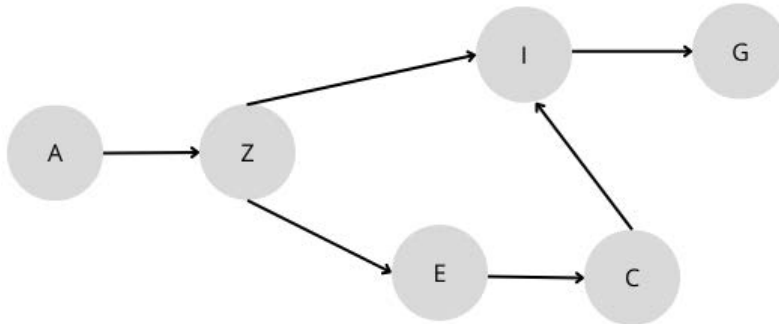


Figura 3.9: Grafo de dependencias Suma de subconjuntos

Elección: AZECIG, siendo los motivos los mismos que los ya explicados.

Mochila 0-1:

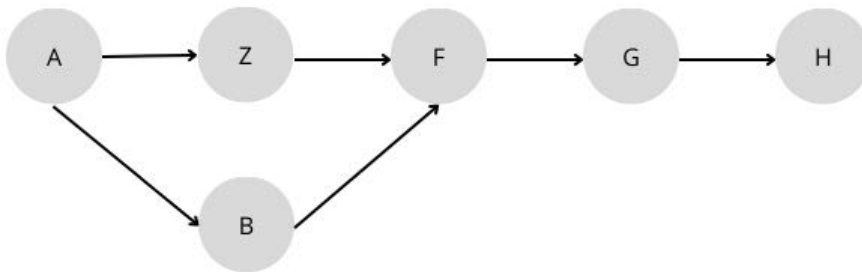


Figura 3.10: Grafo de dependencias Suma de subconjuntos

Elección: AZBFGH, siendo los motivos los mismos que los ya explicados.

Una vez se han modificado y reestructurado estos dos ejercicios volvemos a realizar el grafo de dependencias incluyendo todas las preguntas y las selecciones de cada ejercicio.

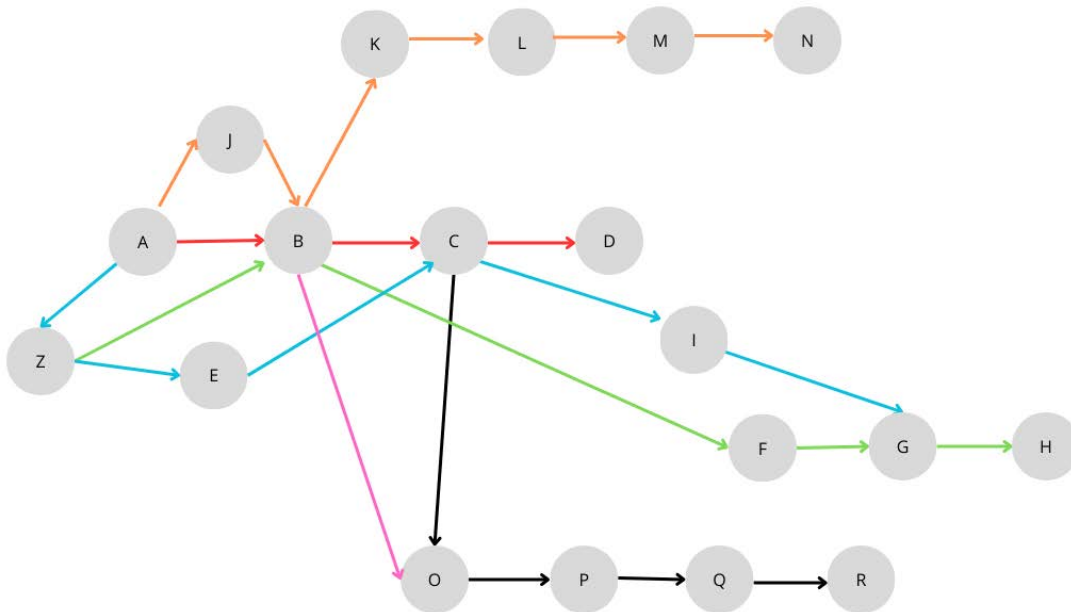


Figura 3.11: Grafo de dependencias todos los problemas

En este nuevo grafo 3.11 podemos comprobar que se ha eliminado el ciclo existente en el anterior y, por tanto, podemos confirmar que las preguntas de todos los ejercicios están bien planteadas y tienen rigor.

3.2.2. Diseño de la interfaz e implementación

En este apartado analizaremos cómo hemos diseñado nuestra aplicación y, a su vez, cómo hemos ido implementándola. Para ello lo dividiremos en tres pantallas: portada, índice y ejercicios.

Como hemos comentado anteriormente, el lenguaje utilizado para el desarrollo ha sido Python. Dentro de este se han utilizado dos librerías: tkinter [8] y pillow.

Tkinter es una librería del lenguaje de programación Python y funciona para la creación y el desarrollo de aplicaciones de escritorio. Esta librería facilita el posicionamiento y desarrollo de una interfaz gráfica de escritorio con Python. Utilizaremos distintos widgets dentro de cada una de las tres ventanas que tendremos, como serán cuadros de texto, botones, etiquetas, scroll, messagebox, etc.

Pillow es una biblioteca adicional gratuita y de código abierto para el lenguaje de programación Python que agrega soporte para abrir, manipular y guardar

muchos formatos de archivo de imagen diferentes. La hemos utilizado para poder importar y utilizar las imágenes creadas para nuestra aplicación.

Pantalla número 1: Portada

Esta pantalla consta de una imagen de fondo en la que aparece escrito “Ejercicios de backtracking” y un botón que consta de una imagen donde se puede leer “Comenzar” y cuya acción será llevarnos a la segunda pantalla donde se encuentra el índice. Dentro de la información de la pantalla se puede ver que nos encontramos en el “Inicio”.

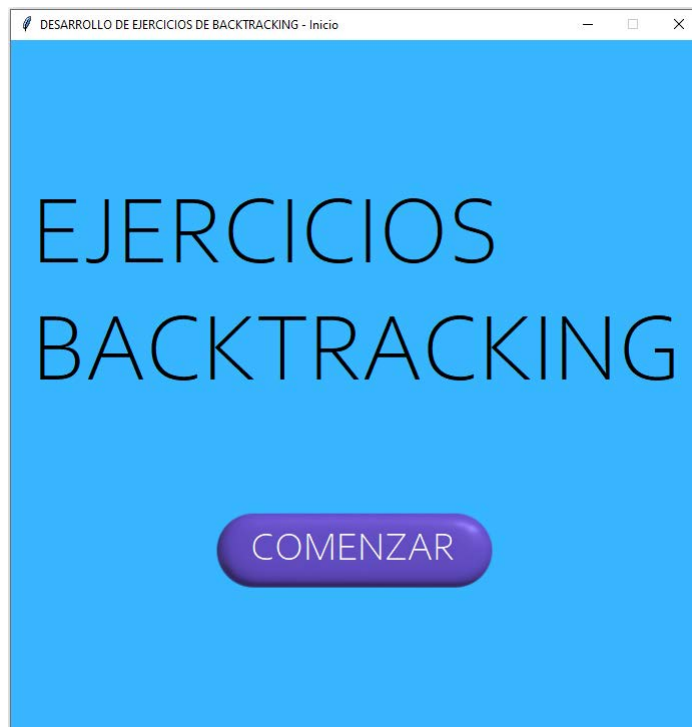


Figura 3.12: Pantalla 1: Portada

Pantalla número 2: Índice

Será el índice donde podremos seleccionar cada uno de los ejercicios. Se compondrá de una imagen de fondo, esta imagen es un rectángulo de color azul con seis espacios divididos mediante tres rayas. En cada uno de los espacios irá el botón representativo de cada uno de los ejercicios, cada uno estará formado por una imagen característica de los mismos y su título. Cuando estos botones sean pulsados, se asignará valor a una variable controladora que nos llevará a la parte de código donde se desarrolla cada uno. También consta de la información de la pantalla, donde se especifica que estamos en el momento de elegir ejercicio.



Figura 3.13: Pantalla 2: Índice

Antes de explicar la estructura y el desarrollo de la tercera y última pantalla de la aplicación cabe mencionar otras cuestiones:

- Se ha importado cada imagen utilizada posteriormente.
- Se han escrito métodos que utilizaremos de forma auxiliar a la hora de implementar cada uno de los ejercicios para evitar que haya repeticiones de código. Esto es, por ejemplo, un método donde aparezcan las opciones de las preguntas y se cambie el texto de cada una de las respuestas, un método que sirva para actualizar el contenido de la caja de las soluciones cambiando o ampliando el contenido anterior, un método donde se cambie la información del *feedback*, un método donde se declare la acción cuando la respuesta elegida sea la correcta o una incorrecta, etc. En conclusión, con esto estamos teniendo métodos que concentren la parte general dentro de cada uso individual según las necesidades de cada problema.

Pantalla número 3: Ejercicios

Se ha declarado:

- Una etiqueta para el enunciado, una caja de texto para el enunciado, una caja de texto para las preguntas, una caja de texto para las indicaciones o *feedback* que recibirá el usuario, una etiqueta para las soluciones, una caja para las soluciones, un scroll en la caja de soluciones por si el tamaño excede el propio de la caja y se necesita.
- Botones de respuesta para los ejercicios, etiquetas para el texto de respuesta de los ejercicios, texto para el enunciado y para la pregunta de los ejercicios, botones de descarga, de siguiente y anterior. Esto se ha declarado una única vez.

Dependiendo del ejercicio que estemos desarrollando, los elementos creados tendrán un contenido o una funcionalidad diferentes. En la información de la pantalla se mostrará en todo momento información acerca de qué ejercicio estamos resolviendo.

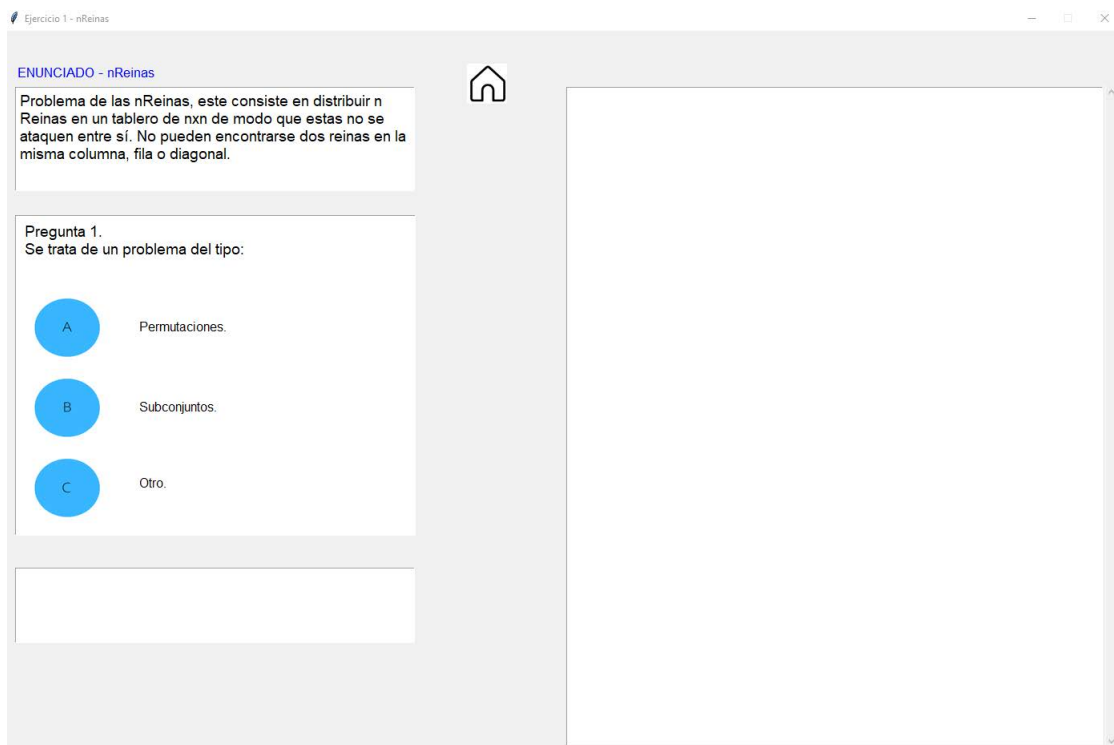
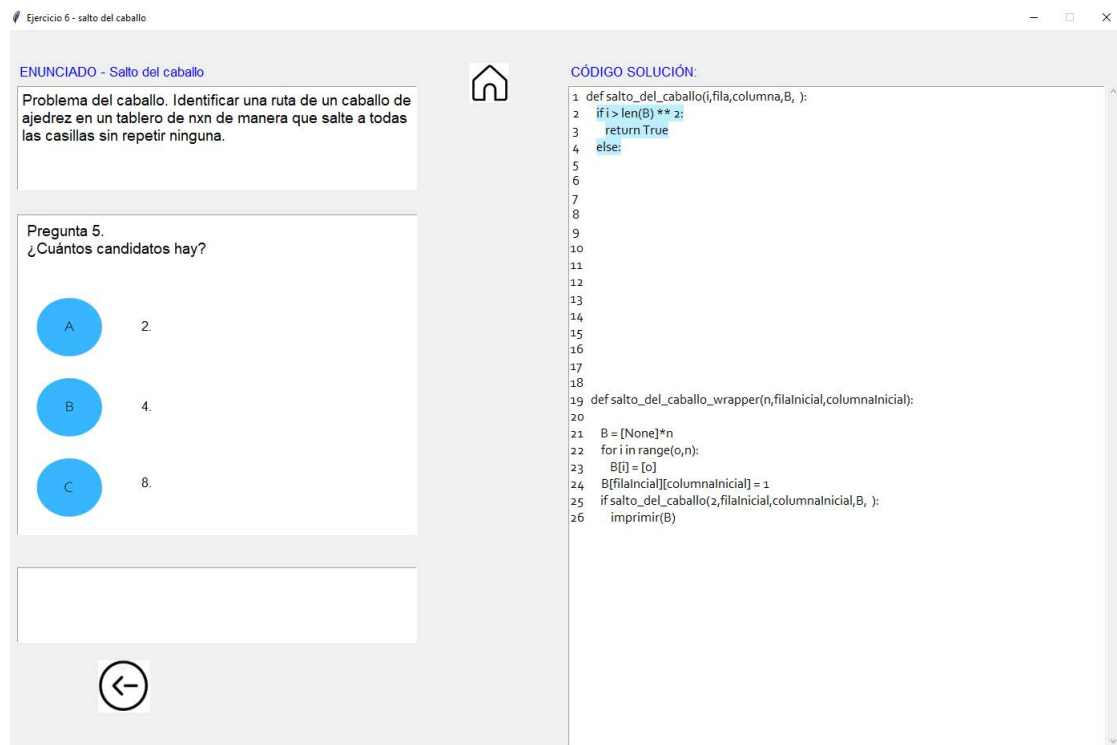


Figura 3.14: Pantalla 3: Ejercicios - Primera pregunta de un ejercicio.

En la imagen 3.14 podemos ver el inicio de uno de los ejercicios, la primera pregunta con sus opciones, y el resto de elementos que se empezarán a utilizar en cuanto se comience a resolver el ejercicio.



Ejercicio 6 - salto del caballo

ENUNCIADO - Salto del caballo

Problema del caballo. Identificar una ruta de un caballo de ajedrez en un tablero de $n \times n$ de manera que salte a todas las casillas sin repetir ninguna.

Pregunta 5.
¿Cuántos candidatos hay?

A 2.

B 4.

C 8.

CÓDIGO SOLUCIÓN:

```
1 def salto_del_caballo(i, fila, columna, B, j):
2     if i > len(B) ** 2:
3         return True
4     else:
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19 def salto_del_caballo_wrapper(n, filainicial, columnainicial):
20
21     B = [None] * n
22     for i in range(0, n):
23         B[i] = [0]
24     B[filainicial][columnainicial] = 1
25     if salto_del_caballo(2, filainicial, columnainicial, B, j):
26         imprimir(B)
```

Figura 3.15: Pantalla 3: Ejercicios - Pregunta intermedia sin responder dentro de un ejercicio

En la imagen 3.15 tenemos una pregunta intermedia aún sin resolver, se puede ver que ya se ha empezado a generar parte del código y que, al no ser la primera pregunta contamos con el botón para poder retroceder a la pregunta anterior. Aparece destacado el código añadido en la pregunta anterior para dar mayor énfasis e información del ejercicio.

Ejercicio 5 - laberinto

ENUNCIADO - Laberinto

Laberinto. Este problema consiste en llegar al final de una cuadrícula sorteando paredes.

Pregunta 6.
¿Cuántos candidatos hay?

A 2.

B 4.

C 8.

Respuesta correcta.
Siendo estas las direcciones para avanzar (arriba, abajo, izquierda, derecha).
Actualizamos código:

← →

CÓDIGO SOLUCIÓN - UNA SOLUCIÓN:

```

1 def laberinto_una_solucion(L, fila, columna, incr, filaSalida, columnaSalida):
2     if fila == filaSalida and columna == columnaSalida:
3         return True
4     else:
5         solFound = False
6         k = 0
7         while not solFound and k < 4:
8             nuevaColumna = columna + incr[k][0]
9             nuevaFila = fila + incr[k][1]
10
11
12
13
14
15         k = k + 1
16         return solFound
17
18
19 def laberinto_una_solucion_wrapper(L, filalinicio, columnalinicio, filaSalida, columnaSalida):
20     incr = [(0,1),(1,0),(0,-1),(-1,0)]
21     L[filalinicio][columnalinicio] = 'P'
22     if laberinto_una_solucion(L, filalinicio, caolumnalinicio, incr, filaSalida, columnaSalida):
23         imprimir(L)

```

Figura 3.16: Pantalla 3: Ejercicios - Acierto de pregunta intermedia dentro de un ejercicio

En la imagen 3.16 tenemos una pregunta intermedia que acaba de ser respondida correctamente, tenemos mensaje de retroalimentación en la caja correspondiente y el botón para avanzar a la siguiente pregunta acaba de ser habilitado.

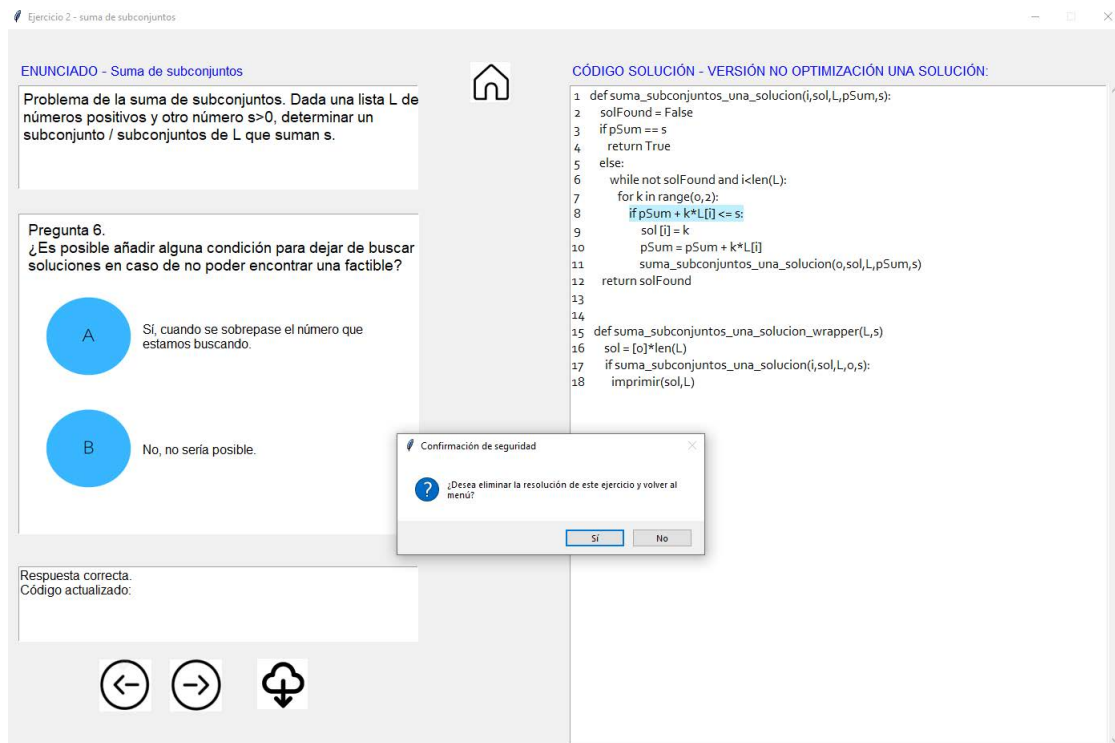


Figura 3.17: Pantalla 3: Ejercicios - Última pregunta de ejercicio y botón menú

En la imagen 3.17 tenemos la última pregunta de un ejercicio, se sabe porque ya está habilitado el botón descarga y porque al pulsar el botón de avanzar a la siguiente pregunta nos aparece el mensaje de confirmación de abandonar el ejercicio, este mensaje también aparecerá cuando pulsemos el botón menú.

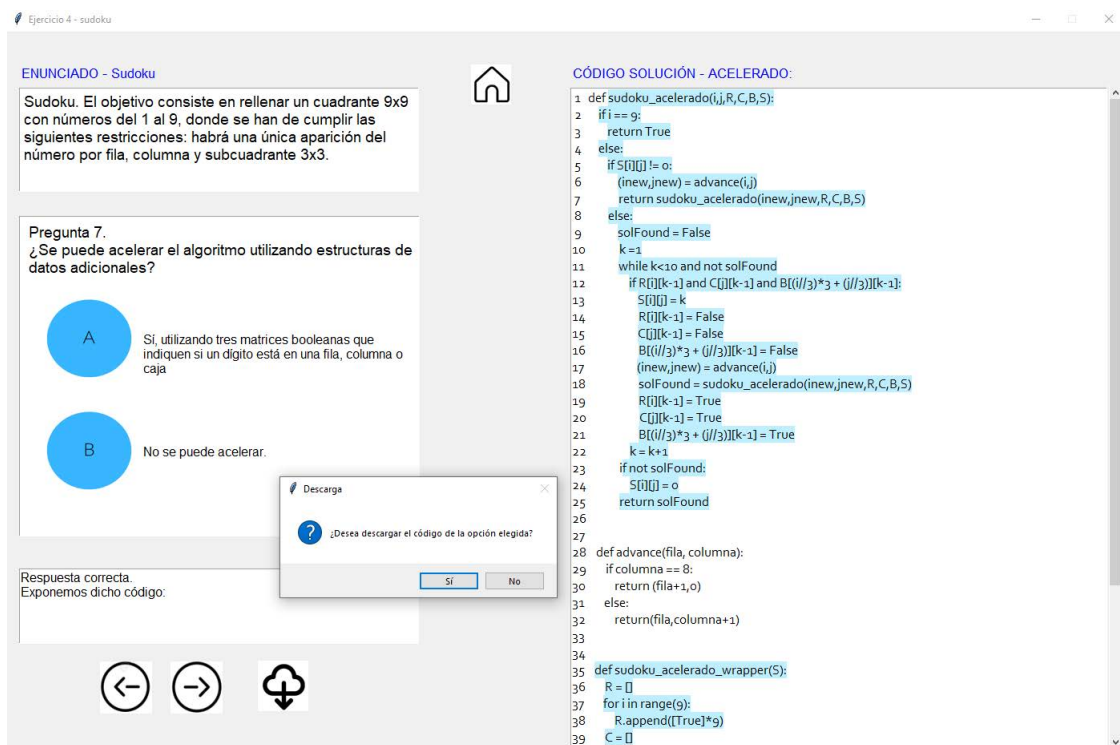


Figura 3.18: Pantalla 3: Ejercicios - Descarga código

En la imagen 3.18 podemos ver la pregunta de seguridad cuando vamos a descargar un código. También se aprecia el *scroll*.

3.2.3. Evaluación de la aplicación

En este apartado evaluaremos la aplicación a través de un par de análisis heurísticos. Esta es una técnica para evaluar la usabilidad de las interfaces.

HEURÍSTICA NIELSEN

Jakob Nielsen es un prestigioso ingeniero de interfaces danés que en 1995 publicó en su blog diez principios acerca de la usabilidad. Estos se deben tener en cuenta al realizar una interfaz. A continuación los analizamos respecto a nuestra aplicación [9] [10].

1. **Visibilidad del estado del sistema.** Se debe mantener a los usuarios informados sobre lo que está sucediendo. Esto se puede ver en la información asignada a cada ventana, donde se nos indica en qué parte estamos o en qué ejercicio. También se puede tener en cuenta la información respecto al ejercicio, colocada encima del enunciado y la opción de la versión que se está desarrollando, colocada encima del código solución.

2. **Relación entre el sistema y el mundo real.** En la aplicación, se están utilizando palabras y frases familiares para los usuarios a los que está dirigida. La información aparece en un orden natural y lógico formando un conjunto (código) a través de ir analizando conceptos.
3. **Libertad y control por parte del usuario.** Los usuarios tendrán vías rápidas para dejar el estado no deseado al que accedieron, podrán regresar al menú siempre que consideren, por si se han equivocado a la hora de seleccionar ejercicio. También podrán volver a las preguntas anteriores a la actual, pudiendo navegar por los ejercicios.
4. **Consistencia y estándares.** Los mismos conceptos se expresan de manera idéntica evitando así posibles confusiones del usuario.
5. **Prevención de errores.** Esto se puede ver cuando se pulsa el botón de volver al menú, el de siguiente pregunta siendo esta la última del ejercicio y el de descarga. Antes de ejecutar ninguna de estas acciones deberán ser confirmadas.
6. **Reconocimiento antes que recuerdo.** El código que se va formando y que constituirá la solución se mantendrá visible constantemente para los usuarios. De esta forma, no tendrán que recordar el progreso ya hecho, solo seguir trabajando con él.
7. **Flexibilidad y eficiencia de uso.** Los usuarios se podrán adaptar sin problema al sistema para usos frecuentes, ya que este funciona de una manera fácilmente comprensible.
8. **Estética y diseño minimalista.** La información que aparecerá en la aplicación será la mínima necesaria para que se pueda desarrollar cada actividad.
9. **Ayudar a los usuarios a reconocer.** En la aplicación no aparece ningún mensaje de error ya que no hay posibilidad de usarla mal.
10. **Ayuda y documentación.** Aunque la aplicación se pueda utilizar sin ningún tipo de documentación se podría crear una guía de uso.

HEURÍSTICA SHNEIDERMAN - OCHO REGLAS DE ORO DE SHNEIDERMAN

Ben Shneiderman es un prestigioso informático estadounidense. Su investigación principal está relacionada con la interacción persona-ordenador con, por ejemplo, las ocho reglas de oro del diseño. A continuación lo analizamos respecto a nuestra aplicación [11].

1. **Esforzarse por la consistencia.** Importancia de la misma terminología en indicaciones, menús y pantallas. En la aplicación está todo desarrollado de manera uniforme, con las mismas fuentes, colores e iconos. También se utilizan mensajes fácilmente comprensibles.
2. **Buscar la usabilidad universal.** La aplicación está dedicada hacia un tema en particular, pero de tal forma que cualquier usuario dentro de este ámbito la pueda utilizar.
3. **Ofrecer comentarios informativos.** La aplicación ofrece retroalimentación para las acciones que realiza el usuario. Esto es, cada vez que responde a alguna de las preguntas de los ejercicios se le ofrece *feedback*.
4. **Diseñar diálogos para producir un cierre.** Dentro de la aplicación la única acción con posible final es la de resolución de un ejercicio, el mensaje de satisfacción mostrado podría ser el propio código que acabamos de generar. No se está mostrando un mensaje como tal, pero se está mostrando el resultado íntegro del proceso.
5. **Prevenir errores.** La aplicación está diseñada de tal manera que ningún usuario pueda cometer ningún error grave.
6. **Permitir la revisión de acciones.** Las acciones son reversibles, hay mensajes de comprobación en los botones que producen los cambios más significativos. Se pedirá confirmación para dejar de resolver un ejercicio y volver al menú, ya sea porque este se ha terminado o porque así lo ha decidido el usuario, también al descargar el código creado.
7. **Mantener a los usuarios en control.** Los usuarios tendrán la sensación de estar a cargo de la aplicación, ya que no hay ningún tipo de proceso o acción que evite esto o que les haga perder esa sensación.
8. **Reducir la carga de memoria a corto plazo.** Los usuarios no deberán recordar ningún tipo de información de una pantalla a otra. El código que vamos generando se muestra en cada pantalla, así se puede ver lo ya añadido, incluso destacando lo último incorporado.

4

Conclusiones

En este capítulo vamos a analizar si la aplicación cumple con los objetivos y metas propuestas.

Por un lado, la idea era desarrollar una aplicación dedicada a la enseñanza de backtracking pero distinta de las ya existentes. Mientras que estas se basan en el análisis y estudio de trazas, representación, árboles de códigos ya creados, con nuestra aplicación buscábamos crear el código desde cero y con la participación activa del usuario. Para lograrlo, también se precisaba de un análisis exhaustivo de cada problema en cuanto a estructura y posibles preguntas, teniendo en cuenta las combinaciones válidas que se podrían originar y eligiendo la que mejor se adecuase a nuestro propósito en cada ejercicio. Una vez realizado el análisis de las preguntas en apartados anteriores y cumpliéndose también la acción del usuario podemos afirmar que esto se logró.

Por otro lado, queríamos que la aplicación cumpliera respecto a la usabilidad una vez evaluada. Teniendo en cuenta la evaluación que realizamos con anterioridad podemos concluir que como primer análisis también se ha cumplido con los objetivos; si bien, la evaluación se podrá realizar de distintas maneras, como comentaremos más adelante.

5

Trabajos futuros

En este capítulo abordaremos qué líneas futuras y desarrollos podrá tener nuestra aplicación.

Se considera:

- **Web:** se podría convertir en una aplicación web. Esto aumentaría su alcance, pudiendo llegar así a un mayor número de personas.
- **Idioma:** originariamente está diseñada para personas hispanoparlantes, se podría ampliar añadiendo nuevos idiomas.
- **Estudio:** realizar el análisis de usabilidad con alumnos. También se podría realizar otro análisis con datos más concretos y distintos grupos de alumnos de los efectos que tiene sobre el aprendizaje.
- **Integración:** se podría estudiar si integrar esta aplicación con algunas de las ya existentes. Esto estaría completando de forma íntegra el proceso de creación y estudio de un ejercicio, algoritmo o problema.
- **Ejercicios:** se puede aumentar el número de ejercicios propuestos.

Bibliografía

- [1] F. J. Almeida-Martínez and A. Pérez-Carrasco, “Srec and vast: Visualizing software with a student-centered aim.” *International Journal of Artificial Intelligence and Interactive Multimedia*, pp. 61–68, 2015.
- [2] M. M. Nasralla, “An innovative javascript-based framework for teaching backtracking algorithms interactively,” *Electronics*, 2004.
- [3] C. Lacave, M. Paredes-Velasco, J. Velázquez Iturbide, and I. Hernán, “Experiencia para la evaluación de visback, una herramienta para la visualización de algoritmos de backtracking.” *Revista iberoamericana de Informática Educativa*, pp. 83–94, 2017.
- [4] M. Rubio-Sánchez, *Introduction to Recursive Programming*. Taylor and Francis Group, 2018.
- [5] A. P. Carrasco, “Sistema generador de animaciones interactivas para la docencia de algoritmos recursivos,” Ph.D. dissertation, Universidad Rey Juan Carlos, 2011.
- [6] A. Andrei and W. T. Mahavier, “Teaching the backtracking method using intelligent games,” Lamar University, Tech. Rep., 2013.
- [7] [Online]. Available: <https://standards.ieee.org/ieee/380/6424/>
- [8] [Online]. Available: <https://docs.python.org/es/3/library/tkinter.html>
- [9] J. Nielsen, “10 usability heuristics for user interface design,” Nielsen Norman Group, Tech. Rep., 2015.
- [10] “La evaluación heurística,” blog, 2018.
- [11] S. L. University, “Las ocho reglas de oro para el diseño de interfaces ben shneiderman,” 2023.

Apéndices



Ejercicios

Tendremos las preguntas junto a sus posibles respuestas de cada uno de los ejercicios. La pregunta correcta se destacará respecto al resto.

NREINAS

Pregunta 1. Se trata de un problema del tipo:

- a. Permutaciones.**
- b. Subconjuntos.
- c. Otro.

Pregunta 2. ¿Se trata de un problema de optimización?

- a. Sí
- b. No**

Pregunta 3. ¿Desea obtener una solución o todas las soluciones?

- a. Una solución.**
- b. Todas las soluciones.**

Pregunta 4. ¿De qué forma se añadirán las restricciones de colocación?

- a. Añadiendo dos nuevos parámetros, uno para cada diagonal (siendo estos dos arrays booleanos) donde se comprobará si el sitio es factible.**
- b. Se recorrerá la matriz comprobando que su colocación es posible.
- c. Una función auxiliar donde se compruebe su funcionamiento.

SUMA DE SUBCONJUNTOS

Pregunta 1. Se trata de un problema del tipo:

- a. Permutaciones.
- b. Subconjuntos.**
- c. Otro.

Pregunta 2. ¿Qué versión desea desarrollar?

- a. Versión optimización.**
- b. Versión no optimización.**

Versión no optimización:

Pregunta 3. ¿Buscamos solución o soluciones con la misma cardinalidad?

- a. Sí
- b. No**

Pregunta 4. ¿Desea obtener una solución o todas las soluciones?

- a. Una solución.**
- b. Todas las soluciones.**

Pregunta 5. Aparte de la variable de la solución parcial será necesaria...

- a. Otra donde almacenemos la suma parcial.**
- b. No serán necesarias más variables.

Pregunta 6. ¿Es posible añadir alguna condición para dejar de buscar soluciones en caso de no poder encontrar una factible?

- a. Sí, cuando se sobrepase el número que estamos buscando.**
- b. No, no sería posible.

Versión optimización:

Pregunta 3. Aparte de la variable de la solución parcial será necesaria...

- a. Otra donde almacenemos la suma parcial.**
- b. No serán necesarias más variables.

Pregunta 4. ¿Es posible añadir alguna condición para dejar de buscar soluciones en caso de no poder encontrar una factible?

- a. Sí, cuando se sobrepase el número que estamos buscando.**
- b. No, no sería posible.

MOCHILA 0-1

Pregunta 1. Se trata de un problema del tipo:

- a. Permutaciones.
- b. Subconjuntos.**
- c. Otro.

Pregunta 2. ¿Qué versión desea desarrollar?

- a. Solución binaria.**
- b. Solución parcial con índices.**

Pregunta 3. ¿Se trata de un problema de optimización?

- a. Sí**
- b. No

Pregunta 4. ¿Cuáles son las variables restantes para optimizar el algoritmo?

- a. Una variable donde iremos actualizando el valor de la capacidad (peso) restante.
- b. Una variable donde iremos actualizando el valor de la solución parcial.
- c. Una variable donde actualizar el valor de la capacidad (peso) restante y otra donde actualizar el de la solución parcial.**

Pregunta 5. ¿Es posible añadir alguna condición para dejar de buscar soluciones en caso de no poder encontrar una factible?

- a. Sí, cuando se sobrepase la capacidad (peso) máxima.**
- b. No, no sería posible.

Únicamente para solución binaria:

Pregunta 6. ¿Desea realizar podas adicionales (descartar soluciones que nunca puedan alcanzar el valor óptimo)?

- a. Sí**
- b. No**

SUDOKU

Pregunta 1. Se trata de un problema del tipo:

- a. Permutaciones.
- b. Subconjuntos.
- c. Otro.**

Pregunta 2. ¿Cómo será la solución parcial?

Será la propia matriz S a rellenar.

- b. Declararemos una nueva matriz que iremos rellenando.

Pregunta 3. ¿Se trata de un problema de optimización?

- a. Sí
- b. No**

Pregunta 4. Sabiendo que la forma de recorrido del sudoku será por filas (desde la 0 hasta la 8), habremos encontrado una solución cuando...

a. La variable que indique la fila sea igual a 9.

- b. La variable que indique la columna sea igual a 9.

Pregunta 5. ¿Qué pasaría si $S[\text{fila}, \text{columna}]$ es distinto de cero (la celda no es vacía)?

- a. Paramos la búsqueda ya que sería imposible encontrar solución.
- b. Procederíamos de la misma manera que si fuese igual a 0.

c. Como la celda ya está ocupada por los números originales (no se puede modificar), debemos tratar de expandir la solución parcial a partir de la siguiente celda.

Pregunta 6. Si $S[\text{fila}, \text{columna}]$ es igual a 0 (la celda está vacía), entonces...

a. Generaremos los candidatos (números del 1 al 9) que podrían aparecer en la celda y, en caso de que sean válidos, expandiremos la solución parcial.

b. Generaremos los candidatos (números del 1 al 9) que podrían aparecer en la celda y expandiremos la solución parcial.

Pregunta 7. ¿Se puede acelerar el algoritmo utilizando estructuras de datos adicionales?

a. Sí, utilizando matrices booleanas que indiquen si un dígito está en una fila, columna o caja.

- b. No

LABERINTO

Pregunta 1. Se trata de un problema del tipo:

- a. Permutaciones.
- b. Subconjuntos.
- c. Otro.**

Pregunta 2. ¿Cómo será la solución parcial?

- Será la propia matriz L a rellenar.**
- b. Declararemos una nueva matriz que iremos rellenando.

Pregunta 3. ¿Se trata de un problema de optimización?

- a. Sí
- b. No**

Pregunta 4. ¿Desea obtener una solución o todas las soluciones?

- a. Una solución.**
- b. Todas las soluciones.**

Pregunta 5. Sabremos que ha finalizado el ejercicio cuando...

- a. Cuando el número de fila sea igual al número de columna.
- b. Cuando estemos en la casilla de salida.**
- c. Cuando el número de la fila y el número de la columna sean $n-1$ y $m-1$, siendo $m \times n$ las dimensiones de la matriz.

Pregunta 6. ¿Cuántos candidatos hay?

- a. 2.**
- b. 4.
- c. 8.

Pregunta 7. Cuando obtengamos un candidato habrá que comprobar que sea válido. Esto es:

- a. Que esté dentro de la matriz.
- b. Que la casilla esté vacía.
- c. Que esté dentro de la matriz y la casilla esté vacía.**

Pregunta 8. ¿Es necesario marcar la celda como vacía ('E') si no se ha encontrado solución?

- a. Sí.**
- b. No.

SALTO DEL CABALLO

Pregunta 1. Se trata de un problema del tipo:

- a. Permutaciones.
- b. Subconjuntos.
- c. Otro.**

Pregunta 2. ¿Cómo será la solución parcial?

Una matriz de tamaño $n \times n$ donde colocaremos el orden de las posiciones del caballo.

- b. Un array donde iremos poniendo la casilla a la que salta el caballo.

Pregunta 3. ¿Se trata de un problema de optimización?

- a. Sí
- b. No**

Pregunta 4. Sabremos que ha finalizado el ejercicio cuando...

a. Cuando se sobrepase(en número de iteraciones) el cuadrado de la longitud de la matriz.

b. Cuando se sobrepase(en número de iteraciones) el doble de la longitud de la matriz.

Pregunta 5. ¿Cuántos candidatos hay?

- a. 2.
- b. 4.
- c. 8.**

Pregunta 6. Cuando obtengamos un candidato habrá que comprobar que sea válido. Esto es:

- a. Que esté dentro del tablero.
- b. Que no se haya visitado ya esa casilla.
- c. Que esté dentro del tablero y no se haya visitado ya esa casilla.**

Pregunta 7. ¿Es necesario marcar la celda vacía si no se ha encontrado solución?

- a. Sí.**
- b. No.