17 de Octubre del 2025

Avance de proyecto

Estructuras de datos

ITIID 4-1

Integrantes:

- Coronado Sánchez Angel Gabriel
- Olazarán López Jared de Jesús
- Reta Limas Georgina
- Torres Martínez Melissa Jazmín

UNIVERSIDAD POLITÉCNICA DE VICTORIA

University Time Tabling

Definición del University Timetabling

La programación de horarios universitarios representa un desafío significativo en la gestión académica, requiriendo la consideración de múltiples factores para optimizar la utilización de recursos y maximizar la satisfacción estudiantil. Este proceso implica la creación de un calendario académico que no solo define las fechas de inicio y fin de los términos, sino que también abarca eventos cruciales como las aperturas de registros, los plazos de retiro y reembolso, y las ceremonias de graduación.

La complejidad de esta tarea radica en la necesidad de equilibrar eficientemente las demandas de los recursos educativos, tales como la asignación de salones de clase, la disponibilidad del profesorado y las preferencias de los estudiantes, todo mientras se asegura una experiencia académica cohesiva y satisfactoria.

Uno de los principales desafíos de la programación de horarios es la eficaz asignación de recursos limitados. Las instituciones educativas, como el Glendale Community College, han implementado mecanismos para priorizar las solicitudes de recursos, clasificándolas en solicitudes de asignación de personal y solicitudes no relacionadas con la contratación. Este enfoque estructurado permite a las universidades manejar eficazmente las solicitudes de recursos, asegurando que las necesidades críticas sean atendidas de manera prioritaria.

La programación de horarios no solo es crucial para la administración eficiente de recursos, sino también para mejorar la satisfacción estudiantil, como lo demuestra la tendencia ascendente en la eficiencia de asignación de recursos y la satisfacción estudiantil del 75.0% y 80.0% en 2020 al 85.0% y 90.0% en 2024, respectivamente. Estas mejoras reflejan un compromiso con la optimización continua del proceso de programación de horarios para satisfacer las expectativas de los estudiantes y el personal académico.

La relación entre la programación de horarios universitarios y el lenguaje de programación C++ es de particular interés, dado que C++ ofrece herramientas poderosas para desarrollar algoritmos eficientes capaces de abordar problemas complejos de optimización. C++ es ampliamente utilizado en la implementación de soluciones algorítmicas debido a su eficiencia y capacidad para manejar estructuras de datos complejas, lo cual es esencial en la programación de horarios, donde se deben considerar múltiples variables y restricciones.

Relación con matrices y grafos.

En el contexto de la programación de horarios, las estructuras de datos como matrices y gráficos juegan un papel fundamental. Las matrices pueden utilizarse para representar la disponibilidad de recursos a lo largo del tiempo, permitiendo una fácil manipulación y acceso a los datos necesarios para tomar decisiones informadas. Los grafos, por otro lado, son útiles para modelar las interacciones y restricciones entre diferentes eventos y recursos, permitiendo una representación visual y algorítmica de las dependencias y relaciones que deben ser gestionadas en el proceso de programación.

Relación con Backtracking

Para abordar el problema de la programación de horarios universitarios, se pueden explorar diversas propuestas de solución. El backtracking es un enfoque sistemático que explora todas las posibilidades de asignación de horarios, retrocediendo y deshaciendo decisiones cuando se detectan conflictos. Aunque este método garantiza encontrar una solución óptima, su desventaja radica en su potencial ineficiencia en problemas de gran escala, donde el espacio de búsqueda puede ser inmenso.

Relación con Brute Force

Por otro lado, el algoritmo de fuerza bruta, aunque conceptualmente simple, implica evaluar todas las combinaciones posibles de horarios para identificar la solución óptima. Sin embargo, debido a su naturaleza exhaustiva, este enfoque es generalmente impracticable para problemas complejos debido a su alta demanda computacional.

Relación con Algoritmos Genéticos

Los algoritmos genéticos ofrecen una alternativa prometedora, inspirada en los procesos de selección natural, para encontrar soluciones aproximadas a problemas complejos como la programación de horarios. Estos algoritmos operan mediante la generación de poblaciones de soluciones potenciales, aplicando operaciones de selección, cruce y mutación para evolucionar hacia una solución óptima. En C++, los algoritmos genéticos pueden ser implementados de manera eficiente gracias a la capacidad del lenguaje para manejar operaciones de bajo nivel y estructuras de datos complejas, permitiendo así una exploración efectiva del espacio de soluciones.

En resumen, la programación de horarios universitarios es un área crítica que requiere un enfoque meticuloso y bien estructurado para garantizar una gestión eficaz de los recursos y una alta satisfacción estudiantil. La aplicación de C++ en este dominio, junto con el uso de estructuras de datos como matrices y gráficos, proporciona un marco robusto para abordar los desafíos inherentes a este problema. Las soluciones algorítmicas, desde el backtracking y la fuerza bruta hasta los algoritmos genéticos, ofrecen diferentes niveles de eficiencia y eficacia, cada uno con sus propias ventajas y limitaciones en la búsqueda de la optimización de horarios en entornos académicos complejos.

Matrices

Introducción a las matrices en programación

Las matrices son estructuras fundamentales en programación que permiten la organización y el procesamiento de datos complejos de manera eficiente. En el contexto del problema de programación de horarios universitarios, su uso es particularmente relevante debido a su capacidad para representar y manipular grandes volúmenes de información estructurada. Una matriz se define como un conjunto de filas y columnas organizadas de tal manera que cada elemento dentro de la matriz puede ser identificado por sus coordenadas (índice de fila

e índice de columna). Las dimensiones de la matriz, que son el número de filas y columnas, determinan su tamaño y capacidad para contener datos.

Las matrices pueden contener diferentes tipos de datos, incluyendo valores numéricos, categóricos o incluso booleanos, dependiendo de la naturaleza de la información que se desea representar. Las operaciones básicas que se pueden realizar con matrices incluyen la suma, resta, multiplicación y transposición. Estas operaciones permiten manipular los datos de manera que se puedan extraer conclusiones útiles o realizar cálculos necesarios para la implementación de soluciones algorítmicas.

En el contexto de la programación de horarios universitarios, las matrices ofrecen una forma estructurada y sistemática de organizar información sobre cursos, aulas, profesores y horarios. Al representar cada uno de estos elementos como una matriz, se puede realizar un seguimiento eficiente de la disponibilidad y restricciones de los recursos, lo cual es esencial para el éxito del proyecto.

Además de las matrices, los grafos juegan un papel crucial en la programación de horarios universitarios. Un grafo es una estructura matemática que consiste en un conjunto de nodos conectados por artistas. En el ámbito de la programación de horarios, los nodos pueden representar cursos, profesores, aulas o bloques de tiempo, mientras que los aristas representan las relaciones o restricciones entre ellos. Esta representación gráfica permite modelar problemas complejos de manera más intuitiva y facilitar la aplicación de algoritmos de optimización para encontrar soluciones eficientes

Los grafos son especialmente útiles para representar conflictos de horarios. Por ejemplo, si dos cursos no pueden dictarse al mismo tiempo debido a que comparten el mismo profesor o aula, se puede representar esta restricción como una arista entre los nodos correspondientes en el grafo. Al utilizar técnicas de coloración de grafos, se pueden asignar diferentes recursos a cada curso de manera que se minimicen los conflictos y se optimice el uso de los recursos disponibles.

El uso combinado de matrices y grafos en la programación de horarios universitarios permite abordar el problema desde dos perspectivas complementarias: la estructuración eficiente de datos y la modelación de relaciones y restricciones complejas. Ambas herramientas son esenciales para desarrollar algoritmos que optimicen la asignación de recursos, asegurando que las restricciones de tiempo y espacio se cumplan de manera efectiva. Además, proporciona una base sólida sobre la cual se pueden implementar soluciones algorítmicas que mejoren la eficiencia y la calidad de la programación de horarios en un entorno universitario.

En conclusión, las matrices ofrecen una metodología para organizar datos de estructura moderada y eficiente, mientras que los grafos permiten modelar relaciones complejas entre esos datos. La implementación del problema de programación de horarios universitarios se beneficia de numerosas herramientas, permitiendo no solo la representación clara de datos, sino también la aplicación de algoritmos avanzados para resolver conflictos y optimizar la asignación de recursos [1]. Estas técnicas constituyen la base sobre la cual se puede construir un sistema robusto y efectivo para la programación de horarios en instituciones educativas, facilitando la gestión de recursos y mejorando la experiencia académica para estudiantes y profesores por igual.

Algoritmos de manipulación de matrices en C++

El problema de programación de horarios universitarios, conocido en inglés como University Timetabling Problem (UTTP), es un desafío común en el ámbito académico que involucra la asignación de una serie de eventos (como clases y exámenes) a un conjunto de recursos limitados (como aulas y profesores), cumpliendo con ciertas restricciones. Este problema es esencialmente una tarea de optimización combinatoria y puede ser abordada mediante diversas técnicas algorítmicas, entre las cuales se encuentran el uso de matrices y gráficos.

Las matrices son estructuras de datos fundamentales que permiten representar y manipular conjuntos de datos organizados en filas y columnas. En el contexto de la programación de horarios, las matrices pueden ser utilizadas para modelar la asignación de tiempos y recursos. Por ejemplo, una matriz podría tener filas que representen diferentes aulas y columnas que representen diferentes períodos de tiempo. Cada celda de la matriz podría indicar si un aula está ocupada o no en un período determinado.

El uso de matrices en C++ para resolver problemas de programación de horarios requiere un manejo eficiente de sus operaciones básicas, como la inicialización, acceso y modificación de sus elementos. Un ejemplo típico de manipulación de matrices en C++ es la implementación de matrices diagonales, donde sólo los elementos de la diagonal principal son no nulos, lo cual puede ser útil para ahorrar espacio[5]. La declaración de una matriz diagonal en C++ podría involucrar el uso de un array unidimensional para almacenar únicamente los elementos necesarios, aprovechando así la indexación eficiente de los arrays en C++.

Por otro lado, los grafos ofrecen una representación más flexible y visual de los problemas de programación de horarios. En un grafo, los nudos pueden representar eventos o recursos, mientras que los aristas pueden representar conflictos o restricciones entre ellos. Un enfoque común es utilizar grafos bipartitos para modelar la asignación de aulas a clases, donde un conjunto de nodos representa aulas y el otro conjunto representa clases, y los aristas indican la posibilidad de asignación.

Para ilustrar cómo se pueden utilizar grafos en la programación de horarios universitarios en C++, consideramos el siguiente ejemplo. Imaginemos que tenemos un conjunto de clases C1, C2, C3 y un conjunto de aulas A1, A2. Podemos definir un grafo representa bipartito donde los nodos en un conjunto representan las clases y los nodos en el otro conjunto las aulas. Si una clase puede llevarse a cabo en un aula, se dibuja un arista entre los nodos correspondientes. La programación de horarios puede entonces formularse como un problema de emparejamiento máximo en grafos bipartitos, donde buscamos asignar el máximo número de clases a aulas sin conflictos.

Grafos

Concepto y tipos de grafos

En el ámbito de la programación de horarios universitarios, es esencial comprender y utilizar estructuras de datos que permitan modelar de manera efectiva las restricciones y requisitos

de los horarios académicos. Dos estructuras fundamentales en este contexto son las matrices y los grafos. Aunque ambos tienen aplicaciones distintivas, los grafos ofrecen una representación más natural y flexible para abordar problemas complejos como el "University Timetabling Problem" (UTTP).

Los grafos son estructuras de datos compuestas por nodos (también llamados vértices) y aristas (o enlaces) que conectan pares de nodos. Estas estructuras se utilizan para modelar relaciones y conexiones entre diferentes entidades. En el contexto de UTTP, los nodos pueden representar cursos, profesores o salones, mientras que los aristas indican la relación entre estos elementos, como la coincidencia de horarios o la necesidad de un recurso compartido.

Existen varios tipos de grafos, cada uno con características y aplicaciones específicas. Los grafos pueden ser dirigidos o no dirigidos. En un grafo dirigido, los aristas tienen una dirección, lo cual es útil para modelar relaciones asimétricas, como la dependencia de un curso respecto a otro. Por otro lado, los grafos no dirigidos son útiles para representar relaciones bidireccionales o simétricas, como la posibilidad de programar dos cursos en el mismo salón.

Una de las principales aplicaciones de los grafos en la programación de horarios universitarios es la detección y resolución de conflictos. Por ejemplo, un grafo de conflictos puede ser construido donde cada nodo representa un curso y un arista entre dos nodos indica que esos cursos no pueden ser programados al mismo tiempo. Este tipo de grafo es crucial para encontrar una asignación de horarios que minimice las colisiones de clases.

En el contexto de programación, los grafos se pueden implementar mediante estructuras de datos como listas de adyacencia o matrices de adyacencia. La elección entre estas dos representaciones depende de la densidad del grafo. Para grafos densos, las matrices de adyacencia son más eficientes, mientras que para grafos dispersos, las listas de adyacencia ofrecen un mejor rendimiento.

Otro concepto relevante en el uso de grafos para la programación de horarios es el de camarillas. Una camarilla en un grafo es un subgrafo completo donde cada par de nodos está conectado por un arista. Identificar camarillas es útil para la programación de horarios, ya que un conjunto de cursos que forman una camarilla no puede ser programado en el mismo intervalo de tiempo sin causar conflictos.

Los algoritmos de búsqueda y recorrido de grafos, como el algoritmo de Dijkstra para encontrar caminos mínimos o el algoritmo de coloreado de gráficos para la asignación de recursos, son herramientas valiosas en la resolución del UTTP. Estos algoritmos permiten explorar diferentes configuraciones de horarios y seleccionar la más óptima bajo ciertas restricciones.

Backtracking

Qué es el Backtracking y sus bases.

El algoritmo de backtracking es una técnica fundamental en la resolución de problemas computacionales que se enfoca en explorar soluciones potenciales construyéndo las de manera incremental. Este método abandona una solución candidata tan pronto como determina que no puede completarse de manera válida. El backtracking se basa en principios como la recursividad y los árboles de decisión, los cuales son cruciales para navegar sistemáticamente a través del espacio del problema.

El concepto de backtracking se asocia con la resolución de problemas de satisfacción de restricciones, donde se generan candidatos para las soluciones y se evalúan en función de su viabilidad. Cuando un candidato no puede llevar a una solución válida, el algoritmo se "retrocede" y prueba una nueva ruta. Este proceso iterativo continúa hasta encontrar una solución óptima o hasta que todas las posibilidades hayan sido exploradas.

Al trabajar con el backtracking, es importante considerar varios aspectos esenciales. En primer lugar, la definición clara del problema y sus restricciones es fundamental, ya que el algoritmo debe evaluar la validez de cada candidato con respecto a estas restricciones. Además, la eficiencia del backtracking puede mejorarse significativamente mediante la implementación de técnicas de poda, que eliminan ramas del árbol de decisión que no pueden llevar a una solución viable. Otro aspecto crucial es la elección del orden en el que se prueban las opciones, lo cual puede influir en el tiempo de ejecución del algoritmo. Un ordenamiento adecuado puede reducir el número de caminos explorados y, por fin, optimizar el proceso.

Para ilustrar el funcionamiento del backtracking, consideramos un ejemplo básico de aplicación en C++. Supongamos que queremos resolver el problema de las n-reinas, donde se busca colocar n reinas en un tablero de ajedrez de n x n sin que se amenacen entre sí. El siguiente código muestra cómo el retroceso puede implementarse para este problema:

```
#include <iostream>
#include <vector>

bool esSeguro(const std::vector<std::vector<int>>& tablero, int fila, int columna, int n) {
    // Verificar la fila
    for (int i = 0; i < columna; ++i)
        if (tablero[fila][i])
            return false;

    // Verificar la diagonal superior izquierda
    for (int i = fila, j = columna; i >= 0 && j >= 0; --i, --j)
        if (tablero[i][j])
            return false;

    // Verificar la diagonal inferior izquierda
```

```
    for (int i = fila, j = columna; i < n && j >= 0; ++i, --j)
        if (tablero[i][j])
            return false;

    return true;
}

bool resolverNReinas(std::vector<std::vector<int>>& tablero, int columna, int n) {
    if (columna >= n)
        return true;

    for (int i = 0; i < n; ++i) {
        if (esSeguro(tablero, i, columna, n)) {
            tablero[i][columna] = 1;
            if (resolverNReinas(tablero, columna + 1, n))
                return true;
            tablero[i][columna] = 0; // retroceder
        }
    }
    return false;
}

void mostrarTablero(const std::vector<std::vector<int>>& tablero) {
    for (const auto& fila : tablero) {
        for (int valor : fila)
            std::cout << valor << " ";
        std::cout << "\n";
    }
}

int main() {
    int n = 8; // Ejemplo para 8 reinas
    std::vector<std::vector<int>> tablero(n, std::vector<int>(n, 0));

    if (resolverNReinas(tablero, 0, n))
        mostrarTablero(tablero);
    else
        std::cout << "No hay solución\n";

    return 0;
}
```

Este código demuestra cómo se utiliza el backtracking para verificar sistemáticamente cada posición en el tablero y retroceder cuando se encuentra una posición no segura, hasta que se encuentra una disposición válida para todas las reinas.

El retroceso es recomendable en situaciones donde el espacio de soluciones es pequeño o donde las restricciones permiten podar significativamente el espacio de búsqueda. Es ideal

para problemas como el sudoku, los laberintos y los problemas de satisfacción de restricciones, donde las se construyen de manera incremental y las restricciones pueden aplicarse directamente para eliminar candidatos no válidos.

Sin embargo, el retroceso puede no ser la mejor opción en casos donde el espacio de soluciones es muy grande y la poda no reduce significativamente el número de posibilidades. En estos escenarios, el algoritmo puede volverse ineficiente, requiriendo un tiempo de ejecución prohibitivo. En tales casos, puede ser más adecuado considerar técnicas alternativas como algoritmos de programación dinámica o heurísticas más avanzadas.

En la actualidad, el backtracking se utiliza combinado en aplicaciones que requieren la generación de todas las permutaciones posibles de un conjunto de elementos o la búsqueda de combinaciones específicas que cumplan con ciertas condiciones. Además, el retroceso es fundamental en la teoría de gráficos, en la resolución de problemas de satisfacción de restricciones en inteligencia artificial, y en la optimización de procesos donde las soluciones deben cumplir con múltiples restricciones simultáneamente.

La teoría adicional relevante incluye el concepto de poda, que es una técnica para reducir el espacio de búsqueda eliminando caminos que no pueden llevar a una solución. También es importante la comprensión de estructuras de datos como los árboles de decisión, que facilitan la representación del espacio de búsqueda y permiten una navegación sistemática a través de las posibilidades. Estas herramientas teóricas son esenciales para implementar el backtracking de manera eficaz y optimizar su rendimiento en la resolución de problemas complejos.

En conclusión, el algoritmo de backtracking es una técnica poderosa para la resolución de problemas que requiere la evaluación de múltiples soluciones potenciales bajo restricciones específicas. Su aplicación es amplia y variada, siendo especialmente útil en problemas de satisfacción de restricciones y en contextos donde la poda puede reducir efectivamente el espacio de búsqueda. Sin embargo, su uso debe evaluarse cuidadosamente en función del tamaño del espacio de soluciones y la eficiencia de la poda aplicable.

Principios y Fundamentos del Backtracking

Principio	Descripción
Recursividad	Una técnica para resolver problemas mediante la descomposición en subproblemas más pequeños, aplicable en el algoritmo de backtracking.
Árboles de Decisión	Estructuras que ayudan a navegar sistemáticamente a través del espacio de problemas, fundamentales en el backtracking.

Aspectos importantes al trabajar con Backtracking

El **Backtracking** es una técnica de programación utilizada para resolver problemas de optimización y búsqueda en espacios de soluciones complejos. Se basa en explorar todas las posibles soluciones de manera sistemática, abandonando aquellas que no cumplen con los criterios establecidos, y retrocediendo en la búsqueda cuando se identifica una solución no viable. Este método es particularmente eficaz en problemas combinatorios donde la cantidad de posibles soluciones es extensa, ya que permite navegar por estos espacios de solución mediante la poda temprana de caminos inválidos..

Al trabajar con Backtracking, es esencial considerar varios aspectos clave para optimizar su uso. Primero, la ***poda de ramas*** es una estrategia crucial para mejorar el rendimiento del algoritmo. Identificar ramas inútiles que pueden ser eliminadas ayuda a reducir el tiempo de ejecución y el uso de recursos. La poda efectiva permite descartar partes del espacio de búsqueda que no contribuyen a la solución, lo que es particularmente útil en problemas donde el número de combinaciones posibles es extremadamente grande.

Otro aspecto importante es la optimización de la memoria. El uso eficiente de estructuras de datos puede reducir significativamente el consumo de memoria al implementar Backtracking. Al manejar adecuadamente las estructuras de datos, se pueden minimizar los costos de almacenamiento y aumentar la eficiencia del algoritmo, permitiendo que se aborden problemas más complejos y extensos sin agotar los recursos disponibles.

Es fundamental identificar problemas adecuados para aplicar Backtracking de manera eficiente. Esta técnica es especialmente recomendable en situaciones donde las soluciones se pueden descomponer en subproblemas más pequeños y donde cada decisión puede ser revertida si no conduce a una solución válida. Los ejemplos comunes incluyen la resolución de rompecabezas como el Sudoku, problemas de satisfacción de restricciones y ciertos tipos de problemas de optimización combinatoria.

Sin embargo, Backtracking puede no ser la mejor opción en situaciones donde el espacio de soluciones es demasiado extenso y la poda de ramas no es efectiva para reducir el número de caminos explorados. En tales casos, el Backtracking puro puede volverse ineficiente, y sería recomendable considerar otras técnicas de optimización como algoritmos heurísticos o aproximaciones probabilísticas que puedan brindar soluciones satisfactorias en menos tiempo.

Las aplicaciones comunes del Backtracking en la actualidad incluyen la solución de problemas de satisfacción de restricciones en inteligencia artificial, planificación de rutas, y la generación de combinaciones en problemas de diseño. Además, el Backtracking es fundamental en la teoría de gráficos para encontrar caminos y ciclos en estructuras complejas.

En conclusión, el Backtracking es una herramienta poderosa para abordar problemas complejos, proporcionando una metodología para explorar grandes espacios de soluciones de manera eficiente. Sin embargo, su eficacia depende de la capacidad de podar ramas y optimizar el uso de recursos, así como de la naturaleza del problema en cuestión. En

situaciones donde estas condiciones no se cumplen, es recomendable evaluar otras técnicas complementarias o alternativas.

Aspecto	Descripción
Técnicas de Optimización de Memoria	Uso eficiente de estructuras de datos para reducir el consumo de memoria al implementar Backtracking
Estrategias de Poda de Ramas	Identificación de ramas inútiles que se pueden eliminar para optimizar el rendimiento
Problemas Adecuados para Backtracking	Casos donde Backtracking puede aplicarse de manera eficiente, mejorando la solución de problemas complejos

Pasos de como realizar un código de Backtracking en C++

El Backtracking es una técnica algorítmica que se utiliza para resolver problemas mediante la exploración de todas las posibles soluciones y la reversión de decisiones cuando se encuentra un camino no viable. Este enfoque se basa en la idea de construir soluciones incrementales, descartando aquellas que no cumplen con los criterios del problema, y retrocediendo en el proceso para explorar nuevas alternativas. La técnica se implementa generalmente mediante el uso de funciones recursivas, que permiten al algoritmo navegar por el espacio de manera eficiente, explorando cada posibilidad hasta encontrar una solución válida o determinar que ninguna solución es posible.

Al trabajar con Backtracking, es esencial considerar varios aspectos importantes. Primero, es fundamental definir claramente los casos base para la función recursiva, ya que estos determinan cuándo el algoritmo debe detenerse y regresar. Los casos base actúan como condiciones de parada, evitando que la función recursiva entre en un ciclo infinito. Segundo, es crucial establecer un criterio de aceptación que permita al algoritmo decidir si una solución parcial puede desarrollarse hacia una solución completa. Tercero, se debe implementar un mecanismo para revertir decisiones y explorar caminos alternativos, lo cual es el núcleo del proceso de Backtracking

El Backtracking es recomendable en situaciones donde el espacio de soluciones es manejable y puede ser explorado completamente, como en problemas de búsqueda donde se requiere encontrar una solución óptima o completa. Los ejemplos típicos incluyen problemas de satisfacción de restricciones, acertijos y problemas de optimización combinatoria. Sin embargo, el Backtracking puede no ser la mejor opción cuando el espacio de soluciones es demasiado grande, lo que puede llevar a una complejidad temporal prohibitiva. En tales casos, se recomienda considerar enfoques alternativos como algoritmos greedy o heurísticos que puedan ofrecer soluciones aproximadas de manera más eficiente.

Teóricamente, el Backtracking se basa en el principio de exploración exhaustiva del espacio de soluciones, pero con la capacidad de reversionar decisiones para evitar caminos improductivos, lo que lo convierte en un enfoque versátil y poderoso para resolver una amplia gama de problemas. Sin embargo, es crucial tener en cuenta la eficiencia del algoritmo, especialmente en términos de tiempo y espacio, y considerar el uso de técnicas complementarias como la poda de ramas no prometedoras para mejorar el rendimiento.

Paso	Descripción
1	Configuración de la función recursiva
2	Definición de casos base
3	Exploración y abandono de caminos

Ejemplo básico de aplicación en C++

Un ejemplo básico que ilustra la aplicación de Backtracking en C++ es la resolución de un rompecabezas Sudoku. Este problema puede resolverse eficientemente mediante Backtracking, donde se intenta colocar números en las casillas vacías y se retrocede cuando se encuentra una contradicción. El siguiente código en C++ muestra cómo se puede implementar esta técnica:

```
#include <iostream>
using namespace std;

#define N 9

bool isSafe(int grid[N][N], int fila, int col, int num) {
   for (int x = 0; x < N; x++)
      if (grid[fila][x] == num || grid[x][col] == num)
         return false;

   int startRow = fila - fila % 3, startCol = col - col % 3;
   for (int i = 0; i < 3; i++)
      for (int j = 0; j < 3; j++)
         if (grid[i + startRow][j + startCol] == num)
            return false;

   return true;
}

bool solveSudoku(int grid[N][N], int fila, int col) {
   if (fila == N - 1 && col == N)
      return true;
```

```
    if (col == N) {
        fila++;
        col = 0;
    }

    if (grid[fila][col] > 0)
        return solveSudoku(grid, fila, col + 1);

    for (int num = 1; num <= N; num++) {
        if (isSafe(grid, fila, col, num)) {
            grid[fila][col] = num;
            if (solveSudoku(grid, fila, col + 1))
                return true;
            grid[fila][col] = 0;
        }
    }
    return false;
}

void printGrid(int grid[N][N]) {
    for (int r = 0; r < N; r++) {
        for (int d = 0; d < N; d++) {
            cout << grid[r][d] << " ";
        }
        cout << endl;
    }
}

int main() {
    int grid[N][N] = {
        {5, 3, 0, 0, 7, 0, 0, 0, 0},
        {6, 0, 0, 1, 9, 5, 0, 0, 0},
        {0, 9, 8, 0, 0, 0, 0, 6, 0},
        {8, 0, 0, 0, 6, 0, 0, 0, 3},
        {4, 0, 0, 8, 0, 3, 0, 0, 1},
        {7, 0, 0, 0, 2, 0, 0, 0, 6},
        {0, 6, 0, 0, 0, 0, 2, 8, 0},
        {0, 0, 0, 4, 1, 9, 0, 0, 5},
        {0, 0, 0, 0, 8, 0, 0, 7, 9}
    };

    if (solveSudoku(grid, 0, 0))
        printGrid(grid);
    else
        cout << "No existe ninguna solución";

    return 0;
}
```

Este código utiliza Backtracking para llenar las casillas vacías del Sudoku, verificando en cada paso si la colocación de un número es segura antes de continuar con la siguiente casilla. En caso de que una colocación no permita completar el Sudoku, se retrocede y se intenta con otro número.

.

El Backtracking es recomendable en situaciones donde el espacio de búsqueda es demasiado grande para ser explorado exhaustivamente por métodos iterativos o de fuerza bruta. Es útil en problemas de satisfacción de restricciones, rompecabezas como el Sudoku, problemas de asignación de tareas y problemas de optimización combinatoria. Sin embargo, puede no ser la mejor opción cuando el problema tiene una solución única que puede ser encontrada con un enfoque más directo o cuando el espacio de búsqueda es tan grande que incluso con poda, el tiempo de cómputo es prohibitivo.

Las aplicaciones comunes de Backtracking en la actualidad incluyen la generación de todas las permutaciones de un conjunto, el cálculo de todas las posibles combinaciones en problemas de mochila, y la resolución de problemas de satisfacción de restricciones como el Sudoku y el problema del laberinto. La teoría adicional relevante incluye el análisis de su complejidad temporal y espacial, así como técnicas de optimización como la poda de ramas y el uso de heurísticas para guiar el proceso de búsqueda.

En resumen, el Backtracking es una técnica poderosa para abordar problemas de búsqueda compleja y optimización, proporcionando una forma sistemática de explorar el espacio de soluciones. Aunque no siempre es la solución más eficiente, su capacidad para manejar problemas complejos con múltiples restricciones lo convierte en una herramienta valiosa en el arsenal de un programador experto.

Propuesta de solución de University TimeTabbling

El backtracking es una técnica de búsqueda exhaustiva que construye soluciones paso a paso y retrocede cada vez que se detecta una violación de las restricciones. En el contexto del UTP, el algoritmo intenta asignar horarios a los cursos de manera recursiva, verificando tras cada asignación si se cumplen las condiciones necesarias: que el profesor esté disponible, que el aula no esté ocupada y que el grupo no tenga solapamientos.

Idea general:

Representar los cursos, profesores, salones y horarios en estructuras de datos (por ejemplo, struct o class en C++).

- Crear una función recursiva que asigne un horario posible a un curso.
- Antes de confirmar la asignación, verificar todas las restricciones duras:
- El profesor no tiene otra clase a esa hora.
- El aula está libre.
- No hay solapamientos en las clases de un mismo grupo.

Si se viola alguna restricción, el algoritmo retrocede y prueba una nueva combinación.
Cuando todos los cursos están asignados correctamente, se obtiene una solución válida.

En C++, este método puede implementarse mediante funciones recursivas que usen estructuras como std::vector, std::map y matrices booleanas de disponibilidad. Su principal ventaja es que garantiza una solución correcta siempre que exista, siendo ideal para instancias pequeñas o para validar la eficacia de otros métodos. Sin embargo, su mayor desventaja es su ineficiencia en problemas grandes, ya que su complejidad es exponencial y el tiempo de ejecución crece drásticamente con el número de cursos y restricciones.

El backtracking es, por tanto, un método exacto y preciso, pero limitado en términos de escalabilidad.

Brute force

Definición y Fundamentos Teóricos

El método de fuerza bruta (brute force) es una técnica computacional clásica que consiste en explorar de manera exhaustiva todas las combinaciones posibles dentro del espacio de soluciones para un problema determinado con el fin de hallar la solución óptima o válida.

Este enfoque se basa en la premisa de que, al evaluar cualquier posible solución sin excepción, no existe riesgo de perder la opción correcta, garantizando así la completitud y la exactitud del resultado.

Desde el punto de vista teórico, la fuerza bruta es el paradigma más básico de algoritmos de búsqueda y forma parte esencial del estudio de problemas de tipo combinatorio y complejidad computacional, particularmente en la resolución de problemas NP-completos y NP-duros donde el espacio de soluciones es exponencialmente grande. Fue desarrollado durante los inicios de la computación para proporcionar un marco de referencia y base para sofisticados métodos posteriores.

En la teoría de algoritmos se acostumbra modelar la complejidad de búsquedas exhaustivas con expresiones del tipo O(b^d) , donde b es el **factor de ramificación** (cuántas opciones hay en cada paso) y ddd la **profundidad** de la decisión (cuántas decisiones se deben tomar). Esa forma expresa el crecimiento exponencial típico de la búsqueda exhaustiva.

Funcionamiento General y Ejemplos de Aplicación

El método funciona mediante una generación y evaluación sistemáticas de todas las posibles configuraciones. Se detallan los pasos típicos:

1. Enumerar el espacio de soluciones: Esto implica listar o generar de manera automatizada todas las posibles combinaciones (permutaciones, subconjuntos, asignaciones, etc.) acorde a la naturaleza del problema.
2. Evaluar cada solución candidata: Aplicando las restricciones y condiciones del problema para verificar si la solución es válida.
3. Almacenar y/o seleccionar la mejor solución: Conforme se avanzan las evaluaciones, se actualiza la solución actual óptima. Se puede detener al encontrar la primera solución válida o tras evaluar todas para hallar la mejor.

4. Entrega de resultados: Dependiendo del tipo de problema, se retorna la solución exacta, un conjunto de soluciones, o se indica que no existe solución válida.

Ejemplos Ilustrativos

- Búsqueda de contraseñas: Probar todas las combinaciones posibles hasta encontrar la correcta. Por ejemplo, una contraseña de 4 dígitos con números de 0-9 implica revisar hasta 10,000 combinaciones.
- Problema de las ocho reinas: Generar todas las posibles formas de colocar ocho reinas en un tablero 8x8 sin que se ataquen mutuamente y filtrar las posiciones válidas.
- Viaje del vendedor (TSP): Enumerar todas las permutaciones de ciudad para encontrar el recorrido de menor costo.
- Asignación de horarios (UTTP): Considerar todas las asignaciones posibles, sitios y tiempos para asignar clases, evaluando restricciones.

Análisis de Complejidad Temporal y Espacial

La característica más crítica de la fuerza bruta es su complejidad exponencial progresiva. Matemáticamente, si:

- n = número de variables o decisiones que el algoritmo debe tomar,
- k = número de opciones posibles por cada variable,

entonces el número de soluciones candidatas a evaluar será del orden:

$$O(k^n)$$

Esto implica que cada incremento en n multiplica el tamaño del espacio por un factor k, causando un crecimiento extremadamente rápido, lo que se traduce en un costo computacional potencialmente inabordable para problemas medianos o grandes.

Ejemplo numérico:

Para un problema con 6 variables binarias (0 o 1), hay 2^6=64 combinaciones; para 20 variables, hay 2^{20} =1,048,576 combinaciones, lo que ya eleva los tiempos de cómputo a muchos segundos o horas.

En cuanto a espacio, la fuerza bruta generalmente no almacena cada solución simultáneamente, sino que genera y evalúa de forma secuencial, pero en problemas más complejos la memoria para guardar estados intermedios o resultados se vuelve considerable.

Ventajas y Desventajas

Ventajas

- Simplicidad y robustez: Fácil de implementar, sin necesidad de estructuras complejas ni optimizaciones.
- Exactitud: Al evaluar todas las soluciones, garantiza hallar la solución óptima o detectar si no existe.
- Versatilidad: Puede aplicarse a cualquier problema con espacio finito de soluciones.

Desventajas

- Ineficiencia extrema: Exponencial crecimiento en tiempo y a veces en memoria, haciéndolo impráctico para problemas grandes.
- No aprovecha estructura: Ignora restricciones o características que podrían evitar explorar soluciones inválidas.
- Escalabilidad limitada: Con el aumento de las variables, la fuerza bruta se vuelve inviable.

Motivos por los que no es recomendable para el UTTP y problemas combinatorios grandes

El **University Timetabling Problem (UTTP)** consiste en generar horarios académicos asignando materias, profesores, aulas y franjas horarias, de manera que se cumplan múltiples restricciones simultáneamente. Entre estas restricciones se incluyen:

- **Conflictos de horario:** un mismo profesor no puede impartir dos materias al mismo tiempo, y los estudiantes no pueden asistir a dos clases simultáneas.

- **Disponibilidad de recursos:** no todas las aulas están disponibles en todos los horarios; algunos profesores tienen restricciones de disponibilidad.

- **Capacidad de aulas:** el número de alumnos por clase debe ajustarse a la capacidad máxima de la sala.

- **Restricciones adicionales:** requisitos curriculares, preferencias de profesores, distribución equilibrada de materias, y reglas específicas de departamentos o programas.

Dadas estas restricciones, el UTTP se convierte en un **problema combinatorio altamente complejo**, donde cada asignación posible de materia, aula, profesor y horario representa un punto en un espacio de soluciones extremadamente grande.

Por qué la Fuerza Bruta no es recomendable

El **método de fuerza bruta**, que consiste en generar y evaluar todas las combinaciones posibles para encontrar soluciones válidas, se enfrenta a limitaciones severas cuando se aplica al UTTP:

1. **Crecimiento exponencial del espacio de búsqueda:**
   Cada variable (materia, aula, profesor, horario) multiplica el número total de combinaciones posibles. Por ejemplo, si tenemos 50 materias, 20 aulas, 30 profesores y 40 franjas horarias, el número de configuraciones posibles es del orden $50 \times 20 \times 30 \times 40 = 1{,}200{,}000$combinaciones solo en la primera aproximación, sin considerar todas las restricciones combinadas, lo que hace que el número real de posibilidades válidas sea astronómico

2. **Tiempo computacional prohibitivo:**
   La exploración exhaustiva de todas las combinaciones crece exponencialmente con el tamaño del problema. En problemas de UTTP de tamaño real, el tiempo requerido para evaluar todas las posibles soluciones puede ser **del orden de años o siglos** incluso en computadores modernos. Esto lo hace completamente inviable para aplicaciones prácticas.

3. **Falta de escalabilidad:**
   La fuerza bruta **no escala**: pequeños incrementos en el número de materias, profesores o aulas generan aumentos dramáticos en el número de combinaciones, haciendo imposible adaptarse a universidades medianas o grandes.

4. **Ausencia de optimización:**
   La fuerza bruta no utiliza **heurísticas, poda de soluciones inválidas ni estrategias de reducción del espacio de búsqueda**. En contraste, técnicas como **backtracking, algoritmos genéticos o búsqueda local** permiten descartar rápidamente combinaciones que violan restricciones, reduciendo drásticamente el tiempo de cómputo y enfocándose en soluciones factibles.

5. **Evidencia en la literatura:**
   Numerosos estudios académicos sobre problemas NP-duros, entre los que se incluye el UTTP, demuestran que los métodos de fuerza bruta son inaplicables en escenarios reales. Investigaciones muestran que para horarios de tamaño mediano (p. ej., 100 materias, 50 profesores y 30 aulas), las soluciones mediante fuerza bruta requieren tiempos que exceden los límites razonables de procesamiento, mientras que algoritmos con **poda inteligente y heurísticas** logran soluciones óptimas o cercanas a óptimas en minutos u horas.

Conclusión

El método brute force es válido en:

- Problemas de pequeña escala.
- Contextos académicos y educativos para entender la búsqueda exhaustiva.
- Validación de modelos y benchmarking.

Sin embargo, en problemas reales y grandes como UTTP u otros NP-duros, es esencial utilizar técnicas avanzadas que reduzcan y guíen la búsqueda de soluciones, evitando la imposibilidad práctica que implica la fuerza bruta.

Genetic Algorithms

Definición y Fundamentos Teóricos

Los Algoritmos Genéticos (AG) son técnicas de optimización que se inspiran en los procesos de evolución biológica, basándose en los principios de la selección natural, la cruza (crossover), la mutación y la evaluación de aptitud (fitness) (CHIRIAC et al., 2023; (Szabó, 2023). Su origen se encuentra en la teoría de la evolución de Darwin, que establece que los individuos con características más adecuadas tienen mayor probabilidad de sobrevivir y reproducirse. Este proceso de evolución se traduce en la forma en que los AG exploran eficientemente un espacio de soluciones al combinar y modificar soluciones existentes para crear nuevas variaciones. Los modelos teóricos en evolución y complejidad sugieren que los AG pueden converger hacia soluciones óptimas en espacios de alta dimensión, aunque el tiempo de convergencia puede variar dependiendo de parámetros como la tasa de mutación y el tamaño de la población (Szabó, 2023).

Las ecuaciones comunes que se encuentran en la literatura incluyen la función de fitness, que se puede definir como (f(x)), donde (x) representa un cromosoma, y la probabilidad de selección de un individuo, que puede ser (P(x*i) \propto f(*xi)). Estos indicadores son esenciales para entender cómo los AG toman decisiones sobre qué soluciones reproducir y mutar, lo que es fundamental para el análisis del UTTP (Szabó, 2023).

Funcionamiento General y Ejemplos de Aplicación

El ciclo evolutivo de un AG consiste en varias etapas clave: generación de poblaciones iniciales, evaluación de fitness, selección, cruzamiento y mutación, seguido de nuevas generaciones (Szabó, 2023). Un cromosoma, que representa una solución potencial al problema, es construido mediante la codificación de los parámetros a ser optimizados. Los operadores genéticos como la selección por torneo y el cruce de un punto son comúnmente utilizados para crear descendencia a partir de padres seleccionados. Por ejemplo, en el contexto de la optimización combinatoria, se ha demostrado la eficacia de los AG aplicados al UTTP, donde varios estudios documentan la generación de horarios que maximizan la utilización de recursos y minimizan conflictos mediante la adaptación y optimización iterativa (Mohsin, 2022).

Un caso específico se presenta con el uso de AG hibridados con métodos de búsqueda local, fortaleciendo la capacidad de los AG para escapar de óptimos locales (Mohsin, 2022). Estos enfoques han mostrado resultados prometedores al comparar la calidad de las soluciones obtenidas con métodos más tradicionales como el backtracking.

Análisis de Complejidad Temporal y Espacial

Los costes computacionales de los AG son influenciados por diversos factores, incluyendo el tamaño de la población, el número de generaciones y la eficiencia de la función de fitness (Szabó, 2023). Elementos fundamentales que afectan el rendimiento incluyen la tasa de mutación y el esquema de selección, los cuales determinan la capacidad del algoritmo para explorar el espacio de búsqueda sin caer en una convergencia prematura (Szabó, 2023). Comparativamente, los AG tienden a ser más flexibles que los métodos de fuerza bruta,

debido a su capacidad de enfocarse en áreas prometedoras del espacio de soluciones, mientras que pueden ser menos eficientes que algunos métodos heurísticos para problemas muy específicos debido a la variabilidad inherente en los resultados (Szabó, 2023).

Ventajas y Desventajas

Los AG presentan varias ventajas en la resolución de problemas NP-duros, destacando su flexibilidad y capacidad para adaptarse a diferentes escenarios problemáticos. Esta adaptabilidad es crucial, especialmente en contextos como el UTTP, donde las restricciones y recursos pueden variar significativamente (Mohsin, 2022). Sin embargo, sus desventajas incluyen la posibilidad de convergencia prematura y la necesidad de un ajuste riguroso de parámetros. Además, su rendimiento puede ser inconsistente, lo que plantea desafíos en aplicaciones prácticas (Szabó, 2023).

Aplicaciones en el University Timetabling Problem (UTTP)

Varios estudios han documentado el uso exitoso de AG para la generación de horarios universitarios, en los que se han aplicado tanto algoritmos estándar como variantes híbridas (Mohsin, 2022). Estos modelos suelen combinar AG con técnicas adicionales como búsquedas locales o restricciones basadas en grafos, lo que mejora la efectividad en la creación de soluciones óptimas. Comparaciones de resultados han mostrado que los AG son competidores sólidos en este ámbito, a menudo superando métodos convencionales en términos de calidad de solución y viabilidad (Mohsin, 2022).

Conclusión y Perspectivas Actuales

En conclusión, los AG han demostrado ser una herramienta eficaz y escalable en la optimización del UTTP. Las tendencias actuales destacan la hibridación de AG con otras técnicas metaheurísticas, así como la utilización de enfoques de aprendizaje automático para mejorar la exploración y explotación del espacio de solución (Szabó, 2023). El potencial para futuras investigaciones incluye el desarrollo de algoritmos meméticos y adaptativos que mejoren aún más la eficiencia y efectividad de los AG en la resolución de problemas complejos.

Referencias

University TimeTabbling

- Academic Calendar | OHSU https://www.ohsu.edu/education/academic-calendar
- Resource Allocation | Glendale Community College https://www.glendale.edu/about-gcc/gcc-overview/institutional-effectiveness/strategic-planning/planning-processes/resource-allocation
- Examples of Academic Challenges in College - The Classroom https://www.theclassroom.com/examples-academic-challenges-college-14490.html
- Assessment under Resource Constraints | Current Issues in Education https://cie.asu.edu/ojs/index.php/cieatasu/article/view/1357

Matrices y grafos

- Matrix methods in C and C++ # | Data-Structures-and-Algorithms https://jfspps.github.io/Data-Structures-and-Algorithms/15_Matrices.html
- Applied Sciences | Free Full-Text | Graph-Based Modeling in Shop Scheduling Problems: Review and Extensions https://www.mdpi.com/2076-3417/11/11/4741/html
- (PDF) A University Timetabling System Based on Graph Colouring and Constraint Manipulation https://www.researchgate.net/publication/235439133_A_University_Timetabling_System_Based_on_Graph_Colouring_and_Constraint_Manipulation
- Graph Colorings | An Introduction to Algebraic Graph Theory https://www.geneseo.edu/~aguilar/public/notes/Graph-Theory-HTML/ch3-graph-colorings.html
- Graph Coloring in University Timetable Scheduling https://www.mecs-press.org/ijisa/ijisa-v15-n3/IJISA-V15-N3-2.pdf
- A solution to the university course timetabling problem using a hybrid method based on genetic algorithms https://www.redalyc.org/journal/496/49668129006
- A survey of approaches for university course timetabling problem https://www.sciencedirect.com/science/article/abs/pii/S0360835214003714

Backtracking

- Backtracking - Wikipedia https://en.m.wikipedia.org/wiki/Backtracking_search
- Backtracking search | Combinatorial Optimization Class Notes | Fiveable | Fiveable https://library.fiveable.me/combinatorial-optimization/unit-10/backtracking-search/study-guide/loLBBLyh50uYNxbv
- 5 Ways To Prune Trees And Promote Growth https://pbswisconsin.org/news-item/5-ways-to-prune-trees-and-promote-growth/
- What is a backtracking algorithm in C and how is it implemented? | BestDivision https://www.bestdivision.com/questions/what-is-a-backtracking-algorithm-in-c-and-how-is-it-implemented
- Recursion and Recursive functions in C++ | What is Recursion and Recursive Functions in C++ | lec#22 - YouTube https://m.youtube.com/watch?v=A7-KsT_1KNc

- Sudoku Solver with C++ using Backtracking | JAYPARMAR https://coderspacket.com/sudoku-solver-with-c-using-backtracking
- Backtracking in C | StudyMite https://studymite.com/blog/backtracking-in-c
- Backtracking Algorithm - Definition, Usecase, and Example https://intellipaat.com/blog/backtracking-algorithm/
- 10.1. Constraint Solving via Backtracking — YSC2229 2019 https://ilyasergey.net/YSC2229/week-10_backtracking.html
- Disadvantages of Backtracking https://iq.opengenus.org/disadvantages-of-backtracking/amp/
- 4.2. Comparing Algorithms — An Introduction to Data Structures and Algorithm A https://opendsa-server.cs.vt.edu/ODSA/Books/mcps/427-62ef763c-0959-4be3-903f-61fda03f170f/fall-2022/Period_7/html/AnalIntro.html
- Constraint Satisfaction Problems with Global Modular Constraints: Algorithms and Hardness via Polynomial Representations | SIAM Journal on Computing https://epubs.siam.org/doi/abs/10.1137/19M1291054
- Backtracking - Backtracking Throughout the book (see in particular Sections 3 and 11), we have - Studocu https://www.studocu.com/en-us/document/massachusetts-institute-of-technology/introduction-to-algorithms/backtracking/54081767
- What is Algorithmic Complexity? https://www.codingblocks.net/podcast/what-is-algorithmic-complexity/
- [Backtracking Technique and Examples | by Shivkumar More | Medium https://medium.com/@shivkumar.more19/backtracking-technique-and-examples-9eb768e86c42
- 2024 Trends In Ai Algorithms | Restackio https://www.restack.io/p/ai-development-trends-answer-2024-ai-algorithms
- https://www.tutor.com/cmspublicfiles/WWW/APA_References_Guide.pdf
- How to Research a Machine Learning Algorithm - MachineLearningMastery.com https://machinelearningmastery.com/how-to-research-a-machine-learning-algorithm

Brute force

- Alkhwaja, I. (2023). Descifrado de contraseñas con algoritmo de fuerza bruta y comparativa entre técnicas paralelas. *Applied Sciences*, *13*(10), 5979. https://doi.org/10.3390/app13105979
- Blum, C., & Roli, A. (2003). Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, *35*(3), 268–308. https://doi.org/10.1145/937503.937505
- Burke, E. K., & Newall, J. (2004). A multi-stage evolutionary algorithm for the timetable problem. *IEEE Transactions on Evolutionary Computation*, *7*(1), 69–82. https://doi.org/10.1109/TEVC.2002.808320
- Garey, M. R., & Johnson, D. S. (1979). *Computers and intractability: A guide to the theory of NP-completeness*. W. H. Freeman.
- GeeksforGeeks. (s.f.). El enfoque de fuerza bruta: ventajas y desventajas. Recuperado de https://www.geeksforgeeks.org/dsa/brute-force-approach-and-its-pros-and-cons/

- Korf, R. E. (1990). Deepening, a general algorithm for polynomial space search. *Artificial Intelligence*, *54*(2), 193–239. https://doi.org/10.1016/0004-3702(90)90033-C
- Orizaola Molinero, P. (2021). *Modelización y comparación del rendimiento entre algoritmos: fuerza bruta y ramificación y poda* [Trabajo de fin de grado]. Universidad de Salamanca. https://gredos.usal.es/bitstream/handle/10366/164079/TFG_Paula_Orizaola_Molinero.pdf
- Russell, S., & Norvig, P. (2021). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson.
- Trujillo, Y. (2025). *Trabajo de algoritmos de fuerza bruta para gestión de datos de vacunación* [Monografía]. Scribd.
- Universidad de Guadalajara. (s.f.). *Análisis de algoritmos: fuerza bruta, voraces, divide y vencerás, recursivos y programación dinámica* [Documento educativo]. https://dcc.cucei.udg.mx/sites/default/files/il355_analisis_de_algoritmos_extenso.pdf

Genetic Algorithms

- Abdipoor, S., Yaakob, R., Goh, S. L., & Abdullah, S. (2023). Meta-heuristic approaches for the University Course Timetabling Problem. *Intelligent Systems with Applications, 19*, 200253. https://doi.org/10.1016/j.iswa.2023.200253
- Abdelhalim, E. A., & El-Sedeek, M. S. (2016). A utilization-based genetic algorithm for solving the university course timetabling problem. *Alexandria Engineering Journal, 55*(2), 1395–1405. https://doi.org/10.1016/j.aej.2016.02.017
- Babaei, H., Karimpour, J., & Hadidi, A. (2015). A survey of approaches for university course timetabling problems. *Computers & Industrial Engineering, 86*, 43–59. https://doi.org/10.1016/j.cie.2014.11.010
- Lewis, R. (2008). A survey of metaheuristic-based techniques for university timetabling problems. *Journal of Scheduling, 12*(2), 129–143. https://doi.org/10.1007/s00291-007-0097-0
- Rezaeipanah, A., Saniei-Nezhad, A., Ebrahimnejad, A., & Javidi, M. M. (2021). A hybrid algorithm for the university course timetabling problem using the improved parallel genetic algorithm and local search. *Applied Intelligence, 51*, 7687–7710. https://doi.org/10.1007/s10489-020-01833-x
- Yu, E., & Sung, K.-S. (2002). A genetic algorithm for a university weekly courses timetabling problem. *International Transactions in Operational Research, 9*(6), 703–717. https://doi.org/10.1111/1475-3995.00383
- Jat, S. N., & Yang, S. (2009). A guided search genetic algorithm for the university course timetabling problem. *MISTA 2009 Conference Paper*. (Disponible en PDF).
- Alhuniti, O., Alwadain, A., Alhushail, A., & Aldaoud, A. (2020). Smart university scheduling using genetic algorithms. *Proceedings of the 2020 3rd International Conference on Computer Applications & Information Security (ICCAIS)*. https://doi.org/10.1145/3436829.3436873
- Xiang, K., Zhong, Z., Wang, S., & Li, M. (2024). Exact and heuristic methods for a university course scheduling problem. *Expert Systems with Applications, 238*, 123383. https://doi.org/10.1016/j.eswa.2024.123383

- Kheiri, A., & Keedwell, E. (2023). Real-world university course timetabling at ITC-2019. *Journal of Scheduling, 26*(4), 547–566. https://doi.org/10.1007/s10951-023-00801-w