PROYECTO FINAL

Análisis y Diseño de algoritmos

3CM1

Jorge Luis Vargas Hernández

17 de enero de 2024

0.1. Introducción

Este proyecto se centra en la implementación y optimización del algoritmo de backtracking en Python para resolver el problema de las N Reinas. Se explorarán dos estrategias específicas de optimización, y se realizará un análisis de rendimiento comparativo, incluyendo gráficas que ilustren el tiempo de ejecución de las diferentes versiones del código

0.2. Desarrollo

0.2.1. Implementación Básica

Desarrollar una implementación inicial del algoritmo de backtracking para resolver el problema de las N Reinas en Python. Verificar la corrección de la solución mediante pruebas exhaustivas.

```
def imprimir_tablero(tablero):
    for fila in range(len(tablero)):
        line = ""
        for columna in range(len(tablero)):
            if columna == tablero[fila]:
                line += "Q "
            else:
                line += ". "
        print(line)
    print("\n")

def resolver_n_reinas(tablero, fila, N):
    if fila == N:
        imprimir_tablero(tablero)
        return
    for columna in range(N):
        if es_seguro(tablero, fila, columna, N):
            tablero[fila] = columna
            resolver_n_reinas(tablero, fila + 1, N)
            tablero[fila] = -1  # backtrack

def n_reinas(N):
    tablero = [-1] * N
    resolver_n_reinas(tablero, 0, N)

n_reinas(8)

def verificar_soluciones(N):
    tablero = [-1] * N
    soluciones = []

    def verificar(tablero, fila, N):
```

```
        if fila == N:
            soluciones.append(tablero.copy())
            return
        for columna in range(N):
            if es_seguro(tablero, fila, columna, N):
                tablero[fila] = columna
                verificar(tablero, fila + 1, N)
                tablero[fila] = -1  # backtrack

    verificar(tablero, 0, N)

    for solucion in soluciones:
        for i in range(N):
            if not es_seguro(solucion, i, solucion[i], N):
                print("¡Error en la solución!")
                return
    print("Todas las soluciones son válidas.")

verificar_soluciones(8)
```

0.3. Optimizaciones Específicas

Estrategia 1: Aplicar una técnica de poda que reduzca la cantidad de nodos explorados durante la búsqueda, mejorando así el tiempo de ejecución. Estrategia 2: Investigar y aplicar heurísticas inteligentes que guíen la colocación inicial de las reinas para acelerar la convergencia a soluciones válidas.

0.3.1. Estrategia 1 Técnica de la Poda:

```
# Técnica de poda
def es_seguro(tablero, fila, columna, N):
    for i in range(fila):
        if tablero[i] == columna or \
           tablero[i] - i == columna - fila or \
           tablero[i] + i == columna + fila:
            return False
    return True

def imprimir_tablero(tablero):
    for fila in range(len(tablero)):
        line = ""
        for columna in range(len(tablero)):
            if columna == tablero[fila]:
                line += "Q "
            else:
                line += ". "
```

```
        print(line)
    print("\n")

def resolver_n_reinas(tablero, fila, N):
    if fila == N:
        imprimir_tablero(tablero)
        return
    for columna in range(N):
        if tablero[fila] == -1 and es_seguro(tablero, fila, columna, N):
            tablero[fila] = columna
            resolver_n_reinas(tablero, fila + 1, N)
            tablero[fila] = -1  # backtrack

# Ejemplo de uso para N = 4
tablero = [-1] * 4
resolver_n_reinas(tablero, 0, 4)
```

0.3.2. Estrategia 2 Heurísitcas Inteligentes:

```
def es_seguro(tablero, fila, columna, N):
    for i in range(fila):
        if tablero[i] == columna or \
           tablero[i] - i == columna - fila or \
           tablero[i] + i == columna + fila:
            return False
    return True

def imprimir_tablero(tablero):
    for fila in range(len(tablero)):
        line = ""
        for columna in range(len(tablero)):
            if columna == tablero[fila]:
                line += "Q "
            else:
                line += ". "
        print(line)
    print("\n")

def resolver_n_reinas(tablero, fila, N):
    if fila == N:
        imprimir_tablero(tablero)
        return
    for columna in range(N):
        if es_seguro(tablero, fila, columna, N):
            tablero[fila] = columna
            resolver_n_reinas(tablero, fila + 1, N)
            tablero[fila] = -1  # backtrack
```

```
def heuristica_colocacion_inicial(N):
    # Inicializar el tablero con todas las reinas fuera del tablero (-1)
    tablero = [-1] * N

    for fila in range(N):
        columna = 0
        while columna < N:
            if es_seguro(tablero, fila, columna, N):
                tablero[fila] = columna
                break
            columna += 1

        if columna == N:
            for i in range(N):
                tablero[i] = -1
            fila = -1
            continue

    return tablero

tablero_inicial = heuristica_colocacion_inicial(4)
resolver_n_reinas(tablero_inicial, 0, 4)
```

0.4. Análisis de Complejidad

Realizar un análisis detallado de la complejidad temporal y espacial de las implementaciones. Evaluar la eficiencia de cada estrategia de optimización y discutir las limitaciones de cada enfoque.

0.4.1. Implementación Básica

Este código llega hasta la solución utilizando un método de algoritmo de backtracking. Para análizar la complejidad del algoritmo es necesario desglosarlo entre las partes que lo componen.

La primera es la inicialización del tablero, esta se hace mediante la función N REINAS que inicializa todas las reinas en la primera columna, no es difícil darnos cuenta que la complejidad es O(N) puesto que el algoritmo tan solo recorre cada fila del tablero.

Respecto a la función RESOLVER N REINAS, nos damos cuenta que aqui se emplea el algoritmo del backtracking con la finalidad de explorar todas las posibles configuraciones del tablero. En el peor caso el algoritmo debe explorar todas las configuraciones posibles, lo que nos da lugar a una complejidad de O(N!).

Viendo el panorama completo, nos damos cuenta de que la función más importante aquí es la función resolver n reinas", puesto que eclipsa completamente a la función encargada de inicializar el tablero. Con todo lo anterior dicho, nos damos cuenta que la complejidad de este método de backtracking es exponencial en relación al tamaño del tablero.

0.4.2. Optimización Poda

El algoritmo se mantiene bastante similar a la implementación básica. En la funcion ES SEGURO, la función verifica si colocar una reina en una posición específica es seguro, comparando con las reinas ya colocadas en filas anteriores. En el peor caso, se deben realizar N comparaciones para cada fila. La complejidad de ES SEGURO en el peor caso es $O(N^2)$.

Es importante destacar que la técnica de poda en este caso no reduce la complejidad asintótica del problema, pero ayuda a evitar explorar ramas innecesarias durante el backtracking, mejorando el rendimiento en la práctica. La complejidad en la práctica puede variar dependiendo de la eficacia de las podas y optimizaciones aplicadas. En conclusión la complejidad total del algoritmo está dominada por el algoritmo de backtracking, y es O(N!).

0.4.3. Heuristicas Inteligentes

Este algoritmo hace una modificación respecto a los dos anteriores agregando una nueva función llamada HEURISTICA COLOCACION INICIAL, esta función se encarga de colocar a las reinas por fila asegurandose de que no existan conflicos. En el peor de los casos la función tendría que verificar cada una de las columnas para cada fila. En la práctica, la eficacia de la heurística puede reducir la cantidad de exploración realizada durante el backtracking, mejorando el rendimiento real del algoritmo. La complejidad teórica proporciona una visión general del comportamiento del algoritmo en términos de tamaño del problema, pero la optimización específica de la heurística puede afectar la eficiencia en casos prácticos. La complejidad total está dominada por el backtracking, ya que la heurística de colocación inicial es $O(N^2)$.

0.5. Documentación Técnica

0.5.1. Explicación de cada una de las funciones creadas

Este es Algoritmo que resuelve el clásico problema de las N reinas utilizando el algoritmo de backtracking, para dar la explicación completa usaremos el algoritmo más completo que tiene la incorporación de una heurística de colocación inicial para mejorar el rendimiento. Aquí se detallan las funciones clave y las estrategias implementadas:

Función ES SEGURO: Esta función verifica si es seguro colocar una reina en una posición específica del tablero. Compara las columnas y diagonales de las reinas ya colocadas para evitar conflictos.

Función IMPRIMIR TABLERO: Esta función imprime el tablero con las reinas colocadas de manera legible.

Función RESOLVER N REINAS: La función principal de backtracking. Intenta colocar reinas en cada fila de manera recursiva y retrocede cuando se encuentra una solución o se descarta una configuración.

Función N REINAS: Una función de conveniencia que inicializa el tablero y llama a resolver$_{nr}einasparaencontrareimprimirtodaslassoluciones$.

Función VERIFICAR SOLUCIONES: Esta función verifica la validez de todas las soluciones encontradas utilizando la técnica de backtracking.

Función HEURISTICA COLOCACION INICIAL: Implementa una heurística que proporciona una colocación inicial para las reinas en el tablero antes de aplicar el algoritmo de backtracking. Esta estrategia busca mejorar el rendimiento del algoritmo al reducir la cantidad de configuraciones que deben explorarse desde el principio.

0.5.2. Decisiones de Diseño y Estrategias de Optimización

Uso de Backtracking: La solución se basa en el algoritmo de backtracking para explorar eficientemente todas las posibles configuraciones del tablero.

Heurística de Colocación Inicial: La función busca una columna segura en cada fila antes de aplicar el backtracking. Esto permite un inicio más eficiente del algoritmo, reduciendo la cantidad de configuraciones exploradas desde el principio.

Función de Poda (ES SEGURO): actúa como una función de poda para evitar configuraciones inválidas, reduciendo así la exploración innecesaria del espacio de búsqueda.

0.5.3. Ejemplos de Uso

```
# Encontrar y mostrar todas las soluciones para N=4 con colocación inicial heurística

tablero_inicial = heuristica_colocacion_inicial(4)
resolver_n_reinas(tablero_inicial, 0, 4)
```

La implementación se demuestra para N=4, pero es escalable para cualquier valor de N. Puedes probar diferentes valores de N para encontrar soluciones para tableros de tamaño variable.

```
# Encontrar y mostrar todas las soluciones para N=8 con colocación inicial heurística

tablero_inicial = heuristica_colocacion_inicial(8)
resolver_n_reinas(tablero_inicial, 0, 8)
```

0.6. Conclusión

En resumen, el problema de las N reinas destaca la importancia de implementar estrategias de optimización para mejorar la eficiencia, especialmente cuando se trata de problemas combinatorios con soluciones exponenciales. Además, demuestra cómo pequeñas mejoras pueden tener un impacto significativo en la resolución práctica del problema.

0.7. Referencias

Salmerón, S. C., Lallena, J. A. R., Ruiz, A. T. (2015). El problema de las n reinas. Boletín de la Titulación de Matemáticas de la UAL, 9(1), 15-16.

Cañari, A. R., Rivas, M. Á. (2009). Aplicación de la metaheurística "Búsqueda tabú" al problema de las N-reinas. Revista de investigación de Sistemas e Informática, 6(2), 27-34.

Chazallet, S. (2016). Python 3: los fundamentos del lenguaje. Ediciones Eni.