Proyectos de Programación

JUSTIFICACIÓN ALGORITMOS

Primera Entrega

Oriol Farrés, Daniela Cervantes, Guiu Carol y Daniel Martínez

Grado GEI-PROP - Curso 2024-25, Quatrimestre de otoño

Identificador equipo: Grupo 42.3

18 de Noviembre

oriol.farres daniela.cervantes guiu.carol daniel.martinez

Entrega 2.0

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Índice general

Capítulo 1

Algoritmos

1.1. Algoritmo de Fuerza Bruta

1.1.1. Pseudocódigo

Pseudocódigo:

Algorithm 1 Algoritmo de Fuerza Bruta

```
1: function AgregarPermutación(raíz, usados, caminoActual)
2:     if EstaCompleto(raíz) then
3:         heur ← Heurística(raíz)
4:         if heur < min then
5:             min ← heur
6:             resultado ← raíz
7:         end if
8:         Retornar
9:     end if
10:    for i ← 0 hasta n_est − 1 do
11:        for j ← 0 hasta n_col − 1 do
12:            if raíz[i][j] == −1 then
13:                for k ← 0 hasta tamaño(estanteria) - 1 do
14:                    if no usados[k] then
15:                        copia ← CopiarMatriz(raíz)
16:                        copia[i][j] ← estanteria[k]
17:                        usados[k] ← verdadero
18:                        AgregarPermutación(copia, Clonar(usados), Clonar(caminoActual))
19:                        usados[k] ← falso
20:                    end if
21:                end for
22:                Retornar
23:            end if
24:        end for
25:    end for
26: end function
```

1.1.2. Justificación de las Estructuras de Datos Utilizadas

En nuestro algoritmo de fuerza bruta, utilizamos las estructuras de datos explicadas a continuación. Incluimos una breve justificación de por qué las hemos escogido y posibles alternativas (junto a sus respectivas ventajas y desventajas).

Matrices Bidimensionales ($int[][]$)

Hemos elegido las matrices bidimensionales para representar la disposición de los productos en las estanterías. Permiten un acceso directo a cualquier posición en tiempo constante, facilitando la verificación y actualización de posiciones específicas. Además, su estructura en filas y columnas es intuitiva para modelar estanterías. Sin embargo, las dimensiones fijas limitan la adaptabilidad a diferentes tamaños de estanterías.

Como alternativa, podríamos usar una lista de listas `List<List<Integer>>`, que ofrece mayor flexibilidad en las dimensiones. No obstante, el acceso a elementos puede es más lento en comparación con las matrices bidimensionales. Hemos priorizado la matriz bidimensional por su coste $O(1)$ en el acceso, considerando que en este caso no nos interesa la flexibilidad adicional de las listas.

Array de Booleanos ($boolean[]$)

Utilizamos un array de booleanos como un *flag* para saber qué productos ya han sido colocados, evitando así duplicaciones. Nos permite verificar rápidamente si un producto ha sido utilizado en tiempo constante $O(1)$ y es sencillo de implementar con un bajo consumo de memoria. La desventaja es que solo indica si un producto ha sido usado, sin almacenar información adicional que podría ser útil para optimizaciones futuras.

Como alternativa, podríamos utilizar array con un `Par<Key, Value>` y guardar más información. Hemos decidido prescindir de ello.

Listas Dinámicas ($List < Integer >$)

Hemos seleccionado las listas dinámicas para mantener el camino actual de productos colocados, facilitando la reconstrucción de la disposición completa. Son flexibles y permiten añadir y remover elementos fácilmente, además de facilitar la clonación para explorar nuevas permutaciones. La sobrecarga de clonación puede ser costosa en tiempo y memoria.

Como alternativa, podríamos usar arreglos estáticos con índices `int[]`, lo que reduciría la sobrecarga de clonación pero disminuiría la flexibilidad para gestionar dinámicamente los elementos. A pesar de la sobrecarga de clonación, hemos elegido las listas dinámicas por su capacidad de manejar permutaciones de manera flexible.

Árboles ($TreeNode < int[][] >$)

Representamos los árboles para estructurar los nodos del árbol de permutaciones, cada uno con una posible disposición de productos. Facilitan la exploración sistemática de todas las permutaciones

y organizan cada nivel del árbol como una decisión de colocación. Sin embargo, consumen mucha memoria ya que cada nodo contiene una copia completa de la matriz y la gestión de nodos y ramas puede ser compleja.

Como alternativa, podríamos usar recursión sin un árbol explícito, lo que reduciría el uso de memoria pero dificultaría el seguimiento de estados parciales. Hemos utilizado una estructura de árbol para mantener una organización clara de las permutaciones, a pesar del alto consumo de memoria (asumiendo que, tenemos un límite de memoria mucho mayor, al límite de tiempo de ejecución, en función del tamaño de la entrada).

1.2. Algoritmo AStarFaster

1.2.1. Pseudocódigo

Pseudocódigo:

Algorithm 2 Algoritmo AStarFaster

```
1: function CALCULAALGORITMO
2:     crear PriorityQueue<Nodo> llamada listaAbierta
3:     crear Map<String, Double> llamada listaCerrada
                                  ▷ Inicializar nodos iniciales para cada producto en la estantería
4:     for cada productoInicial en estanteria do
5:         crear List<Integer> llamada caminoInicial y añadir productoInicial
6:         crear HashSet<Integer> llamada noVisitados con todos los productos de estanteria
   excepto productoInicial
7:         crear Nodo nodoInicial con caminoInicial, noVisitados, productoInicial, 0,0,
   matrizSimilitudes, S_max
8:         añadir nodoInicial a listaAbierta
9:     end for
10:    while listaAbierta no está vacía do
11:        nodoActual ← extraer el nodo con la mayor prioridad de listaAbierta
12:        if noVisitados de nodoActual está vacío then
13:            si similitudTotal de nodoActual es mayor que mejorPuntuacion:
14:            actualizar mejorPuntuacion con similitudTotal
15:            actualizar mejorCamino con camino de nodoActual
16:        end if
17:        continuar
18:
19:        claveEstado ← GENERARCLAVEESTADO(nodoActual)
20:        if listaCerrada contiene claveEstado y listaCerrada[claveEstado] ≥ g de
   nodoActual then
21:            continuar
22:        end if
23:        listaCerrada[claveEstado] ← g de nodoActual
24:        for cada siguienteProducto en noVisitados de nodoActual do
25:            nuevoCamino ← copiar camino de nodoActual y añadir siguienteProducto
26:            nuevosNoVisitados ← copiar noVisitados de nodoActual y remover
   siguienteProducto
27:            costo ← g de nodoActual − matrizSimilitudes[siguienteProductoAnterior][siguienteProducto]
28:            crear Nodo sucesor con nuevoCamino, nuevosNoVisitados, siguienteProducto,
   costo, matrizSimilitudes, S_max
29:            if −sucesor.g + sucesor.h > mejorPuntuacion then
30:                añadir sucesor a listaAbierta
31:            end if
32:        end for
33:    end while
34:    Retornar mejorCamino convertida a matrizSolucion o null si mejorCamino está vacía
35: end function
```

1.2.2. Justificación de las Estructuras de Datos Utilizadas

En el algoritmo `AStarFaster`, se emplean diversas estructuras de datos que optimizan el rendimiento y la eficiencia del proceso de búsqueda. A continuación, se detallan las principales estructuras utilizadas, junto con su justificación y posibles alternativas.

PriorityQueue ($PriorityQueue < Nodo >$)

Utilizamos una `PriorityQueue<Nodo>` denominada `listaAbierta` para gestionar los nodos a explorar. Esta estructura permite extraer de manera eficiente el nodo con la mayor prioridad, determinada por la suma del costo acumulado (`g`) y la heurística estimada (`h`).

Como alternativa, se podría emplear una `List<Nodo>` ordenada manualmente. Sin embargo, esta opción incrementaría el tiempo de extracción del nodo con mayor prioridad, ya que requeriría una búsqueda lineal. La `PriorityQueue` ofrece operaciones de inserción y extracción con complejidad $O(\log n)$, lo que mejora significativamente el rendimiento del algoritmo.

Map ($Map < String, Double >$)

Se utiliza un `Map<String, Double>` llamado `listaCerrada` para almacenar las claves únicas de los estados ya explorados junto con su costo asociado (`g`). Esto permite verificar rápidamente si un estado ha sido visitado previamente y si el nuevo costo es más eficiente.

Una alternativa sería usar una `Set<String>` para almacenar únicamente las claves de los estados. Sin embargo, al utilizar un `Map`, no solo se verifica la existencia del estado sino que también se puede comparar y actualizar el costo asociado, lo que proporciona una gestión más eficiente de los estados explorados.

Listas y Sets ($List < Integer >$ y $HashSet < Integer >$)

Para representar el camino recorrido y los productos no visitados, se utilizan `List<Integer>` y `HashSet<Integer>`.

- **List<Integer>(camino)**: Permite mantener el orden de los productos asignados, lo cual es esencial para reconstruir la solución final.
- **HashSet<Integer>(noVisitados)**: Facilita la verificación y eliminación rápida de productos no visitados en tiempo constante $O(1)$. Esto optimiza el rendimiento al gestionar dinámicamente los productos restantes.

Como alternativa a `HashSet<Integer>`, se podría utilizar una lista `List<Integer>` para `noVisitados`. No obstante, esto incrementaría el tiempo de verificación y eliminación de elementos, ya que `HashSet` ofrece operaciones de búsqueda e inserción en tiempo constante.

List ($List < Integer >$ para mejorCamino)

Se emplea una `List<Integer>` llamada `mejorCamino` para almacenar el mejor camino encontrado durante la ejecución del algoritmo. Esta estructura permite una fácil construcción y acceso secuencial a los elementos, facilitando la conversión final a una matriz de solución.

Arrays Bidimensionales ($int[][]$)

La matriz de similitudes ($\texttt{matriz}_s imilitudes$)$serepresentacomounarraybidimensional$`int[][]`$.Estaestructura$

`Como alternativa, se podría utilizar una lista de listas List<List<Integer`, que ofrecería mayor flexibilidad en las dimensiones. Sin embargo, el acceso a elementos sería menos eficiente comparado con los arrays bidimensionales, que proporcionan un acceso directo y rápido.

Estructura de Nodo

La clase `Nodo` encapsula toda la información relevante para cada estado en la búsqueda, incluyendo el camino recorrido, los productos no visitados, el último producto asignado, el costo acumulado (`g`), la heurística (`h`), y referencias a la matriz de similitudes y al valor máximo de similitud (`S_max`).

Esta encapsulación facilita la gestión de los estados y permite una implementación más limpia y modular del algoritmo A*.

Variables de Control

- **mejorPuntuacion** (*double*): Almacena la mejor puntuación encontrada hasta el momento, lo que permite comparar y actualizar la solución óptima de manera eficiente.
- **mejorCamino** ($List < Integer >$): Guarda el mejor camino encontrado, permitiendo reconstruir la solución final una vez que se completa la búsqueda.

Estas variables son esenciales para mantener el seguimiento de la mejor solución encontrada durante la ejecución del algoritmo, asegurando que al finalizar se obtenga la distribución óptima de productos.

HashMap para listaCerrada

El uso de un `HashMap<String, Double>` para `listaCerrada` permite no solo verificar la existencia de un estado sino también comparar y actualizar el costo asociado. Esto es fundamental para evitar la reexploración de estados que ya han sido visitados con un costo menor o igual, optimizando así la eficiencia del algoritmo.

[1]

En resumen, las estructuras de datos seleccionadas en `AStarFaster` están cuidadosamente elegidas para maximizar la eficiencia y el rendimiento del algoritmo, asegurando una búsqueda óptima y rápida de la mejor distribución de productos en las estanterías.

1.3. Algoritmo de Aproximación - A* más preciso

1.3.1. Pseudocódigo

Pseudocódigo:

[1] Hemos elegido un `HashMap` debido a que proporciona operaciones de búsqueda, inserción y actualización en tiempo constante promedio $O(1)$, lo que mejora significativamente la eficiencia del algoritmo. Otras implementaciones, como `TreeMap`, tienen una complejidad de $O(\log n)$ pero mantienen los elementos ordenados, lo cual no es necesario en este caso.

Algorithm 3 Algoritmo A* más preciso

```
1: function CALCULAALGORITMO
2:     crear PriorityQueue<Nodo> llamada listaAbierta
3:     crear Map<String, Double> llamada listaCerrada
                                        ▷ Añadir nodos iniciales para cada producto de la estantería
4:     for cada productoInicial en estanteria do
5:         crear List<Integer> llamada caminoInicial y añadir productoInicial
6:         crear HashSet<Integer> llamada noVisitados con todos los productos de estanteria
   excepto productoInicial
7:         crear Nodo nodoInicial con caminoInicial, noVisitados, productoInicial, 0,0,
   matriz_similitudes, S_max
8:         añadir nodoInicial a listaAbierta
9:     end for
10:    while listaAbierta no está vacía do
11:        nodoActual ← extraer el nodo con menor f de listaAbierta
12:        if noVisitados de nodoActual está vacío then
13:            if similitudTotal de nodoActual > mejorPuntuacion then
14:                actualizar mejorPuntuacion con similitudTotal
15:                actualizar mejorCamino con camino de nodoActual
16:            end if
17:            continuar
18:        end if
19:        claveEstado ← GENERARCLAVEESTADO(nodoActual)
20:        if listaCerrada contiene claveEstado y listaCerrada[claveEstado] ≤ g de
   nodoActual then
21:            continuar
22:        end if
23:        listaCerrada[claveEstado] ← g de nodoActual
24:        for cada siguienteProducto en noVisitados de nodoActual do
25:            nuevoCamino ← copiar camino de nodoActual y añadir siguienteProducto
26:            nuevosNoVisitados ← copiar noVisitados de nodoActual y remover
   siguienteProducto
27:            costo ← g de nodoActual − matriz_similitudes[siguienteProductoAnterior][siguienteProduct
28:            crear Nodo sucesor con nuevoCamino, nuevosNoVisitados, siguienteProducto,
   costo, matriz_similitudes, S_max
29:            if −sucesor.g + sucesor.h > mejorPuntuacion then
30:                añadir sucesor a listaAbierta
31:            end if
32:        end for
33:    end while
34:    Retornar mejorCamino convertida a matrizSolucion o null si mejorCamino está vacía
35: end function
```

1.3.2. Justificación de las Estructuras de Datos Utilizadas

En el algoritmo `AStarPreciser`, se emplean diversas estructuras de datos que optimizan el rendimiento y la eficiencia del proceso de búsqueda. A continuación, se detallan las principales estructuras utilizadas, junto con su justificación y posibles alternativas.

PriorityQueue ($PriorityQueue < Nodo >$)

Utilizamos una `PriorityQueue<Nodo>` denominada `listaAbierta` para gestionar los nodos a explorar. Esta estructura permite extraer de manera eficiente el nodo con la menor prioridad, determinada por la función de evaluación $f = g + h$.

Como alternativa, se podría emplear una `List<Nodo>` ordenada manualmente. Sin embargo, esta opción incrementaría el tiempo de extracción del nodo con menor prioridad, ya que requeriría una búsqueda lineal. La `PriorityQueue` ofrece operaciones de inserción y extracción con complejidad $O(\log n)$, lo que mejora significativamente el rendimiento del algoritmo.

Map ($Map < String, Double >$)

Se utiliza un `Map<String, Double>` llamado `listaCerrada` para almacenar las claves únicas de los estados ya explorados junto con su costo asociado (`g`). Esto permite verificar rápidamente si un estado ha sido visitado previamente y si el nuevo costo es más eficiente.

Una alternativa sería usar una `Set<String>` para almacenar únicamente las claves de los estados. Sin embargo, al utilizar un `Map`, no solo se verifica la existencia del estado sino que también se puede comparar y actualizar el costo asociado, lo que proporciona una gestión más eficiente de los estados explorados.

Listas y Sets ($List < Integer >$ y $HashSet < Integer >$)

Para representar el camino recorrido y los productos no visitados, se utilizan `List<Integer>` y `HashSet<Integer>`.

- **List<Integer>(camino)**: Permite mantener el orden de los productos asignados, lo cual es esencial para reconstruir la solución final.
- **HashSet<Integer>(noVisitados)**: Facilita la verificación y eliminación rápida de productos no visitados en tiempo constante $O(1)$. Esto optimiza el rendimiento al gestionar dinámicamente los productos restantes.

Como alternativa a `HashSet<Integer>`, se podría utilizar una lista `List<Integer>` para `noVisitados`. No obstante, esto incrementaría el tiempo de verificación y eliminación de elementos, ya que `HashSet` ofrece operaciones de búsqueda e inserción en tiempo constante.

Mapas ($Map < String, Double >$ para listaCerrada)

El uso de un `HashMap<String, Double>` para `listaCerrada` permite no solo verificar la existencia de un estado sino también comparar y actualizar el costo asociado. Esto es fundamental para evitar la reexploración de estados que ya han sido visitados con un costo menor o igual, optimizando así la eficiencia del algoritmo.

2

[2]Hemos elegido un `HashMap` debido a que proporciona operaciones de búsqueda, inserción y actualización en tiempo constante promedio $O(1)$, lo que mejora significativamente la eficiencia del algoritmo. Otras implementaciones, como `TreeMap`, tienen una complejidad de $O(\log n)$ pero mantienen los elementos ordenados, lo cual no es necesario en este caso.

Arrays Bidimensionales (*int*[][])

La matriz de similitudes (`matriz_similitudes`) se representa como un array bidimensional `int[][]`. Esta estructura permite un acceso directo y eficiente a las similitudes entre pares de productos en tiempo constante $O(1)$, lo cual es crucial para calcular los costos y beneficios durante la exploración de nodos.

Como alternativa, se podría utilizar una lista de listas `List<List<Integer`, que ofrecería mayor flexibilidad en las dimensiones. Sin embargo, el acceso a elementos sería menos eficiente comparado con los arrays bidimensionales, que proporcionan un acceso directo y rápido.

Estructura de Nodo

La clase `Nodo` encapsula toda la información relevante para cada estado en la búsqueda, incluyendo el camino recorrido, los productos no visitados, el último producto asignado, el costo acumulado (`g`), la heurística (`h`), y referencias a la matriz de similitudes y al valor máximo de similitud (`S_max`).

Esta encapsulación facilita la gestión de los estados y permite una implementación más limpia y modular del algoritmo A*.

Variables de Control

- **S__max** (*double*): Almacena la máxima similitud entre productos, utilizada para la heurística.
- **mejorPuntuacion** (*double*): Almacena la mejor puntuación encontrada hasta el momento, lo que permite comparar y actualizar la solución óptima de manera eficiente.
- **mejorCamino** ($List < Integer >$): Guarda el mejor camino encontrado, permitiendo reconstruir la solución final una vez que se completa la búsqueda.

Estas variables son esenciales para mantener el seguimiento de la mejor solución encontrada durante la ejecución del algoritmo, asegurando que al finalizar se obtenga la distribución óptima de productos.

Resumen

En resumen, las estructuras de datos seleccionadas en `AStarPreciser` están cuidadosamente elegidas para maximizar la eficiencia y el rendimiento del algoritmo, asegurando una búsqueda óptima y rápida de la mejor distribución de productos en las estanterías. La combinación de `PriorityQueue`, `Map`, `List` y `HashSet` permite gestionar de manera efectiva los estados explorados, optimizar las operaciones de inserción y extracción, y mantener una estructura ordenada y accesible para la reconstrucción de la solución final.