Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2133 — Estructuras de Datos y Algoritmos 1'2024

Interrogación 2

18 de junio de 2024

Condiciones de entrega: Debe entregar solo 3 de las siguientes 4 preguntas.

Tiempo: 2 horas

Entrega: Al final de la prueba tienen 10 minutos para subir la imagen de la prueba a **https://iic2133.org**, cada pregunta por separado y en vertical

Evaluación: Cada pregunta tiene 6 puntos (+1 punto base). La nota es el promedio de las 3 preguntas entregadas.

1. Tablas de hash

Considere el problema de entregar los resultados de la prueba PAES del proceso de admisión a las universidades 2025. Este proceso es de muy alta carga en un período muy corto de tiempo. Ud. recibirá del DEMRE el archivo con los resultados solo 2 horas antes de la hora de publicación: 00:00 del día 20 de enero. Ud. considera que el servidor de bases de datos no será capaz de soportar la carga por lo que considera usar una Tabla de Hash para almacenar los resultados, la cual será poblada (`Insert`) entre las 10:00 y 11:59 del día previo (19 de enero). Los datos están almacenados en un registro con los siguientes campos:

```
Record r(
  RUN Entero (sin DV)
  DV char
  Nombre String 40
  NEM Entero
  Ranking Entero
  Puntaje competencia lectora Entero
  Puntaje competencia matemática 1 Entero
  Puntaje historia y ciencias sociales Entero
  Puntaje ciencias Entero
  Puntaje competencia matemática 2 Entero
)
```

Considere un rango de los RUN 1-25.000.000, pero solo 200 mil personas rinden la prueba PAES y el 80 % tienen un RUN entre 22.000.000 y 22.400.000

(a) (2 puntos) Suponiendo que el registro puede almacenarse directamente en una tabla de Hash (no es necesario el puntero al registro). Proponga una estructura de datos para dicha Tabla, indicando la función de hash utilizada, el tamaño de la tabla y su factor de carga esperado.

(b) (2 puntos) ¿Cómo cambia su solución si cada bucket de la primera tabla de hash contiene solo un puntero a una segunda tabla de hash que almacena los resultados?

(c) (2 puntos) Para el segundo caso, proponga el pseudocódigo de las funciones HashInsert(A,k,j,r) y HashSearch (A,k,j) $\rightarrow$ r.
Siendo A la tabla de Hash, h(r)= k el resultado de la función de hash de la primera tabla y g(r)=j la de la segunda y r el registro conteniendo los resultados de una persona.

Solución:

(a) (2pts) Dominio de RUN 1-25 millones, pero el 80 por ciento está 22 ¡RUN¡22.4 millones, el 20 por ciento restante RUN ¿22.4 millones y RUN ¡22 millones. h(RUN) = Techo ((RUN-22 millones)/2) 200 mil buckets. Factor de carga 100 por ciento. También es válido usar funciones con más buckets. Como el número de registros es fijo y conocido no se necesita mas espacio para nuevas inserciones.

(b) (2 pts)h(RUN) =DV genera una tabla de 11 valores, g(RUN) RUN mod 7 pueden ser otras funciones.

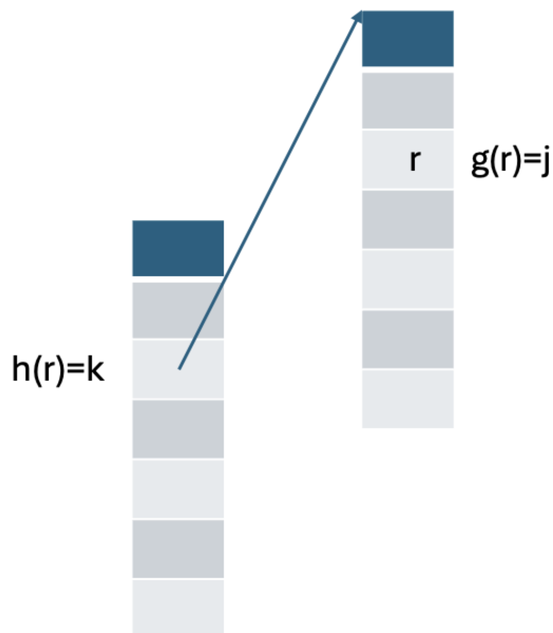


Figura 1: esquema de tablas de hash

(c) (2 pts)

```
k <- h(r)
j <- g(r)
Insert(A,k,j,r) {

    if A[k] = null  A[k] <- alloc(buckets) --buckets=tamaño de tabla secundaria
    else return false
    if ( A[k][j] está vacío) <- r; return true
    else j <-overflow (A[k],j); <- r; return true
    return false
 }
 HashSearch(A,k,j) {
    if A[k]= null return null
    j <- overflow A[k][j]
    if A[k][j]= vacío return null
    else return A[k][j]
}
overflow (p){
    while true
    if no hay overflow return p
```

```
        else r <- busque en el overflow     -- el overflow depende de la técnica elegida
    }
```

2. Backtracking

En una ciudad se quiere instalar estaciones de carga para vehículos eléctricos. La ciudad está dividida en una cuadrícula M de $n \times n$ celdas y cada celda puede contener una estación de carga o estar vacía. Cada estación de carga E_i permite cubrir un cuadrante de $c \times c$ celdas con la estación en su centro y se dispone de una lista Z de las zonas (celdas) de la ciudad que es prioritario que estén cubiertas por al menos una estación de carga. De igual forma, se dispone de una lista S que indica todas aquellas celdas de la ciudad en las que no se pueden instalar estaciones de carga por razones de seguridad. Finalmente, por limitaciones de presupuesto se puede instalar un máximo de k estaciones de carga.

(a) (1 punto) El problema es claramente un CSP, identifique Variables, Dominios y Restricciones.

- 0,33 puntos por cada concepto

Variables: Ubicación de las estaciones $E_i = E[i] = \text{celda}(x_i, y_i)$ para todo i en $1 \leq i \leq$k
Dominios: Para cada E[i] el dominio es M[n $\times n$]
Restricciones: $E[i] \notin S$ para todo i en $1 \leq i \leq k$

(b) (3 puntos) Diseñe un algoritmo en pseudocódigo que permita ubicar las estaciones de carga para cubrir las zonas Z de la ciudad, sin ubicarlas en las celdas seguras S, utilizando a lo más de k de ellas.

- 1 punto por resolver las condiciones de término,
- 1 punto por controlar dominios y restricciones,
- 1 punto por resolver recursión y retroceso.

```
Sea:
    testSol(Z,E): true si la asignación de estaciones de E cubre todas las celdas de Z
    safeCel(celda,S): true si la celda no está en S

    grid(M,Z,S,E,k) // E parte vacío y k en el numero de estaciones
        if testSol(Z,E) // es solución válida
            return true
        elseif k == 0 // no hay estaciones disponibles
            return false
        else
            for celda in M // el dominio completo
                if safeCel(celda,S)     // cumple restricciones
                    E[k] <- celda // Estación k en esa celda
                    if grid(M,Z,S,E,k-1) // una estacion menos para asignar
                        return true
                    E[k] <- null
            return false

    testSol(Z,E)
        Aux[n x n] // Matriz auxiliar para la verificacion
        for celda in E // id que celdas estan cubiertas
            Aux[(c x c)*celda] <- cubierta // centradas en celda
        for zona in Z // verificar celdas prioritarias
            if Aux[zona] <> cubierta
                return false
        return true
```

```
safeCel(celda,S)
    for s in S
        if s == celda
            return false
    return true
```

(c) (2 puntos) Un experto aconseja que se minimice la distancia media desde las celdas de Z a la estación más cercana, siempre utilizando a lo más k estaciones. Modifique su algoritmo anterior de acuerdo con lo señalado. *Hint: Puede asumir que dispone de las funciones: distancia(E_i, Z_i) que le entrega la distancia entre la Zona prioritaria Z_i y la Estación de carga E_i, y la función masCerca(Z_i) que entrega la estación de carga más cercana a Z_i.*

Pueden indicar solo lo que se modifica
- 1 punto por resolver el registro del menor en las condiciones de término,
- 1 punto por resolver el ajuste en recursión y retroceso para evaluar todas las soluciones.

```
Sea:
    testSol(Z,E): true si la asignación de estaciones de E cubre todas las celdas de Z
    safeCel(celda,S): true si la celda no está en S
    dMedia(Z,E): distancia media de Z a E mas cercana

    grid(M,Z,S,E,k,dMin) // E parte vacío, k numero de estaciones, dMin +infinito
        if testSol(Z,E) // es solución válida
            d = dMedia(Z,E)
            if d < dMin // es la de distancia minima
                dMin <- d
                minE <- E // guardamos la solucion
            return true
        elseif k == 0 // no hay estaciones disponibles
            return false
        else
            for celda in M // el dominio completo
                if safeCel(celda,S)     // cumple restricciones
                    E[k] <- celda // Estación k en esa celda
                    if grid(M,Z,S,E,k-1) // una estacion menos para asignar
                        // E[k] es una solucion, next
                    E[k] <- null
            return false

    dMedia(Z,E)
        D <- 0
        m <- cardinalidad(Z)
        for i in Z
            D <- D + distancia(masCerca(i),i)
        return D / m
```

3. DFS y algoritmos codiciosos

El problema de recorrido de laberintos ha sido muy estudiado en el contexto de robots autónomos. Considere un laberinto modelado como cuadrícula, en el cual un robot solo puede realizar dos movimientos: (1) avanzar a la celda directamente en frente o (2) rotar su posición en múltiplos de 90°. El robot posee cámaras en las cuatro direcciones, que le permiten detectar cuántas celdas libres existen hasta encontrar un muro. El objetivo del robot es encontrar una celda específica, que se detecta solo cuando el robot se ubica sobre ella.

(a) (3 puntos) Considere que el mapa del laberinto es desconocido por el robot en un inicio y lo descubre a medida que avanza por él. Describa cómo el robot puede modelar el laberinto para usar DFS y lograr su objetivo. Indique la complejidad de tiempo en el peor caso para la operación del robot en un laberinto de $n \times m$ celdas. No requiere entregar pseudocódigo.

Solución: Para que el robot pueda utilizar DFS y resolver el problema, requerimos modelar el laberinto como grafo. Una posibilidad es considerar las celdas como nodos y los pasos entre celdas adyacentes alcanzables como aristas. Es decir, los muros no se consideran celdas sino que solo impiden la existencia de una arista entre las celdas que separa. Bajo esta modelación, el robot puede almacenar en un grafo originalmente vacío, la secuencia de nodos y aristas que descubre a medida que se mueve por el laberinto. Al almacenar el grafo descubierto, se permite volver atrás, tal como exige DFS cuando ha agotado un camino en profundidad específico. Esta posibilidad es central, pues DFS requiere saber qué nodos ya ha visitado.

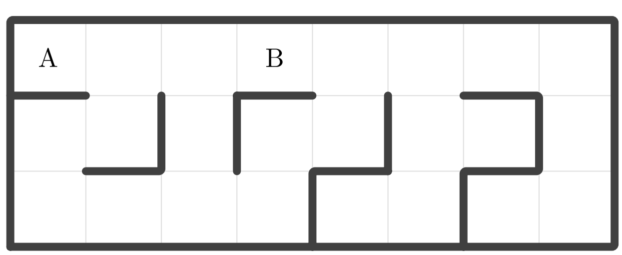
En términos de complejidad, podemos suponer que guardar la celda actual y las aristas que conectan con celdas vecinas no visitadas es una operación constante en el tamaño del laberinto, cuando se utiliza una matriz de adyacencia como estructura para almacenar el grafo. Luego, como cada celda tiene a lo más 4 celdas vecinas alcanzables, la cantidad de aristas es $\mathcal{O}(nm)$. Con esto, tenemos un grafo con tamaños $V \in \Theta(nm)$ y $E \in \mathcal{O}(nm)$. En tal escenario, el recorrido DFS toma tiempo $\mathcal{O}(E+V)$, i.e. $\mathcal{O}(nm)$.

Asignación de puntajes

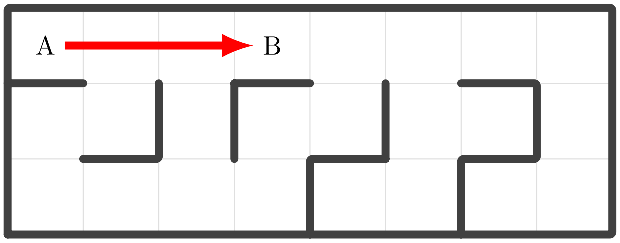
- 1.5 punto por describir la construcción de un grafo (indicando qué representan los nodos y aristas) y describir el uso de DFS sobre esta red. Hay varias modelaciones que pueden usar esta idea, por ejemplo, aquellas que consideran los muros como celdas "bloqueadas".
- 1.5 por describir la complejidad haciendo referencia a los parámetros n y m conocidos.

(b) (3 puntos) La rotación es costosa, por lo que un intento para que el robot rote menos es tomar la siguiente decisión frente a bifurcaciones: "*elegir el camino con mayor número de celdas libres*". Muestre una instancia de laberinto en que esta estrategia reduzca el número de rotaciones requeridas para llegar al objetivo con respecto a la solución de (a). Justifique su respuesta comparando ambos enfoques.

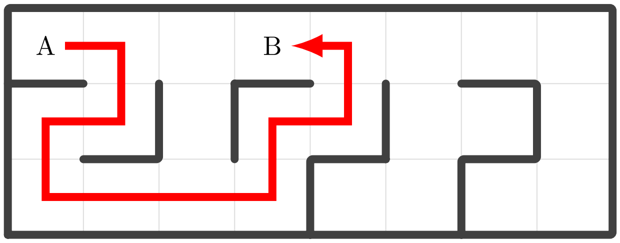
Solución: Consideremos la siguiente instancia de laberinto, donde la posición inicial es la celda A y B es la celda objetivo.



Suponiendo que el robot comienza en la misma posición en ambos enfoques (y posiblemente rota en ambos para ubicarse apuntando a B desde A), la estrategia propuesta obliga al robot a dirigirse directamente a B, llegando a dicha celda en 3 pasos. Dado que DFS escoge en un orden arbitrario qué dirección visitar primero, podemos considerar una implementación que siga la ruta mostrada en la segunda imagen.



Ruta encontrada por la estrategia propuesta



Ruta encontrada por posible ejecución de DFS clásico

En tal instancia, la cantidad de rotaciones es claramente menor con la estrategia propuesta.

Asignación de puntajes

- 1.5 puntos por construir una instancia de laberinto.
- 1.5 por comparación de la solución encontrada.

4. Programación dinámica

Suponga que en una sala se han programado n charlas $c_1, \ldots, c_n$, cada una de las cuales dura exactamente una hora, y tal que cada charla c_{i+1} comienza apenas termina la anterior charla c_i. La asistencia a una charla c_i ($1 \leq i \leq n$) entrega un puntaje p_i (representado por un entero positivo) pero produce tanto cansancio que es imposible asistir a las siguientes d_i charlas, teniendo que descansar. Alice quiere asistir a las charlas y sumar la máxima cantidad de puntaje posible, respetando las reglas de descanso indicadas. En este ejercicio usted tendrá que ayudarla a calcular el puntaje máximo.

(a) (3 puntos) Dados los valores $p_1, \ldots, p_n$ y $d_1, \ldots, d_n$ para las n charlas, escriba una ecuación de recurrencia para la función $S(i)$, la que calcula (de forma exhaustiva) el puntaje máximo que se puede obtener con las charlas $c_i, \ldots, c_n$, para $1 \leq i \leq n$. Además de la definición recurrente, indique claramente los casos base necesarios para la correcta definición de la función. Si lo prefiere, puede usar pseudocódigo para representar la ecuación de recurrencia.

Solución: Para $1 \leq i \leq n$ definimos:

$$S(i) = \begin{cases} 0, & i > n \\ \text{máx}\,\{S(i+1), S(i+d_i+1)+p_i\}, & i \leq n. \end{cases}$$

Claramente, la solución al problema original se obtiene con $S(1)$. La versión pseudocódigo es:

Algoritmo 1: $S(i)$

if $i > n$ **then**
  return 0
else
  return $\text{máx}\,\{S(i+1), S(i+d_i+1)+p_i\}$

(b) (3 puntos) Implemente el algoritmo del punto anterior usando programación dinámica, tal que evite recalcular subproblemas antes calculados. Su solución puede ser iterativa o recursiva, según su preferencia.

Solución: En este caso se mantiene un arreglo $M[1..n]$ con todas sus entradas inicializadas con $\emptyset$. La versión de programación dinámica recursiva es:

Algoritmo 2: $\mathit{SPD}(i)$

if $i > n$ **then**
  return 0
else
  if $M[i] = \emptyset$ **then**
    $M[i] \leftarrow \max\{\mathit{SPD}(i+1), \mathit{SPD}(i+d_i+1)+p_i\}$
  return $M[i]$

También es válida la versión iterativa.