Proyecto de curso
Calculando el plan de riego óptimo de una finca

Estudiantes:

Sebastián Idrobo Avirama - 2122637

Brayan Andrés Sanchez Lozano - 2128974

Jose Luis Hincapié Bucheli - 2125340

Carlos Andrés Hernandéz Agudelo - 2125653

Asignatura: Análisis y Diseño de Algoritmos II

Docentes: Juan Francisco Diaz Frias - Jesús Alexander Aranda

Universidad del Valle
Facultad de Ingeniería
Santiago de Cali
2024

Tabla de contenido

Algoritmo de Fuerza Bruta

Complejidad

Para poder determinar la complejidad del algoritmo, es importante identificar el método principal y evaluar las complejidades de los submétodos que utiliza, si los hay. En este caso, el método principal es **roFB**.

El método **roFB** es responsable de devolver los índices de los tablones en el orden en que deben regarse, de acuerdo a la solución óptima. Esté comienza invocando el sub-método **calcular**, que tiene una complejidad de tiempo $O(n!)$, donde **n** es el número de tablones.

El método **calcular** genera todas las permutaciones posibles de la lista de tablones, que es una operación con una complejidad temporal de $O(n!)$. Para cada permutación, calcula el costo de regar los tablones en ese orden, lo cual es una operación con una complejidad temporal de $O(n)$, pero este costo es dominado por la generación de permutaciones.

Después de invocar a **calcular**, **roFB** procede a construir una lista de índices que representan el orden óptimo de riego de los tablones. Este proceso tiene una complejidad de tiempo de O(n), ya que implica iterar sobre cada tablón en el resultado.

Sin embargo, la complejidad de tiempo total del algoritmo está dominada por el llamado al método **calcular**, por lo que la complejidad del algoritmo termina siento $O(n!)$.

La complejidad espacial está dada por el método **calcular** con un valor de $O(n)$, en vista de que se elige la mejor combinación de tablones en el instante que se obtiene, siendo solo reemplazada si se encuentra otra con menor costo.

Corrección

El algoritmo de fuerza bruta funciona generando todas las permutaciones posibles de los tablones y calculando el costo de riego para cada permutación. Luego, selecciona la permutación, el costo mínimo, y dado a esta naturaleza de explorar exhaustivamente todas las posibles soluciones, **siempre encontrará la solución óptima.**

Ahora bien, se debe tener en cuenta que la complejidad del algoritmo es $O(n!)$, donde **n** es el número de tablones. Por lo que el tiempo de ejecución del algoritmo crece muy rápido a medida que aumenta el número de los tablones. Por lo que, a pesar de que el algoritmo es correcto, puede no ser práctico para fincas con un gran número de tablones, traduciéndose en una solución correcta pero no eficiente.

Algoritmo Dinámico

Caracterización de la estructura de una solución óptima

"Toda solución óptima de un problema se forma de soluciones óptimas de subproblemas"

Realización de escogencia

Sea $F = \{T_1, T_2, ..., T_{n-1}\}$ una secuencia de tablones en las que cada tablón es una tupla tal que $T_i = \{ts_i^F, tr_i^F, p_i^F\}$ donde ts_i^F representa el tiempo de supervivencia, tr_i^F el tiempo de regado y p_i^F la prioridad del tablón i de la finca F, para $0 \leq i < n$, es posible caracterizar la estructura de una solución óptima de la siguiente manera:

$$roPD(F_{n-1}) = roPD(F_{n-2}) - T_{n-1}$$

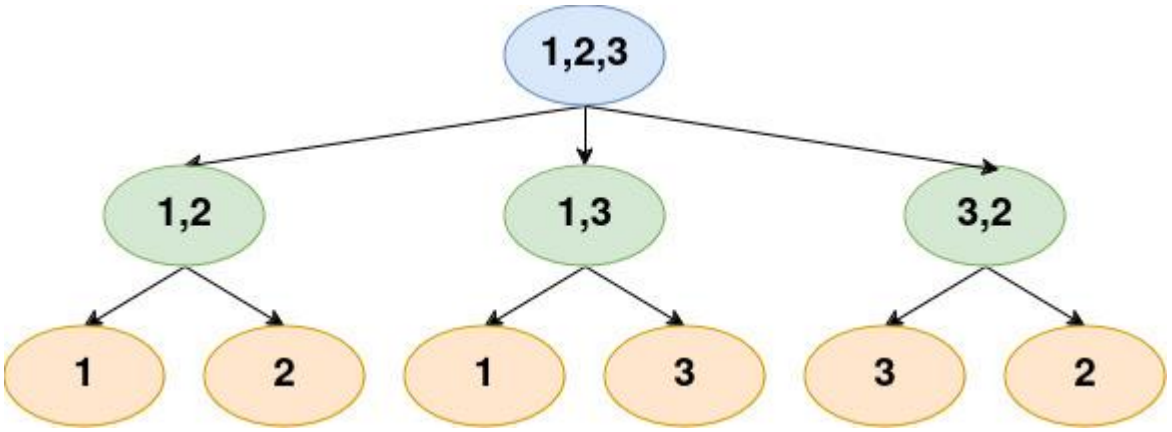
Dicho en otras palabras, la estructura óptima del programa de riego producido por $roPD(F_{n-1})$ se compone de la estructura óptima del programa de riego producido por $roPD(F_{n-2})$, siendo F_{n-2} la secuencia de tablones descontando el último tablón y T_{n-1} el último tablón en la programación de riego.

Asunción que la escogencia lleva a la solución

Se espera entonces encontrar una programación de riego $\prod$ para la finca F tal que $CR_F^{\prod}$ sea mínimo, y es que es posible asumir que esta escogencia lleva a la solución óptima debido a que los tablones de F irán siendo removidos y acomodados en el orden necesario, correspondiendo con el presente problema.

Determinar los subproblemas asociados

Para determinar los subproblemas correspondientes de la escogencia, es posible realizarlo a partir de una visualización del respectivo llamado recursivo:



En primer lugar, el factor de ramificación empieza por n, disminuyendo progresivamente en cada llamado recursivo de $roPD(F)$

Teniendo en cuenta que el factor de ramificación empieza por n, y que el mismo disminuye progresivamente en cada llamado recursivo debido a que a la finca F se le descuenta de manera seguida el última tablón, y que también la cantidad de llamados recursivos (O visto gráficamente, la profundidad del árbol) es n, esto se puede expresar así:

$$\sum_{i=0}^{n} i!$$

Demostración que las soluciones de los subproblemas son óptimas

Si la escogencia no es óptima, significa que hay una mejor manera de organizar la finca F de tal manera que j corresponda a un índice con menor costo, y consigo que el costo asociado a $CR_F^{\prod}$ es mínimo, lo que implica que hay una mejor programación de riego.

Tal situación es contradictoria debido a que la estructura óptima se encarga de caracterizar todos los posibles subproblemas que tiene la estructura, y consigo se garantiza que todas las posibles combinaciones han sido tomadas, por lo que se debe de haber encontrado con anterioridad el índice j en el problema y subproblemas que garantizan el menor costo.

Definición recursiva del valor de una solución óptima

La definición recursiva del costo de una solución óptima para cada subproblema puede ser vista de la siguiente manera:

$$f(x) = \begin{cases} 0 & si\ \ n = 0 \\ \\ min_{i=1}^{n}(C_F^R[i] + costRoPD(F_{n-2} - T_{n-1})) & si\ \ x \geq 1 \end{cases}$$

Explicado de manera textual, es posible deducir el caso base que será útil para la resolución del problema, el cual se trata de una finca F tal que se encuentra vacía, y debido a que no hay tablones para calcular el costo de riego asociado a los mismos, es posible decir que el costo de riego de la misma finca es 0.

Por otro lado, si se trata de una finca F tal que se encuentra compuesta por uno o más tablones, el costo de la solución óptima se encuentra calculado por evaluar todas las posibles combinaciones de regar los tablones de la finca F tal que se itera sobre cada uno de los subproblemas obteniendo las sub-programaciones de riegos mínimas para cada uno de ellos hasta obtener el programa de riego $\prod$ óptimo para la finca F.

Descripción del algoritmo para calcular el costo de una solución óptima

Teniendo en cuenta la definición recursiva, es posible calcular el costo de una solución óptima de la siguiente manera:

```python
def costoRoPD(finca):
  if (len(finca == 0)):
    return 0

  else:
    costoF = ∞
    costosCandidatos = []

    for i in range(finca.len, -1, -1, -1):
      tablonRestante = finca[len(finca) - 1]
      fincaRestante = finca[0:len(finca) - 1]
      tiempoRestante = calcularTiempo(fincaRestante)

      costoTablon = evaluarCosto(tablon, tiempoR)

      costo = roPD(fincaRestante)

      costo = costo + costoTablon
      costosCandidatos.append(costo)

    for costo in costosCandidatos:
      if (costo < costoF):
        costoF = costo

  return costoF
```

El pseudocódigo anterior funciona así:

1. Se define el caso base que funcionará como caso de parada para la función recursiva $costRoPD$
2. Si la finca ingresada trae un tablón o más, es necesario calcular el costo recursivamente, calculando el costo del último tablón respecto al tiempo de riego calculado de la finca F sin el último tablón escogido. Calcular el tiempo de esta manera no tiene inconveniente alguno debido a que, no importa la organización que tome el subproblema $costoRoPD(F_{n-2} - \{T_{n-1}\})$, la suma de los tiempos de riegos será el mismo.

Descripción del algoritmo para calcular una solución óptima

Por último, para obtener una solución óptima, se realiza de la siguiente manera:

```python
def roPD(self, finca, memo={}):
	if (len(finca) == 0):
		return (0, [])

	else:
		costoF = float('inf')
		ordenF = []
		ordenesCandidatos = []

		for i in range (len(finca) - 1, -1, -1):
			fincaRestante = finca[:]
			tablon = fincaRestante.pop(i)[:]
			tiempoR = self.calcularTiempo(fincaRestante)

			costoTablon = self.evaluarCosto(tablon, tiempoR)

			if ((tuple(fincaRestante)) in memo):
				costo, orden = memo[(tuple(fincaRestante))]
			else:
				costo, orden = self.calcular(fincaRestante, memo)
				memo[tuple(fincaRestante)] = (costo, orden)

			costo += costoTablon
			orden = orden + [tablon]

			ordenesCandidatos.append((costo, orden))

		for orden in ordenesCandidatos:
			if (orden[0] < costoF):
			costoF = orden[0]
			ordenF = orden[1]
	return (costoF, ordenF)
```

1. Se define el caso base que funcionará como caso de parada para la función recursiva $RoPD$.
2. Si la finca ingresada trae un tablón o más, es necesario calcular el costo y almacenar el orden de los tablones recursivamente, calculando el costo del último tablón respecto al tiempo de riego calculado de la finca F sin el último tablón escogido. Cabe resaltar que el bucle *for* se encarga de intercambiar los tablones que irán en la última posición, de tal manera que este es el método para probar todas las posibles permutaciones.
    a. Cabe resaltar de nuevo que calcular el tiempo de esta manera no tiene inconveniente alguno debido a que, no importa la organización que tome el

subproblema $RoPD(F_{n-2} - \{T_{n-1}\})$, la suma de los tiempos de riego será el mismo.

3. Hasta lo descrito, $roPD$ se trata de una función de fuerza bruta, por lo que para pasar a un algoritmo de programación dinámica se utiliza *memoization* con una tabla hash. Durante cada llamado recursivo se evalúa si el grupo de fincas F_{n-2} ya ha sido evaluado para evitar re-calcular el costo del mismo. En caso de que ya lo haya sido y se encuentre dentro de la tabla hash, se retorna el costo directamente, y en caso de que no, se procede con el llamado recursivo de $RoPD(F_{n-2} - \{T_{n-1}\})$ y se guarda el respectivo resultado del costo una vez ha sido calculado, almacenando como llave la secuencia de tablones de F_{n-2}. La ventaja de este método se explica en la complejidad temporal.

Complejidad

Complejidad temporal

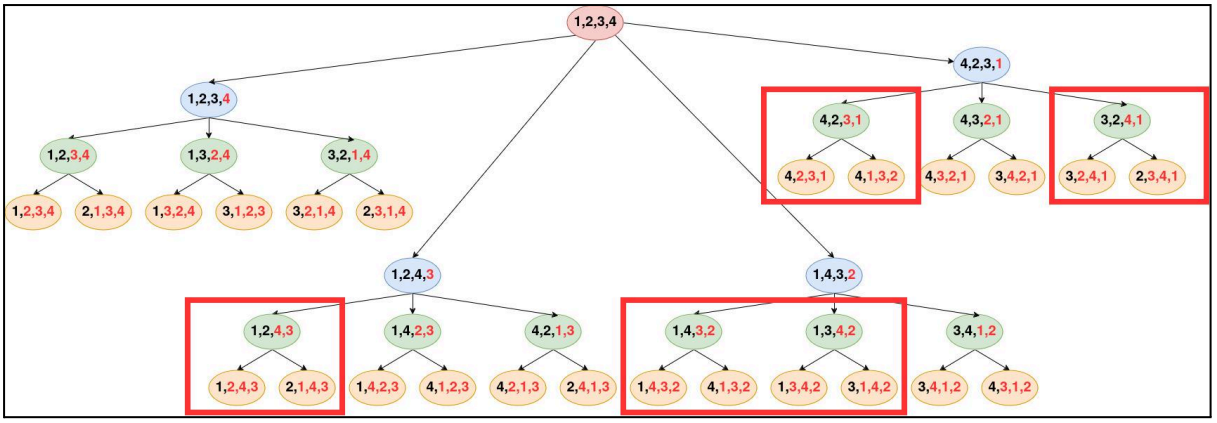
La complejidad de este algoritmo es $O(n^3)$, considerando n la cantidad de tablones de la finca F. Esto debido a que, como es posible observar en el árbol de llamados recursivos, el factor de ramificación del árbol se encuentra acotado superiormente por n llamados recursivos, de la misma manera que la profundidad del árbol se encuentra acotado por n debido a que en cada llamado recursivo se elimina un tablón de la finca.

Esto nos obtendría una complejidad $O(n!)$, pero el uso de la estructura de tabla hash para guardar los datos ya calculados permite que el algoritmo haga un menor número de cálculos, lo que significa que $roPD$ debe de realizar n llamados recursivos para cada subproblema, y en cada llamado recursivo se tiene que hay n posibilidades de ramificación, y porque cada uno de estas combinaciones son guardadas para ser reusadas, en total existen $n * n$ nodos que el algoritmo puede recorrer.

Por último, debido a que se tratan de permutaciones y no combinaciones -Ya que el orden de los tablones afecta el costo del mismo-, esto significa que el resultado de los $n * n$ combinaciones dependen de la cantidad de tablones para la finca, es decir, no es lo misma configuración (1,2) que (2,1), por lo que teniendo en cuenta la cantidad de tablones n, se tiene como resultado final que el algoritmo $roPD$ consta de una complejidad temporal $O(n^3)$ [1].

En aras de mostrar gráficamente como el uso de la tabla hash ayuda a la disminución de la complejidad temporal del algoritmo, se expone el árbol de recursión en los que los recuadros rojos contemplan aquellas llamadas recursivas que no son calculadas ya que fueron almacenadas anteriormente.

[1] Este resultado no toma en cuenta que en cada cálculo recursivo realizado se realiza una copia de la finca restante como llave en la tabla hash. Si se toma en cuenta esta operación, se tiene la complejidad $O(n^4)$



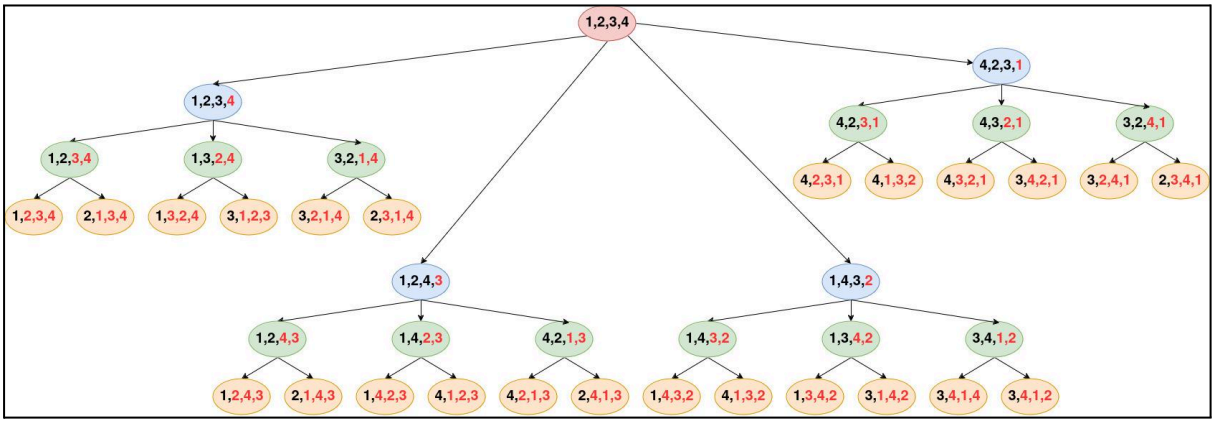
Complejidad espacial

La complejidad de este algoritmo es $O(n^2)$, esto debido a que la tabla hash solo almacena aquellos nodos que se encuentran en medio del árbol de recursión (No almacena al nodo cuya profundidad es 0, y no almacena a los nodos cuya profundidad es n), lo que implica que almacenará n^2 elementos dentro de la misma, dando como resultado la complejidad anteriormente mencionada.

De hecho, si se suma la cantidad de nodos de profundidades 2 y 3 en un árbol generado por una finca F que contiene 4 tableros, esta complejidad se cumplirá.

Corrección

La corrección de este algoritmo es posible explicar bajo el siguiente árbol de llamados recursivos:

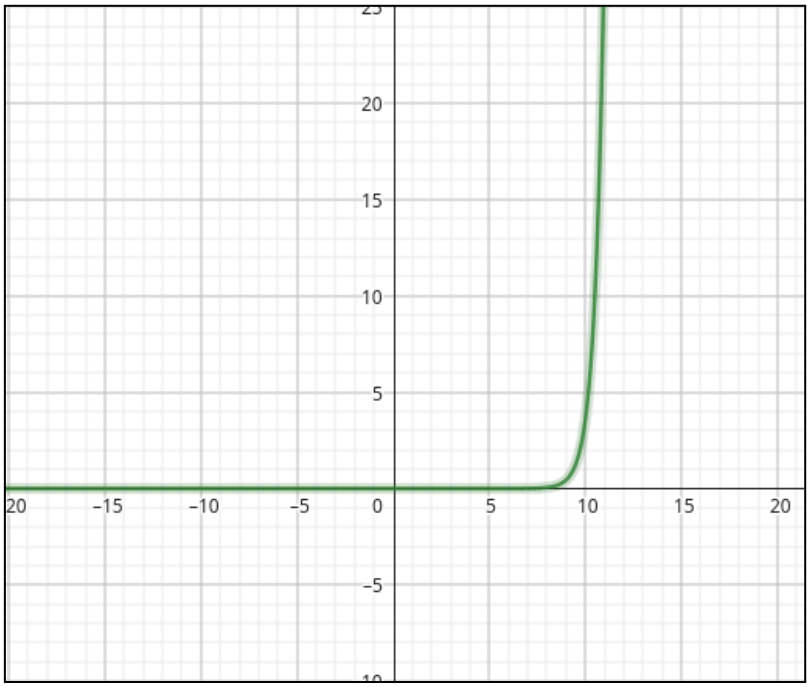


Considerando que 1,2,3,4 se trata de los tablones que conforman la finca F, debido a que el algoritmo realiza todas las permutaciones como es posible observar en el árbol, la correctitud del algoritmo se explica en que el mismo evalúa todas las posibles soluciones óptimas para calcular y conseguir así la solución óptima.

¿Es útil en la práctica?

Respecto al tiempo

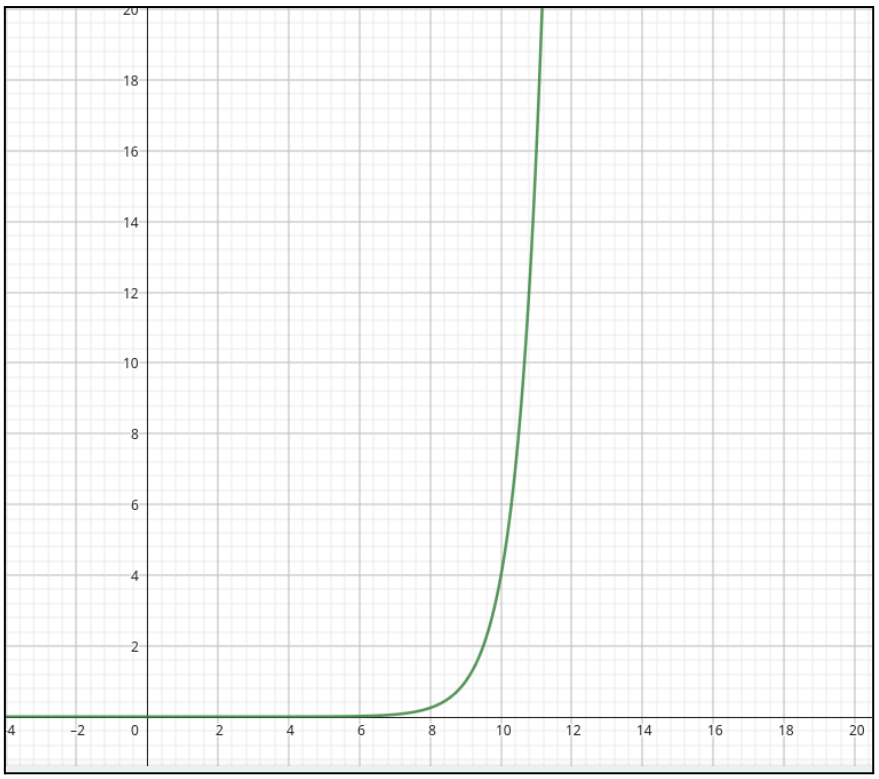
El tiempo que se tomará el equipo en resolver el problema es aceptable mientras que $k < 12$, ya que en este valor el programa se demoraría aproximadamente 4 horas en resolver el problema. Este resultado dejaría entrever que es el algoritmo es útil respecto al tiempo, ya que significa que podría calcular el programa de riego óptimo para una finca tal que contiene 2^{12} tablones, lo que es un valor suficiente para cualquier finca que necesite del algoritmo para tener una programación de riego óptima.



Gráfica realizada a partir de la función f(k) = ((2^k)^3)/3*10^8. Como es de esperarse se observa el comportamiento de una función polinomial, ya que el algoritmo $roPD$ lo contiene, de la misma manera, se observa que es cuando k es cercano a 10 en que el crecimiento aumenta rápidamente.

Respecto a la memoria

Considerando que se utilizan 4 bytes para el almacenamiento por celda, el almacenamiento que tomará el equipo en resolver el problema es aceptable mientras que $k < 13$, ya que en este valor el programa tomaría aproximadamente 256 MB lo que significa una cantidad considerable de memoria. Este resultado dejaría entrever que el algoritmo es útil respecto a la memoria, ya que significa que podría calcular el programa de riego óptimo para una finca tal que contiene 2^{13} tablones, lo que es un valor suficiente para cualquier finca que necesite del algoritmo para tener una programación de riego óptima.



Gráfica realizada a partir de la función f(k) = ((2^k)^2)/(((2^10)/4)*2^10). Como es de esperarse se observa el comportamiento de una función polinomial, ya que el algoritmo $roPD$ lo contiene, de la misma manera, se observa que es cuando k es cercano a 9 en que el crecimiento aumenta rápidamente.

Algoritmo Voraz

Descripción del algoritmo

La estrategia aplicada en sí misma es bastante sencilla, teniendo $F = \{T_1, T_2, ..., T_{n-1}\}$ una secuencia de tablones en las que cada tablón es una tupla tal que $T_i = \{ts_i^F, tr_i^F, p_i^F\}$ donde ts_i^F representa el tiempo de supervivencia, tr_i^F el tiempo de regado y p_i^F la prioridad del tablón i de la finca F, para $0 \leq i < n$, se aplica a cada tablón una función heurística que permita asignar una puntuación. Dicha puntuación servirá para otorgar una idea sobre cuál debe ser el tablón a regarse en la iteración en cuestión. El método principal **roV** se incluye a continuación:

```python
def roV(self, fincaIngresada=None):
        if fincaIngresada == None:
            finca = self.finca.copy()
        else:
            finca = fincaIngresada.copy()

         # Se añade a cada tablon un índice que representa su posición en la
finca.
        for i in range(len(finca)):
            finca[i] = (finca[i][0], finca[i][1], finca[i][2], i)

        riegoOptimo = []
        costoRiego = 0
        tiempo = 0
        n_tablones = self.n

        while n_tablones > 0:
            prioridades = []
            for i in range (n_tablones):
                prioridades.append(self.heuristica(finca[i]))

            indice_tablonOptimo = prioridades.index(min(prioridades))
            tablonOptimo = finca[indice_tablonOptimo]
            riegoOptimo.append(tablonOptimo[3])

            costoRiego += self.calcularCosto(tablonOptimo, tiempo)
            tiempo += tablonOptimo[1]

            finca.pop(indice_tablonOptimo)
            n_tablones -= 1

        return (costoRiego, riegoOptimo)
```

De forma muy general, se crea una copia de la finca ingresada, luego, se añade la posición dentro de la finca a cada tablón (esto, simplemente con la finalidad de evitar problemas a la hora de presentar la solución cuando existen elementos repetidos), posteriormente, se comienza a iterar sobre los tablones, otorgando a cada uno una puntuación o "prioridad" (nótese que es diferente a la prioridad p incluída en el tablón), se selecciona el tablón con la menor puntuación (podría haber sido el mayor, esto depende exclusivamente de cómo fue definida la heurística), se añade su índice a la programación de riego óptima, así mismo como su costo al costo total del riego, y se elimina el tablón en cuestión de la finca, y se repite el mismo proceso hasta que no queden tablones.

La heurística utilizada en este caso fue:

$$heurística = ts + \frac{ts + tr^{factor}}{p}$$

Donde:

$ts = tiempo\ de\ supervivencia\ del\ tablón$

$tr = tiempo\ de\ riego\ del\ tablón$

$p = prioridad\ del\ tablón$

$factor = valor\ que\ incrementa\ a\ medida\ que\ hay\ más\ cultivos\ en\ la\ finca$

Descripción de la heurística: Como se puede apreciar, esta heurística cuenta con dos componentes principales, primero, el tiempo de supervivencia, lo que al ser sumado con el otro componente de la heurística, asigna una puntuación mayor a aquellos cultivos que pueden sobrevivir más tiempo sin agua, por lo que su "prioridad" (diferente a la prioridad p del tablón), será menor que la de los cultivos con un menor tiempo de supervivencia.

El otro componente es una proporción entre el tiempo de supervivencia y el tiempo de riego respecto a la prioridad del tablón. En esta sección se otorga más prioridad a aquellos cultivos que tienen un tiempo de riego corto, un tiempo de supervivencia largo y una alta prioridad.

Es decir, en forma muy general, la heurística favorece a los cultivos con un tiempo de supervivencia largo (ts), un tiempo de riego corto (tr), y una alta prioridad (p).

¿Por qué esta heurística? Se escogió esta heurística en particular por varias razones, primero, es más eficiente en términos de memoria y tiempos de ejecución respecto a otras heurísticas, a su vez manteniendo cierta calidad en los resultados, en particular, para la batería de pruebas otorgada con los requerimientos del proyecto. Por otro lado, la heurística es relativamente simple, fácil de entender, y por lo tanto, de probar y depurar. Es además ciertamente flexible, puesto que cuenta con un factor que crece a medida que aumenta el número de tablones en la finca. Finalmente, fue escogida porque tiene un sentido intuitivo, priorizando factores importantes a considerar a la hora de decidir el orden de riego.

Salidas del algoritmo respecto a casos de prueba

Para los ejemplos presentados en el enunciado del proyecto, estos fueron los resultados obtenidos:

$F1 =<< 10, 3, 4 >, < 5, 3, 3 >, < 2, 2, 1 >< 8, 1, 1 >, < 6, 4, 2 >>$

Resultado: $(19, [2, 1, 4, 0, 3])$

¿Es la solución óptima? No, la solución óptima es $(17, [2, 1, 0, 3, 4])$

$F2 =<< 9, 3, 4 >, < 5, 3, 3 >, < 2, 2, 1 >, < 8, 1, 1 >, < 6, 4, 2 >>$

Resultado: $(23, [2, 1, 4, 0, 3])$

¿Es la solución óptima? No, la solución óptima es $(16, [2, 1, 0, 3, 4])$

Prueba	N. de Tablones	Solución Óptima	Solución Voraz	¿Es la solución Voraz óptima?	Porcentaje de error
Prueba1.txt	5	17	19	No	11,76%
Prueba3.txt	5	93	93	Si	0,00%
Prueba4.txt	5	84	136	No	61,90%
Prueba6.txt	5	187	192	No	26,74%
Prueba8.txt	8	245	278	No	13,47%
Prueba10.txt	8	152	187	No	23,03%
Prueba12.txt	12	351	379	No	7,98%
Prueba14.txt	12	513	564	No	9,94%
Prueba16.txt	15	594	622	No	4,71%
Prueba18.txt	15	1470	1489	No	1,29%
Prueba20.txt	18	805	895	No	11,18%
Prueba22.txt	20	3088	3272	No	5,96%
Prueba23.txt	20	780	825	No	5,77%

Complejidad

Para poder calcular la complejidad total del algoritmo, debe hacerse este proceso también con sus funciones auxiliares. En primer lugar, se tiene la función auxiliar **heuristica:**

```python
def heuristica(self, tablon, n):
        tr = tablon[1]
        p = tablon[2]
        ts = tablon[0]
        factor = 0

        if n <= 5:
            factor = 1
        elif n <= 10:
            factor = 1.7
        elif n <= 15:
            factor = 2.2
        else:
            factor = 2.5

        return ts + (ts + tr**factor)/p
```

Cuyo costo, tanto temporal como espacial es O(1) puesto que su función se basa simplemente en hacer cálculos y no almacena estructuras adicionales. Por otro lado, se considera la función auxiliar **calcularCosto:**

```python
def calcularCosto(self, tablon, tiempo):
        if (tablon[0] - tablon[1]) >= tiempo:
            return tablon[0] - (tiempo + tablon[1])
        else:
            return tablon[2] * ((tiempo + tablon[1]) - tablon[0])
```

Cuyo costo, tanto temporal como espacial también es O(1), por la misma razón que la función anterior. Finalmente, se tiene el método principal, **roV** (cuyo código se encuentra un par de páginas más arriba, en el apartado **"Descripción del algoritmo":**

Ya que este algoritmo implica iterar sobre todos los tablones de la finca, y a su vez, seleccionar el tablón óptimo, la complejidad temporal de este apartado es $O(n^2)$, siendo "n" el número de tablones de la finca. Por su parte, el costo espacial de este algoritmo se ve representado por el almacenamiento de la lista de prioridades, lo cual es $O(n)$. Ya que las funciones auxiliares tuvieron costo constante, el costo tanto temporal como espacial de toda la estrategia voraz es el mismo que el obtenido en la función **roV,** es decir,

Complejidad temporal: $O(n^2)$

Complejidad espacial: $O(n)$

Corrección

El algoritmo no siempre da la mejor respuesta, por lo que no es óptimo. Para un caso en particular, podría esto ser demostrado con un contraejemplo:

$F1 = << 10, 3, 4 >, < 5, 3, 3 >, < 2, 2, 1 >, < 8, 1, 1 >, < 6, 4, 2 >>$

Resultado: $(19, [2, 1, 4, 0, 3])$ **¿De dónde salió este resultado?**

La forma de obtenerlo se basa en aplicar a cada tablón la función de heurística, resultando en:

$$< 10 + \frac{10+3^1}{4}, 5 + \frac{5+3^1}{3}, 2 + \frac{2+2^1}{1}, 8 + \frac{8+1^1}{1}, 6 + \frac{6+4^1}{2} > =$$

$$< 7.57142, \quad 7.66666, \quad 6, \quad 17, \quad 11 >$$

Posteriormente, se obtiene el índice con el menor valor en la lista, en este caso, 2, que corresponde al número 6. Se selecciona el tablón en la finca con el mismo índice (2), y se calcula el costo de regarlo:

$$CR_{\mathcal{F}}^{\Pi}[i] = \begin{cases} ts_i^{\mathcal{F}} - (t_i^{\Pi} + tr_i^{\mathcal{F}}) & \text{si } ts_i^{\mathcal{F}} - tr_i^{\mathcal{F}} \geq t_i^{\Pi} \\ p_i^{\mathcal{F}} * ((t_i^{\Pi} + tr_i^{\mathcal{F}}) - ts_i^{\mathcal{F}}) & \text{sino} \end{cases}$$

Ya que se cumple la primera condición, se realiza el primer cálculo.

$< 2, 2, 1 >$ 2 - (0 + 2) = 0.

Se acumula dicho costo en una variable, se retira el tablón de la finca y se procede de la misma manera con los demás. De antemano se sabe cuál será el orden de salida de los tablones, que sería ordenar la lista de las puntuaciones otorgadas por la heurística a cada tablón.

¿Es la solución óptima? No, la solución óptima es $(17, [2, 1, 0, 3, 4])$

Para que el algoritmo sea considerado como óptimo, siempre debe otorgar la solución óptima.

Ahora, para realizar una análisis sobre los casos en los que el algoritmo es óptimo, y en los que no, se ha decidido tomar una pequeña muestra y tratar de analizar patrones existentes en ella. En particular, este es un **conjunto de fincas para los que el algoritmo retorna voraz retorna una solución óptima**:

finca3 = [(10, 9, 3), (9, 3, 4), (8, 8, 1), (10, 10, 3), (8, 3, 3)]

finca4 = [(20, 8, 4), (21, 9, 4), (14, 3, 1), (30, 3, 2), (9, 6, 2)]

finca5 = [(30, 8, 4), (25, 9, 4), (12, 7, 1), (10, 4, 4), (25, 5, 3)]

finca10 = [(6, 1, 4), (13, 10, 2), (24, 4, 4), (24, 3, 2), (5, 7, 3)]

finca13 = [(27, 10, 3), (18, 10, 3), (5, 6, 1), (29, 10, 2), (28, 8, 4)]

finca22 = [(7, 6, 2), (24, 5, 2), (24, 7, 4), (6, 2, 1), (28, 3, 2)]

finca24 = [(8, 7, 2), (14, 6, 1), (18, 4, 1), (15, 2, 3), (5, 2, 2)]

finca26 = [(12, 10, 2), (28, 9, 1), (18, 7, 2), (28, 6, 4), (7, 4, 3)]

Y este es un conjunto de fincas para los que no se obtiene solución óptima:

finca14 = [(30, 1, 3), (10, 7, 2), (23, 6, 2), (22, 1, 2), (14, 5, 4)]

finca15 = [(23, 6, 3), (14, 4, 2), (16, 4, 1), (15, 5, 2), (11, 5, 2)]

finca16 = [(22, 9, 1), (9, 5, 4), (28, 5, 4), (13, 5, 3), (6, 5, 1)]

finca17 = [(25, 10, 1), (17, 8, 3), (9, 7, 4), (30, 2, 2), (23, 4, 4)]

finca18 = [(20, 7, 4), (10, 6, 2), (21, 10, 3), (26, 8, 4), (7, 4, 2)]

finca19 = [(20, 3, 1), (30, 6, 1), (7, 8, 1), (17, 10, 1), (26, 8, 4)]

finca20 = [(24, 4, 4), (19, 2, 2), (13, 3, 2), (29, 1, 4), (14, 9, 1)]

En particular, no hay mucho que se pueda analizar de esta muestra. Algo que sí es notable sobre otras características, es que los tiempos de supervivencia suelen ser generalmente más altos, lo que indica que el algoritmo funciona mejor entre menos penalización haya. En particular, esta afirmación es casi trivial para todo algoritmo voraz, pues bien, al no haber penalización, la importancia de optimizar el orden de riego disminuye notablemente. Cualquier heurística que termine organizando los tablones de manera razonable podría ser considerada como buena en un caso “promedio”, sin importar lo ingenua que esta sea. En muy pocos casos el algoritmo otorga la solución óptima, estas son observadas en valores de n pequeños, por ejemplo, 5 y 8. No está claro bajo qué circunstancia específica el algoritmo otorga una solución óptima en un caso promedio.

Existen otras condiciones triviales bajo las cuales el algoritmo puede dar la solución óptima, por ejemplo, que todos los tablones estén ordenados de manera óptima según el costo de riego. En esta situación el algoritmo simplemente tendría que seguir ese orden. Un posible caso desfavorable para la heurística sería una finca con tiempos de supervivencia muy similares pero con tiempos de riego y prioridades muy diferentes. Al no otorgar el tiempo de supervivencia distinción entre los diferentes tablones, puede que el factor del tiempo de riego no sea lo suficientemente para compensar las diferencias en las prioridades.

Comparación de resultados

Se realizaron pruebas de los algoritmos con 30 fincas diferentes, con el fin de evaluar la eficiencia de los tres enfoques utilizados para resolver el problema de riego óptimo. A continuación se puede evidenciar los tiempos de ejecución obtenidos.

Tener en cuenta que se promedio los tiempos de ejecución en aquellas fincas cuyo número de tablones era el mismo para cada algoritmo.

Tamaño de la Finca	Fuerza Bruta (segundos)	Dinámico (segundos)	Voraz (segundos)
5	0,000355107	0,000141348	0,000042915
8	0,124075254	0,001507680	0,000047445
12	-	0,044633484	0,000067711
15	-	0,607240836	0,000078519
18	-	8,118826866	0,000097156
20	-	50,324780464	0,000120084
25	-	-	0,000174403
30	-	-	0,000203371
35	-	-	0,000256538
40	-	-	0,000320673

Esta tabla evidencia de manera clara lo impracticable que puede usar algoritmos que aseguran una solución óptima pero terminan siendo ineficientes a medida que aumenta el tamaño de la finca, como lo son el algoritmo de fuerza bruta y el de programación dinámica, que a diferencia del algoritmo de programación voraz no lograron dar resultado a todos los casos de prueba propuestos. Entre las principales conclusiones al comparar los resultados se obtuvieron las siguientes:

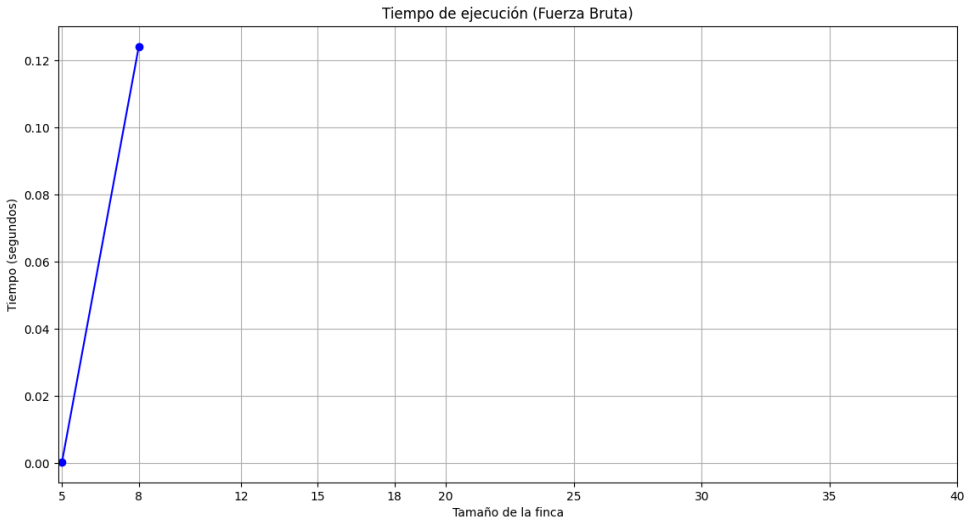
- **Eficiencia Temporal:** El enfoque de programación voraz demostró ser el algoritmo más eficiente en términos de tiempo de ejecución para todas las pruebas realizadas, incluso a medida que aumentaba el número de los tablones. Por otro lado, tanto el algoritmo de fuerza bruta como el de programación dinámica mostraron tiempos de ejecución significativamente más largos, siendo el de programación dinámica más eficiente que el de fuerza bruta en la mayoría de los casos.

- **Escalabilidad:** El algoritmo de fuerza bruta mostró ser impracticable para fincas superiores a 8 números de tablones, cosa que se puede prever con el análisis realizado en secciones anteriores, pues el tiempo de ejecución de este algoritmo aumenta exponencialmente con el tamaño de la finca. La programación dinámica, aunque fue más eficiente que la fuerza bruta, también mostró limitaciones para encontrar el

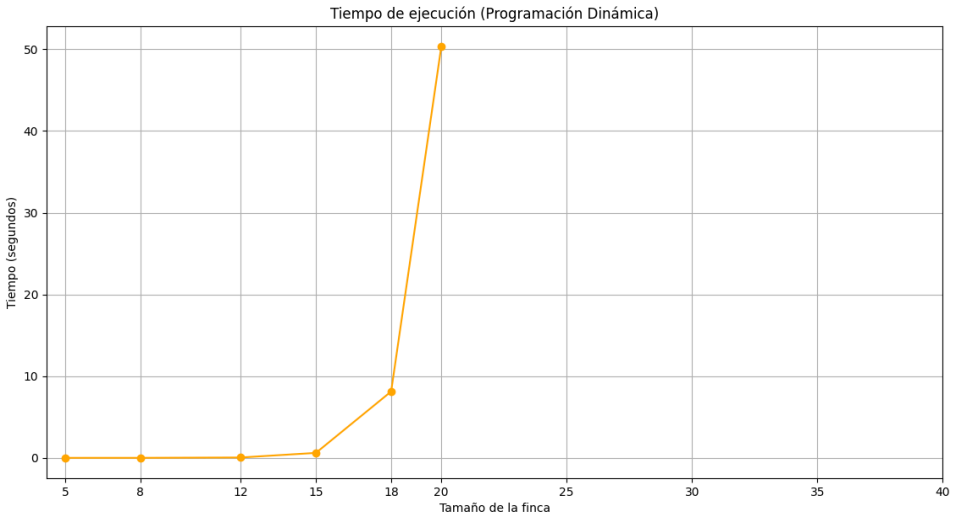
resultado en fincas que superan los 20 tablones. Por último, y a contraste de los dos anteriores, el enfoque voraz se mantuvo eficiente incluso para fincas de mayor tamaño, lo que nos da a entender que es un algoritmo con mejor escalabilidad.

- **Calidad de la solución:** Aunque el enfoque voraz mostró una eficiencia temporal superior a los otros dos enfoques realizados, es importante tener en cuenta que no siempre se produjo la solución óptima con este algoritmo, cosa que se vió en secciones anteriores del informe. Por lo que es importante entender que el algoritmo de programación voraz sacrifica ligeramente la búsqueda de la solución óptima en favor de la eficiencia en el tiempo de ejecución.
-

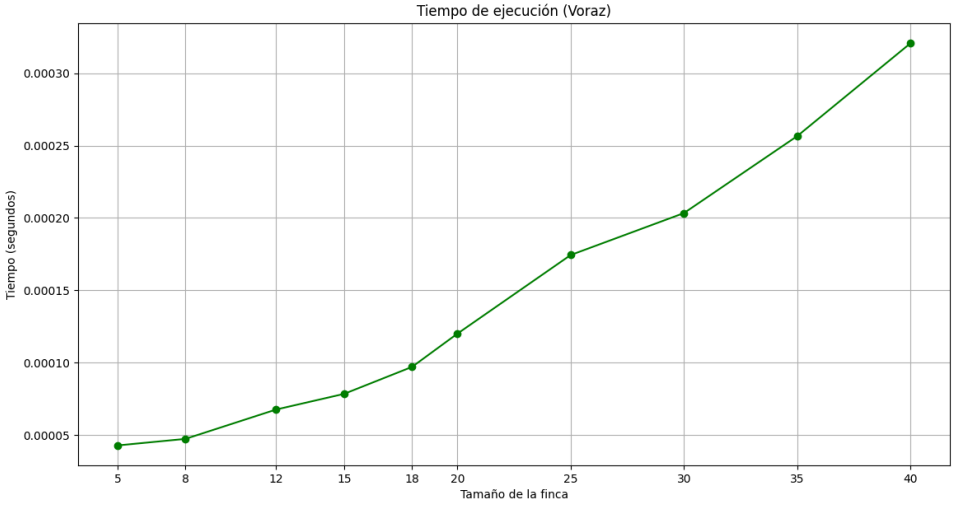
A continuación se evidencia en gráficas los resultados de la anterior tabla.



Se observa una tendencia exponencial en la línea de tiempo de ejecución a medida que aumenta el número de tablones. Esta tendencia sugiere que el algoritmo se vuelve rápidamente inviable debido a su comportamiento exponencial.



Inicialmente se observa un crecimiento gradual y predecible en el tiempo de ejecución. Sin embargo, a partir de alrededor de 18 a 20 tablones, se produce un aumento significativo en el tiempo de ejecución, dicando que el algoritmo deja de ser eficiente a cierto número de tablones.



La gráfica muestra un aumento gradual en el tiempo de ejecución a medida que aumenta el tamaño de la finca, pero este aumento es mucho más moderado en comparación a los otros algoritmos, evidenciando un crecimiento relativamente estable.

Conclusiones

El uso de estrategias como la programación dinámica y voraz permite resolver problemas de manera rápida y eficiente, en contraparte con soluciones más ingenuas como el uso de fuerza bruta, si bien suele ser el primer acercamiento por su naturaleza demuestra ser poco eficiente y útil en problemas.

La programación dinámica desde sus distintos acercamientos (bottom-up y top-down) ofrece una solución óptima en un tiempo menor a formas más ingenuas como fuerza bruta, sin embargo, la dificultad de su realización, sus características propias de uso y complejidad temporal, son grandes puntos a tener en cuenta, especialmente el tema del tiempo de ejecución representa una mejora considerable respecto a un algoritmo de fuerza bruta, garantizando siempre encontrar una solución óptima, no obstante frente a algoritmos voraces óptimos no resulta la opción más atractiva.

Los algoritmos voraces basan sus elecciones en decisiones rápidas que pueden ser o no las más óptimas, a cambio de tener una menor complejidad temporal y para ciertos casos espacial. El mayor inconveniente que se presenta es la posibilidad de que el problema no se pueda modelar de tal forma que el algoritmo voraz siempre arroje la solución óptima, por lo que es necesario evaluar si el problema a tratar permite tolerar cierto porcentaje de error, y si es así, cuánto es. Otro gran obstáculo es encontrar el patrón o estrategia que permita tomar la mejor decisión, cosa que no es fácil puesto que requiere un gran entendimiento de la subestructura óptima del problema.