-

Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Práctica 3

Programación dinámica

Jorge Solán Morote 816259

Francisco Javier Pizarro 821259

09/04/2023

Planteamiento del problema

El problema claramente tiene una gran importancia en la parte de manipulación de textos, de llamadas recursivas y de uso de una estructura de datos que permita almacenar un diccionario de palabras con un tiempo de consulta de si existe o no una palabra en el mismo en tiempo O(1). Por estas razones se ha elegido emplear Golang como lenguaje para el desarrollo de la práctica ya que al ser compilado ofrece un rendimiento muy alto, además de satisfacer todos los requisitos, concretamente destaca en la manipulación de Strings gracias a sus Slices siendo mejor opción que Python por la gran diferencia de rendimiento.

No obstante también existe otro subproblema implícito en la práctica la generación de un diccionario de palabras y de una frase en formato de fichero para cargar los datos de forma dinámica en el programa principal y realizar pruebas con datos de entrada aleatorios, para esta labor al ser relativamente sencilla en coste computacional y para no perder las buenas costumbres de las prácticas anteriores hemos recurrido a Python.

También como es costumbre para la automatización del lanzamiento del programa para probarlo se ha empleado un script en Bash así como para borrar los archivos temporales generados durante la ejecución.

El enfoque inicial del problema fue simplemente resolverlo por fuerza bruta recursiva, dicho enfoque para un diccionario relativamente pequeño es relativamente bueno, no obstante cuando el tamaño del diccionario aumenta y lo mas importante cuantas más posibles combinaciones de palabras que tienen como resultado otra palabra del diccionario al aumentar tanto las posibles combinaciones muchos cálculos se repiten. Para mejorar esta implementación empleamos la programación dinámica al vuelo es decir empleando las mismas funciones de fuerza bruta pero cada vez que encontramos una palabra completa en la búsqueda la almacenamos en una matriz para que la próxima vez que dicha palabra aparezca no tengamos que volver a buscarla es decir si hay una palabra de 10 caracteres ya registrada en cuenta de 10 llamadas recursivas para buscar dicha palabra solo se realizará 1 y se avanzará al final de la misma. El código de esta solución se encuentra en el Anexo I.

Este enfoque ya era significativamente mejor que el anterior no obstante quisimos llevar el problema un paso más allá, en cuenta de calcular la matriz sobre la marcha, esta es precalculada al principio del programa la gran ventaja de esta solución respecto de la anterior aparte del evidente ahorro en llamadas recursivas es que esta búsqueda es extremadamente paralelizable y al estar empleando concretamente el lenguaje Golang el cual está muy centrado en la concurrencia de forma nativa, realizamos dicha implementación con concurrencia en la búsqueda del precálculo, como resultado se obtiene una solución mucho más óptima que la anterior. El código de esta nueva solución se encuentra en el Anexo II.

El código que se encarga de generar los diccionarios y frases se encuentra en el Anexo III.

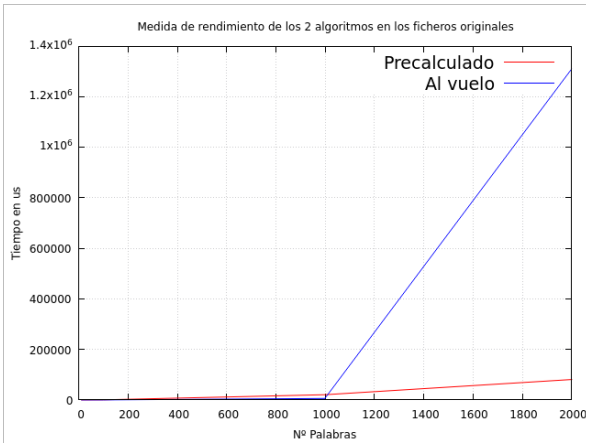
A pesar de que el servidor hendrix.cps.unizar.es no cuenta con el compilador de Golang este se encuentra disponible en el servidor central.cps.unizar.es así como en los ordenadores del Lab 102(155.210.154.191-208)

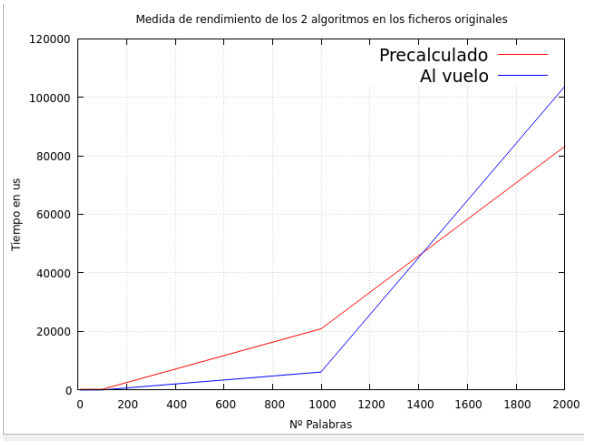
Tests

Teniendo en cuenta que en este caso siempre vamos a ejecutar el programa mediante un script y que el propio programa genera sus propios ficheros de entrada no hemos realizado tests para el control de excepciones por parámetros de entrada erróneos ya que esto nunca va a poder suceder.

Durante el desarrollo de ambas soluciones se empleó un diccionario y una frase elegidas a conciencia y no autogeneradas para buscar errores en su funcionamiento.

Lo interesante de este problema y sus soluciones es la comparación de rendimientos, para ello se han seguido las directrices del guión de prácticas sobre la generación de frases, para aportar ese factor de tener más de una posible combinación en el diccionario existen palabras que combinadas son equivalentes a otras palabras del propio diccionario, concretamente el 7.25% de palabras son el resultado de combinar otras palabras, la longitud de las palabras oscila entre 30 y 60 carácteres, la longitud de la frase oscila entre el número de palabras total entre 100 y entre 10 siendo siempre el mínimo de longitud 1.
En el script de ejecución se emplean 10,100,1000 y 2000 palabras para realizar las pruebas.





Resulta muy evidente la mejora significativa de la solución precalculada de forma concurrente respecto a la solución que genera la matriz al vuelo, conforme más aumenta el número de palabras más evidente resulta esta diferencia, en el proceso de pruebas se han tratado de emplear 3000 palabras pero para esta cantidad el algoritmo concurrente tarda en torno a 7 segundos mientras que el algoritmo original no finaliza en menos de 3 minutos.

ANEXO I

```
func buscarCadenaPD(dic map[string]bool, cadenaBuscada string, n int, encontrados []string,
pseudomatrix [][]string, alreadyCalculated int) bool {
   // Realiza una busqueda del tipo fuerza bruta de forma descendente, en el proceso
   // cada palabra encontrada es almacenada al vuelo en la pseudomatriz para agilizar
   // las siguientes recursiones que la necesiten
   // En caso de que la frase sea una combinación de palabras del diccionario devuelve true

   if cadenaBuscada == "" {
       // fmt.Println(encontrados);
        return true}
   if n > len(cadenaBuscada) {return false}
   if n == 1 && len(pseudomatrix[alreadyCalculated]) != 0 {
       for _, element := range pseudomatrix[alreadyCalculated] {
           buscarCadenaPD(dic, cadenaBuscada[len(element)-1:] , 1,
append(encontrados,element),pseudomatrix,alreadyCalculated+(len(element)-1))
       }
   }
   _, exists := dic[cadenaBuscada[:n]]
   encontradaGlobal := 0
   encontradaLocal := 0
   encontradosNuevos := encontrados
   if exists {
       encontradosNuevos = append(encontrados,cadenaBuscada[:n])
   }
   // esta asignación a entero en cuenta de hacerlo con booleanos directamente
   // es porque en golang el || esta cortocircuitado y el | que usa enteros no.
   if (exists && buscarCadenaPD(dic, cadenaBuscada[n:], 1, encontradosNuevos, pseudomatrix,
alreadyCalculated + 1)) {
       encontradaLocal = 1
       pseudomatrix[alreadyCalculated + 1 - n] = append(pseudomatrix[alreadyCalculated + 1 - n],
cadenaBuscada[:n])

   }
   if (buscarCadenaPD(dic, cadenaBuscada, n+1, encontrados, pseudomatrix, alreadyCalculated + 1))
{encontradaGlobal = 1}
   return  ((encontradaLocal) | (encontradaGlobal)) == 1
}
```

ANEXO II

```
func searchWords(cadenaBuscada string, dic map[string]bool, precalcMatrix [][]string) {
   // Función de precalculo de la matriz
   // Para cada letra de la frase original lanza una gorutina, dicha gorutina busca
   // todas las palabras existentes en el diccionario que comiencen a partir de dicha letra
   // Devuelven los resultados por un canal y se almacena para el indice de dicha letra en la matriz

   // Crear e iniciar canales
   chs := make([]chan []string, len(cadenaBuscada))
   for i := range chs {
       chs[i] = make(chan []string)
   }

   // Crear y lanzar gorutinas
   for i := 0; i < len(cadenaBuscada); i++ {
       go func(start int,ch chan<- []string) {
           var encontradas []string
           for j := start; j < len(cadenaBuscada); j++ {
               prefix := cadenaBuscada[start : j+1]
               if dic[prefix] {
                   encontradas = append(encontradas, prefix)
               }
           }
           ch <- encontradas
       }(i, chs[i])
   }

   // Guardar resultados
   for i, ch := range chs {
       precalcMatrix[i] = <-ch
   }
}

func buscarPDPrecalc(cadenaBuscada string, precalcMatrix [][]string,alreadyCalculated int,
combination string) bool {
   // Empleando la matriz precalculada previamente busca empezando por el principio
   // de la matriz todas las combinaciones de las palabras existentes en la matriz de forma
recursiva
   // Si llega al final de forma exitosa devuelve true

   if cadenaBuscada == "" {
       return true}
   hasCombination := false
   for _, element := range precalcMatrix[alreadyCalculated] {
       hasCombination = hasCombination || buscarPDPrecalc(cadenaBuscada[len(element):],
precalcMatrix ,alreadyCalculated+(len(element)),combination + " " + element)
   }
   return hasCombination
}
```

ANEXO III

```
basic_words = set(''.join(random.choice(string.ascii_lowercase) for _ in
range(random.randint(30,60))) for _ in range(num_words*925//1000))
compound_words = set()
while len(compound_words) < num_words*75//1000:
   bw1 = random.choice(list(basic_words))
   bw2 = random.choice(list(basic_words))
   compound_words.add(bw1 + bw2)
all_words = basic_words.union(compound_words)

first_line = ''.join(random.sample(list(all_words), random.randint(num_words//100 + 1,
num_words//10)))
def modify_sentence(sentence):
   LF = len(sentence)
   probability = 1 / (LF * 10)
   new_sentence = ""
   for index, letter in enumerate(sentence):
       if random.random() <= probability:
           new_sentence += random.choice(string.ascii_lowercase)
       else:
           new_sentence += letter
   return new_sentence

with open('./tmp/f' + sys.argv[1] + '.txt', 'w') as f:
   f.write(first_line + '\n')
   for word in all_words:
       f.write(word + '\n')

with open('./tmp/fMod' + sys.argv[1] + '.txt', 'w') as f:
   f.write(modify_sentence(first_line) + '\n')
   for word in all_words:
       f.write(word + '\n')
```