"Camel Trading" Algoritmo Greedy

Juan Andrés Young Hoyos

2024

Universidad EAFIT
Estructura de Datos y Algoritmos 2

1 El lenguaje de programación utilizado y por qué fue seleccionado

Para resolver este problema he utilizado **Python**. Python fue seleccionado debido a su simplicidad en la manipulación de expresiones matemáticas, la facilidad con la que se pueden procesar listas, y la capacidad de trabajar de forma eficiente con strings y operadores. Además, Python permite una rápida prototipación, lo que facilita realizar pruebas y ajustes al algoritmo de manera ágil.

2 Descripción del algoritmo en pseudocódigo. ¿Por qué funciona?

El algoritmo utiliza una estrategia greedy que agrupa sumas o multiplicaciones, dependiendo de si buscamos maximizar o minimizar el valor de la expresión. El objetivo es obtener las interpretaciones máxima y mínima de una expresión sin paréntesis.

Función para dividir la expresión:

```
dividir_expresion(expresión):
  resultados = []
  numero_actual = ''

  para cada caracter en expresión:
    si el caracter es un dígito:
      agregar a numero_actual
    si el caracter es un operador:
      agregar numero_actual a resultados
      agregar el operador a resultados
      vaciar numero_actual
  si numero_actual no está vacío:
    agregar numero_actual a resultados
  retornar resultados
```

Función para maximizar:

```
maximizar(expresión_completa):
  resultados = []
  actual = primer número de la expresión

  para cada par (operador, número) en la expresión:
    si el operador es '+':
      actual += número
    si el operador es '*':
      agregar actual a resultados
```

```
      actual = número

  agregar actual a resultados
  final_result = 1
  para cada número en resultados:
    final_result *= número
  retornar final_result
```

Función para minimizar:

```
minimizar(expresión_completa):
  resultados = []
  actual = primer número de la expresión

  para cada par (operador, número) en la expresión:
    si el operador es '*':
      actual *= número
    si el operador es '+':
      agregar actual a resultados
      actual = número

  agregar actual a resultados
  final_result = suma de todos los elementos en resultados
  retornar final_result
```

El algoritmo funciona correctamente porque se basa en la propiedad conmutativa de la suma y la multiplicación. Al agrupar sumas primero para la maximización, se garantiza que los números más grandes se multiplican, maximizando el resultado. Al agrupar multiplicaciones primero para la minimización, los números pequeños se suman, minimizando el valor final.

3 Demostración matemática de por qué funciona

El algoritmo se apoya en las siguientes propiedades matemáticas:

- La multiplicación es distributiva respecto a la suma:

$$a \times (b + c) = a \times b + a \times c$$

Esta propiedad nos permite reorganizar las operaciones sin cambiar el valor de la expresión.

- **Maximización**: Al agrupar las sumas primero, estamos asegurando que los valores más grandes se multiplican después. Esto maximiza el resultado total, ya que la multiplicación de números grandes genera un resultado mayor:

$$a + b \times c = (a + b) \times c$$

Si sumamos primero, el valor final será mayor que si multiplicamos primero.

- **Minimización**: Agrupando las multiplicaciones primero, reducimos los valores intermedios, lo que resulta en un valor total menor:

$$a \times b + c = (a \times b) + c$$

Aquí, al multiplicar antes de sumar, el resultado final será más pequeño.

Estas propiedades aseguran que el algoritmo greedy resuelva correctamente el problema de encontrar los valores máximo y mínimo posibles.

4 Descripción de la estrategia codiciosa

La estrategia codiciosa radica en tomar decisiones locales basadas en las operaciones que se están procesando:

- **Maximización**: Se agrupan las sumas antes de las multiplicaciones porque, al sumar primero, se obtiene un valor más grande que luego será amplificado por las multiplicaciones. - **Minimización**: Se agrupan las multiplicaciones antes de las sumas para reducir el valor intermedio, logrando así un resultado menor.

Una vez que se toma una decisión en cada paso (resolver una suma o multiplicación), no se revisa. Esta estrategia es efectiva porque las propiedades conmutativas y distributivas permiten reorganizar las operaciones sin cambiar el significado de la expresión.

5 Demostración de que el algoritmo codicioso es exacto

El algoritmo es exacto porque, como se ha demostrado matemáticamente, agrupar sumas primero maximiza el resultado, y agrupar multiplicaciones primero lo minimiza. En cada paso, las decisiones greedy locales conducen al resultado global óptimo.

6 Definición de las funciones utilizadas

6.1 Función `split_expression`

- **Nombre de la función:** `split_expression`
- **Parámetros que recibe:** `expression` (string): La expresión matemática a dividir.
- **Tipo de dato que retorna:** `list` de strings (números y operadores).
- **Excepciones que produce:** Ninguna.
- **Descripción corta de qué hace:** Convierte una expresión matemática en una lista de tokens (números y operadores).

6.2 Función `maximize`

- **Nombre de la función:** `maximize`
- **Parámetros que recibe:** `full_expression` (list): Lista de números y operadores que representan la expresión completa.
- **Tipo de dato que retorna:** `int`: El valor máximo calculado.
- **Excepciones que produce:** Ninguna.
- **Descripción corta de qué hace:** Calcula el valor máximo posible de la expresión agrupando sumas primero.

6.3 Función `minimize`

- **Nombre de la función:** `minimize`
- **Parámetros que recibe:** `full_expression` (list): Lista de números y operadores que representan la expresión completa.
- **Tipo de dato que retorna:** `int`: El valor mínimo calculado.
- **Excepciones que produce:** Ninguna.
- **Descripción corta de qué hace:** Calcula el valor mínimo posible de la expresión agrupando multiplicaciones primero.

7 Descripción de las variables

Nombre de la variable	**Tipo de dato**	**Para qué se utiliza**	**Visibilidad**	**Ciclo de vida**
`full_expression`	list (de strings)	Almacena los números y operadores de la expresión	Local en `main`	Se crea en `main` y se destruye al finalizar `main`
`number`	str	Construye los números mientras se procesa la expresión	Local en `split_expression`	Se crea en `split_expression` y se destruye al finalizar
`current`	int	Almacena el valor acumulado en cada operación	Local en `maximize` y `minimize`	Se crea al inicio de cada función y se destruye al finalizar
`results`	list (de int)	Almacena los valores intermedios de sumas o multiplicaciones	Local en `maximize` y `minimize`	Se crea al inicio de cada función y se destruye al finalizar
`op`	str	Almacena el operador actual (+ o *)	Local en `maximize` y `minimize`	Se crea en cada iteración del ciclo y se destruye inmediatamente después
`num_cases`	int	Número de casos de prueba	Local en `main`	Se crea al leer el input y se destruye al finalizar `main`

Table 1: Descripción de las variables