



Análise de algoritmos

ALGORITMO DE DIJKSTRA

Baseado no livro Entendendo Algoritmos, Um guia ilustrado para programadores e outros curiosos. Aditya Y. Bhargava, Novatec, 2017.





Análise de algoritmos

ALGORITMO DE DIJKSTRA

Figuramente, você está planejando uma expedição para escalar uma montanha, uma vez que o objetivo é encontrar o caminho mais curto e seguro para chegar ao topo. O algoritmo de *Dijkstra* funciona de maneira similar, mas em forma de grafos ponderados ou valorados, salvo para ciclos em grafos e para pesos negativos nas arestas em que não é aplicável. O peso de um grafo poderá indicar distância, custo ou tempo.

Retomando o conceito de *Breadth-First Search* ou pesquisa em largura, é melhor indicado para calcular o caminho com o menor número de arestas, ou seja, quando trabalhamos com grafos não ponderados (não valorados), por outro lado, o caminho mais rápido é obtido pelo *Dijkstra*. Para entendermos como ocorre na prática, vamos percorrer as etapas:

- Encontrar o vértice mais “barato” que indicará o caminho mais rápido;
- Verificar se há um caminho alternativo que seja mais “barato” para os vizinhos do vértice, atualizando os custos dos vizinhos do vértice atual,





Análise de algoritmos

ALGORITMO DE DIJKSTRA

- outrora indefinido e tido como infinito;
- Repetir as etapas até o final do grafo e;
- Calcular o caminho final.

Conforme abordado anteriormente, um grafo com ciclos nunca indicará o caminho de menor peso, visto que ao viajar num vértice e terminar nele mesmo adicionará mais peso no peso total. E assim como a pesquisa em largura, podemos implementar o grafo ponderado fazendo uso de tabelas *hash* (estrutura estudada anteriormente), divididos em tabelas destinadas aos nomes das arestas do grafo, custo das arestas e pais das arestas.

Referente aos tempos de execução Big O no pior caso, a complexidade do algoritmo Big O é de $O(V^2)$ (número de vértices do grafo) + E (número de arestas no grafo)), já no médio caso é mais complexa a análise com exatidão, pois depende da distribuição dos pesos das arestas e estrutura do grafo. O desempenho no médio caso é próximo ao pior caso,





Análise de algoritmos

ALGORITMO DE DIJKSTRA

especialmente em grafos mais densos.

APLICAÇÕES

De maneira geral, podemos encontrar o algoritmo de grafos ponderados nos seguintes casos:

- Roteamento em rede de computadores: rota mais curta entre dispositivos e para tráfego de dados;
- Planejamento de rotas: rotas otimizadas;
- Análise de redes sociais: relacionamento de grau mais próximo possível entre usuários e análise de propagação de informações;
- Análise de circuitos elétricos: caminho de menor resistência;
- Bioinformática: análise de sequência genética e identificar padrões;
- Jogos e inteligência artificial: caminho mais rápido para atingir o objetivo e algoritmos de busca em jogos de estratégia e;
- Logística: resolução de problemas de otimização logísticos.





Análise de algoritmos

ALGORITMO DE DIJKSTRA

De forma geral, as vantagens da implementação de um algoritmo de grafo consistem na versatilidade, simplicidade, eficiência e completude. Por outro lado, temos como desvantagens o consumo de memória, ineficiência em alguns grafos e pode não encontrar o caminho ideal para todos os casos.

Para o algoritmo abaixo, teremos um algoritmo que fará o caminho de esforço mínimo em uma matriz 2D (*heights*) de tamanho *rows* x *columns*, onde *heights[row][col]* representa a altura da célula (*row*, *col*). Partindo da célula superior esquerda, (0, 0), é possível viajar pra cima, pra baixo, esquerda ou direita, em busca da rota que exija menos esforço possível. O esforço de uma rota é a diferença absoluta máxima de altura entre duas células consecutivas da rota. Retorne o esforço mínimo necessário para viajar da célula superior esquerda para a célula inferior direita.





Análise de algoritmos

ALGORITMO DE DIJKSTRA

1	2	2
3	8	2
5	3	5

Extraído do Leetcode.

Entrada: **alturas** = `[[1,2,2],[3,8,2],[5,3,5]]`

Saída: 2

Explicação: A rota de `[1,3,5,3,5]` tem uma diferença absoluta máxima de 2 em células consecutivas. Isso é melhor do que a rota de `[1,2,2,2,5]`, onde a diferença absoluta máxima é 3.





🔍 ALGORITMO DE DIJKSTRA



```
package main

import (
    "bufio"
    "container/heap"
    "fmt"
    "math"
    "os"
    "strconv"
    "strings"
)
```





ALGORITMO DE DIJKSTRA



```
type Item struct {  
    effort int  
    x      int  
    y      int  
}  
  
type PriorityQueue []Item  
  
func (p PriorityQueue) Len() int { return len(p) }  
  
func (p PriorityQueue) Less(i, j int) bool {  
    return p[i].effort < p[j].effort  
}
```





🔍 ALGORITMO DE DIJKSTRA



```
func (p PriorityQueue) Swap(i, j int) bool {  
    p[i], p[j] = p[j], p[i]  
}
```

```
func (p *PriorityQueue) Push(x interface{}) {  
    *p = append(*p, x.(Item))  
}
```

```
func (p *PriorityQueue) Pop() interface{} {  
    old := *p  
    n := len(old)  
    item := old[n-1]
```





🔍 ALGORITMO DE DIJKSTRA



```
*p = old[0 : n-1]
return item
}

func minimumEffortPath(heights [][]int) int {
    rows, columns := len(heights), len(heights[0])
    distance := make([][]int, rows)
    for i := range distance {
        distance[i] = make([]int, columns)
        for j := range distance[i] {
            distance[i][j] = math.MaxInt32
        }
    }
}
```





Q ALGORITMO DE DIJKSTRA



```
}  
distance[0][0] = 0  
directions := [][]int{{0, 1}, {0, -1}, {1, 0}, {-1, 0}}  
  
p := &PriorityQueue{Item{0, 0, 0}}  
heap.Init(p)  
  
for p.Len() > 0 {  
    item := heap.Pop(p).(Item)  
    effort, x, y := item.effort, item.x, item.y  
    if effort > distance[x][y] {  
        continue  
    }  
}
```



ALGORITMO DE DIJKSTRA



```
}  
if x == rows-1 && y == columns-1 {  
    return effort  
}  
for _, dir := range directions {  
    nx, ny := x+dir[0], y+dir[1]  
    if nx >= 0 && nx < rows && ny >= 0 && ny <  
columns {  
        newEffort := int(math.Max(float64(effort),  
math.Abs(float64(heights[x][y]-heights[nx][ny])))  
        if newEffort < distance[nx][ny] {  
            distance[nx][ny] = newEffort  
        }  
    }  
}
```





ALGORITMO DE DIJKSTRA



```
        heap.Push(p, Item{newEffort, nx, ny})
    }
}
return -1
}
```

```
func main() {
    reader := bufio.NewReader(os.Stdin)

    fmt.Print("enter the number of rows: ")
}
```





Q ALGORITMO DE DIJKSTRA



```
rows, _ :=
strconv.Atoi(strings.TrimSpace(readLine(reader)))

fmt.Print("enter the number of columns: ")
columns, _ :=
strconv.Atoi(strings.TrimSpace(readLine(reader)))

heights := make([][]int, rows)
for i := 0; i < rows; i++ {
    fmt.Printf("enter row %d: ", i+1)
    line := strings.TrimSpace(readLine(reader))
    values := strings.Split(line, " ")
    heights[i] = make([]int, columns)
```





🔍 ALGORITMO DE DIJKSTRA



```
    for j := 0; j < columns; j++ {  
        heights[i][j], _ = strconv.Atoi(values[j])  
    }  
}
```

```
minEffort := minimumEffortPath(heights)  
fmt.Printf("minimum effort: %d\n", minEffort)  
}
```

```
func readLine(reader *bufio.Reader) string {  
    text, _ := reader.ReadString('\n')  
    return strings.TrimSpace(text)  
}
```





SAÍDA DO PROGRAMA



CENÁRIO OTIMISTA

enter the number of rows: 5
enter the number of columns: 5
enter row 1: 1 2 1 1 1
enter row 2: 1 2 1 2 1
enter row 3: 1 2 1 2 1
enter row 4: 1 2 1 2 1
enter row 5: 1 1 1 2 1
minimum effort: 0





Análise de algoritmos

CONSIDERAÇÕES

Para o exemplo apresentado, seguiu-se as boas práticas de desenvolvimento com *clean code* e SOLID, na tentativa de simular um cenário otimista.

Essa iniciativa vai de encontro com a ideia de trazer conteúdos relevantes altamente abordados em processos seletivos e desmitificar a ideia de algoritmos e estrutura de dados. Espero que seja de bom proveito e bons estudos.