CS112.L21.KHCL.N06 – Giảng viên: Nguyễn Thanh Sơn

COMPLETE SEARCH BACKTRACKING

Nhóm 6:
Bùi Thị Bích Hậu - 19521483
Trần Tiến Hưng – 19521587
Lê Vinh Quang - 19522093

Tìm đường đi cho mê cung

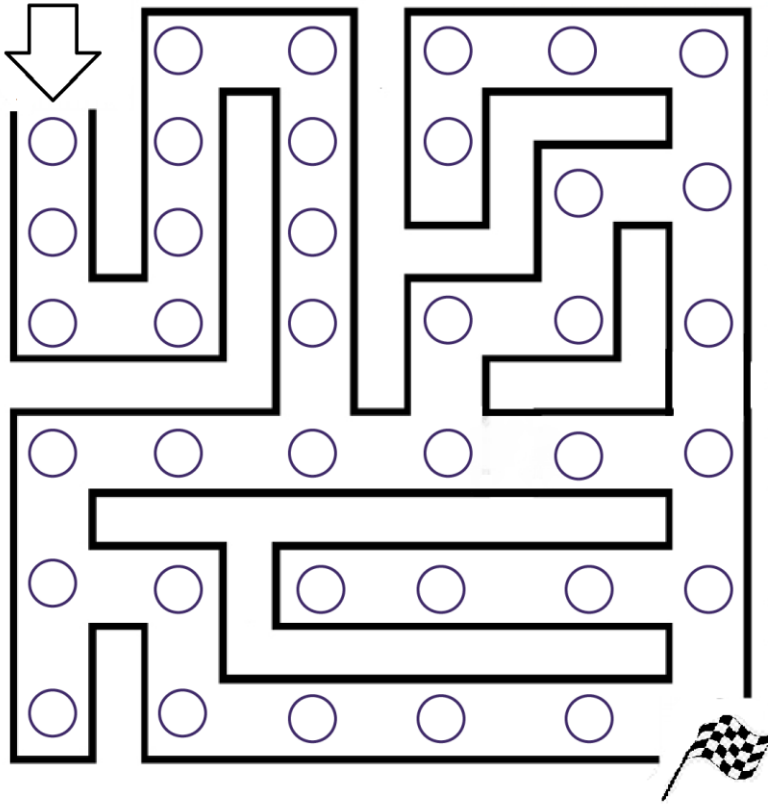


Table of Content

1. **Định nghĩa:**
   a) Backtracking là gì?
   b) Khái niệm State-Space Tree.

2. **Nhận dạng bài toán:**
   a) Vấn đề của Backtracking.
   b) Cách nhận dạng Backtracking.

3. **Phương pháp thực hiện:**
   a) Bài tập ứng dụng.

4. **Đặc điểm**
   a) Ưu điểm
   b) Nhược điểm

1. Định nghĩa

a) Backtracking:

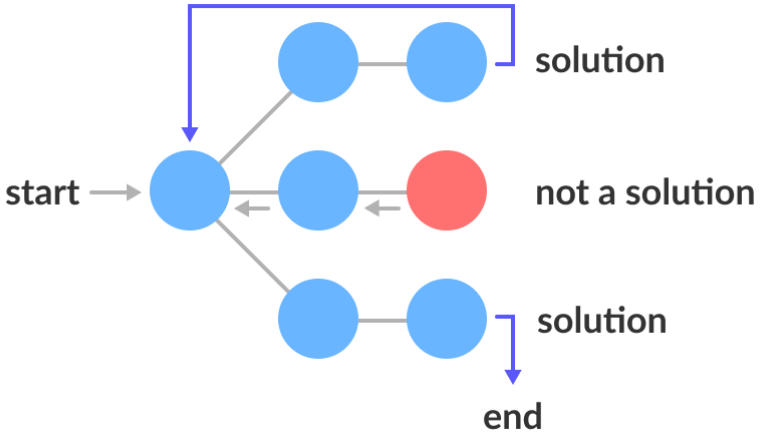
- Backtracking là một kỹ thuật tổng quát xem xét việc **tìm kiếm mọi lời giải** có thể có để giải quyết một vấn đề tính toán.

b) Ý tưởng:

- Xây dựng giải pháp theo từng thành phần và đánh giá từng thành phần đó
- Nếu thành phần được xây dựng phát triển tiến dần tới giải pháp mà không bị ràng buộc bởi các điều kiện bài toán, thì nó được phát triển từ các thành phần của sự chọn lựa thành phần giải pháp tiếp theo.
- Ngược lại, nếu không có thành phần hợp lý hay thành phần thay thế nào không phát triển được, thì thuật toán sẽ quay lui về thành phần được xây dựng cuối cùng bởi giải pháp với sự lựa chọn tiếp theo.

1. State-Space Tree:

- State-Space Tree (cây không gian trạng thái) là cây **đại diện** cho **tất cả** các trạng thái có thể có (giải pháp hoặc không giải) của bài toán từ gốc là trạng thái ban đầu đến lá là trạng thái cuối.



2. Nhận dạng:

a) Vấn đề của Backtracking:

Backtracking được sử dụng khi bài toán gồm những vấn đề sau:

- Vấn đề **Quyết định** - Tìm kiếm một giải pháp khả thi.
- Vấn đề **Tối ưu hóa** – Tìm kiếm giải pháp tốt nhất.
- Bài toán **Liệt kê** – Tìm kiếm tất cả các giải pháp khả thi.

2. Nhận dạng:

b) Nhận dạng bài toán sử dụng backtracking:

- Mọi **vấn đề có các ràng buộc** và được xác định rõ ràng đối với bất kỳ giải pháp khách quan nào, sẽ dần dần xây dựng ứng viên đến giải pháp và loại bỏ một thành phần (“backtracks”) ngay khi xác định rằng thành phần đó không thể hoàn thành giải pháp, có thể được giải quyết bằng Backtracking

* State-Space Tree:

a) Cách thức hoạt động:

Gốc của cây thể hiện input của bài toán trước khi thực hiện công việc tìm kiếm lời giải.

Các node ở tầng [1] tượng trưng cho các giải pháp như là thành phần đầu tiên.

Các node ở tầng [2] thể hiện sự chọn lựa cho thành phần thứ 2, và cứ thế cho các tầng phía dưới.

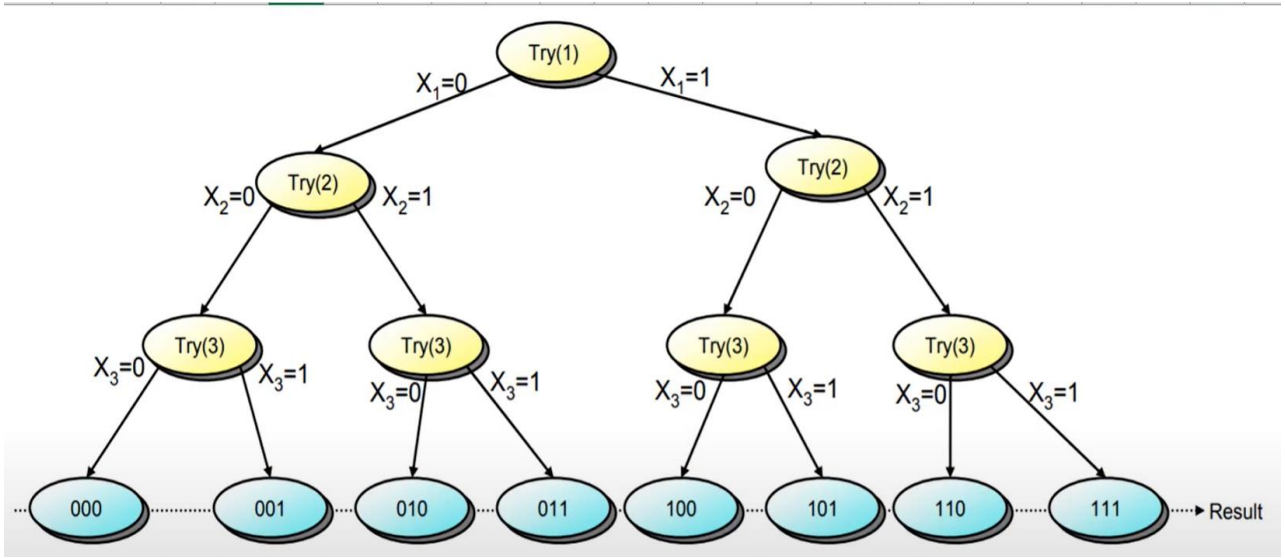
Một node (thành phần) không bị ràng buộc bởi các điều kiện bài toán thì được gọi là thỏa mãn, ngược lại là không thỏa mãn.

Node lá được xem như là 1 đường cụt của lời giải hoặc là đích đến của một lời giải hoàn chỉnh.

State-Space Tree được xây dựng tuân theo phương pháp Depth-first Search (DFS).

3. Phương pháp thực hiện:

Bài toán : Sinh ra tất cả các khả năng của chuỗi nhị phân n bit



3. Phương pháp thực hiện:

```
1
2   n=int(input())
3   a=[]
4   for i in range(n):
5       a.append(0)
6
7   def print1(array1):
8       for i in array1:
9           print(i,end='')
10      print()
11  def Try (i):
12      for j in range(2): #duyet tat ca kq co the thu
13          a[i]=j
14          if i==n-1:
15              print1(a)
16          else:
17              Try(i+1)
18  Try(0)
19
20
21
```

3. Phương pháp thực hiện:

Cho tập A có **n** phần tử và số nguyên **k**, hãy tìm tất cả **các tập con** của A có **tổng các phần tử** bằng **k.**

- Ta sẽ phát sinh ra tất cả các chuỗi nhị phân n bit tương ứng với các tập hợp con

Ví dụ n=4 thì tập có dạng { x1, x2, x3, x4 }

0000: {}
0001: { x4 }
0010: { x3 }
0011: { x3, x4 }
....
1111: { x1, x2, x3, x4 }

Khi phát sinh được các tập hợp con ta sẽ kiểm tra nếu có tổng bằng k thì in ra màn hình

```python
n,k=input().split()
n=int(n)
k=int(k)

example=list(map(int,input().split()))
bitarray=[]
for i in range(n):
    bitarray.append(0)

def Try (i):
    for j in range(2): #duyet tat ca kq co the thu
        bitarray[i]=j
        if i==n-1:
            sum=0
            b=[]  # tao list la tap con cua mang
            for i in range(n):
                if bitarray[i] ==1:    # dựa theo dãy bit mà tạo được tập con tương ứng
                    b.append(example[i])
                    sum+=example[i]
            if sum==k:
                print(b)


        else:
            Try(i+1)
Try(0)
```

4. Đặc điểm:

a) Ưu điểm:

- Kỹ thuật đơn giản và dễ mã hóa.
- Các trạng thái khác nhau được lưu trữ thành ngăn xếp để dữ liệu hoặc thông tin có thể được sử dụng bất cứ lúc nào.

4. Đặc điểm:

b) Nhược điểm:

- Không hiệu quả để giải quyết các vấn đề chiến lược.
- Thời gian chạy tổng thể của thuật toán thường chậm.
- Cần dung lượng bộ nhớ lớn để lưu trữ các chức năng trạng thái khác nhau trong ngăn xếp cho vấn đề lớn.

Bài toán

Bài toán tìm đường đi trong mê cung

Ý tưởng:

- Khởi tạo ma trận 0
- Tạo hàm đệ qui, input là ma trận ban đầu, output là ma trận mới g ồm vị trí [x,y] ban đầu.
- Nếu vị trí không nằm trong ma trận hoặc nằm ngoài ngoại vi của ma trận -> return False
- Đánh dấu [x,y] = 1 trong ma trận mới và check nếu vị trí đó có p hải là đích đến hay không. Nếu phải thì in ma trận mới
- Gọi đệ qui theo [x+1, y] và [x, y+1]
- Nếu không đến đích theo các 4 hướng thì [x, y] = 0 (Backtrack)

Nguồn tham khảo: https://www.geeksforgeeks.org/rat-in-a-maze-backtracking-2/

```python
# Hàm in ma trận đường đi
def printSolution( sol ):

    for i in sol:
        for j in i:
            print(str(j) + " ", end ="")
        print("")
```

```python
# Hàn Check nếu [x, y] thoả điều kiện (constraints)
def isSafe( maze, x, y ):

    if x >= 0 and x < N and y >= 0 and y < N and maze[x][y] == 1:
        return True

    return False
```

```
""" Hàm dùng để giải dựa vào đệ qui với những contraints, trả về
    False nếu không tìm thấy hướng giải, nếu có giải pháp thì trả về
    ma trận mới chứa giá trị 1 là đường đi"""
def solveMaze( maze ):

    # Tạo ma trận mới 4*4
    sol = [ [ 0 for j in range(4) ] for i in range(4) ]

    if solveMazeUtil(maze, 0, 0, sol) == False:
        print("Solution doesn't exist");
        return False

    printSolution(sol)
    return True
```

```python
# Đệ qui dùng để giải
def solveMazeUtil(maze, x, y, sol):

    # Nếu đạt tới đích return True
    if x == N - 1 and y == N - 1 and maze[x][y]== 1:
        sol[x][y] = 1
        return True

    # Check các constraints:
    if isSafe(maze, x, y) == True:
        # Check nếu vị trí đó đã có trong vị trí đã được thăm hay không.
        if sol[x][y] == 1:
            return False

        # Đánh dấu trong ma trận sol[x,y]
        sol[x][y] = 1

        # Tiến theo chiều x
        if solveMazeUtil(maze, x + 1, y, sol) == True:
            return True

        # Nếu tiến theo chiều x mà dính đường cụt
        # Tiến theo chiều y
        if solveMazeUtil(maze, x, y + 1, sol) == True:
            return True

        # Nếu tiến theo chiều y mà dính đường cụt
        # Lùi về x-1
        if solveMazeUtil(maze, x - 1, y, sol) == True:
            return True

        # Nếu bước x-1 bị đường cụt
        # Lùi về y-1
        if solveMazeUtil(maze, x, y - 1, sol) == True:
            return True

        # Nếu tất cả 4 hướng không dẫn tới đích
        # BACKTRACK: Bỏ ô [x, y] được đánh dấu trong đường đi
        sol[x][y] = 0
        return False
```

Tham khảo:

Anany Levitin, Introduction to the Design and Analysis of Algorithms, 3rd Edition, 2014File

https://www.geeksforgeeks.org/rat-in-a-maze-backtracking-2/

https://www.youtube.com/watch?v=CBXlJcPqj1w

Bài tập về nhà: (19521587@gm.uit.edu.vn)

Cho mảng A gồm n phần tử. Liệt kê tất cả các mảng con chứa ít n hất 2 phần tử có vị trí liên tiếp nhau trong mảng A.. Nếu không c ó thì print('False').

Input:

- Dòng đầu tiên là n phần tử.
- Dòng hai là mảng A gồm n phần tử.

Output:

- Các tập con chứa ít nhất 2 phần tử có vị trí liên tiếp nhau.

INPUT	OUTPUT
4 3 5 6 7	6 7 5 6 3 5 3 6 7 3 5 7 3 5 6 3 5 6 7