

# Laboratórios de Algoritmia II

Teórica 4

# Pesquisa exaustiva

- Técnica genérica de resolução de problemas
- Também conhecida por pesquisa por “força bruta”
- Enumera todos os possíveis candidatos a solução até encontrar um que seja válido
- Eficiência proporcional ao número de candidatos
- Infelizmente, o número de candidatos tende a crescer muito rápido quando comparado com o tamanho do problema...
- Mas para muitos problemas é a melhor técnica de resolução

# Algoritmo básico

```
# devolve o primeiro candidato para um problema p
def first(p):

# dado um candidato c devolve o próximo candidato para p
def next(p,c):

# testa se um candidato c é uma solução válida para p
def valid(p,c):

# procurar solução para p com pesquisa exaustiva
def search(p):
    c = first(p)
    while (c != None):
        if valid(p,c):
            return c
        c = next(p,c)
```

# String matching

- Dadas duas strings  $s$  e  $m$ , determinar em que posição  $m$  ocorre em  $s$  usando pesquisa exaustiva
- Os candidatos a solução são todas posições de  $s$  onde  $m$  pode ocorrer

# String matching

```
def first(s,m):  
    return 0  
  
def next(s,m,c):  
    if c < len(s):  
        return c+1  
  
def valid(s,m,c):  
    return s[c:c+len(m)] == m  
  
def search(s,m):  
    c = first(s,m)  
    while (c != None):  
        if valid(s,m,c):  
            return c  
        c = next(s,m,c)
```

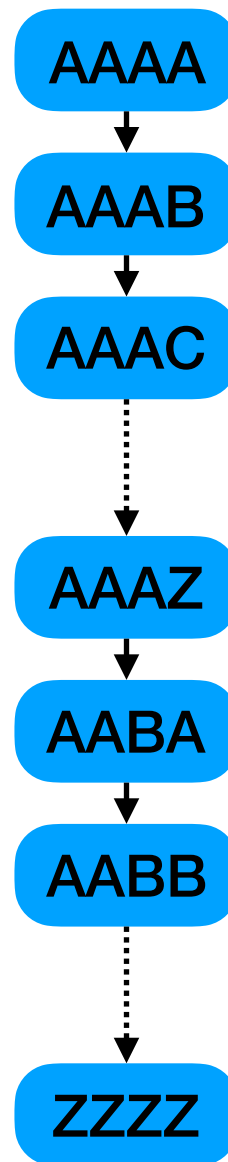
# Password hacking

- Um sistema tem passwords de 4 caracteres maiúsculos
- Por questões de segurança só é guardado e comparado o hash da password
- Um hacker rouba o ficheiro com os hashes
- Dado um hash  $h$  quer descobrir a respectiva password usando pesquisa exaustiva
- Os candidatos a solução são todas as sequências possíveis de 4 caracteres maiúsculos

# Password hacking

```
def first(h):  
    return ['A']*4  
  
def next(h,c):  
    i = len(c) - 1  
    while i >= 0 and c[i] == 'Z':  
        c[i] = 'A'  
        i = i-1  
    if i < 0:  
        return None  
    c[i] = chr(ord(c[i])+1)  
    return c  
  
def valid(h,c):  
    return h == hash(c)  
  
def search(h):  
    c = first(h)  
    while (c != None):  
        if valid(h,c):  
            return c  
        c = next(h,c)
```

# Password hacking





# Enumeração exaustiva com backtracking

- Quando os candidatos são compostos por vários elementos (sequências, dicionários, etc) é possível usar uma técnica de enumeração diferente
- Os candidatos são construídos incrementalmente acrescentando um elemento de cada vez até o candidato estar completo
- Depois de testar um candidato completo ou abandonar um candidato parcial inviável, faz-se backtracking para explorar outras alternativas
- Pode levar a ganhos de eficiência consideráveis

# Backtracking

## (versão para listas)

```
# testa se o candidato c está completo
def complete(p,c):

# enumera as extensões válidas para o candidato parcial c
def extensions(p,c):

# testa se um candidato c é uma solução válida para p
def valid(p,c):

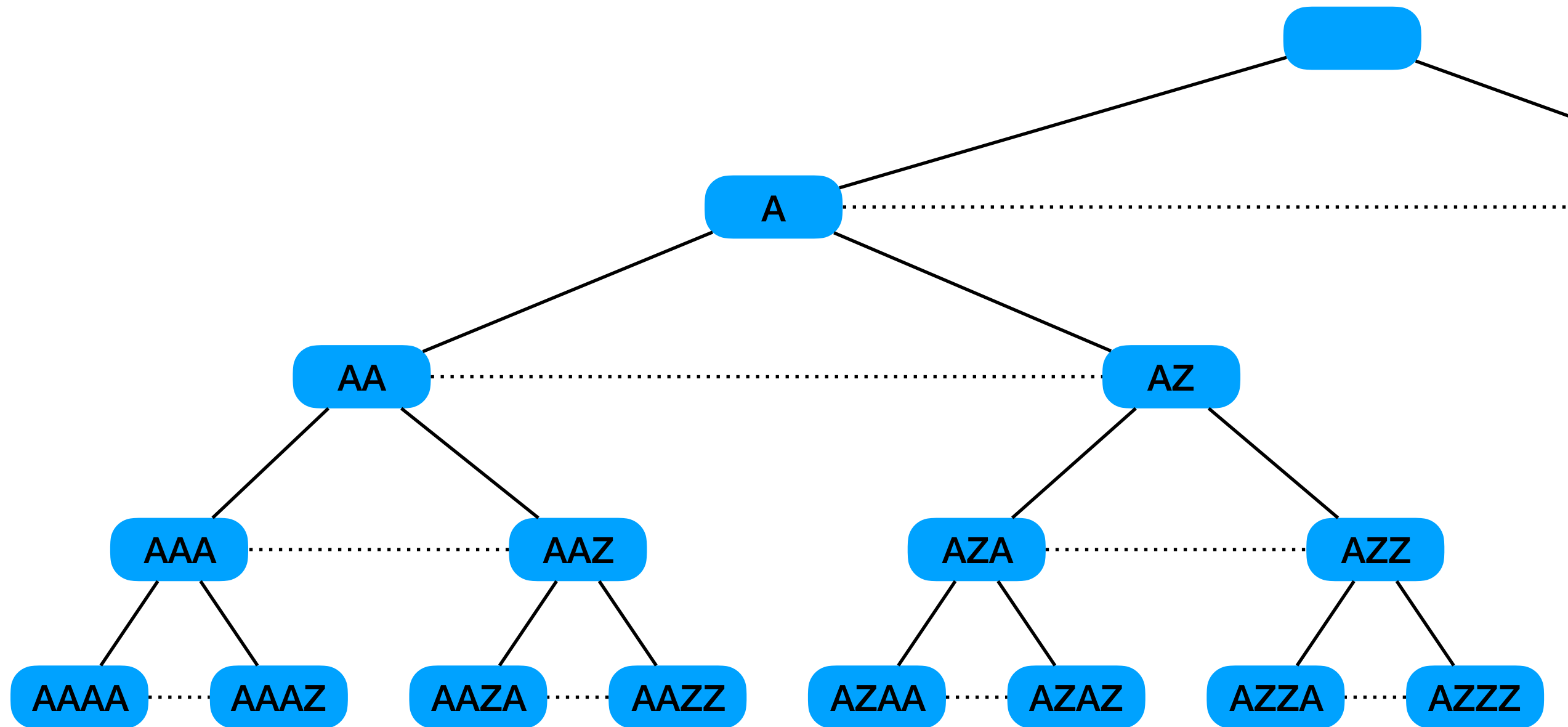
# procurar solução para p com pesquisa exaustiva
def search(p):
    c = []
    if aux(p,c):
        return c

def aux(p,c):
    if complete(p,c):
        return valid(p,c)
    for x in extensions(p,c):
        c.append(x)
        if aux(p,c):
            return True
    c.pop()
    return False
```

# Password hacking

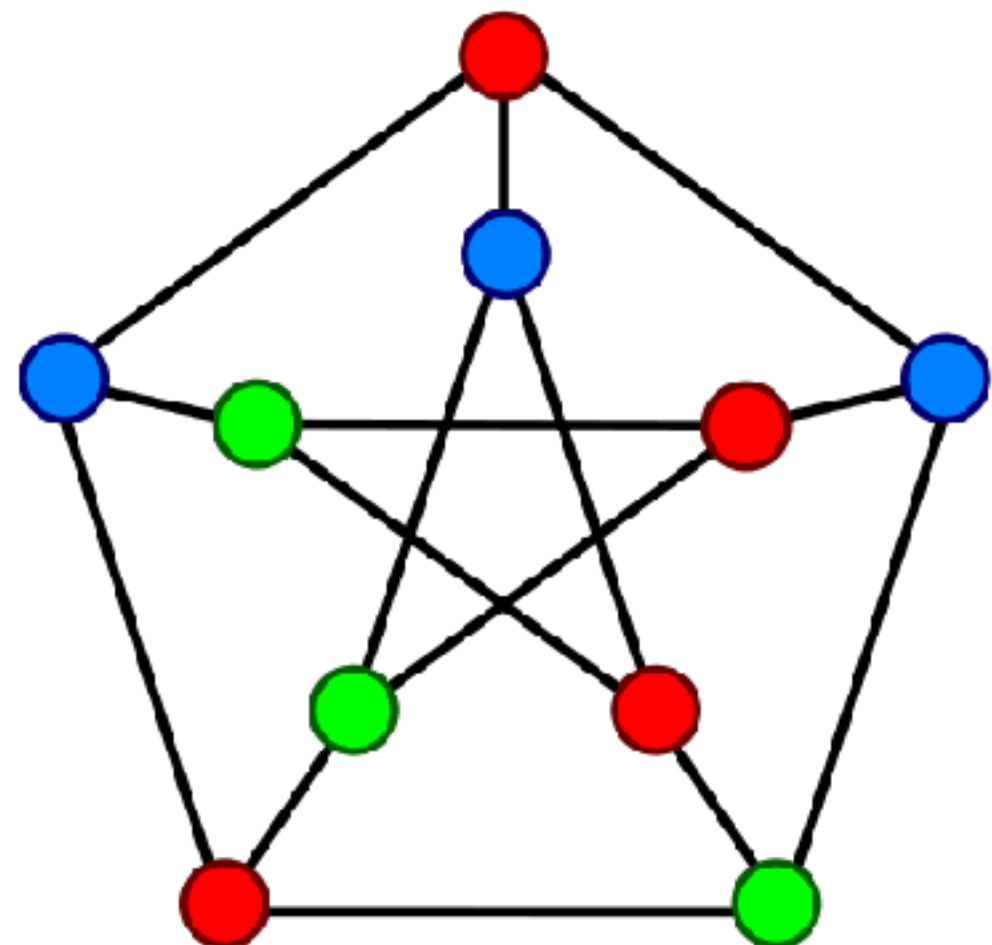
```
def complete(h,c):  
    return len(c) == 4  
  
def extensions(h,c):  
    return [chr(x) for x in range(ord('A'),ord('Z')+1)]  
  
def valid(h,c):  
    return h == hash(c)  
  
def search(h):  
    c = []  
    if aux(h,c):  
        return c  
  
def aux(h,c):  
    if complete(h,c):  
        return valid(h,c)  
    for x in extensions(h,c):  
        c.append(x)  
        if aux(h,c):  
            return True  
        c.pop()  
    return False
```

# Password hacking



# Coloração de grafos

- Testar se é possível colorir os nós de um grafo com  $k$  cores
- Uma coloração só é válida se nós adjacentes tiverem cores diferentes
- O número cromático de um grafo é o número mínimo de cores que é necessário para colorir um grafo



# Backtracking

## (versão para dicionários)

```
# testa se o candidato c está completo
def complete(p,c):

# enumera as extensões válidas para o candidato parcial c
def extensions(p,c):

# testa se um candidato c é uma solução válida para p
def valid(p,c):

# procurar solução para p com pesquisa exaustiva
def search(p):
    c = {}
    if aux(p,c):
        return c

def aux(p,c):
    if complete(p,c):
        return valid(p,c)
    for i,x in extensions(p,c):
        c[i] = x
        if aux(p,c):
            return True
        c.pop(i)
    return False
```

# Coloração de grafos

```
def complete(adj,k,c):  
    return len(adj) == len(c)  
  
def extensions(adj,k,c):  
    i = list(adj.keys())[len(c)]  
    return [(i,x) for x in range(k)]  
  
def valid(adj,k,c):  
    for o in adj:  
        for d in adj[o]:  
            if c[o] == c[d]: return False  
    return True  
  
def search(adj,k):  
    c = {}  
    if aux(adj,k,c): return c  
  
def aux(adj,k,c):  
    if complete(adj,k,c): return valid(adj,k,c)  
    for i,x in extensions(adj,k,c):  
        c[i] = x  
        if aux(adj,k,c): return True  
        c.pop(i)  
    return False
```

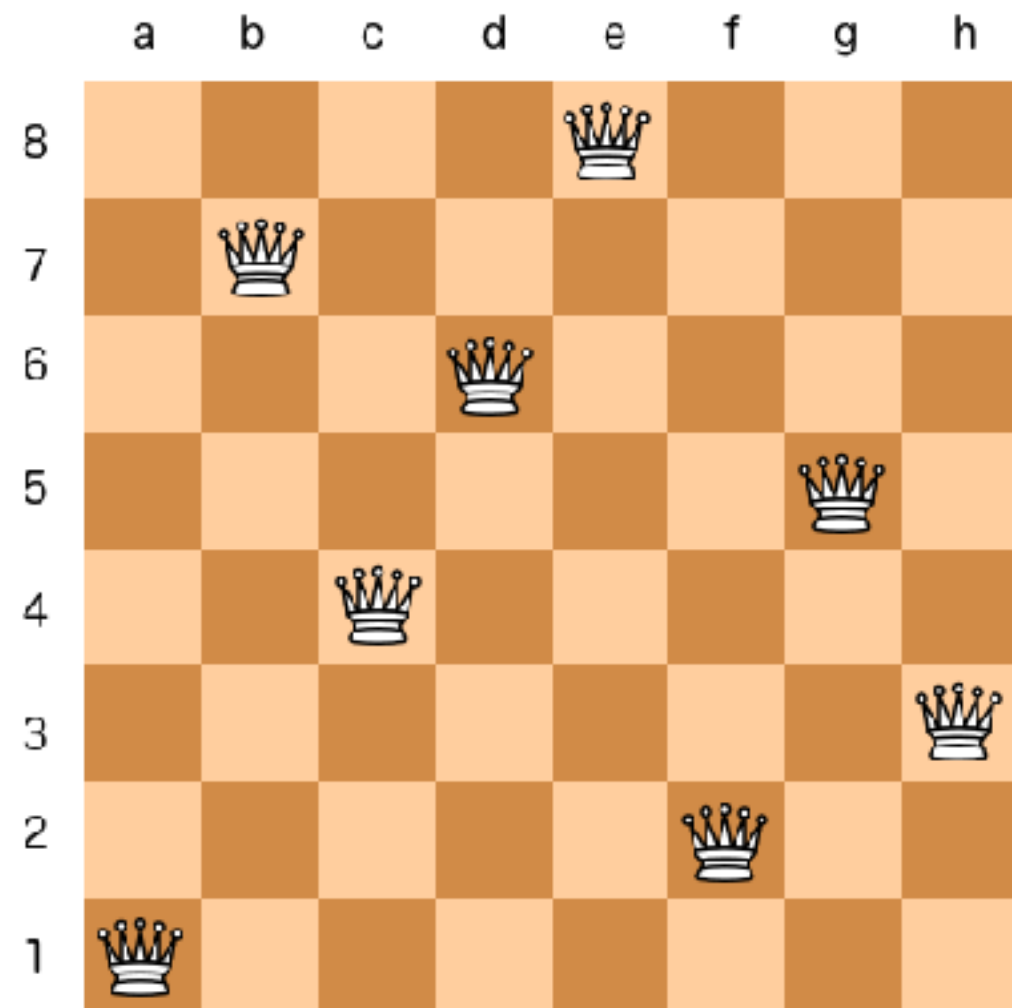
# Coloração de grafos

```
def complete(adj,k,c):  
    return len(adj) == len(c)  
  
def extensions(adj,k,c):  
    i = list(adj.keys())[len(c)]  
    cv = [c[x] for x in adj[i] if x in c]  
    return [(i,x) for x in range(k) if x not in cv]  
  
def valid(adj,k,c):  
    return True  
  
def search(adj,k):  
    c = {}  
    if aux(adj,k,c): return c  
  
def aux(adj,k,c):  
    if complete(adj,k,c): return valid(adj,k,c)  
    for i,x in extensions(adj,k,c):  
        c[i] = x  
        if aux(adj,k,c): return True  
        c.pop(i)  
    return False
```



# Rainhas

- Considere um tabuleiro de xadrez de dimensão  $n$
- Determinar se é possível posicionar  $n$  rainhas no tabuleiro sem que se ataquem mutuamente



# Rainhas

```
def complete(n,pos):  
    return len(pos) == n  
  
def extensions(n,pos):  
    return [(x,y) for x in range(n) for y in range(n)]  
  
def valid(n,pos):  
    for i in range(n):  
        for j in range(i+1,n):  
            (x0,y0) = pos[i]  
            (x1,y1) = pos[j]  
            if x0==x1 or y0==y1 or x0+y0==x1+y1 or y0-x0==y1-x1: return False  
    return True  
  
def search(n):  
    pos = []  
    if aux(n,pos): return pos  
  
def aux(n,pos):  
    if complete(n,pos): return valid(n,pos)  
    for p in extensions(n,pos):  
        pos.append(p)  
        if aux(n,pos): return True  
    pos.pop()  
    return False
```

# Rainhas

```
def complete(n,pos):  
    return len(pos) == n  
  
def extensions(n,pos):  
    return [(len(pos),y) for y in range(n)]  
  
def valid(n,pos):  
    for i in range(n):  
        for j in range(i+1,n):  
            (x0,y0) = pos[i]  
            (x1,y1) = pos[j]  
            if y0==y1 or x0+y0==x1+y1 or y0-x0==y1-x1: return False  
    return True  
  
def search(n):  
    pos = []  
    if aux(n,pos): return pos  
  
def aux(n,pos):  
    if complete(n,pos): return valid(n,pos)  
    for p in extensions(n,pos):  
        pos.append(p)  
        if aux(n,pos): return True  
    pos.pop()  
    return False
```

# Rainhas

```
def complete(n,pos):  
    return len(pos) == n  
  
def ok(n,pos,x,y):  
    for (x0,y0) in pos:  
        if y0==y or x0+y0==x+y or y0-x0==y-x: return False  
    return True  
  
def extensions(n,pos):  
    return [(len(pos),y) for y in range(n) if ok(n,pos,len(pos),y)]  
  
def valid(n,pos):  
    return True  
  
def search(n):  
    pos = []  
    if aux(n,pos): return pos  
  
def aux(n,pos):  
    if complete(n,pos): return valid(n,pos)  
    for p in extensions(n,pos):  
        pos.append(p)  
        if aux(n,pos): return True  
    pos.pop()  
    return False
```