3. agentes de procura

Os agentes reativos têm uma visão local do ambiente, controlador fixo, pouco memória.

Enquanto puder:
analisa estado
determina regras/comportamento aplicáveis
escolhe regra/comportamento por ordem
aplica regra/comportamento

Os agentes de procura têm mais memória, relações entre estados, controlador mais sofisticado com simulação/antevisão, restrições, operadores de mudança de estado.

Enquanto não resolvido e tiver alternativas:
analisa estado
determina regras aplicáveis
aplica de modo simulado todas as regras
escolhe controladamente um estado resultante

Alguns conceitos comuns são os de estado, operadores de mudança de estado e espaço de procura. O estado deve ser indicado de forma completa em cada situação.

algoritmo geral de procura

```
def general_search(problem, strategy):
    # 1. Initialize the search tree with the initial state of the problem
    search_tree = initialize_tree(problem.initial_state)

    while True:
        # 2.1 If there are no candidates left to expand
        if not has_candidates(search_tree):
            # 2.1.1 Return failure
            return "failure"

        # 2.2 Choose a state from the frontier to expand, according to the
strategy
        state = select_state(search_tree, strategy)

        # 2.3 If the state contains the goal
        if problem.is_goal(state):
            # 2.3.1 Return the corresponding solution
            return extract_solution(state)
        else:
            # 2.3.2 Expand the state and add its successors to the search tree,
according to the strategy
            successors = expand_state(state, problem)
            add_successors(search_tree, successors, strategy)
```

desempenho

estratégia

- **completa**: se existir uma solução, será encontrada *em tempo finito*

- **discriminadora**: caso existam várias soluções, escolhe a melhor
- **complexidade temporal**
- **complexidade espacial**

tipos de soluções

- sequência de ações
- a solução
- uma solução
- a melhor solução

agentes de procura cega

breadth-first search (largura primeiro)

- avançar horizontalmente pela árvore: todos os nós de um nível são visitados antes de descer para o próximo nível
- os nós a visitar são mantidos numa *FIFO queue*

```
# graph is a dict where the key is a char representing the node in question and
the value a list of chars representing the nodes connected to it
def BFS(graph, root, target):
    seen = []
    queue = []

    seen.append(root)
    queue.append(root)

    while queue:
        s = queue.pop(0)
        print(s, end=" ")

        if s == target:
            return s

        for n in graph[s]:
            if n not in seen:
                seen.append(n)
                queue.append(n)

    return None
```

(sendo *n* o nível de um nó, neste caso, do nível da solução, e *r* o fator de ramificação)

A complexidade temporal advém de, para uma solução no nível *n*, para todo o nível k de 1 até *n*, visitamos r^k nós, pelo que o total de nós visitado é $1 + r^2 + r^2 + \ldots + r^k$. O mesmo acontece para a complexidade espacial, pois para analisar os nós de um dado nível tem de os manter todos em memória.

depth-first search (profundidade primeiro)

- avançar verticalmente pela árvore
- os nós a visitar são mantidos numa *stack*

```
# graph is a dict where the key is a char representing the node in question and
the value a list of chars representing the nodes connected to it
def DFS(graph, root, target):
    seen = []
    stack = deque()

    visited.append(root)
    queue.append(root)

    while stack:
        s = stack.pop()
        print(s, end=" ")

        if s == target:
            return s

        for n in reverse(graph[s]):
            if n not in seen:
                seen.append(n)
                stack.append(n)

    return None
```

(sendo *n* o nível de um nó, neste caso, do nível da solução, e *r* o fator de ramificação)

A complexidade temporal advém de, para uma solução no ramo mais à esquerda do nível *n*, serão analisado *n+1* nós. Se estiver no mias à direita, será parecido com a procura em largura. Para a memória, para uma solução no nível *n*, temos em memória no máximo $n * (r - 1) + 1$.

Para a **complexidade espacial** (memória ocupada), os cálculos são mais simples, pelo menos para o cálculo do **máximo** de memória necessária quando a solução se encontra no nível **n**. Para nos auxiliar, reparemos na Fig. 3.24. Quando estamos para analisar Portalegre, temos guardado em memória os 3 sucessores de Coimbra, mais os 3 sucessores de C. Branco, mais Portalegre.

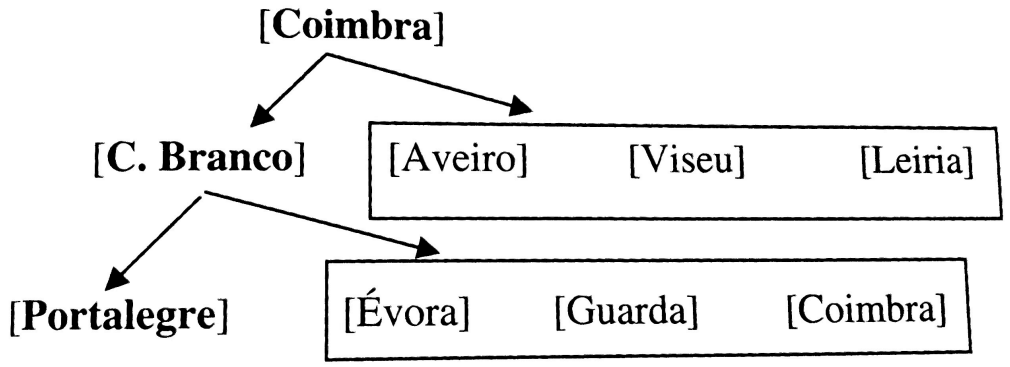


Fig. 3.24 – Árvore de procura em profundidade primeiro

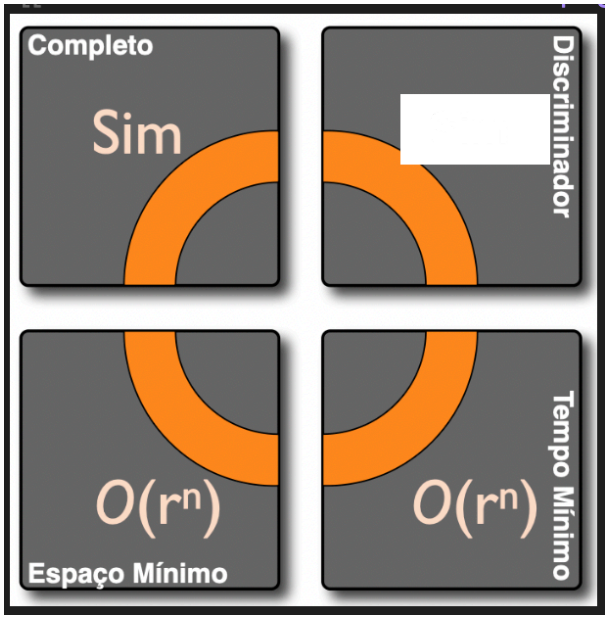
custo uniforme

- variação do BFS que escolhe para expansão o nó da fronteira com menor custo/distância
- ao contrários do BFS, dá o caminho ótimo! (no BFS escolhe-se o nó mais perto da raíz em termos de altura na árvore, mas isso nem sempre é o caminho mais curto)
- se a função de custo for apenas o nível da árvore, então fica-se com um simples BFS

```
# graph: dict where keys are nodes, values are lists of tuples (neighbor, cost)
def UCS(graph, root, target):
    queue = [(0, start)]
    seen = set()
```

```
    while queue:
        cost, node = heapq.heappop(queue)
        print(node, end=" ")

        if node == target:
            return node, cost

        if node not in seen:
            seen.add(node)
            for neighbor, edge_cost in graph.get(node, []):
                if neighbor not in seen:
                    heapq.heappush(queue, (cost + edge_cost, neighbor))

    return None, None
```

> **Note**
>
> O custo de um caminho não pode diminuir à medida que descemos de nível. Para tal basta não haver custos negativos. Desta forma, o algoritmo é **completo** e **discriminador**, pois não poderão existir ciclos de valor decrescente de custo - os custos que estão na pilha são o custo acumulado até chegar aos nós da pilha, naquele mesmo estado.



(sendo *n* o nível de um nó, neste caso, do nível da solução, e *r* o fator de ramificação)

profundidade limitada

- igual ao DFS mas com uma profundidade máxima para a qual o algoritmo pode ir
- evitar ciclos infinitos
- se se conhecer o nível máximo em que a solução pode estar, torna-se **COMPLETO** (mas continua a ser não discriminador)
- sendo l a profundidade máxima, temos de **complexidade temporal** $O(r^l)$ **e espacial** $O(r * l)$

afundamento progressivo

- para quando não sabemos o nível máximo mas queremos evitar ciclos infinitos
- iterar pelos vários níveis, executando algoritmo de profundidade limitada

```
def progressive_depth_search(graph, start, target):
    depth = 1
    while true:
        result = DFS_limited_depth(graph, start, target, depth)
        if result:
            break
        depth += 1
```

- **COMPLETO**
- **NÃO DISCRIMINADOR** (a não ser caso o custo seja sempre igual)
- **COMPLEXIDADE TEMPORAL**: se a solução estiver no nível k, a raíz será analisada k+1 vezes, o nível 1 será k vezes, o nível 2 será k-1 vezes, ... Para um nível de ramificação elevado, a sobrecarga da complexidade temporal tende a diminuir, não dependendo essa sobrecarga do nível! Perde-se $r/r-1$, sendo r a ramificação. Bom para pesquisa cega num grande espaço de procura e em que não se sabe o nível da solução.
- **COMPLEXIDADE ESPACIAL**: $O(r*n)$ - a mesma da DFS

procura heurística

- usam informação, conhecimento, sobre o problema para fazer uma procura mais eficiente e eficaz
- essa informação vai permitir estimar o custo caminho desde o nó atual até ao nó solução

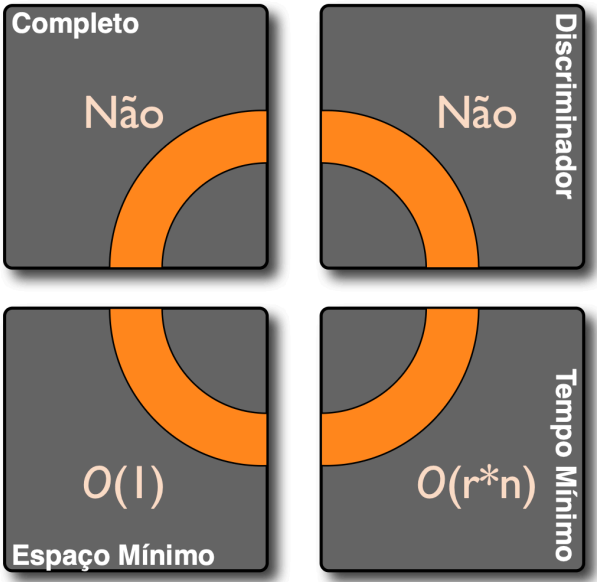
Considerar que a função de estimativa, *h(n)*, será a distância em linha reta entre duas cidades. Considerar *g(n)* como a função que retorna o custo real.

trepa colinas

- **VISÃO LOCAL**: avalia apenas os vizinhos do nó atual
- **MEMÓRIA LIMITADA**
- envolve ir melhorando, podendo partir de uma solução candidata, como o caso do problema da n rainhas
- é bom para encontrar máximos locais -> para globais, retomar o algoritmo noutra posição
- "planaltos" podem levar a escolhas aleatórias -> vizinhança maior pode resolver o problema

```
# graph: dict where keys are nodes, values are the neighbours
# h: function that will return the neighbour with the minimum estimate of the cost
of de distance from the current node to the target (it is the function h that
indirectly tells which vertex is the target)
def hill_climbing(graph, start, h):
    node = start

    while True:
        neighbours = graph[node]
        if not neighbours:
            return node
```

```
        next_node = min(neighbours, key=h)

        if h(next_node) > h(node):
            return node

        node = next_node
```



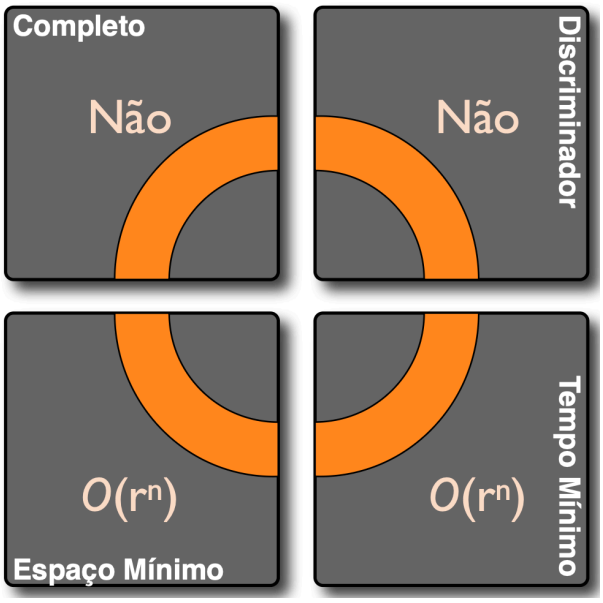
- **COMPLEXIDADE ESPACIAL**: só tem de guardar um nó
- **COMPLEXIDADE TEMPORAL**: similar à DFS

pesquisa sôfrega

- expandir o estado mais promissor de acordo com *h(n)*
- faz com que a fronteira esteja ordenada por *h(n)*
- **VISÃO GLOBAL**
- **MEMÓRIA NÃO LIMITADA**

```
# graph is a dict where the key is a char representing the node in question and
the value a list of chars representing the nodes connected to it
# h: function that will return the neighbour with the minimum estimate of the cost
of de distance from the current node to the target (it is the function h that
indirectly tells which vertex is the target)
def BFS(graph, root, h):
    seen = set()
    queue = []

    seen.append(root)
    queue.append(root)

    while queue:
        s = queue.pop(0)
        print(s, end=" ")

        if s == target:
            return s
```

```
        for n in graph[s]:
            if n not in seen:
            seen.append(n)
            queue.append(n)
            sorted(queue, key=h)

    return None
```



- **NÃO É COMPLETO**: pode ter ciclos infinitos
- Em termos de complexidade temporal, a expansão é um híbrido de BFS e DFS, pelo que no pior dos casos adota o pior: BFS
- Na complexidade espacial, toda a vizinhança tem de ser mantida em memória -> $O(r^n)$

A*

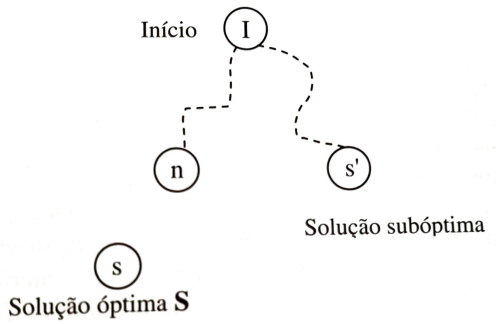
- usa como função de cálculo do custo $f(n) = g(n) + h(n)$, ou seja, $f(n)$ é o custo estimado de um caminho que passe por *n* (incluí a estimativa do que falta até ao alvo e o custo real de chegar até ao nó atual)
- junta a procura sôfrega com o custo uniforme
- visão **GLOBAL**
- **MEMÓRIA NÃO LIMITADA**
- o algoritmo é **COMPLETO** e **DISCRIMINADOR** se:
  - no caso de *h(n)* ser **admissível**: por admissível significa que **nunca sobrestima o custo real**: $h(n) \leq hreal(n)$
  - *h(n)* pode ser então a distância em linha reta
  - o fator de ramificação **r deve ser finito**
  - os arcos devem ter sempre **custo positivo** $(c(n, m) > 0)$
- a única diferença para a pesquisa sôfrega é o uso conjunto de *g(n)* e *h(n)*

- *f(n)* é monótona não decrescente

- caso, por algum motivo, o valor de f(n) para f(m) decresça (por exemplo, a distância real de n para m é 1, mas a distância prevista de m até ao fim é duas unidades menor que a de n), podemos fazer com que $f(m) = max(f(n), g(m) + h(m))$

```
# graph is a dict where the key is a char representing the node in question and
the value a list of chars representing the nodes connected to it
# f: g + h

def f(node):
    return g(node) + h(node)

def A_star(graph, root, f):
    seen = set()
    queue = []

    seen.append(root)
    queue.append(root)

    while queue:
        s = queue.pop(0)
        print(s, end=" ")

        if s == target:
            return s

        for n in graph[s]:
            if n not in seen:
            seen.append(n)
            queue.append(n)
            sorted(queue, key=f)

    return None
```

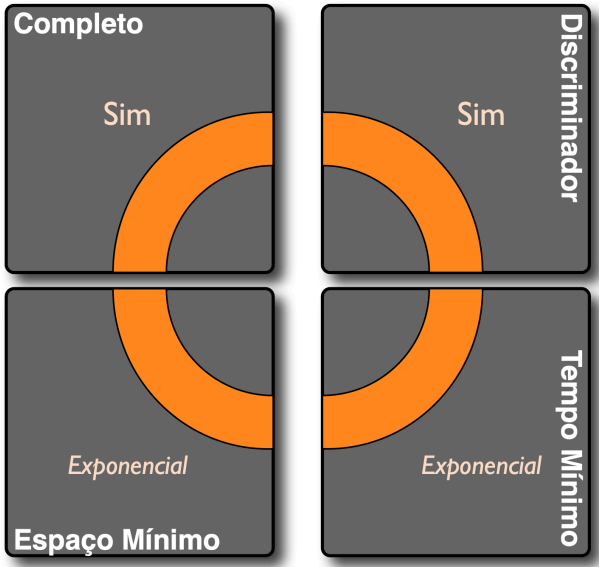
- para não ser completo, teria de haver uma infinidade de nós com valor de *f* inferior ao *f* da solução, o que só é possível se o fator de ramificação for infinito, ou se existisse um caminho com um número infinito de nós mas custo finito (impossível, pois os custos são sempre positivos)
- **PORQUE É QUE É DISCRIMINADOR - DÁ SOLUÇÃO ÓTIMA?**
  - PROVA POR *CONTRADIÇÃO*
    Assumindo que *s'* é escolhido em vez de *n*



Sabendo que h é admissível, temos que $fótimo \geq f(n)$. Por outro lado, como *n* não foi escolhido é porque $f(n) \geq f(s')$. Logo, $fótimo \geq f(s')$, e como *s'* é uma solução ($h(s') = 0$),

vem que $fótimo \geq g(s')$, o que implica que s' não é uma solução subótima, o que contradiz a hipótese inicial.

- **COMPLEXIDADE**: nós na fronteira cresce exponencialmente, e são todos gravados em memória -> tanto temporal como espacial são *exponenciais*



procura estocástica

- algoritmo cego
- escolhe de forma aleatória qual o próximo nó a ser expandido
- não se mantém em memória toda a árvore de procura, trabalhando-se apenas com a fronteira da árvore que contém os nós ainda não visitados

```
def random_search(graph, root, target):
    seen = []
    candidates = []

    seen.append(root)
    candidates.append(root)

    while candidates:
        current = random.choice(candidates)
        candidates.remove(current)
        print(current, end=" ")

        if current == target:
            return current

        for neighbor in graph[current]:
            if neighbor not in seen:
                seen.append(neighbor)
                candidates.append(neighbor)

    return None
```

- o algoritmo salta de estado em estado sem nenhum critério
- para espaços de procura grandes, a probabilidade de escolher um estado por onde já se passou é baixa e em situações sem informação prévia pode ser considerado

recristalização simulada

- similar ao trepa colinas mas evita cair em máximos locais
- em cada momento é selecionada aleatoriamente um sucessor do nó atual; se o seu valor heurístico for melhor que o atual, então é selecionado e o processo repete-se; no entanto, **mesmo que o nó seja pior que o atual, ainda poderá ser escolhido**
  - quanto pior for o nó novo em relação ao atual, menor a probabilidade de ser escolhido
  - quanto mais etapas tiverem passado, menos provável será ser escolhido o nó pior

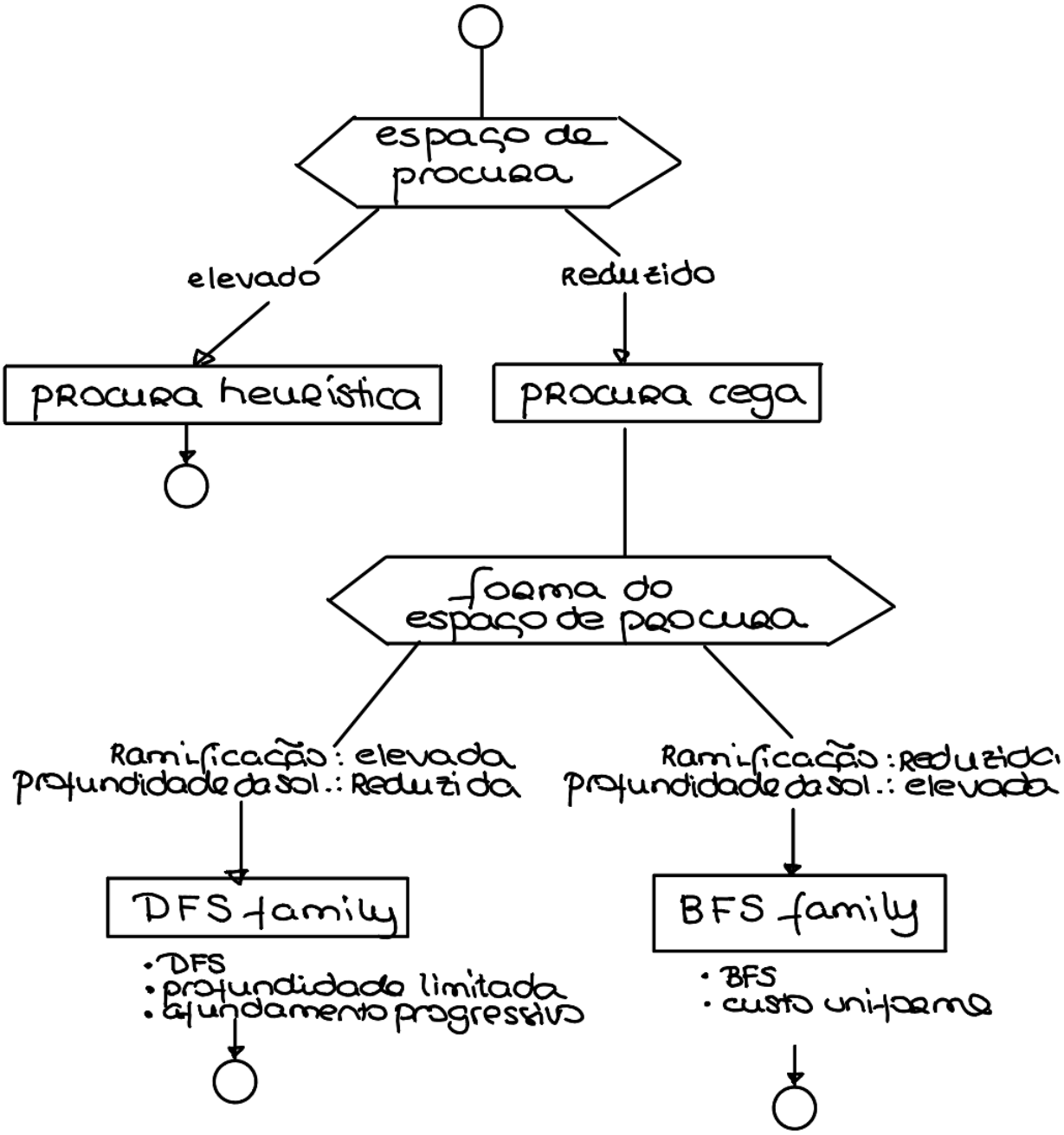
```
# temp_evol: returns the evolution of the temperature value for the given
iteration
def progressive_recrystallization(graph, root, h, temp_evol):
    current = root

    t = 1
    while True:
        T = temp_evol(t)

        if T <= 0:
            return node

        next_node = random.choice(graph[current])

        if h(next_node) > h(current):    # in this case, higher h is better (we
are finding the global max)
            current = next_node
        else:
            delta_E = h(next_node) - h(current)
            if random.random() < math.exp(delta_E / T):    # prob = e^(ΔE/T)
                current = next

        t += 1
```

```
    return None
```

- o parâmetro **T** é a **temperatura**: quanto menor o seu valor, menor a probabilidade de saltar para um nó com pior qualidade
    - um arrefecimento mais lento de T aumenta a probabilidade de encontrar o máximo local (mas aumenta o número de iterações do algoritmo)

- herda as propriedades do trepa colinas, pelo que não é completo nem discriminador
- só guarda um nó em memória - espaço mínimo constante

critério de escolha



exemplo: Charade de 15

- **Fator de ramificação médio**: 3
- **Dimensão do espaço de procura**: 16!
  - aproximadamente r^n, sendo *r* o fator de ramificação e *n* a profundidade máxima da árvore
    - **Profundidade máxima**: $log(16!)/log(3) = 28$

FATOR DE RAMIFICAÇÃO BAIXO E PROFUNDIDADE RAZOÁVEL -> BFS FAMILY

No caso de usar uma heurística, há que ser admissível. Poderia ser o valor do número de número na posição errada, o que seria admissível visto que para colocarmos na posição correta seria preciso pelo menos um movimento, pelo que o valor de movimento seria sempre igual ou superior ao da heurística. Também poderia ser considerar o número mínimo de movimentos para colocar um número na sua posição final, admitindo o caminho livre.

como escolher uma heurística?

- a que distinguir melhor os estados

- a que aproxime mais o valor estimado do real (ou seja, a heurística com valores maiores)
    - no caso anterior, seria a segunda pois o valor da segunda será no mínimo igual ao da primeira, pelo que **analisará em média menos nós**
- uma regra empírica poderá ser tentar definir uma heurística para uma versão menos restringida do problema