NÍVEIS	NÓS	TEMPO	ESPAÇO
12	**10^{12}**	**35 anos**	**111 terabytes**

BUSCA POR CUSTO UNIFORME:

Explora primeiro o nó de menor custo, obtendo assim a solução de menor custo. O custo de um nó é dado pela soma do caminho até seu pai mais o custo da aresta. Não se importa com o número de passos e sim com o custo total. Se todos os passos têm custo igual, busca com custo uniforme é igual a busca por amplitude. A busca é guiada pelo custo do caminho e não pela profundidade, assim a complexidade de tempo e espaço não usam *b* e *d.*

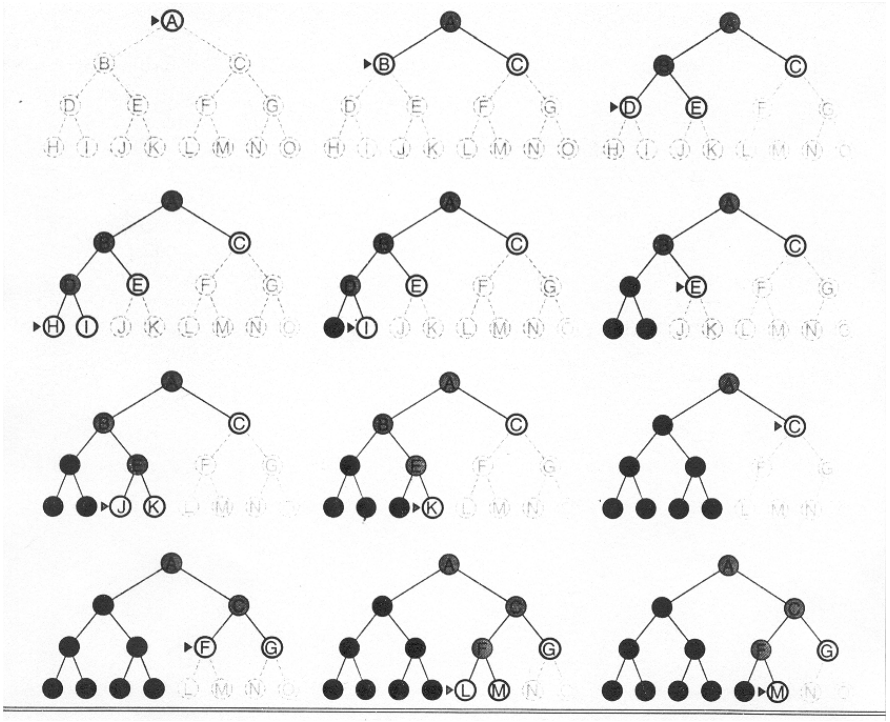
Propriedades:

- **Completo:** sim, se custo por passo positivo;
- **Ótimo:** sim, se

$\forall n\ g(suc(n)) \geq g(n)$, ou seja se o custo for crescente.

BUSCA POR PROFUNDIDADE:

Explora o nó mais a esquerda, depois o filho deste nó e assim recursivamente. Funciona como uma pilha: o último a ser descoberto será o primeiro a ser explorado. Não garante o melhor resultado. Implementado usando uma LIFO.



Propriedades:

- **Completo:** não, pode falhar se a árvore tem ramos infinitos ⇒ evitar estados repetidos.
- **Tempo:** $O(b^m)$: em problemas com várias soluções, ele pode ser mais rápido por que vai até o nodo folha antes de analisar os demais. O problema é se m >> d....
- **Espaço:** *O(bm).*: o gasto de memória é modesto por que armazena somente o caminho atual.
- **Ótimo:** não

Variação do Algoritmo: *BackTracking Search:* somente um nó é explorado por vez: *O(m)* ao invés de *O(bm)*.

BUSCA POR PROFUNDIDADE LIMITADA:

Busca por profundidade com limite de profundidade *l*. Na implementação, os nós de profundidade *l* são tratados como se não tivessem sucessores. Busca em profundidade por ser visto com $l = \infty$.

Propriedades:

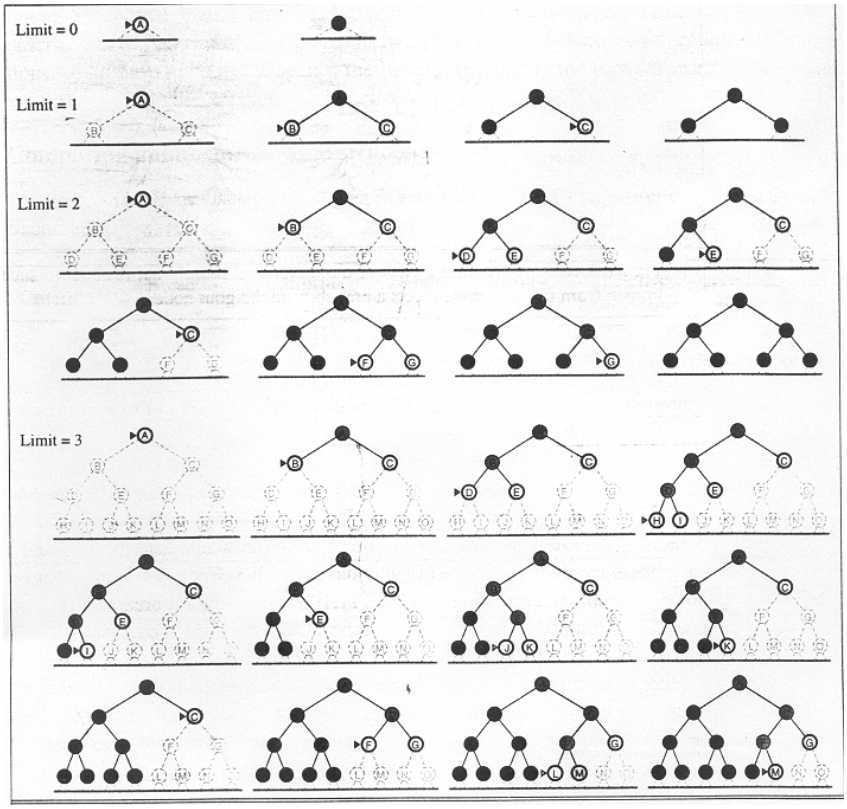
- **Completo:** não O grande problema está em definir um bom *l*.
- **Tempo:** $O(b^l)$
- **Espaço:** *O(bl)*, se definir l < d.
- **Ótimo:** *não,* mesmo com l > d.

O valor de *l* pode ser definido de acordo com algumas características do problema. Exemplo das cidades: se existem 20 cidades, escolhe-se *l* = 19 (diâmetro da solução).

BUSCA POR APROFUNDAMENTO ITERATIVO:

Técnica utilizada em conjunto com a busca em profundidade para encontrar o melhor limite de profundidade. Isso é feito incrementando gradualmente o limite – primeiro 0, depois 1, e então 2, e assim por diante, até o objetivo ser alcançado.

Combina os benefícios da busca em profundidade com a busca em largura. Assim como busca em profundidade, os requisitos de memória são modestos *O(bd)*, e assim como busca em largura, é completo se o fator de ramificação *b* for finito, e ótimo quando o custo do caminho é crescente em função da profundidade do nodo.



A busca por aprofundamento iterativo pode ser considerada dispendiosa visto que explora os mesmos nodos repetidamente. No entanto, os nodos do *d-ésimo* nível são explorados somente uma vez, os nodos do *(d-1)-ésimo* nível são explorados duas vezes, os do *(d-2)-ésimo* nível são explorados 3 vezes, e assim sucessivamente. Assim, o número total de nodos explorados é

$$N(IDS) = (d)b+(d-1)b^2+...+(1)b^d.$$

Propriedades:

- **Completo:** sim ⇒ como busca por amplitude
- **Tempo:** $(d+1)b^0+db^1+(d-1)b^2+...+b^d = O(b^d)$
- **Espaço:** O(bd) ⇒ como busca por largura
- **Ótimo:** sim, se custo fixo por expansão ⇒ como busca por amplitude

BUSCA BIDIRECIONAL

Começa uma árvore de busca do estado inicial para a solução e outra da solução para o estado inicial. Algumas coisas devem ser levadas em consideração

- Como pesquisar invertido? Os predecessores do nodo n são todos aqueles que tem n como sucessor. Na busca invertida nós começamos da solução gerando os predecessores sucessivamente;
- Quando os operadores são reversíveis os predecessores e os sucessores são um conjunto idêntico; podem existir problemas em que calcular os predecessores é muito difícil.
- Quando temos uma lista de soluções, deve-se realizar a busca a partir de cada uma delas? O que fazer se existir muitos estados? Como determinar este conjunto?
- Devemos testar eficientemente se um nodo já foi visitado pela outra parte da busca
- Devemos escolher que tipo de busca será feira em cada parte.

3.4. EVITANDO ESTADOS REPETIDOS

Um ponto crítico do processo de busca é a possibilidade de perder tempo expandindo estados que já foram alcançados e expandidos.

Para alguns problemas, essa possibilidade nunca ocorre, visto que o espaço de estados é uma árvore e existe somente um caminho para cada estado. Para outros problemas, estados repetidos são inevitáveis. Isso inclui todos os problemas para os quais as ações são reversíveis, tais como encontrar rota entre duas cidades e jogos com deslizamento de blocos (8-peças).

Pode-se tentar evitar estados repetidos, não retornando a um estado que deu origem ao estado corrente, ou seja, não gerar um sucessor direto que seja o mesmo estado que o nodo pai do estado corrente $\Rightarrow$ essa estratégia só evita loops de três arestas, mas dependendo do problema é suficiente.

Para busca em profundidade, somente os nodos do caminho da raiz até o nodo corrente são mantidos na memória. Isso garante que um espaço de estados finitos não gera uma árvore de busca infinita, devido a loops causados por estados já visitados. No entanto, isso não garante que não haverá repetição de estados em ramos diferentes.

Para evitar estados repetidos, é fundamental armazenar mais nodos na memória $\Rightarrow$ deve-se manter uma relação aceitável entre espaço e tempo. Algoritmos que "esquecem" o seu histórico, estão condenados a repetir estados. Para verificar se um dado estado já foi visitado anteriormente, é necessário tempo e espaço $O(bd)$ pois devemos manter informações sobre os estados que já foram explorados $\Rightarrow$ solução normalmente adotada quando é possível manter o grafo completo na memória, não somente a fronteira.