UNIVERSIDADE DE SÃO PAULO – USP
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO
DEPARTAMENTO DE SISTEMAS DE COMPUTAÇÃO

TORRES DE HANÓI

Alunos:
Adams Vietro Codignotto da Silva - 6791943
Ana Clara Kandratavicius Ferreira - 7276877
Frederico Facco - 8532206

São Carlos
2014

Introdução

Neste trabalho, iremos resolver o famoso problema das Torres de Hanói, utilizando uma representação em grafo e técnicas aprendidas em IA.

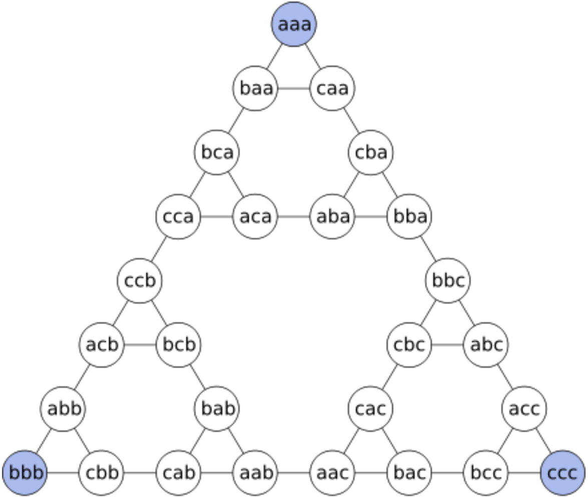
Modelando o problema

A meta é obter o número mínimo de movimentos. O número mínimo pode ser obtido utilizando recorrência.

Tomando n como número de discos, a, b e c como os pinos e $H(n, a, b, c)$ como a quantidade de movimentos para passar do pino a para o pino b, utilizando o auxiliar c, temos:

$$H(1, a, b, c) = 1 \quad (a \rightarrow c)$$
$$H(n, a, b, c) = H(n-1, a, b, c), \quad (a \rightarrow b), H(n-1, c, b, a)$$

Podemos dizer então que para um número n de discos, podemos gerar uma árvore contendo todas as possibilidades, com 3^n nós.



Árvore de possibilidades com 3 discos

Podemos dividir o problema da Torre de Hanói de acordo com o número de discos:

- Para n=1, basta 1 movimento
- Para n=2, precisamos de 3 movimentos (trivial)
- Para n=3, podemos resolver o problema para 2 discos (3 movimentos), mover o disco maior para o pino restante (1 movimento), e movemos os outros 2 discos para o pino final (3 movimentos) totalizando 7 movimentos.
- Para n=4, Resolvemos o problema para três discos(7 movimentos), depois movemos o maior disco (1 movimento), após isso trazemos os três discos que já estão no outro pino para cima do maior disco (7 movimentos), totalizamos 15 movimentos.

Podemos perceber que temos a seguinte seqüencia:

- $1 = 2^1 - 1$
- $3 = 2^2 - 1$
- $7 = 2^3 - 1$
- $15 = 2^4 - 1$

Ou seja, temos $2^n - 1$ movimentos necessários para uma quantidade n de pinos.

Implementação

Utilizando a árvore de possibilidades, podemos utilizar a seguinte estrutura:

```
typedef struct grafo{
    int key[20];
    int flag;
    Lista *L;
}Grafo;
```

Estrutura do Grafo

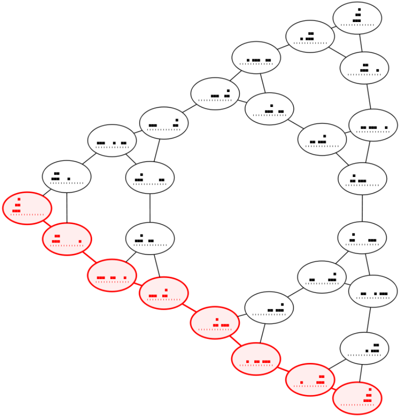
```
typedef struct bloco {
        int valor;
        int flag;
        struct bloco *prox;
} no;
typedef struct {
        no *inicio;
        no *fim;
} Lista;
```

Estrutura da lista de Adjacência

Algoritmos de busca

Utilizamos inicialmente as buscas *DFS (Depth-first Search, Busca em Profundidade)* e *BFS (Breadth-first search, Busca em Largura)* como buscas cega. Podemos percorrer o grafo pelo lado esquerdo, chegando sempre à solução ótima.

Porém, como se trata de uma busca cega, a DFS irá chegar ao resultado ótimo, enquanto a BFS irá percorrer muito mais nós.



Heurísticas

Para o Algoritmo A*, usamos as heurísticas:

- Um contador de movimentos, onde a cada estado que passamos é incrementado.
- A distância do estado até o estado desejado, onde o peso atribuído ao nó é a quantidade de discos que ainda não estão na posição final (no terceiro pino)

Para o algoritmo A*, dado um vértice inicial v, e seja d_{min} a distância total mínima, mantenha uma fila de prioridades (ou uma pilha) de nós a serem visitados, ordenado em ordem decrescente de custo:

1. Retire o primeiro elemento da fila e some o peso do elemento ao peso de seus vizinhos.
2. Reinsira os vizinhos na fila.
3. Se a distancia até o estado final for maior que a distância atual, a distância nova é a distância atual para o vértice v.
4. Continue até encontrar o alvo ou até que não tenha mais elementos na fila.

Todos os nós devem manter um histórico de seu predecessor para que futuramente possa ser resgatado o menor caminho possível até o alvo.

Análise dos Algoritmos

Número de Discos	BFS	DFS	A*	Ideal
1	1	1	1	1
2	6	8	3	3
3	24	26	8	7
4	70	74	15	15
5	232	220	25	24

Note que: A* necessita de mais memória que DFS e BFS juntos.