Escola de Engenharia
Universidade do Minho

Complementos de Programação de Computadores – Aula 11
Estuturas de Dados: Árvores

Mestrado Integrado em Electrónica Industrial e Computadores

Luís Paulo Reis

lpreis@dsi.uminho.pt

Professor Associado do Departamento de Sistemas de Informação, Escola de Engenharia, Universidade do Minho, Portugal

(Slides Baseados em Cortez 2011 , Reis, Rocha e Faria, 2007)

Escola de Engenharia
Universidade do Minho

Recursividade

- **Recursividade em C++: Função/método chamar-se a si própria/o de modo recursivo**
- **Para evitar ciclos infinitos, tem de haver uma condição de paragem.**

Exemplo: factorial

- `#include <iostream>`

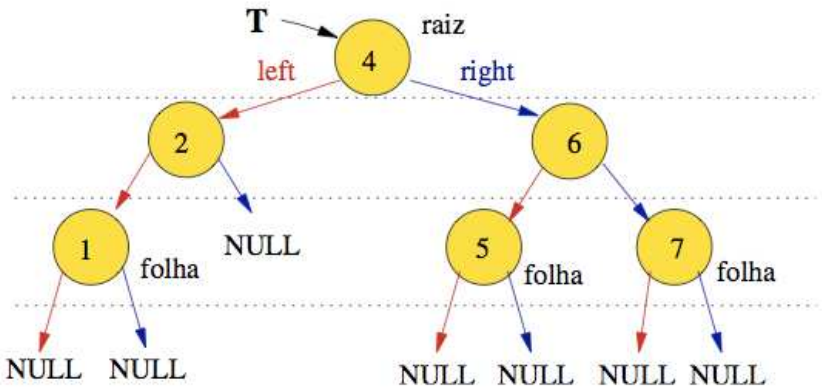
```
using namespace std;
int factorial(int n)
{
   if(n==0) return 1;            // condição de paragem
   else return n*factorial(n-1); // chamada recursiva
}
int main()
{
   cout << factorial(3) << "\n";
}
```

$$5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$$

factorial(3)=3*2*1*1
factorial(2)=2*1*1
factorial(1)=1*1
factorial(0)=1

Árvores Binárias

- Dados estruturados numa árvore
- Cada nodo tem somente 2 ramos: esquerda e direita
- Se ordenada, a pesquisa é mais eficiente: O(log2n) (se árvore balanceada!)
- Aplicações: bases de dados, expressões de texto, jogos, …



Árvores Binárias

Manipulação (operações/métodos) sobre árvores:

- remove - remove elemento da árvore – O(log2n), se balanceada
- find – procura o nodo do elemento na árvore – O(log2n), se bal.
- insert_sort – insere elemento de modo ordenado – O(log2n), se bal.
- empty - determina se a árvore está vazia – O(1)
- size - devolve o número de elementos – O(n)
- init (construtor) – cria uma árvore vazia – O(1)
- destroy (destrutor) – elimina toda a árvore – O(n)

Implementação proposta:

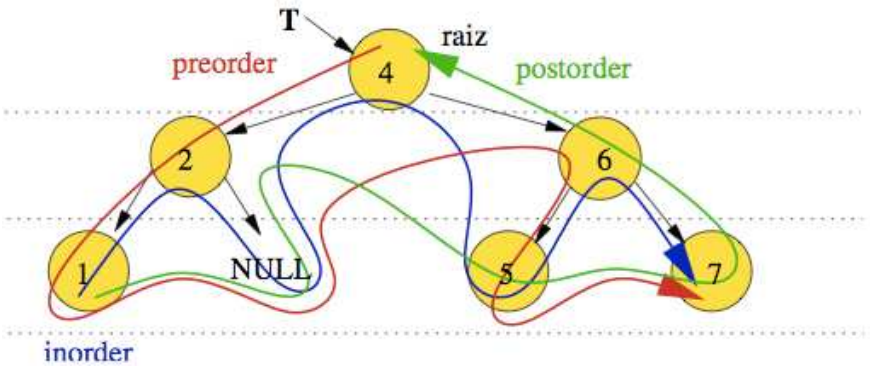
- Existem 2 classes: TNode e BTree;
- Node: contém o atributo d_left, d_right e d_data (será do tipo int, mas pode ser uma qualquer classe…
- BTree: contém o atributo root, do tipo TNode* e que aponta para a raiz da árvore…

Árvores Binárias

```
#include <iostream> // exemplo num único ficheiro tree.cpp
class TNode // nodo de uma arvore binaria
{
private:
   int d_data; // pode ser string ou ate uma classe
   TNode *d_left; TNode *d_right;
public:
   TNode(int data, TNode *left, TNode *right) {
         d_data=data; d_left=left; d_right=right;
   }
   void setData(int data) { d_data=data;}
   void setLeft(TNode *left) { d_left=left;}
   void setRight(TNode *right) { d_right=right;}
   int data() const { return d_data;}
   TNode *left() const { return d_left;}
   TNode *right() const { return d_right;}
   void print() const { cout << d_data << " ";}
};
```


Árvores Binárias

- **Existem 3 tipos de travessias: inorder, preorder e postorder:**
  - Inorder: esquerda, nodo, direita (1,2,4,5,6,7)
  - Preorder: nodo, esquerda, direita (4,2,1,6,5,7)
  - Postorder: esquerda, direita, nodo (1,2,5,7,6,4)



Árvores Binárias

- **Muitos dos métodos (quase todos...) para manipular árvores binárias, incluindo as travessias, funcionam via uma recursividade. Exemplo: motores das travessias**

```
Tipo inorder(TNode *N) {
    if(N!=0) { inorder(N->left());
    // usar nodo
    inorder(N->right());}
}
Tipo preorder(TNode *N) {
    if(N!=0) {
        // usar nodo
        preorder(N->left());
        preorder(N->right());
    }
}
Tipo postorder(TNode *N) {
    if(N!=0) {
        postorder(N->left());
        postorder(N->right());
        // usar nodo
    }
}
```


Árvores Binárias

```
class BTree   // arvore de int, continuação do tree.cpp
{
    private: TNode *root; // nodo raiz
  public:
    BTree(){ root=0;} // lista ligada vazia
    ~BTree() // destrutor
        { destroy(root);}
    void destroy(TNode *N) { // O(N)
       if(N!=0) {
         destroy(N->left()); destroy(N->right()); delete N;}
    }
    bool empty() const { return (root==0);} // O(1)
    int size() const { return size(root);} // O(N), trav. preorder
    int size(TNode *N) const { // travesia preorder
        if(N!=0) { return 1+size(N->left())+size(N->right());}
        else return 0;
    }
```

Árvores Binárias

```
// continuacao do tree.cpp
void print() const {
    print(root); cout << "\n";} // O(N), travesia preorder
void print(TNode *N) const { // travesia preorder
    {
        if(N!=0) { N->print(); cout << "[";
        print(N->left()); cout << "][";
        print(N->right()); cout << "]";}
    }
TNode *find(int elem) const {
        find(root,elem);
    } // O(log2n), se bal.
TNode *find(TNode *N,int elem) const { // preorder
    if(N==0 || N->data()==elem) return N;
    else if(N->data()>elem) return find(N->left(),elem);
    else return find(N->right(),elem);
}
```


Árvores Binárias

```
// continuação do tree.cpp
void print() const { print(root); cout << "\n";}
               // O(N), travessia preorder
void print(TNode *N) const { // travesia preorder
   if(N!=0) { N->print(); cout << "[";
   print(N->left()); cout << "][";
   print(N->right()); cout << "]";}
}
TNode *find(int elem) const {  // O(log2n), se bal.
   find(root,elem);
}
TNode *find(TNode *N,int elem) const {  // preorder
   if (N==0 || N->data()==elem) return N;
   else if(N->data()>elem) return find(N->left(),elem);
   else return find(N->right(),elem);
}
```

Árvores Binárias

- **Imagine que o resultado de: T.print() é:**

  5 [3 [1 [][2 [][]]][4 [][]]][7 [6 [][]][8 [][9 [][]]]]

Desenhe a árvore binária correspondente

Árvores Binárias

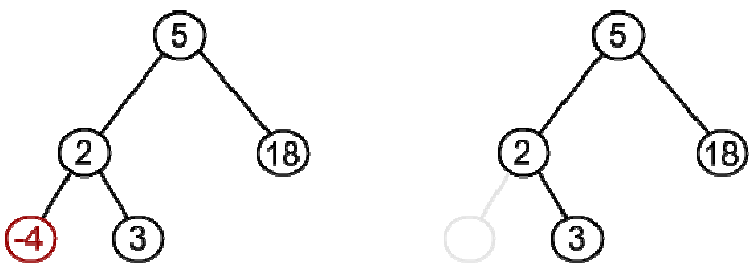
```
// continuacao do tree.cpp
TNode *insert_sort(int elem) {
    root=insert_sort(root,elem);} // O(log2n), se bal.
TNode *insert_sort(TNode *N,int elem) {
    TNode *res=0;
    if(N==0){res=new TNode(elem,0,0);}
    else if(N->data()>elem) {
        N->setLeft(insert_sort(N->left(),elem));res=N;}
    else { N->setRight(insert_sort(N->right(),elem));res=N;}
    return res;
}
int minData(TNode *N) const { // valor minimo a partir de um TNode
    if(N->left()==0) return N->data(); // min=a folha mais à
                                       esquerda de N
    else return minData(N->left());}
}; // fim da class BTree
```

Árvores Binárias

```
int main()
{
    BTree T;
    T.insert_sort(4);
    T.insert_sort(2);
    T.insert_sort(3);
    T.insert_sort(7);
    T.insert_sort(6);
    T.insert_sort(9);
    T.print();
    int s = T.size();
    cout << "size:" << s << "\n";
    TNode *N = T.find(7);
    cout << "data:" << N->data() << "\n";
    cout << "min. de 7:" << T.minData(N) << "\n";
    return 0;
}
```

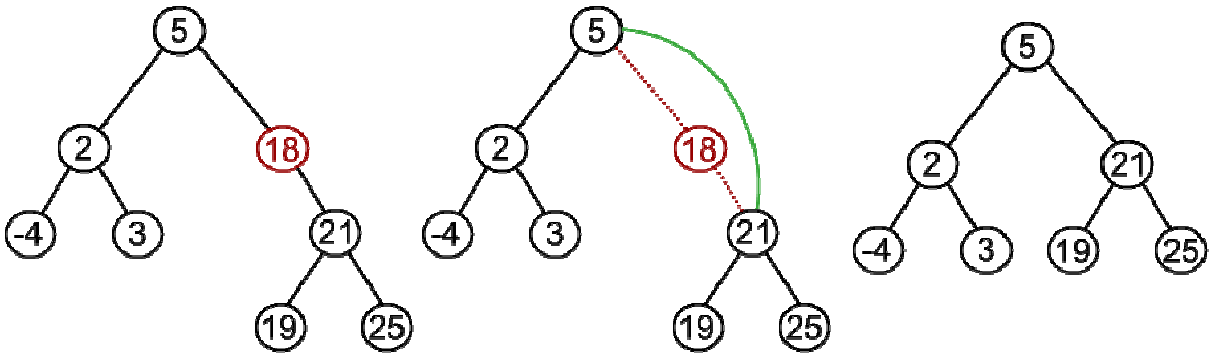

Árvores Binárias

- **Em termos do algoritmo, a remoção em árvores é bem mais elaborada do que a inserção (insert_sort) e procura de elementos (find);**
- **Ver todos detalhes em:**
  - http://www.algolist.net/Data_structures/Binary_search_tree/Removal
- **Existem 3 cenários:**
  - 1 – o nodo a remover não tem filhos; (simples)



Árvores Binárias

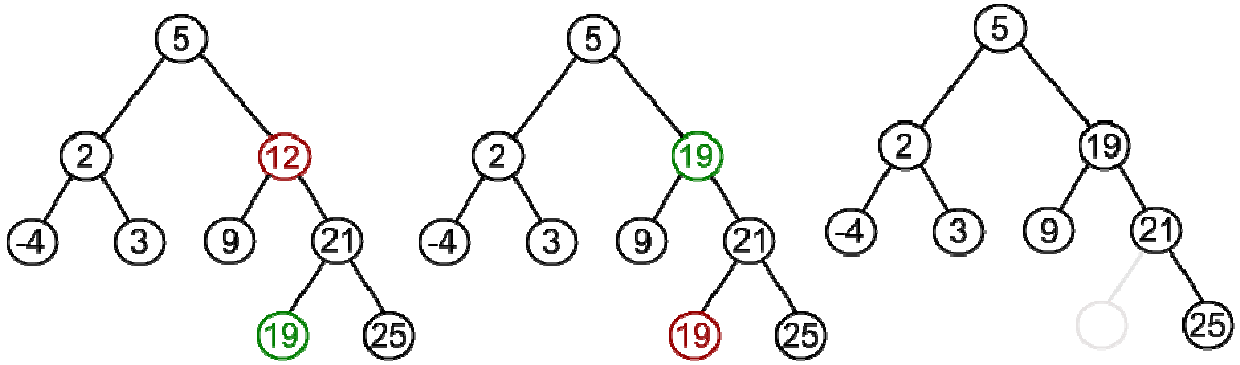
- **Existem 3 cenários:**
  - **2) nodo a remover tem 1 filho**



Árvores Binárias

Cenário 3 – O nodo a remover tem 2 filhos;

- **Caso mais complexo, tendo que fazer-se o seguinte:**
  - Encontrar o valor mínimo na subárvore à direita
  - Substituir o valor do nodo a remover com o mínimo
  - Aplicar a remoção da subárvore à direita para remover o duplicado (cenário 1 ou 2)



Árvores Binárias

```
// acrescentar dentro da class BTree:
void remove(int elem) // O(log2n) se balanceada
    { if(root==0)
     cerr << "Error: " << elem << "not found."; // elem nao existe
     else { if(root->data() == elem) // remover a root!
     { TNode* aux=new TNode(0,root,0);
     TNode* removedNode = remove(root,aux,elem); // metodo auxiliar
     root = aux->left();
     if(removedNode!=0) delete removedNode;
   }
   else // remover outro nodo
   { TNode* removedNode = remove(root,0,elem); // metodo auxiliar
     if(removedNode != NULL) delete removedNode;
   }
   }
}
```


Árvores Binárias

```
// acrescentar dentro da class BTree:
TNode *remove(TNode *N, TNode *parent, int elem) // metodo auxiliar
     { if(elem < N->data()) {
        if (N->left()!= 0) return remove(N->left(),N,elem);
        else return 0;}
       else if(elem > N->data()){
           if (N->right() !=0) return remove(N->right(),N,elem);
           else return 0;}
        else{ if(N->left()!=0 && N->right()!=0) // cenario 3
        { N->setData(minData(N->right()));
        return remove(N->right(),N,N->data());
       } else if(parent->left()==N)
       { if(N->left()!=0) parent->setLeft(N->left()); // cenario 2
       else parent->setLeft(N->right()); // cenario 1
       return N;
       } else if(parent->right()==N)
       { if(N->left()!=0) parent->setRight(N->left()); // cenario 2
      else parent->setRight(N->right()); // cenario 1
      return N;
     }}
}
```

Árvores Binárias

```
// continuação…
int main()
{
    BTree T;
    T.insert_sort(4);
    T.insert_sort(2);
    T.insert_sort(3);
    T.insert_sort(7);
    T.insert_sort(6);
    T.insert_sort(9);
    T.remove(2);
    T.remove(3);
    T.remove(4);
    T.print();
    return 0;
}
```


Algoritmos de Pesquisa de Soluções

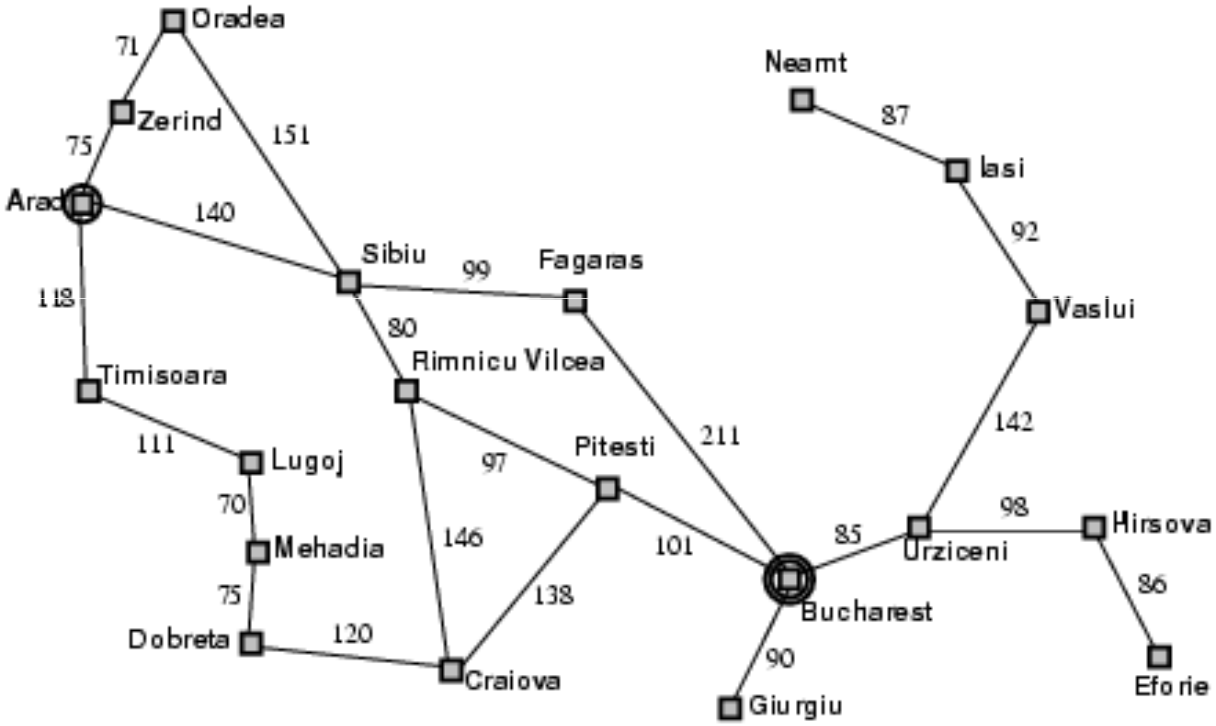
- **Terminologia:**
  - **Fronteira**: coleção de nós que foram gerados mas não expandidos (nós abertos).
  - **Nó Folha:** qualquer elemento da fronteira (sem sucessores na árvore)
  - **Estratégia de Pesquisa :** função que seleciona o próximo nó a ser expandido da fronteira
- **Algoritmos de Pesquisa Cega (Sem Informação):**
  - Pesquisa em Largura
  - Pesquisa por Custo Uniforme
  - Pesquisa em Profundidade
  - Pesquisa em Profundidade Limitada
  - Pesquisa em Profundidade Interativa
  - Pesquisa Bidirecional
- **Algoritmos de Pesquisa Inteligente (Informada):**
  - Pesquisa Gulosa
  - Algoritmo A*

Algoritmos de Pesquisa de Soluções

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  static: seq, an action sequence, initially empty
          state, some description of the current world state
          goal, a goal, initially null
          problem, a problem formulation
  state ← UPDATE-STATE(state, percept)
  if seq is empty then do
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

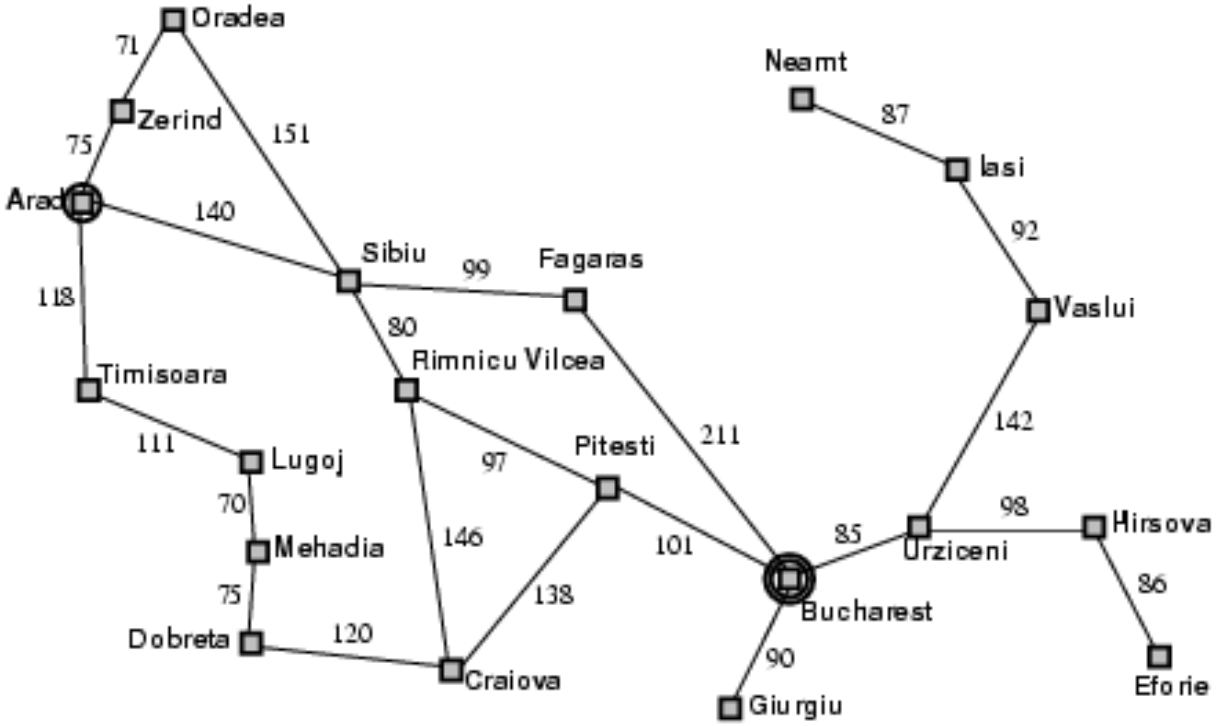

Algoritmos de Pesquisa de Soluções

- **Exemplo: Mapa da Roménia (Russel e Norvig, 1995)**

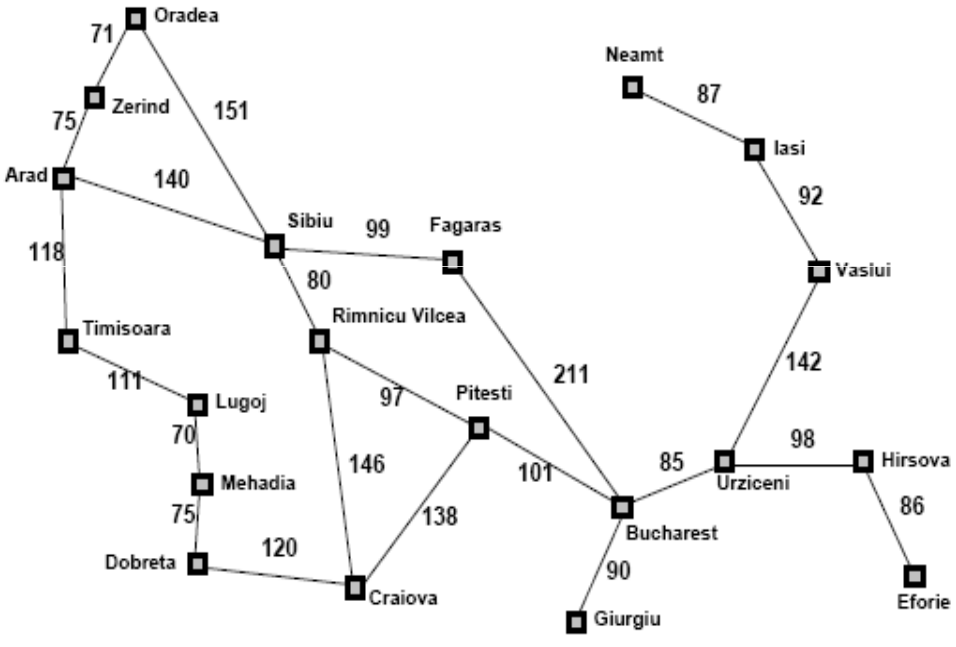


Algoritmos de Pesquisa de Soluções

- **Exemplo: Mapa da Roménia (Russel e Norvig, 1995)**



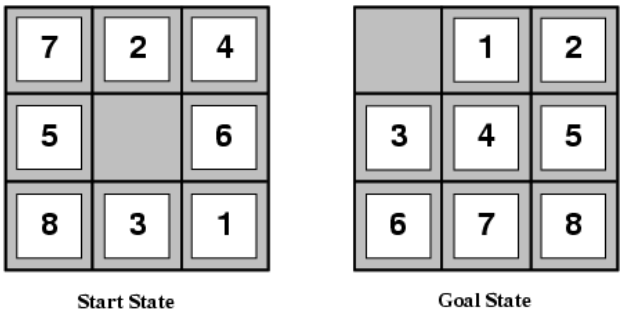
Algoritmos de Pesquisa de Soluções



Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Algoritmos de Pesquisa de Soluções

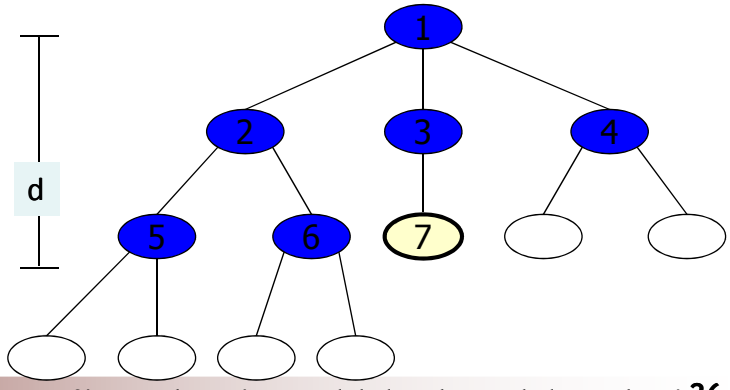
- **Exemplo: Npuzzle**
  - Estados?
  - Acções?
  - Teste Objectivo?
  - Custo Acções?



Algoritmos de Pesquisa de Soluções

- **Pesquisa em Largura:**
  - Completude: Sim. Encontra a solução óptima, se o fator de ramificação *b* é finito
  - Optimalidade: Sim, se o custo do caminho for uma função não decrescente da profundidade do nó (ou seja, se todos os caminhos tiverem o mesmo custo)
  - Complexidade no Tempo: $1 + b + b^2 + b^3 + ... + b^d + (b^{d+1}-b) = O(b^{d+1})$
  - Complexidade no Espaço: $O(b^{d+1})$ – guarda todos os nós na memória
  - Grande quantidade de espaço e tempo exigida. Pode facilmente gerar muitos MB de nós que devem ser guardados.

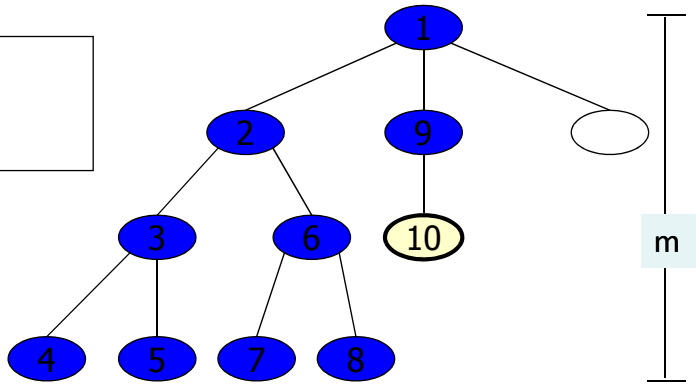
b = número máximo de filhos
(ou fator de ramificação)
d = altura do nó solução



Algoritmos de Pesquisa de Soluções

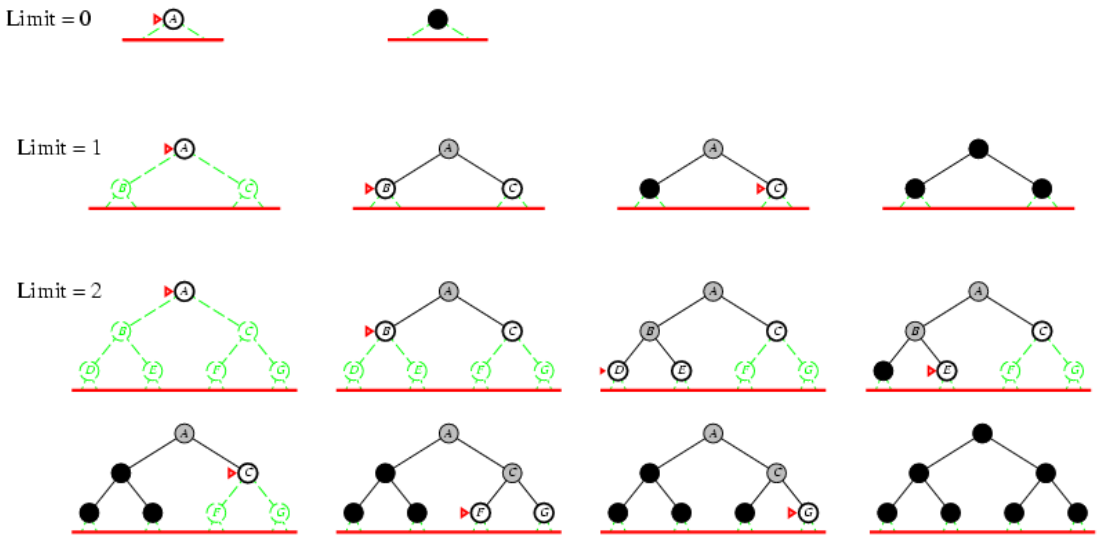
- **Pesquisa em Profundidade (Limitada):**
  - Completude: Sim, somente se o espaço de estados não tiver ciclos.
  - Complexidade de Memória: Armazena só um caminho simples da raiz até à folha
    - Para um fator de ramificação *b* e uma profundidade máxima de *m* armazena *bm* nós (busca em largura = b^d)
  - Complexidade no Tempo: $O(b^m)$ no pior caso -> examinar todos os ramos
    - Muito mau se *m* é muito maior que *d* (m - profundidade máxima de qq nó)
  - Otimalidade: Não. Necessita de um espaço de busca finito e não cíclico

b = número máximo de filhos
(ou fator de ramificação)
m = altura máxima da árvore



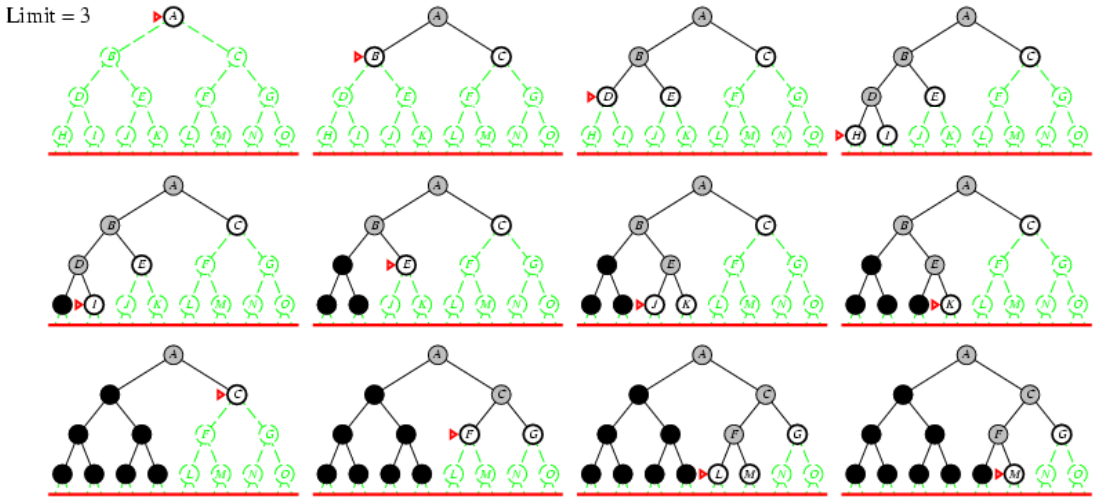
Algoritmos de Pesquisa de Soluções

- **Pesquisa em Profundidade Iterativa:**
  - Tenta todos os possíveis limites de profundidade
  - Combina os benefícios da pesquisa em largura e em profundidade



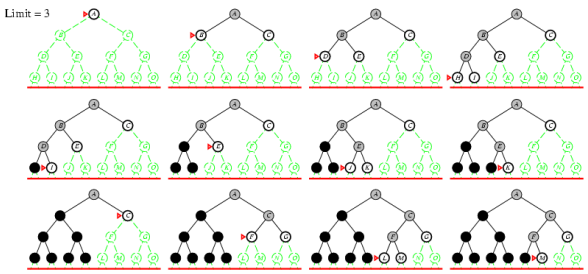
Algoritmos de Pesquisa de Soluções

- **Pesquisa em Profundidade Iterativa:**
  - Tenta todos os possíveis limites de profundidade
  - Combina os benefícios da pesquisa em largura e em profundidade



Algoritmos de Pesquisa de Soluções

- **Pesquisa em Profundidade Iterativa:**
  - Completude: Sim
  - Otimalidade: Sim (se o custo do caminho for uma função não decrescente da profundidade do nó,ou seja, quando todos os caminhos tiverem o mesmo custo)
  - Tempo: $O(b^d)$ – alguns nós podem ser gerados várias vezes. Mas isso acontecerá nos níveis superiores que normalmente têm poucos nós
  - Espaço: O(bd)
  - Preferida quando o espaço de pesquisa é muito grande e a profundidade da solução não é conhecida
  - É o método de pesquisa sem informação preferido quando existe um espaço de pesquisa grande e a profundidade da solução não é conhecida



b = número máximo de filhos
(ou fator de ramificação)
d = altura do nó ótimo

Algoritmos de Pesquisa de Soluções

- **Comparação das Estratégias de Pesquisa**

	Largura	Custo Uniforme	Profun-didade	Profun-didade limitada	Profun-didade Interativa	Bidirecional (se aplicável)
Tempo	$O(b^{d+1})$	$O(b^{d+1})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Espaço	$>>b^d$	$>>b^d$	O(bm)	$O(b\ell)$	O(bd)	$O(b^{d/2})$
Optima?	Sim[3]	Sim	Não	Não	sim[3]	Sim[3,4]
Completa?	sim[1]	sim[1,2]	Não	Sim	Sim[1]	Sim[1,2]

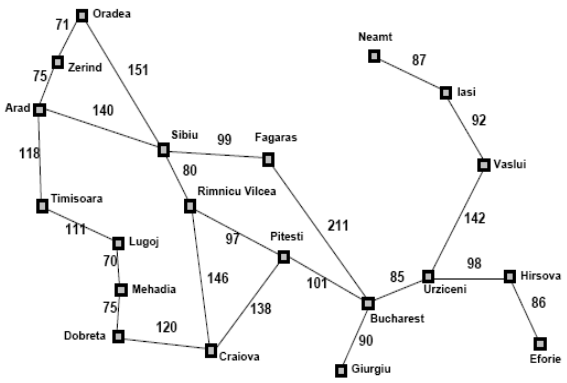
1 – completa se b é finito
2 – completa se o custo do passo é >= c, para c positivo
3 – ótima se o custo dos passos são todos idênticos
4 – se ambos os sentidos utilizam pesquisa em largura

Algoritmos de Pesquisa de Soluções

- **Pesquisa Gulosa:**
  - Minimizar o custo estimado para alcançar o objetivo
  - Muitas vezes o custo para se alcançar o objetivo pode ser estimado mas não pode ser determinado exactamente
  - A pesquisa gulosa (*greedy*) expande o nó que aparenta estar mais próximo do objetivo
  - Função de Avaliação utiliza somente *h(n)*

Para o exemplo da Roménia:

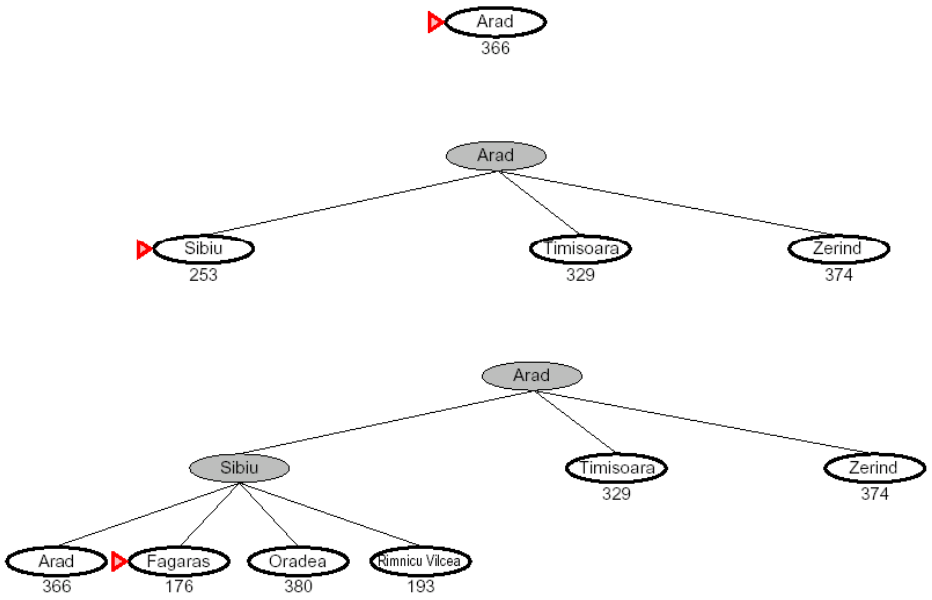
$h_{DLR}(n)$ = distância em linha reta de *n* até Bucarest



Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Algoritmos de Pesquisa de Soluções

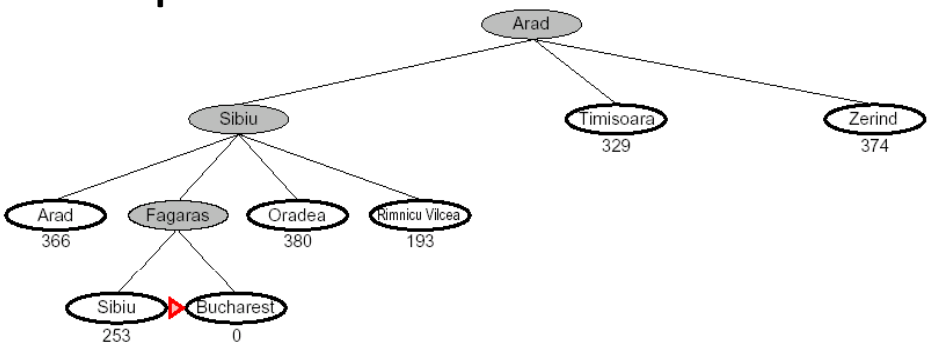
- **Pesquisa Gulosa:**



Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Algoritmos de Pesquisa de Soluções

- **Pesquisa Gulosa:**



Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

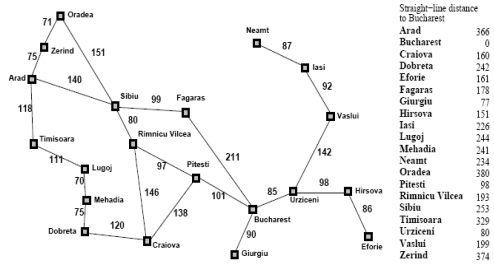
- O solução encontrada não é a solução óptima!
  - Arad ➔ Sibiu ➔ Fagaras ➔ Bucharest
    (140) + (99) + (211) = 450km
- A Solução ó+tima passa por Rimnicu Vilcea
  - Arad ➔ Sibiu ➔ Rimnicu Vilcea ➔ Pitesti➔ Bucharest
    (140) + (80) + (97) + (101) = 418km

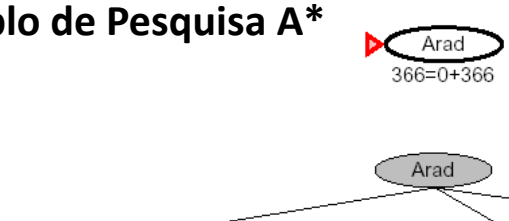
Algoritmo A*

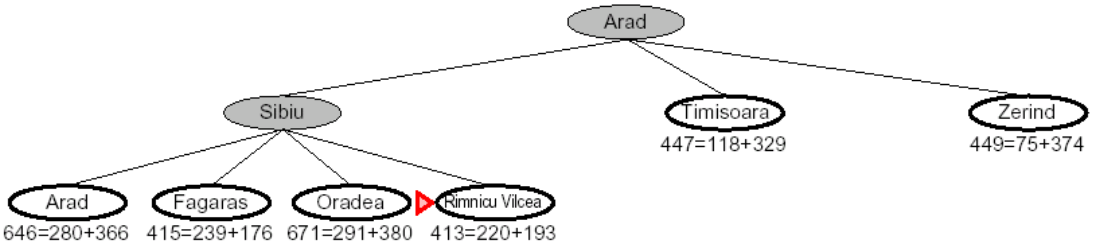
- **Pesquisa gulosa**
  - minimiza o custo estimado de *n* até o objetivo => *h(n)*
  - Não é completa nem óptima
- **Pesquisa de custo uniforme**
  - minimiza o custo do caminho da raiz até *n* => *g(n)*
  - É completa e óptima – mas analisa muitos nós
- **Algoritmo A*: combina as duas estratégias**
  - Evitar expandir caminhos que já são caros
  - Função de Avaliação *f(n) = g(n) + h(n)*
    - *f(n)* = custo estimado total da solução de custo mais baixo passando por *n*

Algortimo A*

- **Exemplo de Pesquisa A***

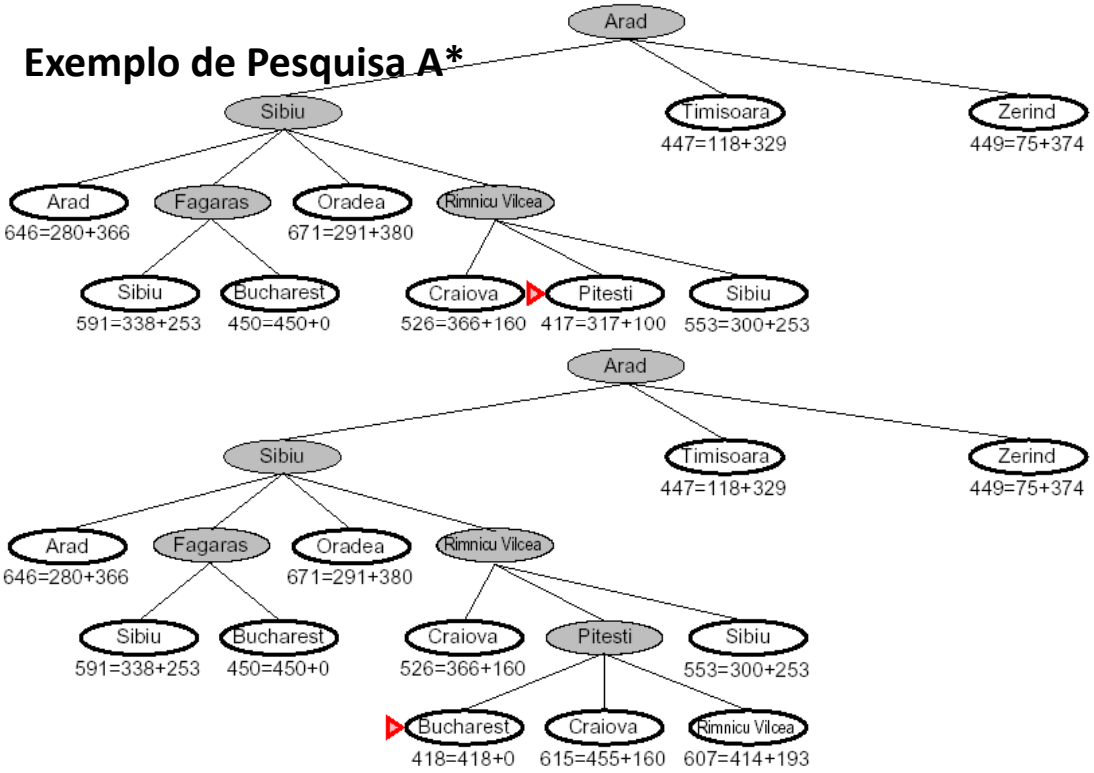






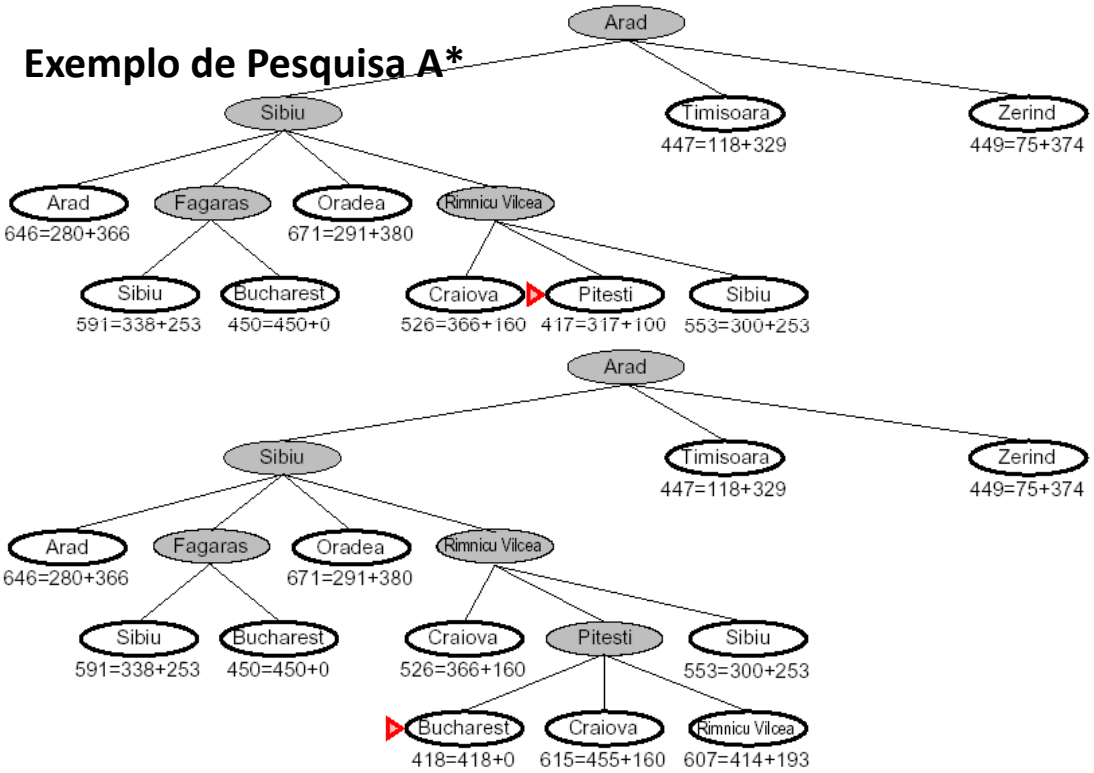
Algortimo A*

- **Exemplo de Pesquisa A***



Algortimo A*

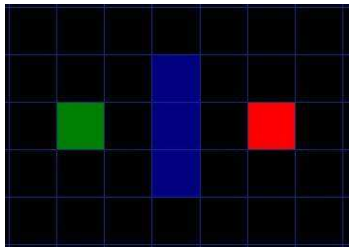
- **Exemplo de Pesquisa A***

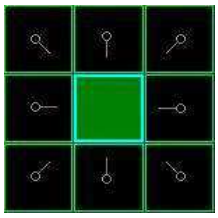


Algoritmo A*

F(n) = G(n) + H(n) , onde

- **G = é o custo do movimento para se mover do estado inicial até ao estado atual**
- **H = é a heurística, ou seja, o custo estimado para se mover do estado atual até ao destino final (estado solução)**
- **Nota: Não sabemos realmente a distância real entre o estado atual e a solução (senão sabíamos os passos para resolver o problema e não precisávamos de pesquisa...)**





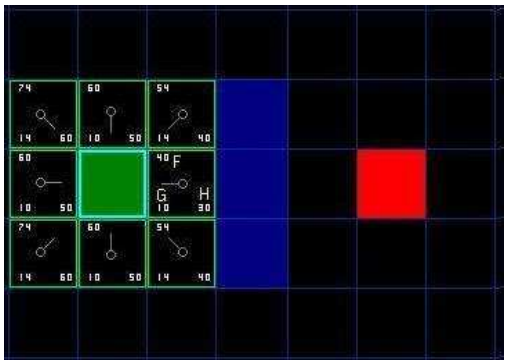
Algoritmo A*

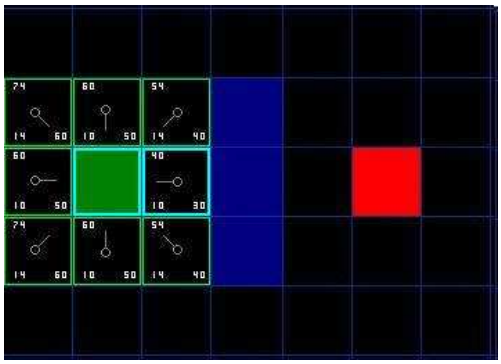
F(n) = G(n) + H(n) , onde

- **G = é o custo do movimento para se mover do estado inicial até ao estado atual**
- **H = é a heurística, ou seja, o custo estimado para se mover do estado atual até ao destino final (estado solução)**

Sendo G(N) = 10 mov. em linha reta e 14 mov. na diagonal

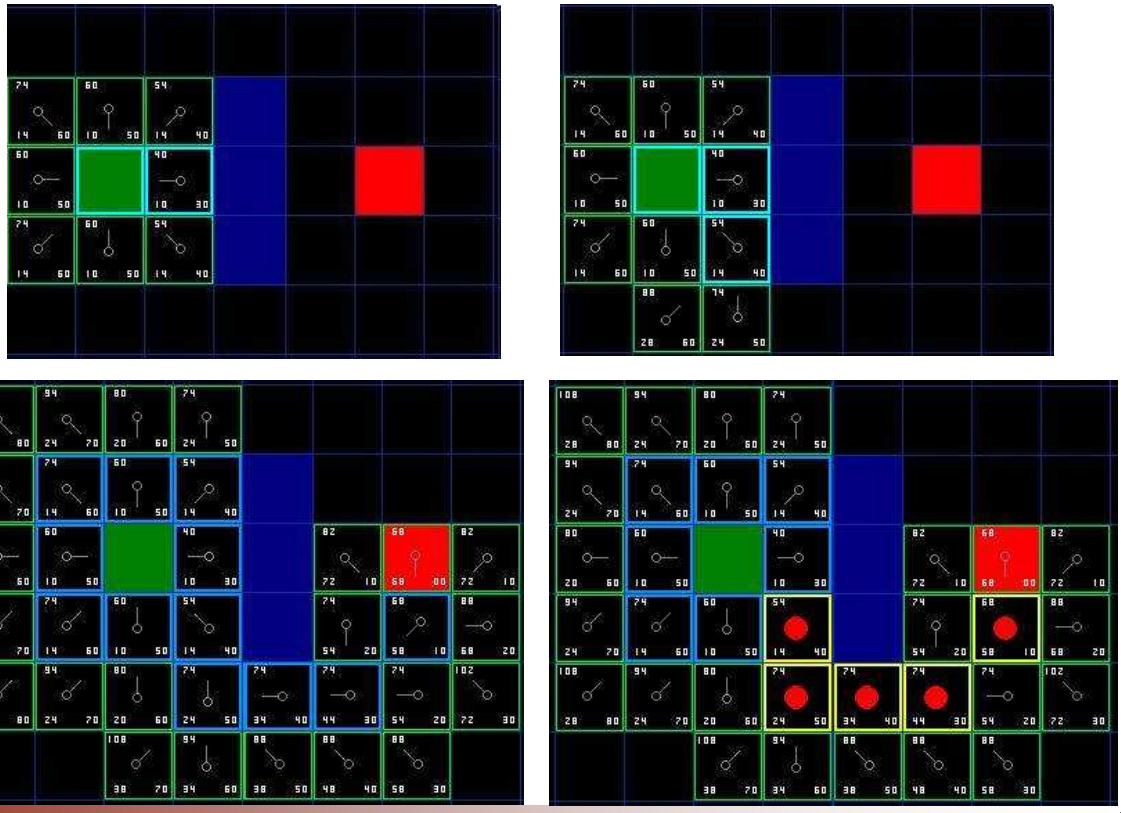
Sendo H(N) = sqrt((Xf-Xi)^2 + (Yf-Yi)^2)





Escola de Engenharia
Universidade do Minho

Algoritmo A*



Escola de Engenharia
Universidade do Minho

Complementos de Programação de Computadores – Aula 11
Estuturas de Dados: Árvores

Mestrado Integrado em Electrónica Industrial e Computadores

Luís Paulo Reis

lpreis@dsi.uminho.pt

Professor Associado do Departamento de Sistemas de Informação, Escola de Engenharia, Universidade do Minho, Portugal

(Slides Baseados em Cortez 2011 , Reis, Rocha e Faria, 2007)