Força Bruta / Backtracking (parte 2)

SCC0210 - Alg. Avançados e Aplicações

Baseado em várias fontes:

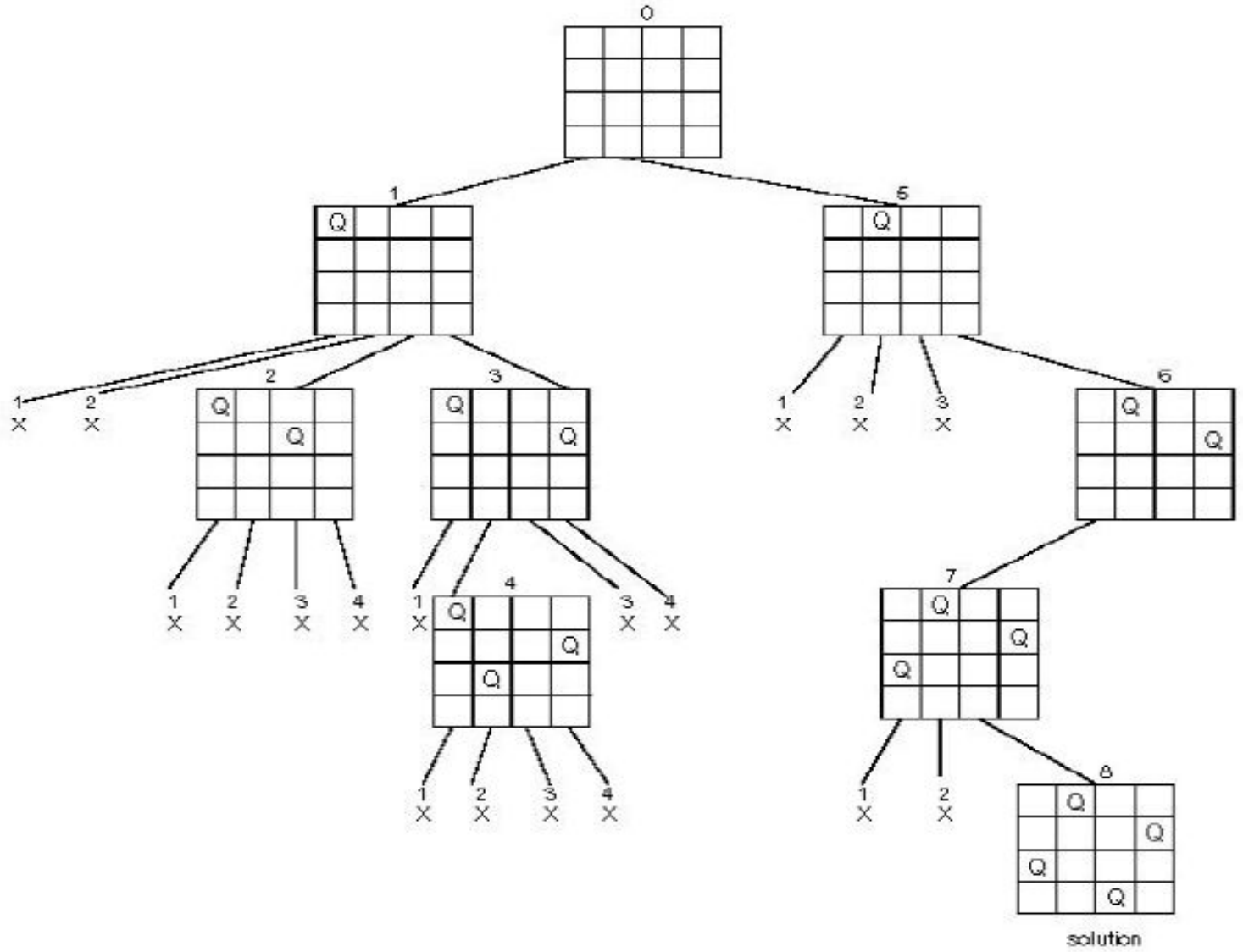
- Artificial Intelligence: A Modern Approach – Russell & Norvig
  http://aima.eecs.berkeley.edu/slides-ppt/
- Livro do Halim – Competitive Programming 3
- Vários sítios na internet

Relembrando : Backtracking

- Técnicas de busca exaustiva geram todas as soluções candidatas e então identificam aquela (ou aquelas) com a propriedade desejada

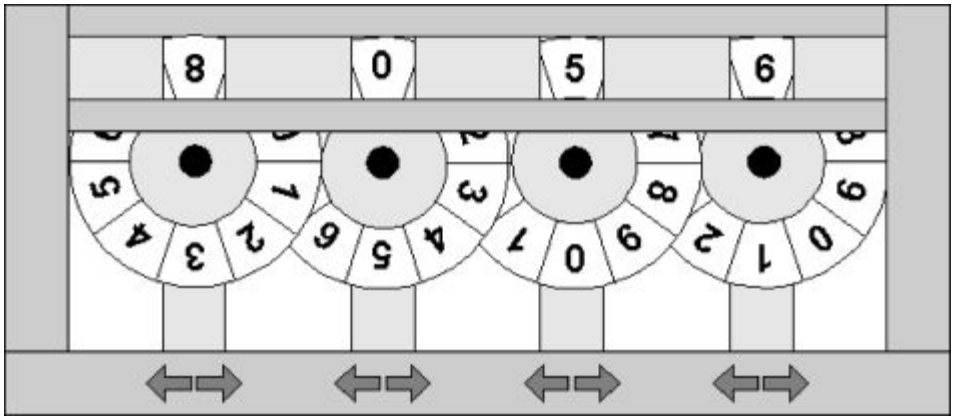
- A ideia do Backtracking, ao contrário, é construir soluções parciais e avaliá-las da seguinte maneira:
  - Se a solução parcial pode ser continuada, sem violar os objetivos, então, faça-o, incorporando um próximo componente legítimo
  - Se não há nenhuma opção legítima, nenhuma alternativa restante precisa ser considerada.
  - Backtracking usa conceito de árvore de estado-espaço.

Backtracking: 4 Queens!



Playing with Wheels

- A partir de uma configuração inicial ($S_1 S_2 S_3 S_4$), um conjunto de configurações proibidas ($F_{i1} F_{i2} F_{i3} F_{i4}$) $(1 \leq i \leq n)$ e uma final ($T_1 T_2 T_3 T_4$), escreva um programa que calcule a menor qtd de movimentações nas rodas para sair de S e chegar em T...



Playing with Wheels

- Qual é o grafo por trás deste problema?
  - Vértices: ?
  - Arestas: ?
- Qual é o grau de cada vértice deste grafo?
- Como é possível encontrar o caminho mínimo?

Playing with Wheels

- Qual é o grafo por trás deste problema?
  - Vértices: estados (números de 4 dígitos);
  - Arestas: possíveis transições entre estados.
- Qual é o grau de cada vértice deste grafo?
  - Exatamente oito, pois existem oito possíveis transições a partir de cada estado.
- Como é possível encontrar o caminho mínimo?
  - A forma mais simples é utilizar uma busca em largura, pois o grafo é não-ponderado.

Playing with Wheels

- É necessário representar o grafo explicitamente?
- Como saber por quais estados já passou?

Playing with Wheels

- É necessário representar o grafo explicitamente?
  - Não. Podemos construir uma função que retorna os próximos estados dado o estado atual;
  - Backtracking com ***fringe*** organizado em uma fila.

- Os estados já visitados podem ser marcados em uma matriz.

Busca X BackTracking

- Mesmo conceito:
  - Busca em grafos: grafo explícito;
  - Backtracking: grafo implícito
- Ou seja:
  - Busca em grafos: pode-se consultar o grafo para se saber os vértices adjacentes;
  - Backtracking: deve-se ter uma sub-rotina que gera os próximos “vértices” e verifica se são válidos.

Playing with Wheels

```
#include<stdio.h>
#include<queue>

using namespace std;

struct state {
    char digit[4];
    int depth;
};

char moves[8][4] = {{-1,  0,  0,  0 },
                    { 1,  0,  0,  0 },
                    { 0, -1,  0,  0 },
                    { 0,  1,  0,  0 },
                    { 0,  0, -1,  0 },
                    { 0,  0,  1,  0 },
                    { 0,  0,  0, -1 },
                    { 0,  0,  0,  1 }};
```

Playing with Wheels

```
void next_states(state s, state nexts[8]) {
    int i,j;

    for (i=0; i < 8; i++) {
        nexts[i] = s;
        nexts[i].depth = s.depth+1;
        for (j = 0; j < 4; j++) {
            nexts[i].digit[j] += moves[i][j];
            if (nexts[i].digit[j] < 0)
                nexts[i].digit[j] = 9;
            if (nexts[i].digit[j] > 9)
                nexts[i].digit[j] = 0;
        }
    }
}
```

```
int main(void) {
    int nr_tests, test, forbidden, i, j, k, l, d;
    char visited[10][10][10][10];
    state initial, final, aux;

    scanf("%d", &nr_tests);
    for (test=0; test < nr_tests; test++) {
        scanf("%d %d %d %d", &initial.digit[0], &initial.digit[1],
                             &initial.digit[2], &initial.digit[3]);
        scanf("%d %d %d %d", &final.digit[0], &final.digit[1],
                             &final.digit[2], &final.digit[3]);
        scanf("%d", &forbidden);
        for(i=0; i<10; i++)
            for(j=0; j<10; j++)
                for(k=0; k<10; k++)
                    for(l=0; l<10; l++)
                        visited[i][j][k][l] = 0;
        for(i=0; i<forbidden; i++) {
            scanf("%d %d %d %d", &aux.digit[0], &aux.digit[1],
                                 &aux.digit[2], &aux.digit[3]);
            visited[aux.digit[0]][aux.digit[1]]
                   [aux.digit[2]][aux.digit[3]] = 1;
        }
        initial.depth=0;
        printf("%d\n", bfs(initial,final,visited));
    }
    return 0;
}
```

```
int bfs(state current, state final, char visited[10][10][10][10]) {
    state nexts[8];
    int i;
    queue<state> q;
                        /* Argh, o estado inicial pode ser proibido */
    if (!visited[current.digit[0]][current.digit[1]]
                [current.digit[2]][current.digit[3]]) {
        visited[current.digit[0]][current.digit[1]]
                [current.digit[2]][current.digit[3]] = 1;
        q.push(current);
        while (!q.empty()) {
            current = q.front();
            q.pop();
            if (equal(current, final)) return current.depth;
            next_states(current, nexts);
            for (i = 0; i < 8; i++)
                if (!visited[nexts[i].digit[0]][nexts[i].digit[1]]
                            [nexts[i].digit[2]][nexts[i].digit[3]]) {
                    visited[nexts[i].digit[0]][nexts[i].digit[1]]
                            [nexts[i].digit[2]][nexts[i].digit[3]] = 1;
                    q.push(nexts[i]);
                }
        }
    }
    return -1;
}
```

```
int bfs(state current, state final, char visited[10][10][10][10]) {
    state nexts[8];
    int i;
    queue<state> q;
                        /* Argh, o estado inicial pode ser proibido */
    if (!visited[current.digit[0]][current.digit[1]]
                [current.digit[2]][current.digit[3]]) {
        visited[current.digit[0]][current.digit[1]]
                [current.digit[2]][current.digit[3]] = 1;
        q.push(current);
        while (!q.empty()) {
            current = q.front();
            q.pop();
            if (equal(current, final)) return current.depth;
            next_states(current, nexts);
            for (i = 0; i < 8; i++)
                if (!visited[nexts[i].digit[0]][nexts[i].digit[1]]
                            [nexts[i].digit[2]][nexts[i].digit[3]]) {
                    visited[
                            [
                    q.push(ne
                }
        }
    }
    return -1;
}
```

```
int equal(state s, state e) {
    int i;

    for (i=0; i < 4; i++)
        if (s.digit[i] != e.digit[i])
            return 0;
    return 1;
}
```

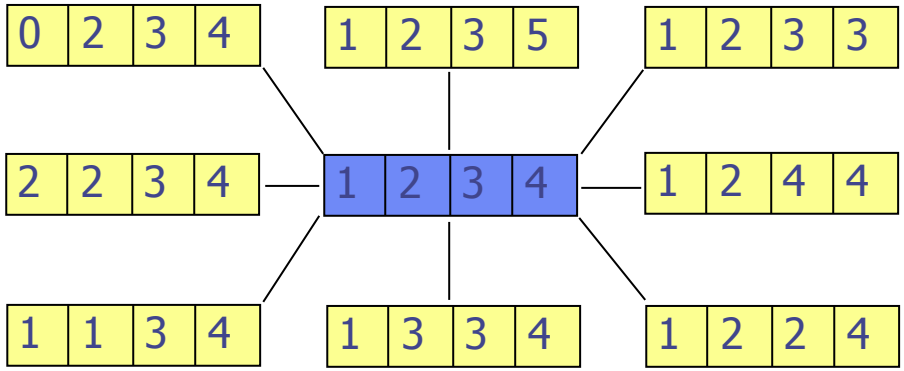
Playing with Wheels

- Vc consegue explicar porque este algoritmo funciona e retorna a “shortest path” para o problema???? (já estudou isso em grafos…)

- Tem como tornar a solução mais inteligente?

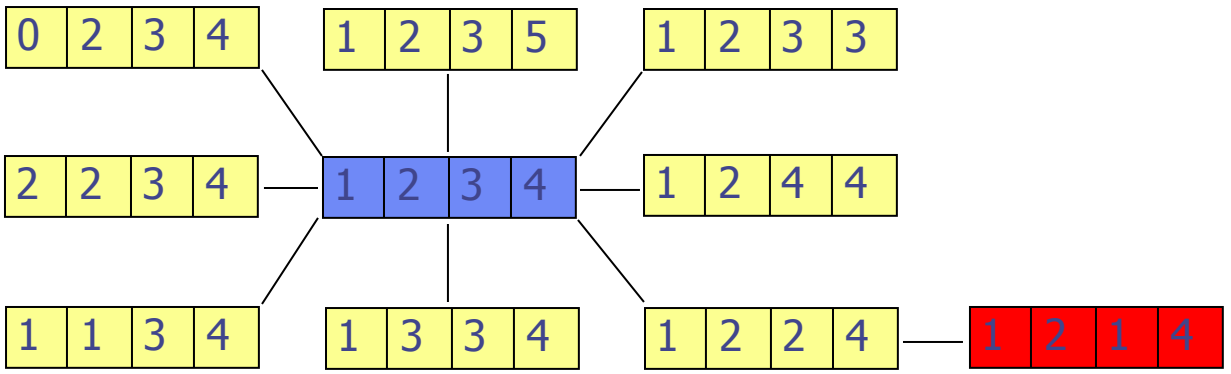
Playing with Wheels: heurística

- Com a busca em largura, dado um certo estado (vértice - [1,2,3,4]), todos os vértices adjacentes não visitados são inseridos na fila *q*.



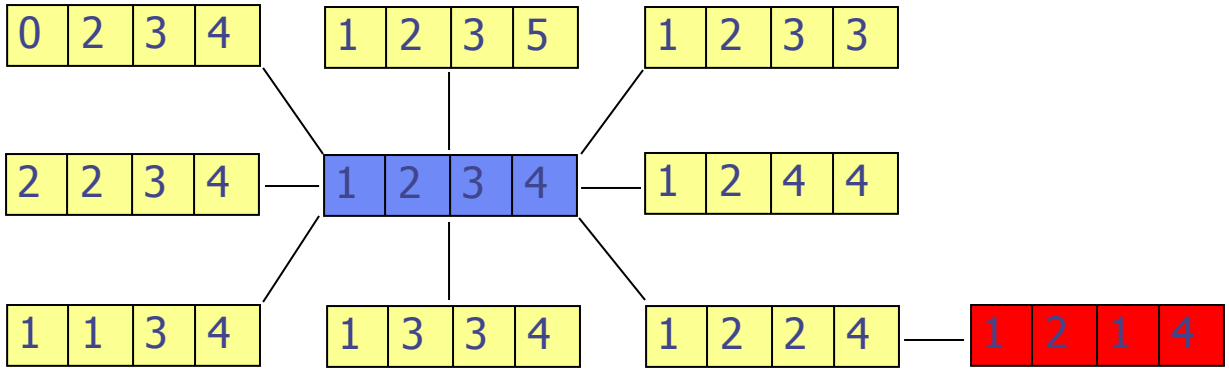
Playing with Wheels: heurística

- Com a busca em largura todos os vértices adjacentes não visitados são inseridos na fila *q*.
- Sendo [1,2,3,4] o estado inicial, imagine que o estado final é [1,2,1,4]. Então é melhor seguir pelo vértice [1,2,2,4].



Playing with Wheels: heurística

- A função heurística pode ser então a distância entre o estado analisado e o estado final:
  - D([1,2,2,4], [1,2,1,4]) = 1 é a menor dentre todos os 8 possíveis vizinhos



Playing with Wheels: heurística

- Um detalhe importante neste caso:
  - **A existência de estados proibidos faz com que a função heurística seja uma estimativa otimista.**
  - O que significa ser "otimista" ????

Playing with Wheels: heurística

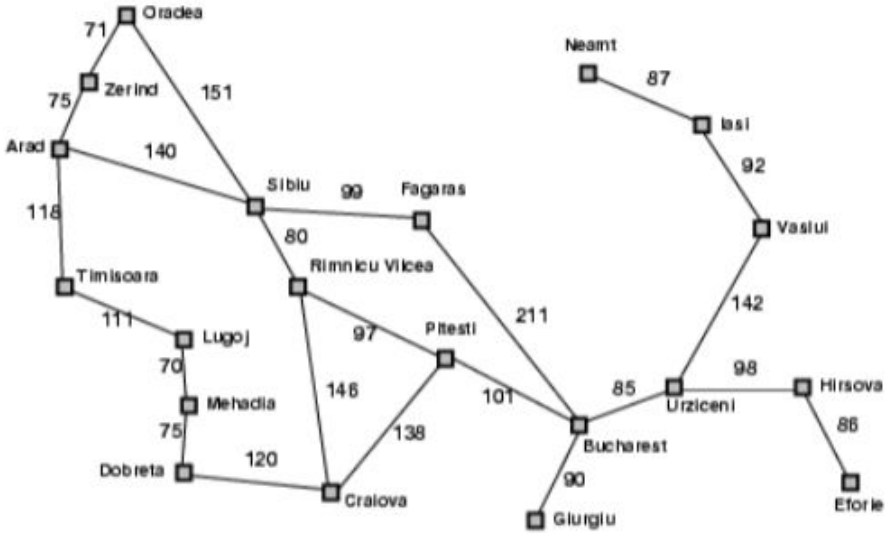
- Um detalhe importante neste caso:
  - **A existência de estados proibidos faz com que a função heurística seja uma estimativa otimista.**
  - O que significa ser "otimista" ????
    - Nunca superestima o custo real de se chegar ao destino. Seja n um estado atual e h(n) a função heurística -> h(n) é sempre > ou = ao real valor para se chegar até o destino final.

Deixemos o problema de lado e falemos sobre heurísticas

- Intuição
- Uma maneira rápida de estimar o quão próximo estamos do objetivo. (não estamos pensando em Dijkstra!!)
- “aquela observação que muita gente faz de maneira simplista, baseada na intuição)

Heurísticas

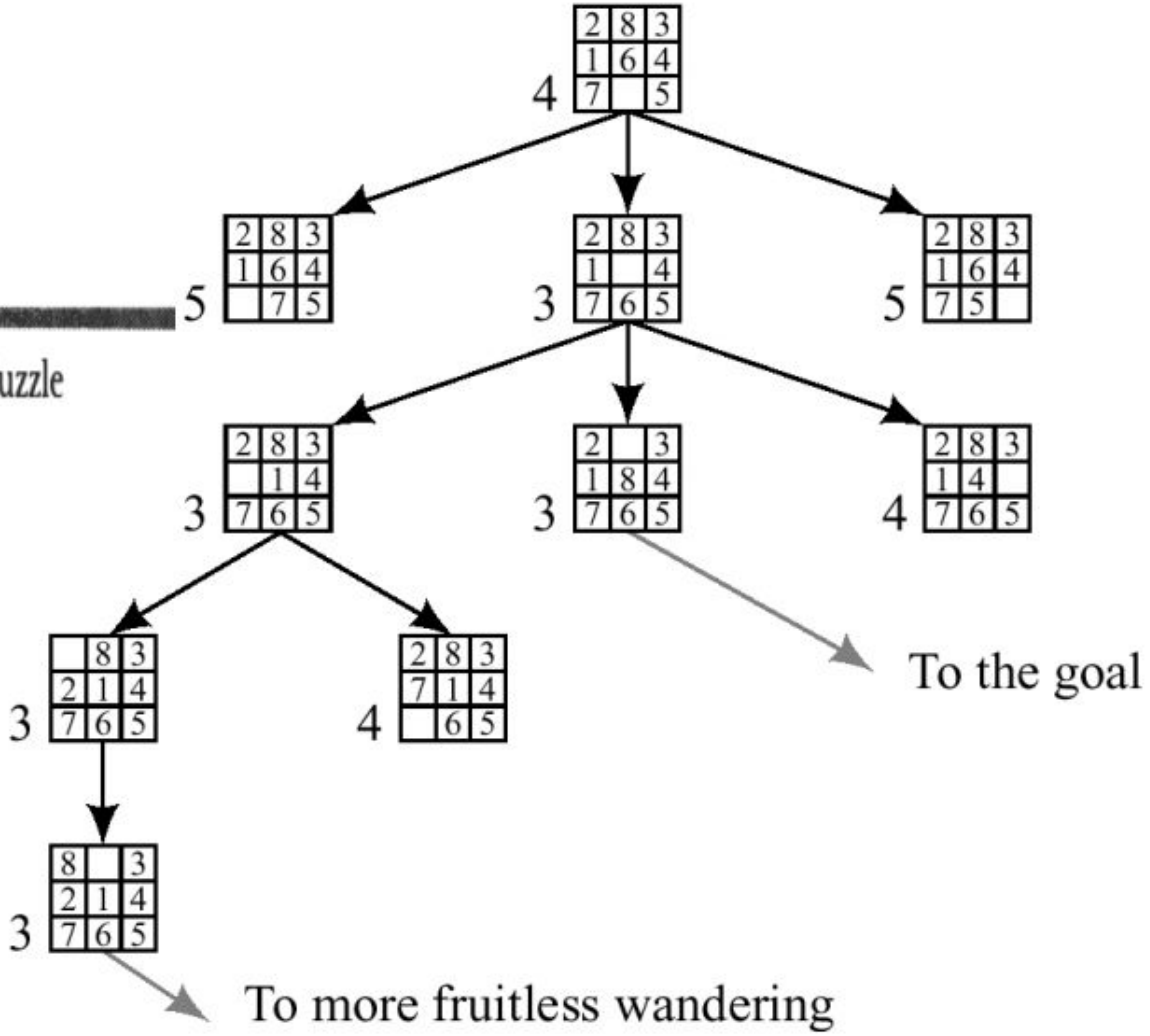
- Para o jogo dos tijolos
  - h(n): nro de peças fora do lugar
  - h(n): distância manhattan.
- Para o mapa:
  - A Dist. Euclidiana (linha reta entre cada cidade à Bucareste.



Mas será que só isso resolve?

Figure 8.1

Start and Goal Configurations for the Eight-Puzzle



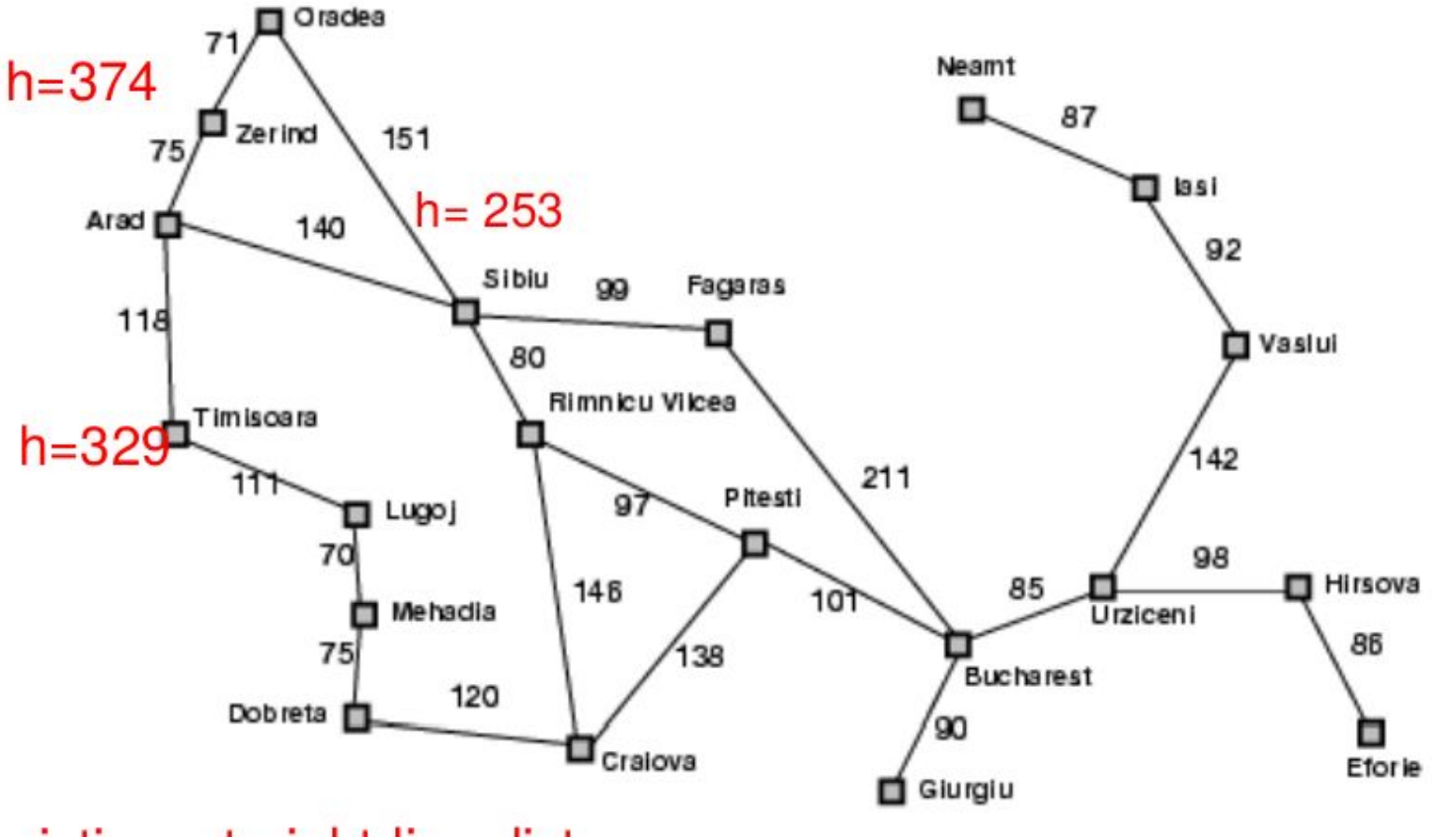
Mas será que só isso resolve?

- Vimos que não.
- O que acabamos de fazer é uma técnica conhecida como “Greedy best-first search”

Greedy Best-first Search

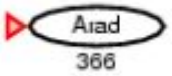
- Função de avaliação f(n) = h(n)
  - Estimativa do custo de n até o destino
- No caso do mapa, h(n) = h_{SLD}(n) -> Straight Line Distance de n até Bucareste

Greedy Best-first Search



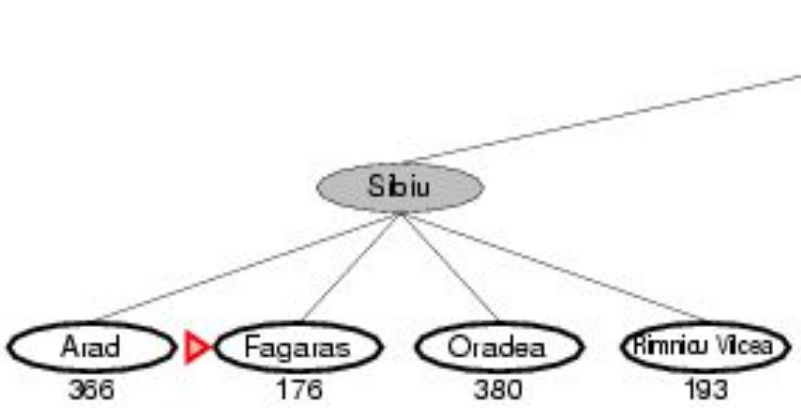
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy Best-first Search



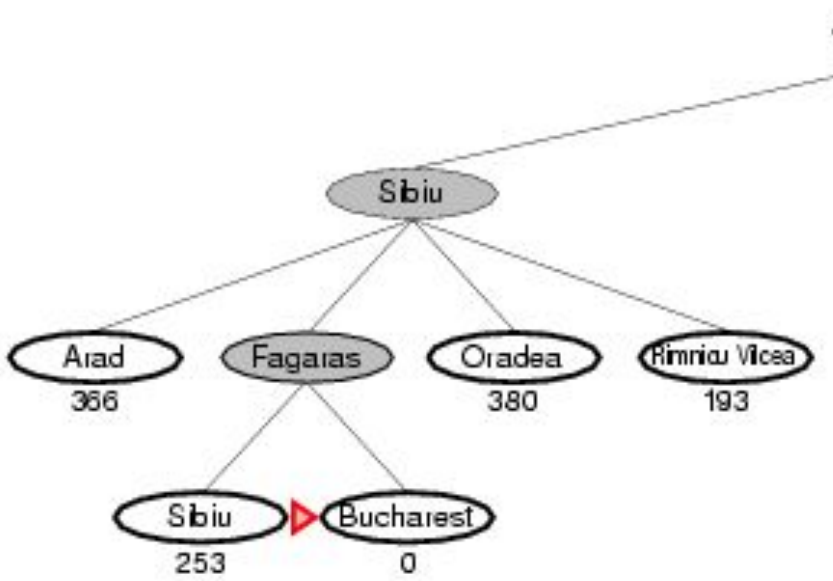
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy Best-first Search



Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy Best-first Search



Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Best-first Search: problemas

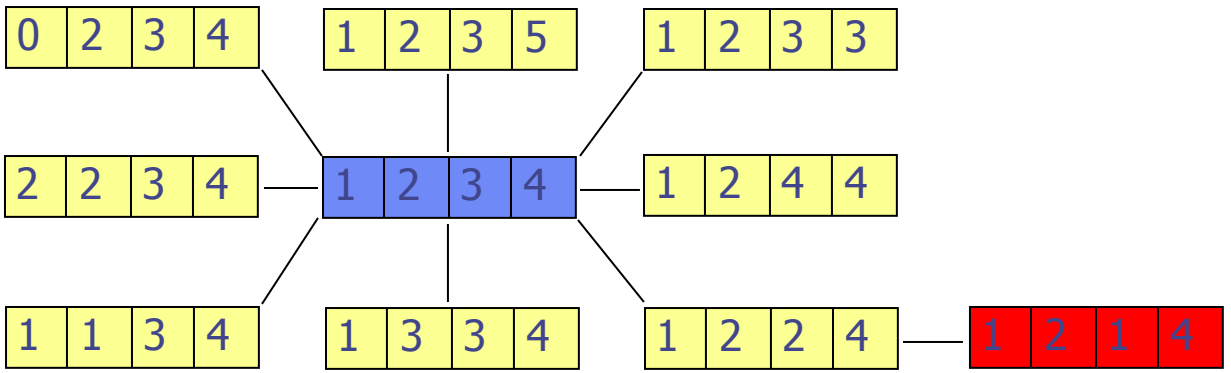
- Busca não completa
  - Nem sempre acha uma solução, se existe
- Loops infinitos
- Não é ótima

Pseudo-código Best-first Search

1. Put the start node *s* on a list called *OPEN* of unexpanded nodes.
2. If *OPEN* is empty exit with failure; no solutions exists.
3. Remove the first *OPEN* node *n* at which f is minimum (break ties arbitrarily), and place it on a list called *CLOSED* to be used for expanded nodes.
4. If *n* is a goal node, exit successfully with the solution obtained by tracing the path along the pointers from the goal back to *s*.
5. Otherwise expand node *n*, generating all it's successors with pointers back to *n*.
6. For every successor *n'* on *n*:
   a. Calculate *f* (*n'*).
   b. if *n'* was neither on *OPEN* nor on *CLOSED*, add it to *OPEN*. Attach a pointer from *n'* back to *n*. Assign the newly computed *f(n')* to node *n'*.
   c. if *n'* already resided on *OPEN* or *CLOSED*, compare the newly computed *f(n')* with the value previously assigned to *n'*. If the old value is lower, discard the newly generated node. If the new value is lower, substitute it for the old (*n'* now points back to *n* instead of to its previous predecessor). If the matching node *n'* resides on *CLOSED*, move it back to *OPEN*.
7. Go to step 2.

Playing with Wheels: o retorno

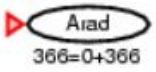
- A função heurística pode ser incorporada alterando-se a fila da busca em largura por uma fila de prioridade (ordenada pela função heurística).
  - *pq* = (1:[1,2,2,4], 3:[1,2,4,4], 3:[1,2,3,3], 3:[1,2,3,5], 3:[0,2,3,4], 3:[2,2,3,4], 3:[1,1,3,4], 3:[1,3,3,4])

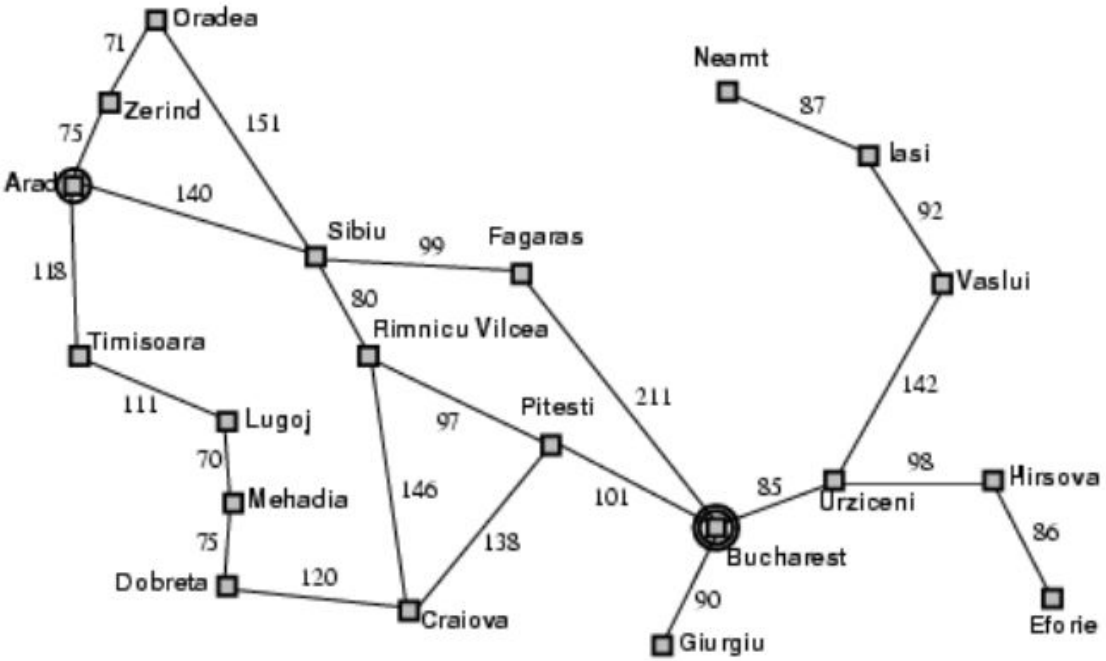


Busca A*

- Ideia
  - evitar expandir trilhas de custo alto
  - dar prioridade àquelas que são promissoras
- Função de avaliação f(n) = g(n) + h(n)
- g(n) = custo **da origem** até n
- h(n) = custo **estimado** de n até o destino

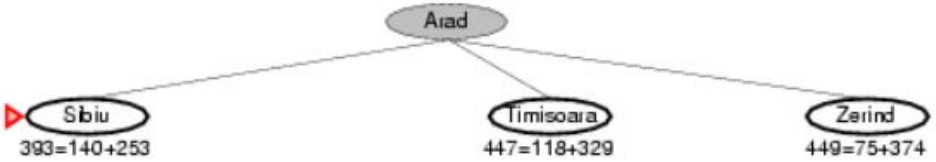
Busca A*

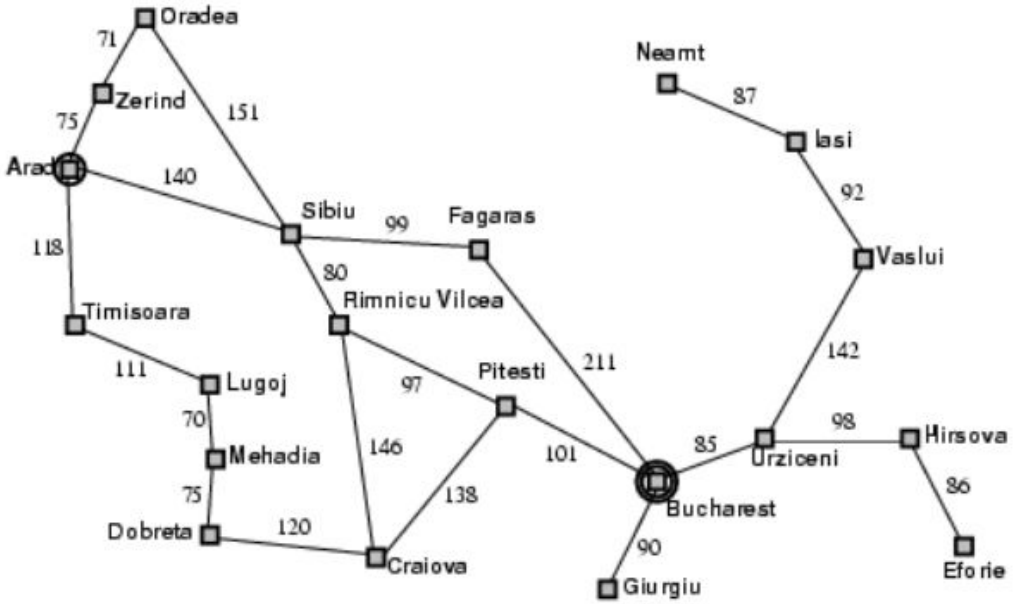




Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

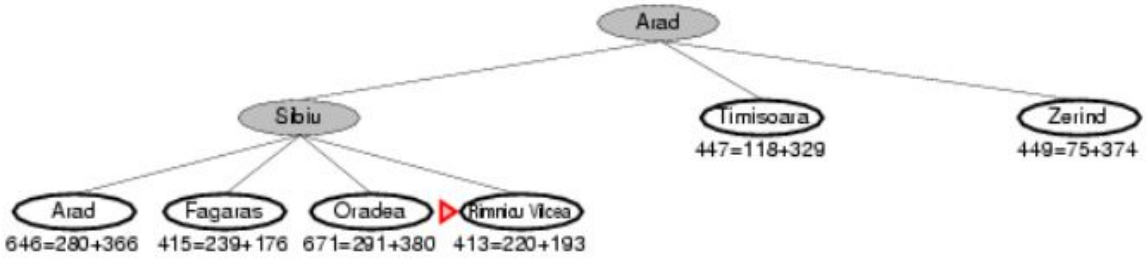
Busca A*

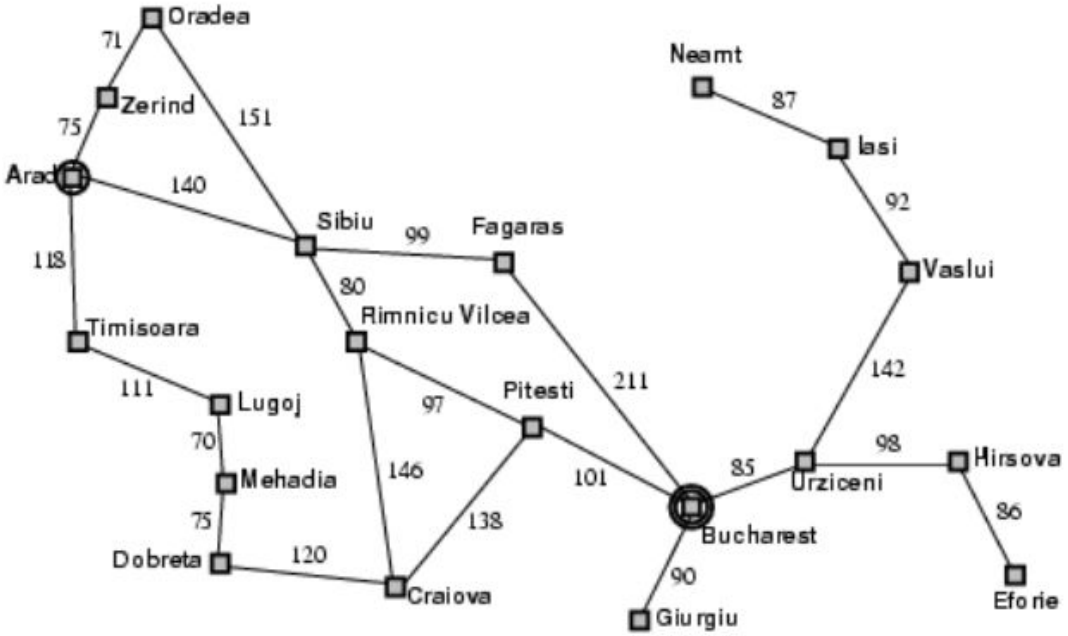




Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

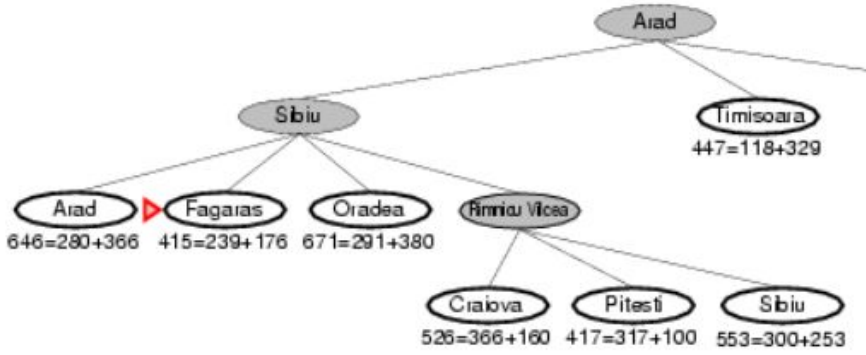
Busca A*

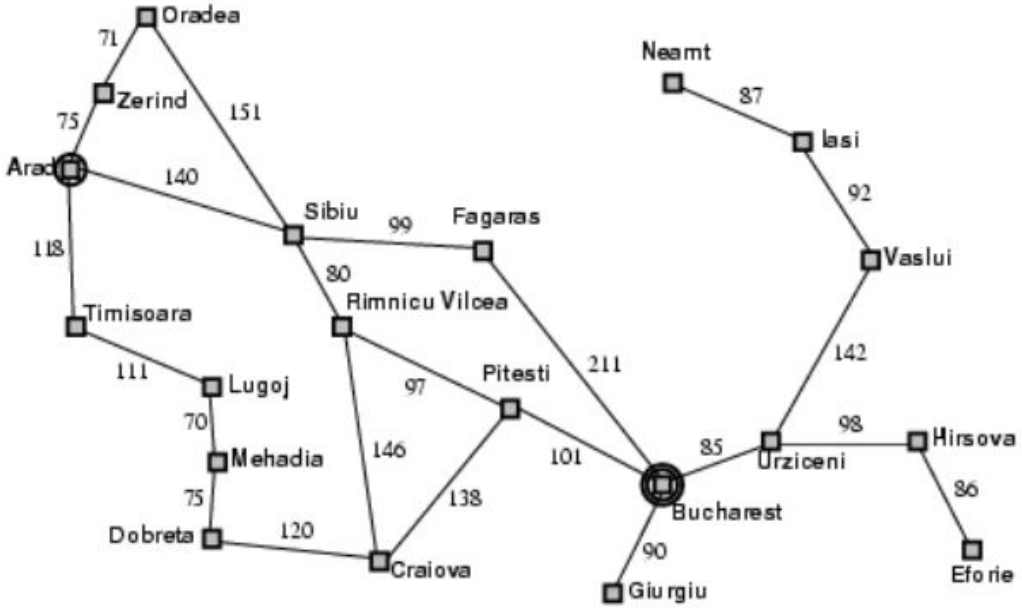




Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

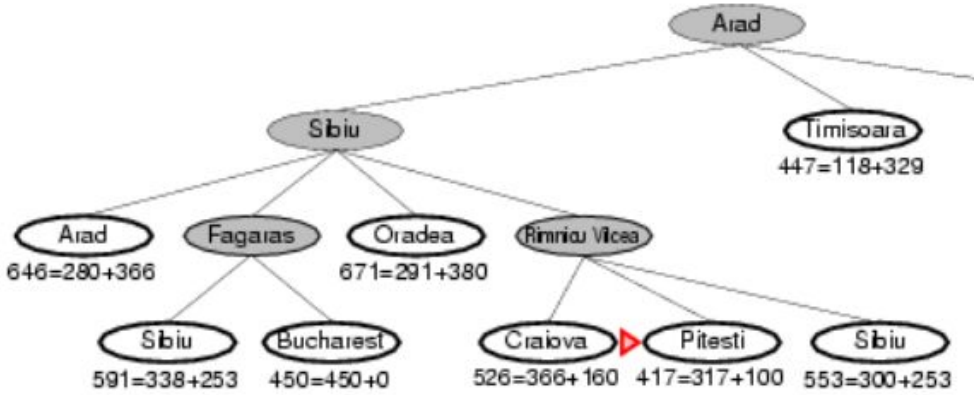
Busca A*

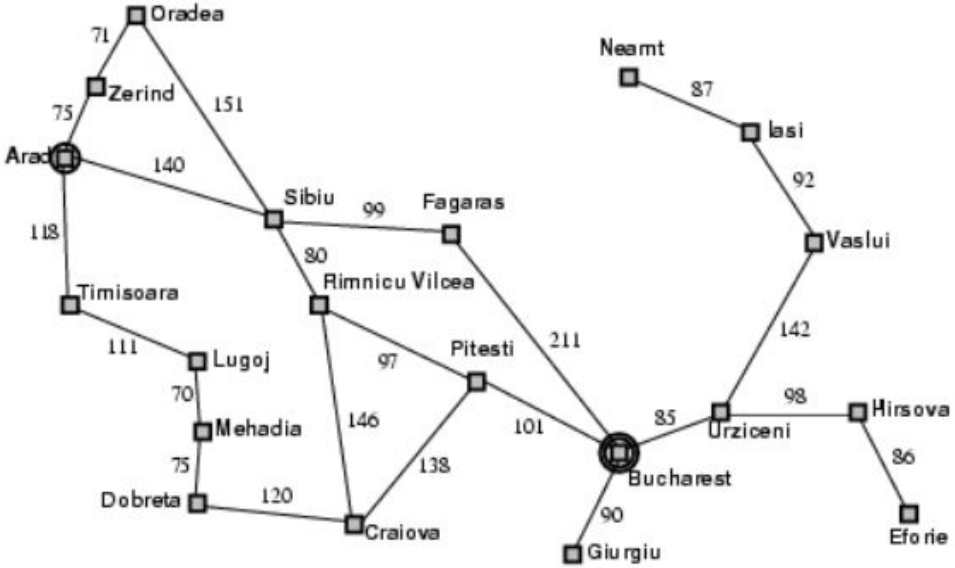




Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

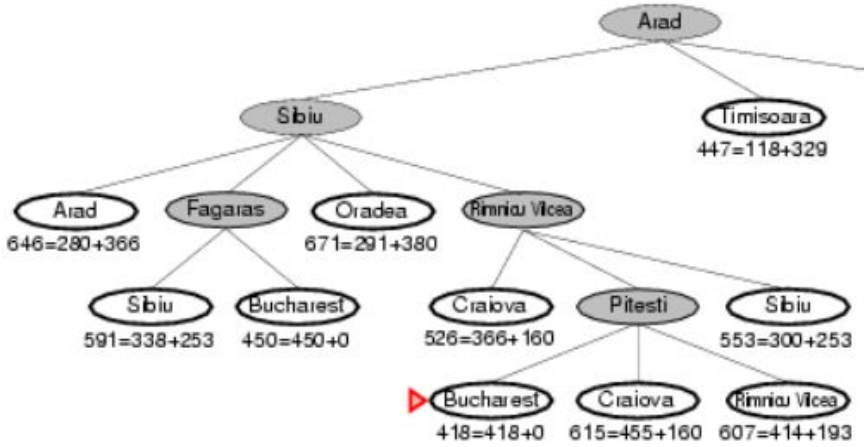
Busca A*

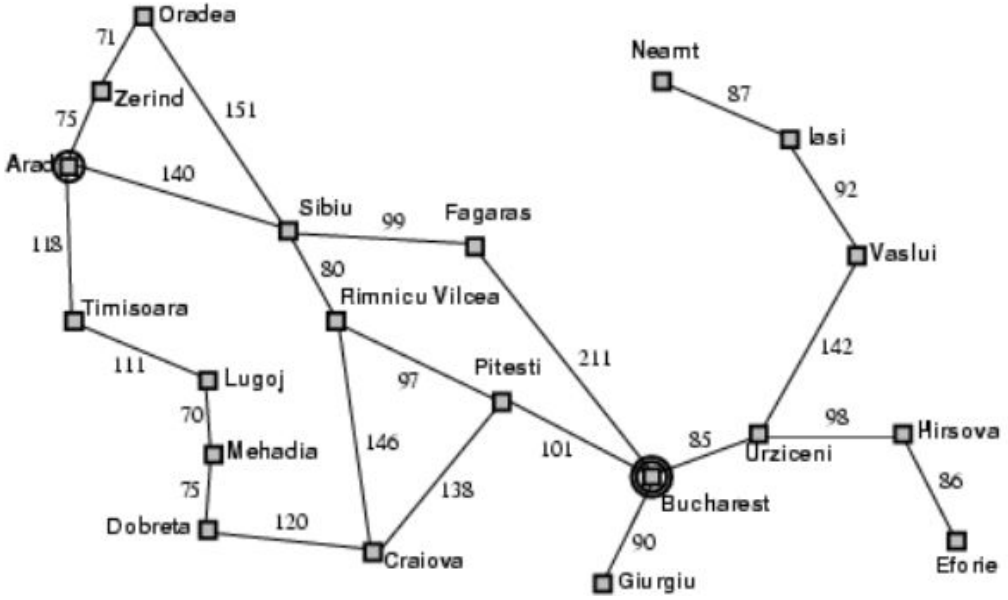




Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

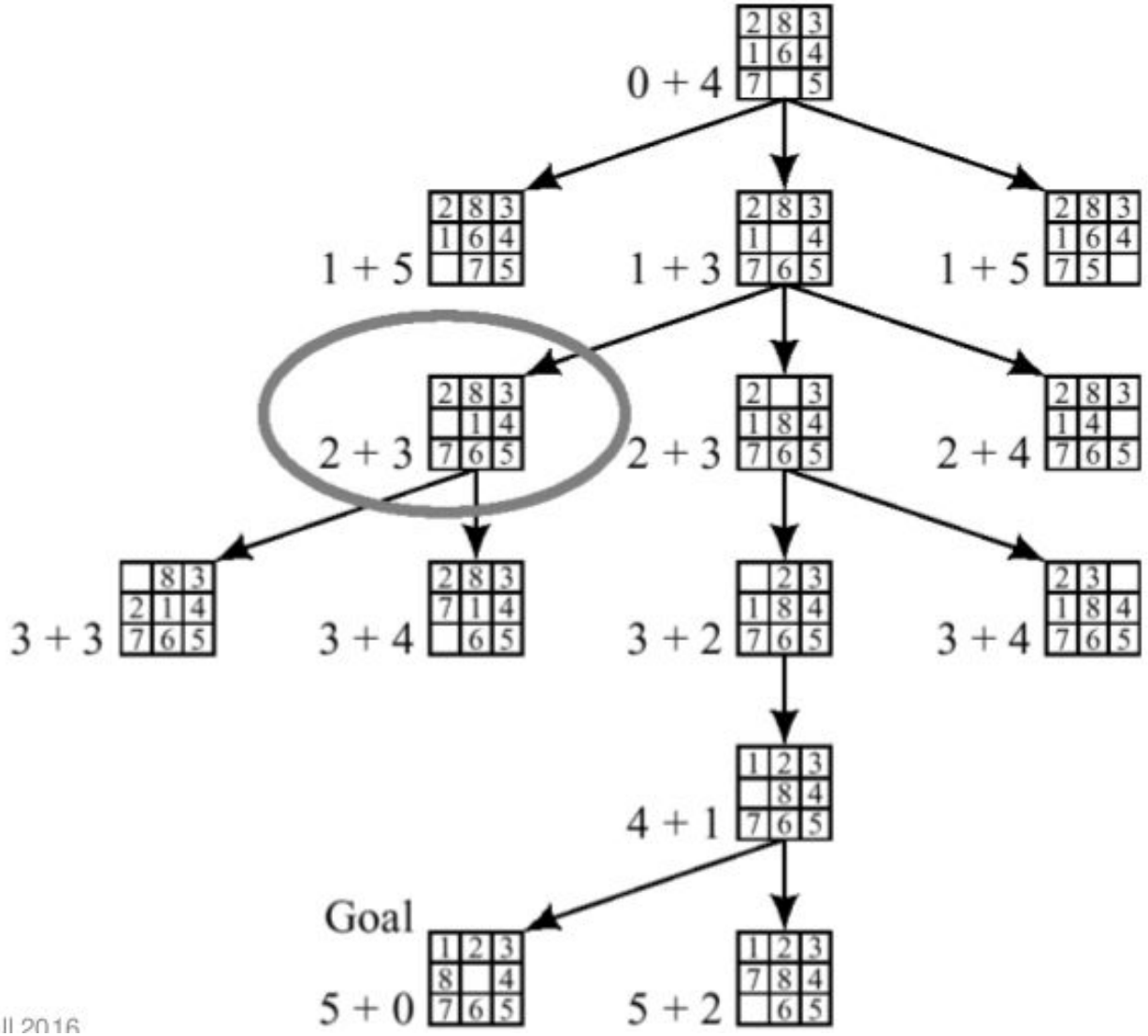
Busca A*



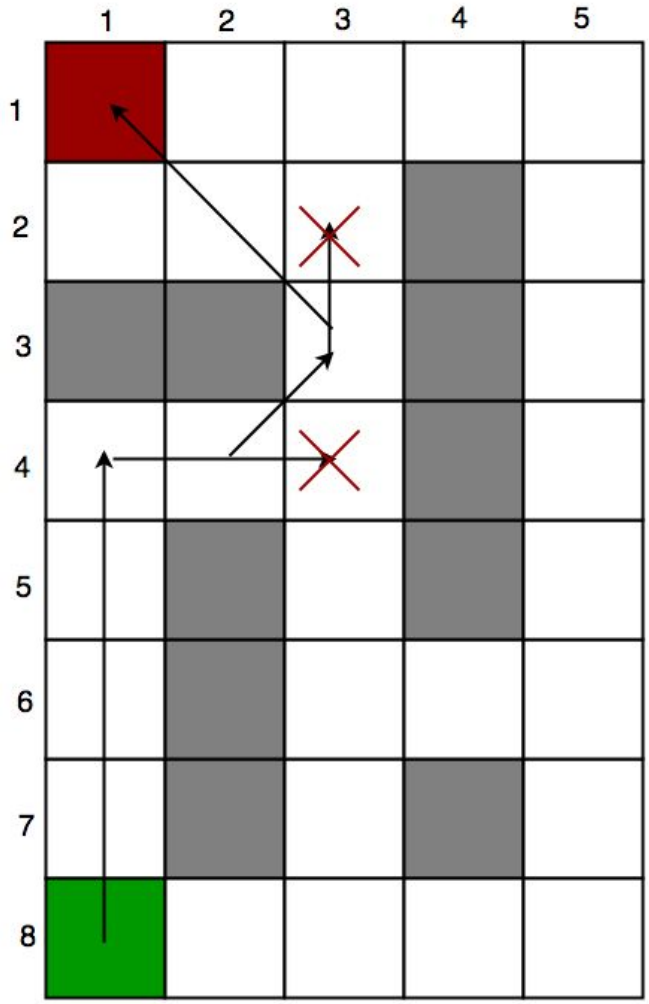


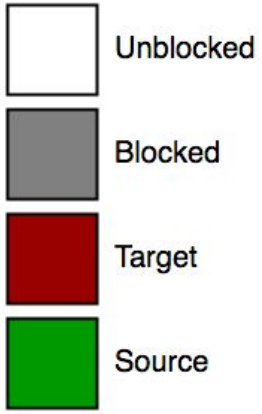
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Busca A*: para o jogo...



Busca A*: para labirinto do rato





A* Search Algorithm makes the most intelligent choice at each step. Hence you can see that algorithm goes from (4,2) to (3,3) and not (4,3) (shown by cross).

Similarly the algorithm goes from (3,3) to (2,2) and not (2,3) (shown by cross).

Formalizando A*

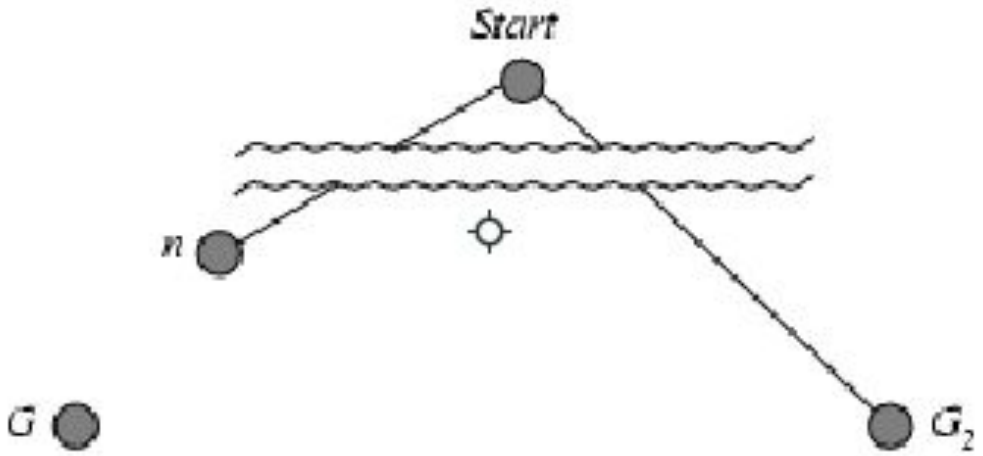
- Notação:
  - c(n, n') = custo real da aresta (n,n')
  - g(n) = custo do caminho atual do início a n
  - h(n) = estimativa do menor custo de uma trilha de n até o destino
  - função de avaliação: f(n) = g(n) + h(n)
  - h*(n) é o menor custo real de n até o destino
- A heurística h(n) é **admissível** se, para todo nó n $h(n) \leq h^*(n)$
- Uma heurística admissível jamais superestima a distância real, ou seja, é **otimista**

Algoritmo A*

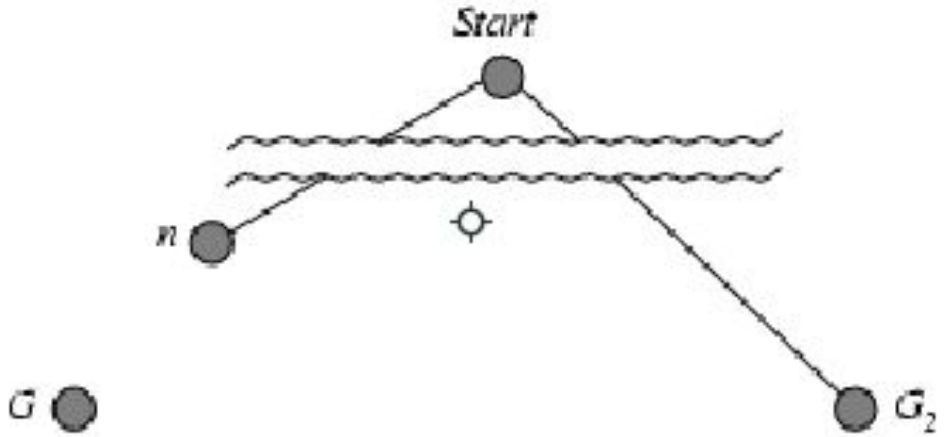
- Input: an implicit search graph problem with cost on the arcs
- Output: the minimal cost path from start node to a goal node.
  - 1. Put the start node s on OPEN.
  - 2. If OPEN is empty, exit with failure
  - 3. Remove from OPEN and place on CLOSED a node n having minimum f.
  - 4. If n is a goal node exit successfully with a solution path obtained by tracing back the pointers from n to s.
  - 5. Otherwise, expand n generating its children and directing pointers from each child node to n.
    - For every child node n' do
      - evaluate h(n') and compute $f(n') = g(n') + h(n') = g(n) + c(n,n') + h(n')$
      - If n' is already on OPEN or CLOSED compare its new f with the old f. If the new value is higher, discard the node. Otherwise, replace old f with new f and reopen the node.
      - Else, put n' with its f value in the right order in OPEN
  - 6. Go to step 2.

Prova da optimalidade de A*

- Suponha que um destino subótimo G_2 foi gerado e está na fringe (fila prioridade, nó aberto). Seja n um nó **não expandido** na fringe tal que n está na trilha mais curta de um nó G.



Prova da optimalidade de A*

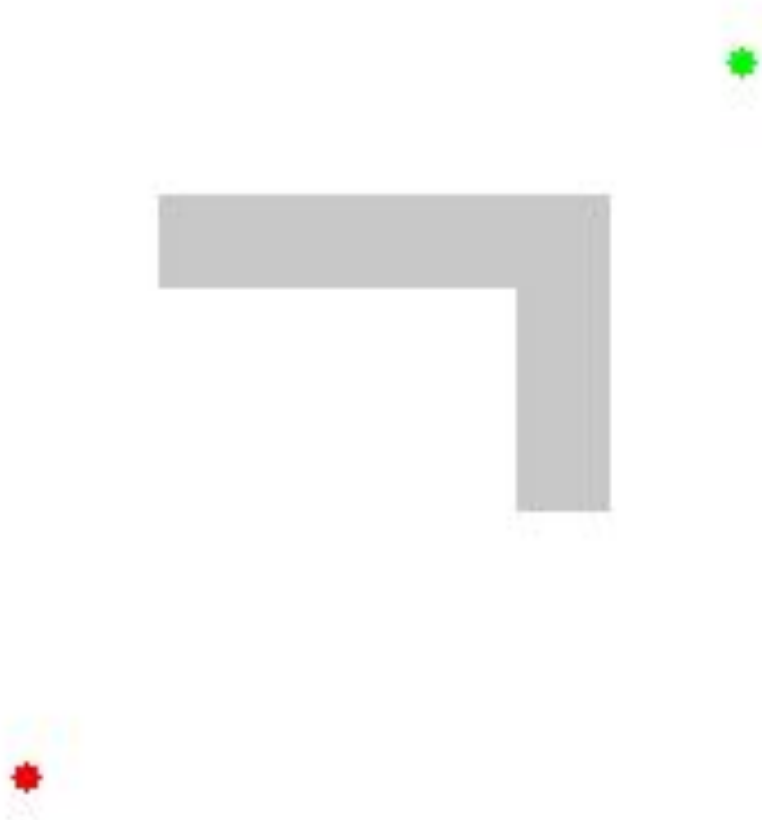


- $f(G_2) = g(G_2)$ ->> porque $h(G_2) = 0$
- $g(G_2) > g(G)$ ->> lembre-se que G_2 é subótimo
- $f(G) = g(G)$ ->> porque $h(G) = 0$
- $f(G_2) > f(G)$ ->> a partir do exposto acima
- $h(n) \leq h^*(n)$ ->> h é admissível
- $g(n) + h(n) \leq g(n) + h^*(n)$
- $f(n) \leq f(G)$
-
- Portanto, $f(G_2) > f(n)$ e A* jamais selecionará G_2 para expansão !!!!!

Busca A*

- A* possui as seguintes características:
  - **Completa**: garantia de encontrar uma solução quando existe;
  - **Ótima**: encontra a melhor solução entre várias soluções não ótimas;
  - **Eficiência ótima**: nenhum outro algoritmo da mesma família expande um número menor de nós que o A*.
- Por outro lado, A* pode consumir muita memória.
  - Uma possível solução: IDA* (Russel e Norvig, 2002).

A* : “visualização”



A* Playing with Wheels

- Para modificar a solução atual para A* é necessário:
  - Definir uma função heurística e uma função custo;
  - Transformar a fila em uma fila de prioridades, na qual os estados são ordenados pelos valores da função heurística + função de custo;
  - Utilizar como próximo estado a ser processado o estado de menor combinação heurística + custo, fornecido pela fila de prioridades.