Algoritmos Avançados SCC-210

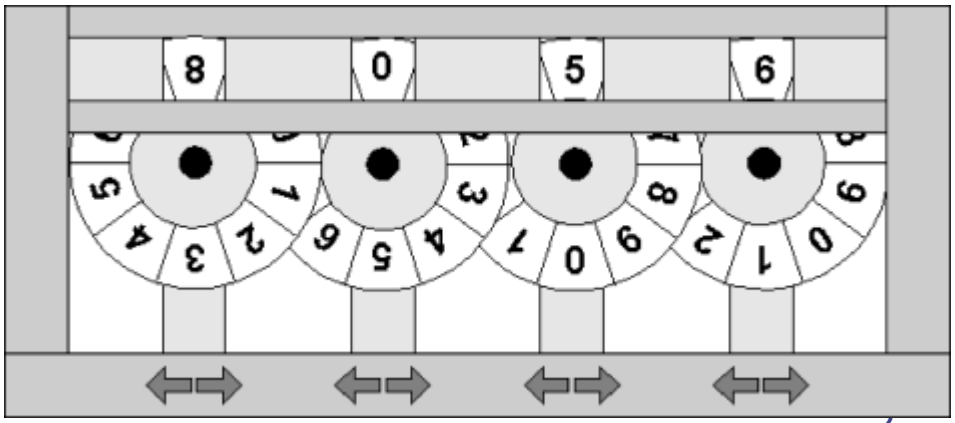
Backtracking II

Joao Batista

Playing with Wheels

Popularidade: C, Sucess rate: average, Level: 2

Consider the following mathematical machine. Digits ranging from 0 to 9 are printed consecutively (clockwise) on the periphery of each wheel. The topmost digits of the wheels form a four-digit integer. For example, in the following figure the wheels form the integer 8,056. Each wheel has two buttons associated with it. Pressing the button marked with a *left arrow* rotates the wheel one digit in the clockwise direction and pressing the one marked with the *right arrow* rotates it by one digit in the opposite direction.



Playing with Wheels

We start with an initial configuration of the wheels, with the topmost digits forming the integer $S_1S_2S_3S_4$. You will be given a set of n forbidden configurations $F_{i1}F_{i2}F_{i3}F_{i4}$ $(1 \leq i \leq n)$ and a target configuration $T_1T_2T_3T_4$. Your job is to write a program to calculate the minimum number of button presses required to transform the initial configuration to the target configuration without passing through a forbidden one.

Playing with Wheels

Input

The first line of the input contains an integer *N* giving the number of test cases. A blank line then follows. The first line of each test case contains the initial configuration of the wheels, specified by four digits. Two consecutive digits are separated by a space. The next line contains the target configuration. The third line contains an integer *n* giving the number of forbidden configurations. Each of the following *n* lines contains a forbidden configuration. There is a blank line between two consecutive input sets.

Output

For each test case in the input print a line containing the minimum number of button presses required. If the target configuration is not reachable print “-1”.

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 utiliza uma busca em largura, pois o grafo é não-ponderado.

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 versus 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;
        }
    }
}
```

```c
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;
}
```

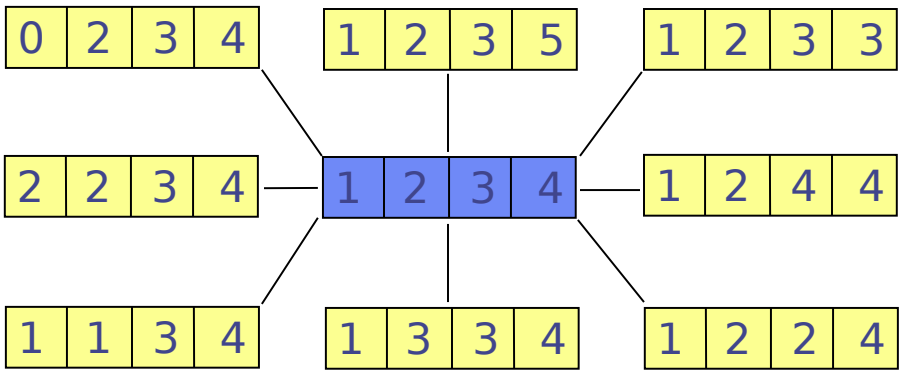
Playing with Wheels

- Suponha que esse problema tenha uma resposta de tempo limite excedido.

- É possível encontrar um solução mais eficiente?
  - Uma possibilidade é incorporar alguma heurística.

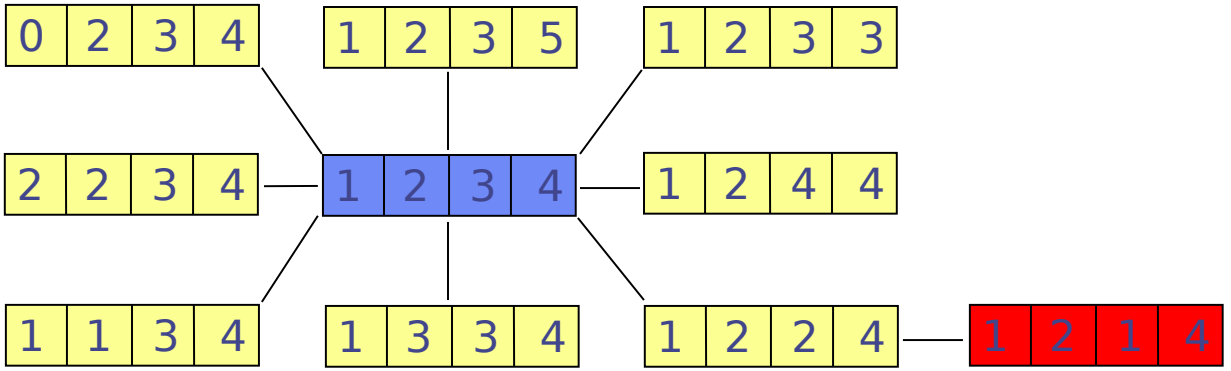
Playing with Wheels: Heurística

- Com a busca em largura, dado um estado (vértice) todos os vértices adjacentes não visitados são inseridos na fila *q*.



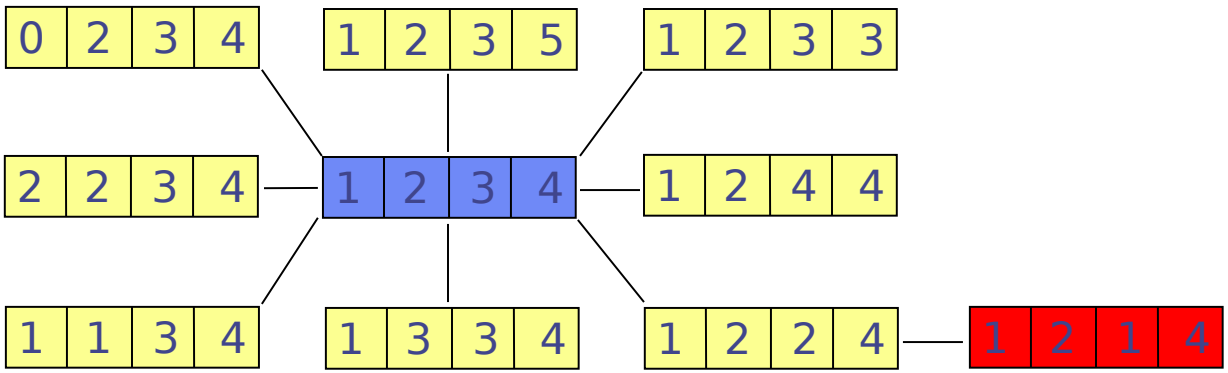
Playing with Wheels: Heurística

- Com a busca em largura, dado um estado (vértice) 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 possiveis vizinhos

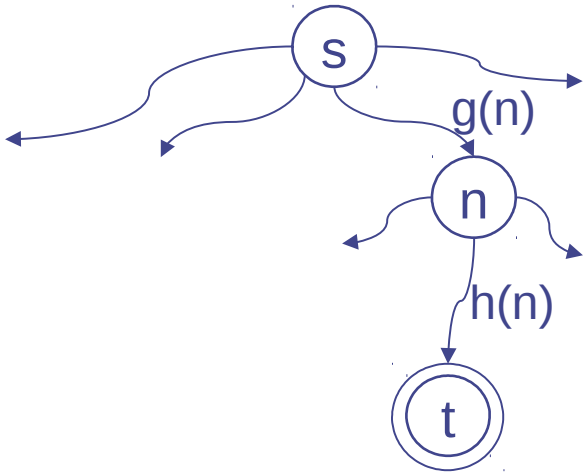


Playing with Wheels: Heurística

- A função heurística pode ser então a distância entre o estado analisado e o estado final:
  - d([1,2,3,4], [1,2,2,4]) = 1
  - d([1,2,3,4], [1,2,1,4]) = 2
- A existência de estados proibidos faz com que a função heurística seja uma estimativa otimista.

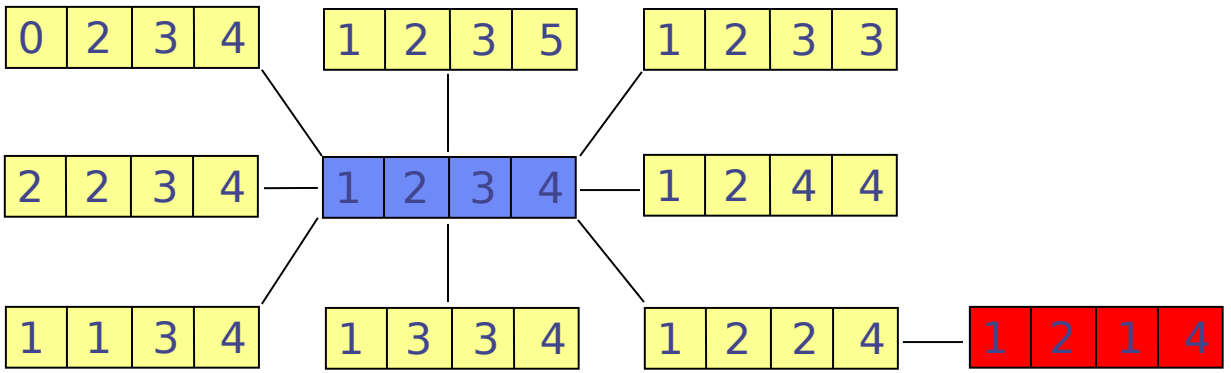
Playing with Wheels: A*

- Uma função heurística admissível nunca superestima a custo real da melhor solução.
- A função heurística fornece uma estimativa do esforço futuro.
- A função de custo fornece o custo do caminho até o momento.



Playing with Wheels: Heurística

- 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])

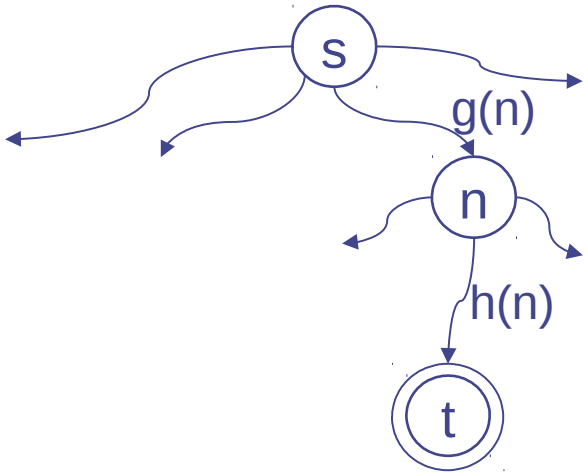


Playing with Wheels: A*

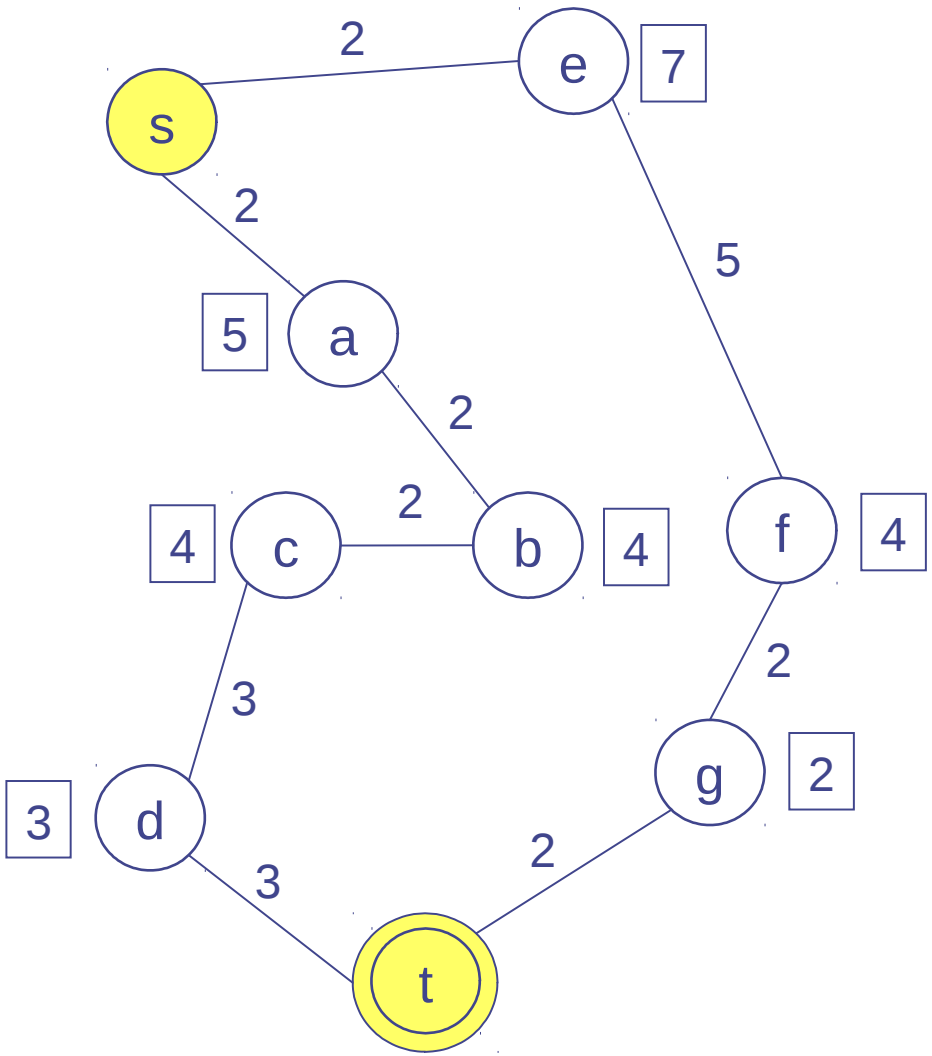
- A busca descrita é chamada de *best-first*.
- Mas essa busca possui uma limitação: ela não garante que o menor caminho será encontrado.
- Uma variação dela, chamada A*, fornece essa garantia.
- A* = *best-first* + função de custo E função heurística **admissível**.

Playing with Wheels: A*

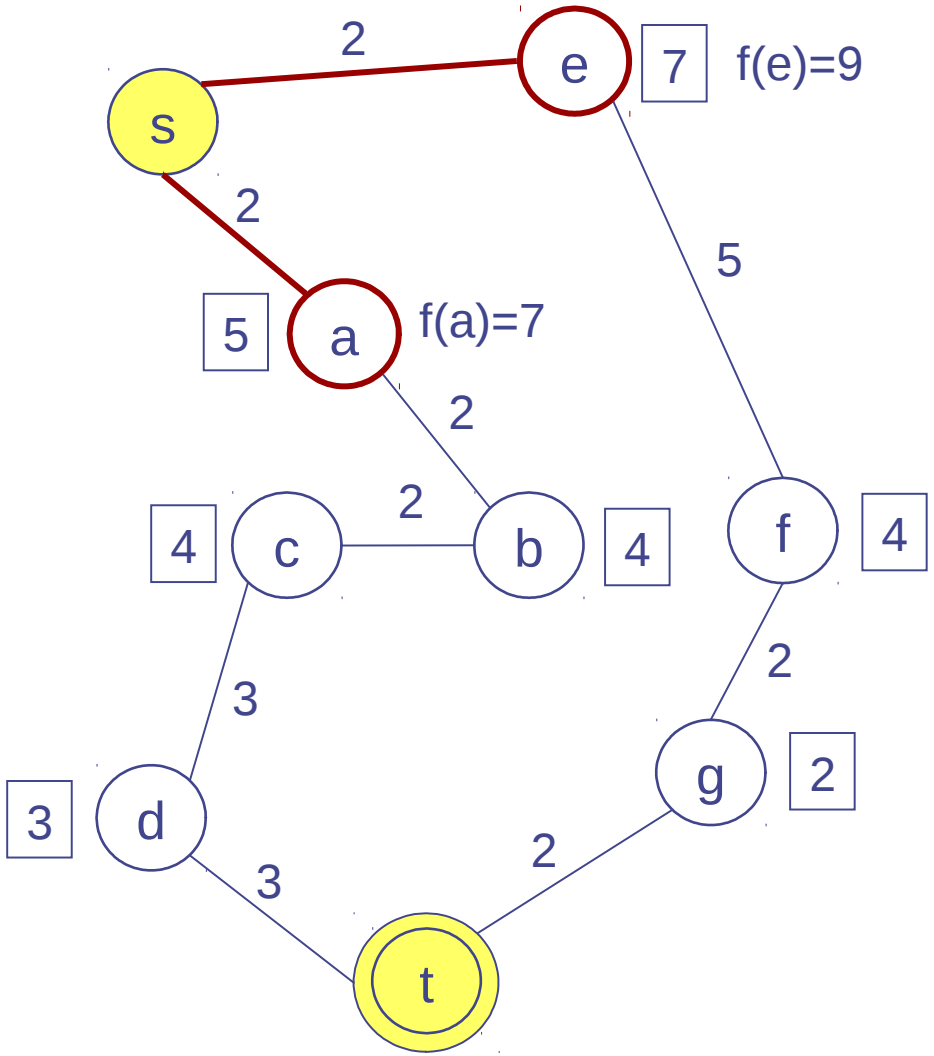
- Uma função heurística admissível nunca superestima a custo real da melhor solução.
- A função heurística fornece uma estimativa do esforço futuro.
- A função de custo fornece o custo do caminho até o momento.



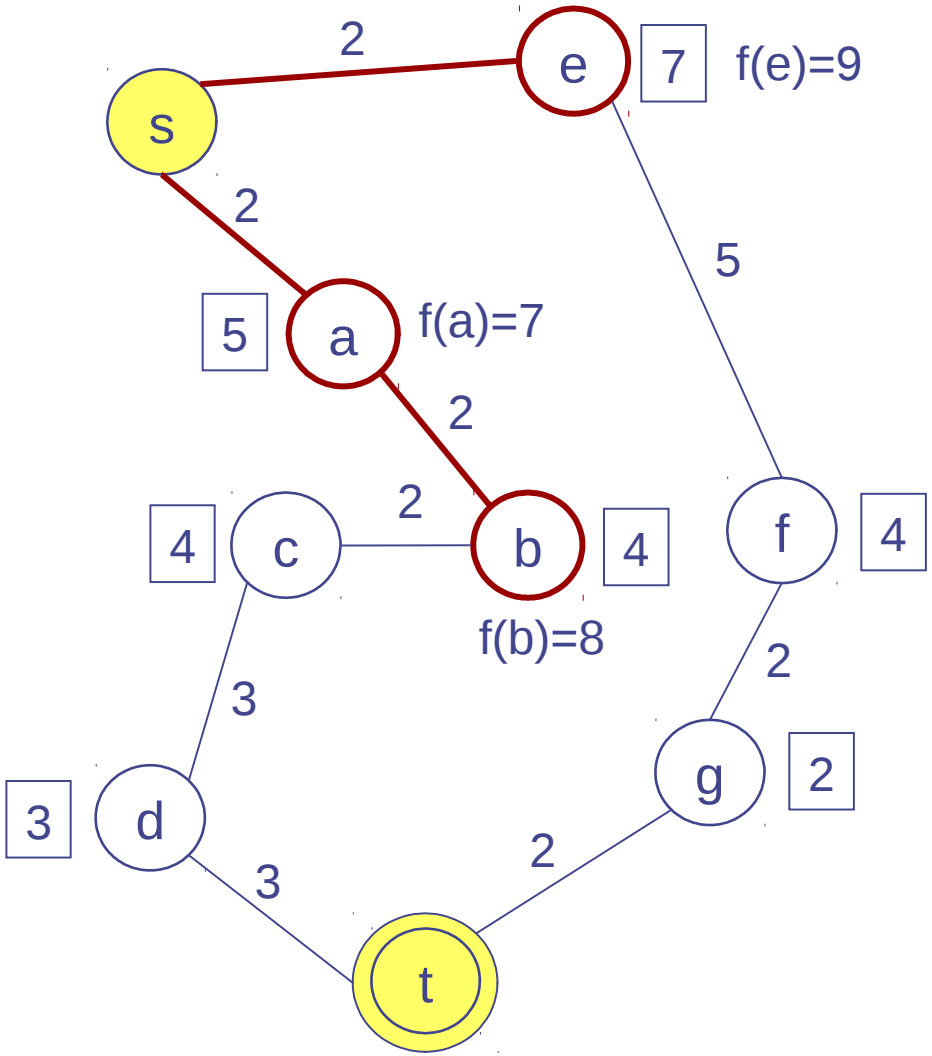
Playing with Wheels: A*



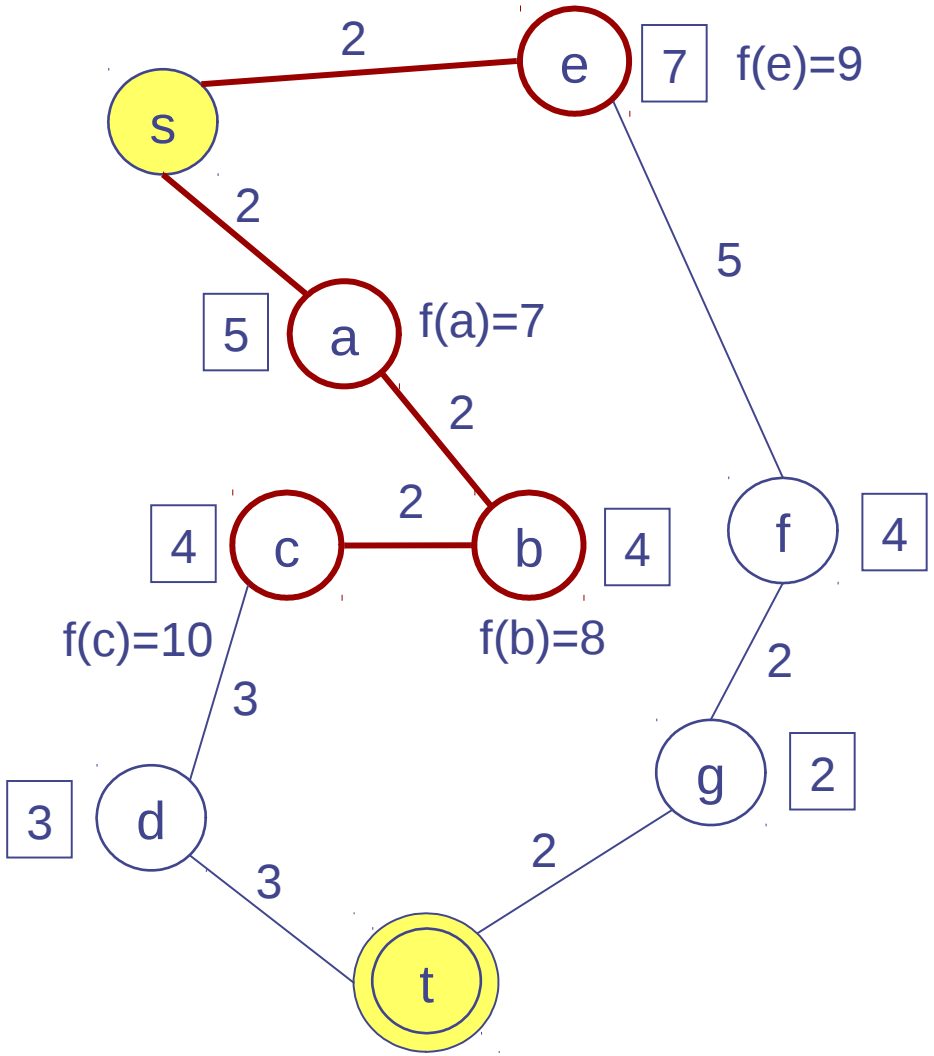
Playing with Wheels: A*



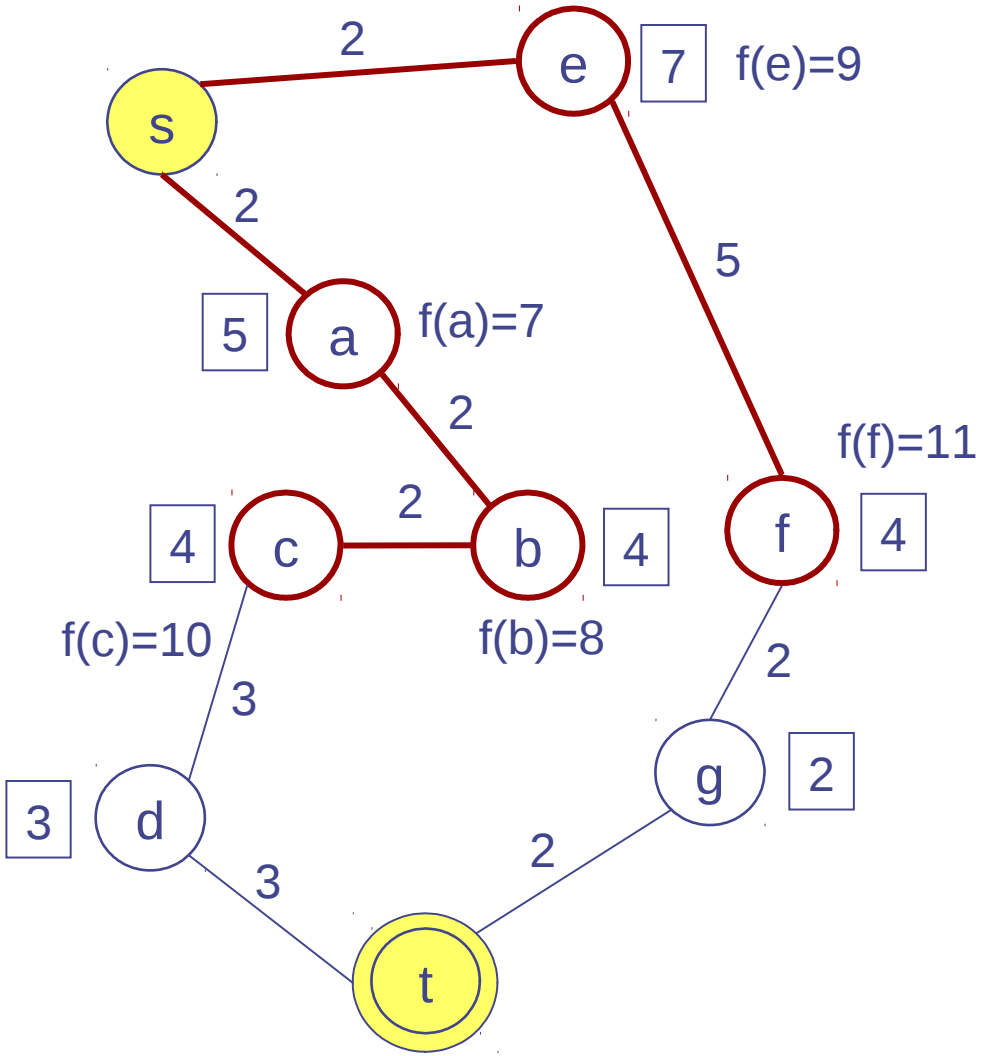
Playing with Wheels: A*



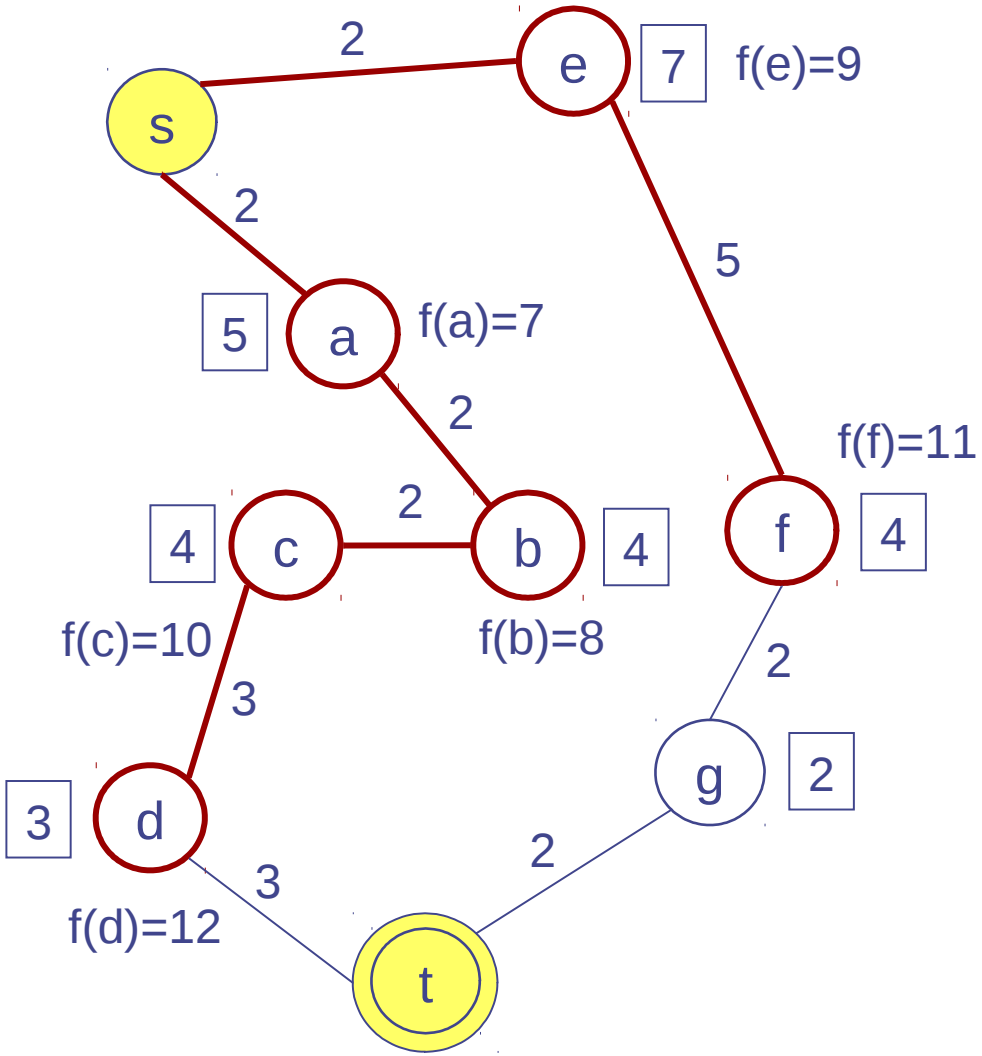
Playing with Wheels: A*



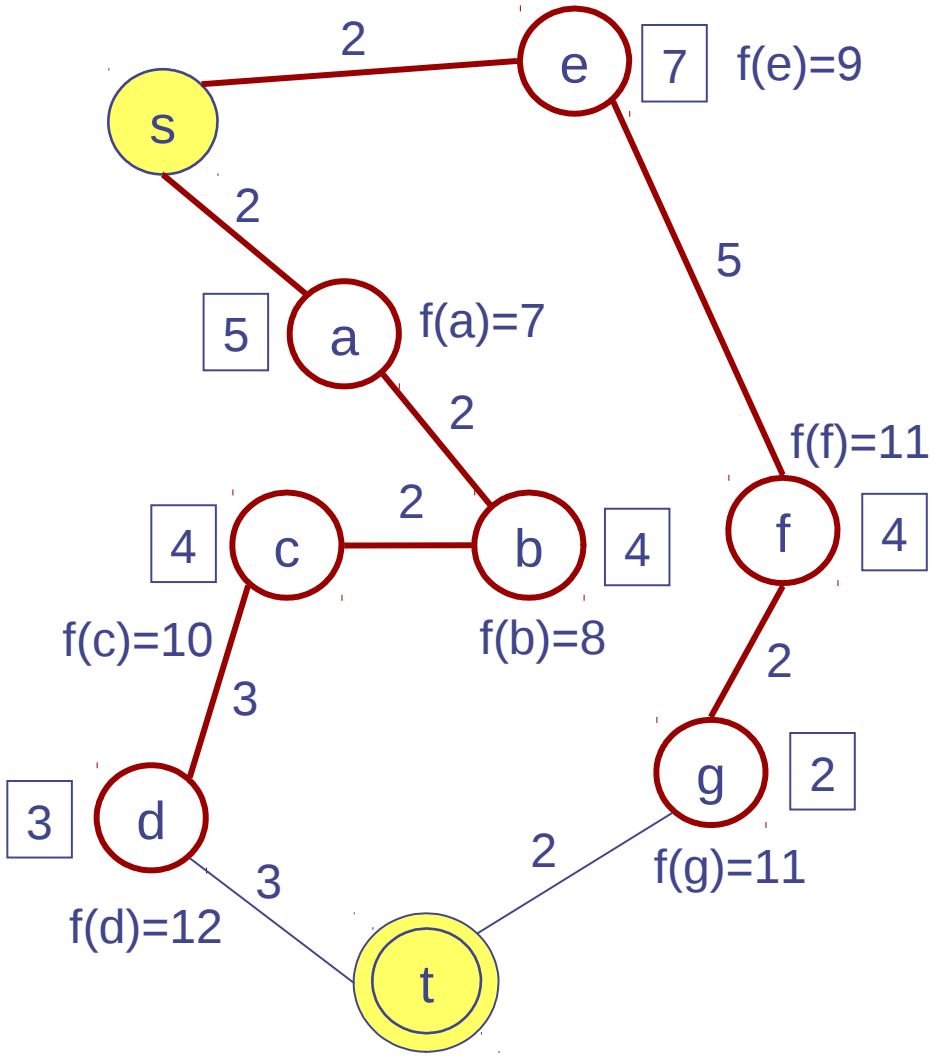
Playing with Wheels: A*



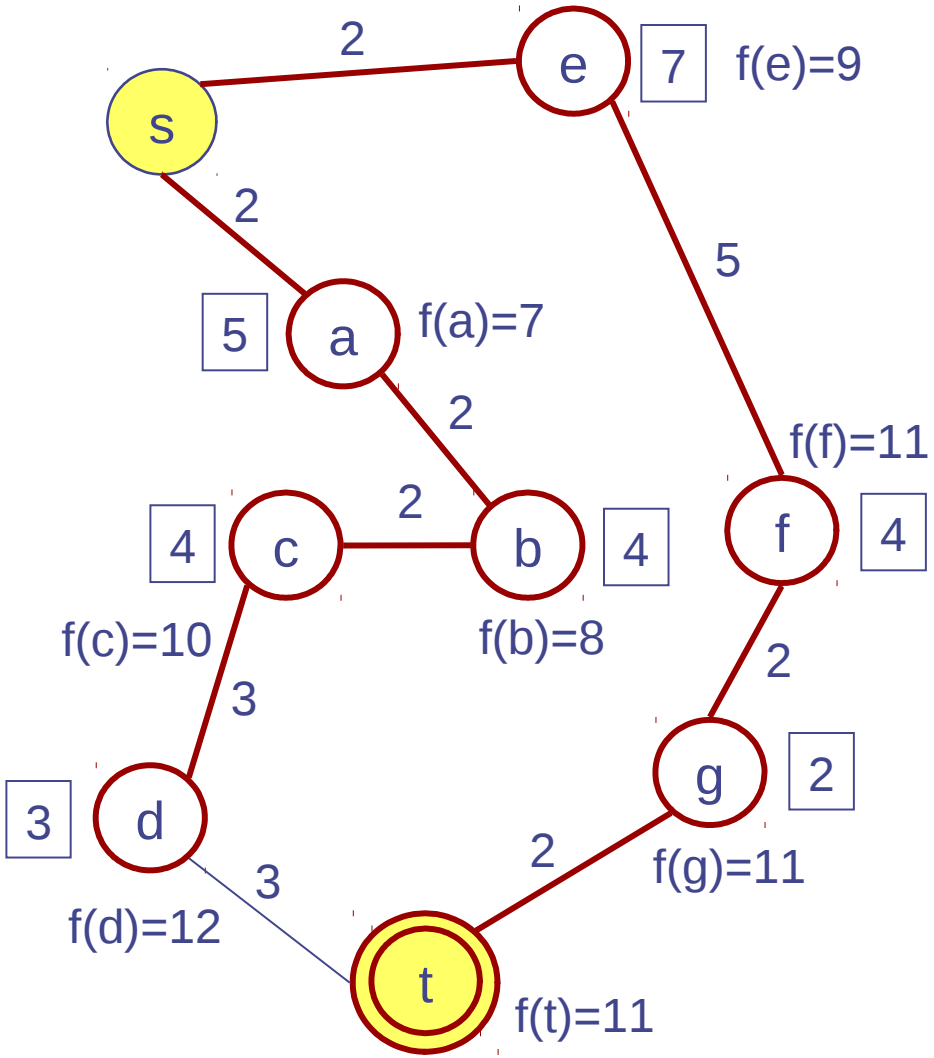
Playing with Wheels: A*



Playing with Wheels: A*



Playing with Wheels: A*



Playing with Wheels: 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.
  - Solução: IDA* e SMA* (Russel e Norvig, 2002).

Playing with Wheels: A*

- 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.

Playing with Wheels: A*

- Dicas para implementacao
- Struct state: acrescentar :
  - uma variável int t.
  - Sobrecarga do operador < para definir a entrada na fila de prioridade
- Crie as funcoes custo e heuristica:
  - Combine tudo numa única função
    - distancia manhattan entre os dois estados + a profundidade na árvore do estado de corrente!
- Apenas uma única linha (chamada da funcao heuristica_custo e atulizacao de t) será acrescentada na funçao bfs !