Lecture 3

- Relaxing the Optimality Condition of the A* Algorithm
- AO* Algorithm

Relaxing the Optimality Condition of the A* Algorithm

- The A* algorithm finds the optimal solution if the heuristic component h is admissible
- In many cases, the A* algorithm uses a lot of time when trying to choose between different paths with almost the same cost
- The admissibility properties can become sometimes a limit and can increase the problem solving time

Relaxing the Optimality Condition of the A* Algorithm

- Depending on the problem requirements, the admissibility property of the A* algorithm can be relaxed and the solution will be suboptimal, but the search time will be decreased
- There are three possible situations:
  - 1. Minimizing the search effort
  - 2. Finding a solution close to the minimal cost solution
  - 3. Using an ε-admissible function

Minimizing the Search Effort

- There are problems for which it is less important to obtain a minimal cost solution and the goal is to **minimize the search effort**
- The reason for including the function g in the evaluation function f is to add in the search process a search component on each level and to guide the search for discovering the optimal solution

Minimizing the Search Effort

- Without the function g, the function f(S) will always estimates, for each node S, the remaining distance up to the final state
- If the goal is to minimize the search effort and not the solution cost, then the weight of the function h must be as high as possible
- In order to adjust the weights between the optimal cost and the quick advance towards the solution it is possible to use a weighted definition of the function f

Minimizing the Search Effort

- $f(S) = (1 - p)*g(S) + p*h(S)$
- where p is a positive constant
- If $p = 0$, the search algorithm becomes a uniform cost search strategy
- If $p = ½$, the standard version of the A* algorithm is obtained
- If $p = 1$, the best-first search is obtained, which minimizes the search effort

Minimizing the Search Effort

- If h is admissible, then the algorithm is admissible in the range $p \in [0 , ½]$, but it can loose the admissibility for the domain $p \in (1/2, 1)$, depending on the distance of the function h with respect to h*

Finding a Solution Close to the Minimal Cost Solution

- There are problems for which it is necessary to obtain the minimal cost solution, but the problem is so hard, such that an admissible A* algorithm is impossible to be executed up to the end, due to efficiency criteria
- In such a situation it is useful to find a **solution close to the minimal cost solution** in a reasonable time
- The function f can be defined by a dynamic weight of the component h

Finding a Solution Close to the Minimal Cost Solution

- $f(S) = g(S) + c(S) * h(S)$
- where $c(S)$ is a weighted function, which depends on the node S
- A possibility to define such a function is:
- $f(S) = g(S) + h(S) + \varepsilon * (1 - d(S)/N) * h(S)$
- where $d(S)$ is the depth of the node associated to the state S and N is the estimated depth of the final state node

Finding a Solution Close to the Minimal Cost Solution

- If the function h is admissible, then the A* algorithm which uses the previous definition of the function f will find a suboptimal solution, with the cost different with almost ε from the optimal solution cost

Using an ε-Admissible Function

- There are problems for which the determination of a good enough admissible heuristic function h, that is close enough to the real function h*, is very difficult or even impossible
- An admissible function h with much lower values than the values of the function h* makes the A* algorithm to degenerate in an uninformed search strategy

Using an ε-Admissible Function

- If it is not possible to find a good enough heuristic function h, an ε-admissible function can be used
- An heristic function h is called **ε-admissible** if **$h(S) \leq h^*(S) + \varepsilon$**
- where $\varepsilon > 0$ is a constant

Using an ε-Admissible Function

- The A* algorithm which uses an evaluation function f with an ε-admissible function h always finds a solution with a higher cost with almost ε than the optimal solution cost
- Such an algorithm is called **ε-admissible A* algorithm** and the solution found is called **ε-optimal solution**

The AO* Algorithm

- Obtaining the optimal solution in the problem decomposition in subproblems can be done with a similar algorithm to the A* algorithm
- The difference between the two algorithms consists in the problem solution nature, that is the presence of AND nodes, which shows a set of subproblems which must be solved

The AO* Algorithm

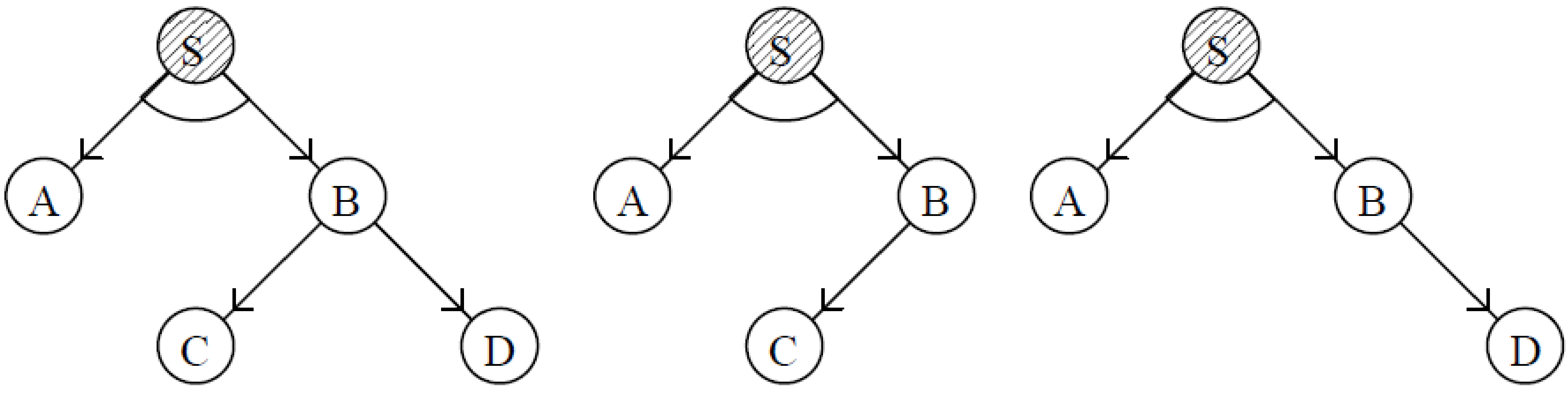
- The specific aspects which must be taken into account in the case of an AND/OR solution tree are:
  - 1. How can be used the heuristic information to search for the optimal solution
  - 2. How an optimal solution is defined

The AO* Algorithm

- When executing a search algorithm in the state space, there is a one-to-one correspondence between the candidate nodes for expansion and the partial solutions built
- When finding the solution in an AND/OR graph, this one-to-one correspondence between the node choosen for expansion and the potential solution to be extended is not kept anymore

The AO* Algorithm

- Each partial solution can contain many candidate nodes for expansion and a given node can be part of many potential solution trees
- Expanding the AND node S means generating two potential solution trees



The AO* Algorithm

- In these conditions, the heuristic information can be used in two steps of the search
- First step – the **most promising solution** is identified, using a **graph evaluation function f**
- Second step – from this partial solution is selected the **next node to be expanded**, based on a **node evaluation function f_n**

The AO* Algorithm

- These two functions, with different roles, offer two estimation types:
  - 1. f estimates the properties of the solution trees which can be generated from a current candidate tree
  - 2. f_n estimates the information quantity which can be offered by the expansion of a node about the importance of the graph which contains that node

The AO* Algorithm

- The function f establishes the optimality of the solution, based on associated cost of the problem decomposition into subproblemes process
- The most promising solution tree can be determined based on the cost associated to the trees generated during search
- The cost of an AND/OR solution tree can be defined in two ways: the sum cost and the maximum cost, based on the costs associated to the edges of the graph

Definitions

- The **sum cost** of a solution tree is the sum of the costs of all the edges from the tree
- The **maximum cost** of a solution tree is the sum of the costs on the most expensive path between the root and a terminal node
- If each edge of the solution tree has a unit cost, then the sum cost is the number of edges in the tree and the maximum cost is the depth of the farthest node from the root

Definitions

- The **cost of an optimal solution tree**, denoted **c**, in an AND/OR graph search space is computed by:
  - 1. If S is a terminal node labelled with an elementary problem, then c(S) = 0
  - 2. If S is an OR node with the successors S_1, S_2, …, S_k then:

$$c(S) = \min_{j=1,k} \left(cost\_arc(S, S_j) + c(S_j)\right)$$

Definitions

- 3. If S is an AND node with the succesors S_1, S_2, …, S_m and the **sum cost** is used, then

$$c(S) = \sum_{j=1}^{m} (cost\_arc(S, S_j) + c(S_j))$$

- 4. If S is an AND node with the succesors S_1, S_2, …, S_m and the **maximum cost** is used, then

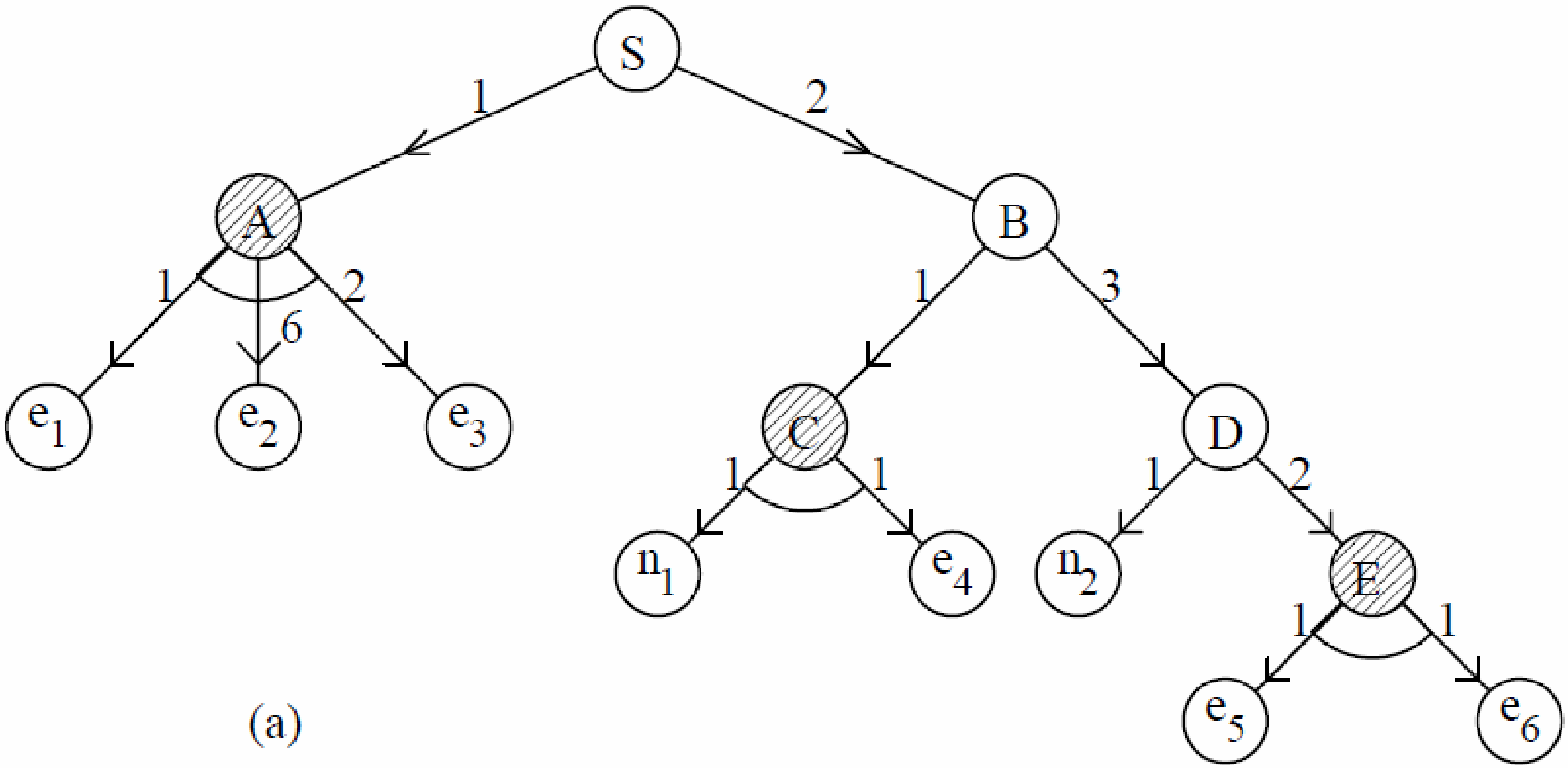
$$c(S) = \max_{j=1,m} (cost\_arc(S, S_j) + c(S_j))$$

- 5. If S is a terminal node labelled with a non-elementary problem, than $c(S) = \infty$

Example

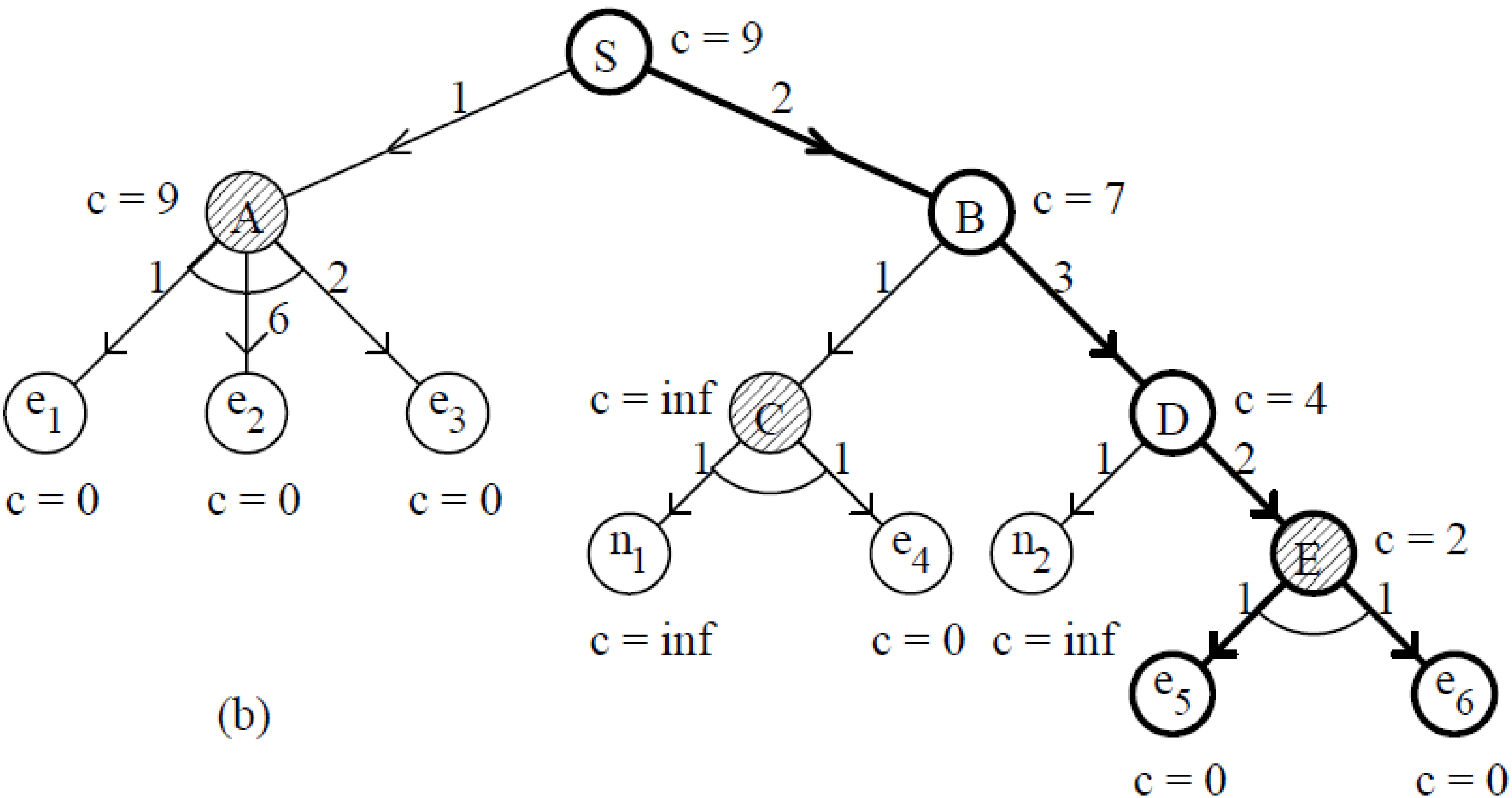
- Consider the AND/OR tree in which e_i denotes terminal nodes labelled with elementary problems and n_i terminal nodes labelled with non elementary problems
- The terminal nodes e_1, e_2, e_3, e_4, e_5 and e_6 have an associated zero cost, because they correspond to elementary problems
- The terminal nodes n_1 and n_2 have an associated infinite cost, because they correspond to non-elementary problems

AND/OR Tree



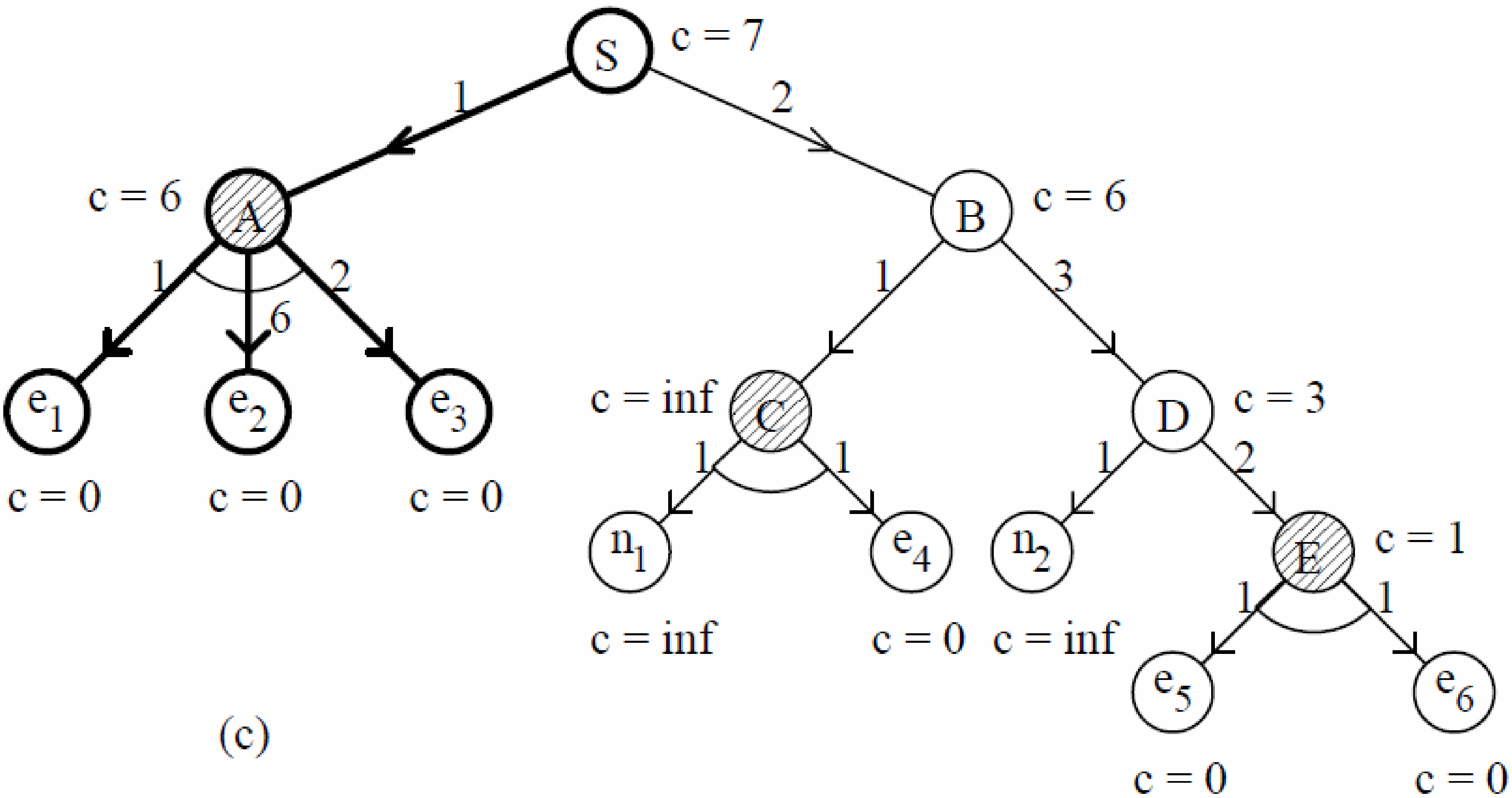
(a)

The Sum Cost



S
c = 9
1
2
A
c = 9
B
c = 7
1
6
2
1
3
e_1
e_2
e_3
c = 0
c = 0
c = 0
C
c = inf
D
c = 4
1
1
1
2
n_1
e_4
n_2
E
c = 2
c = inf
c = 0
c = inf
1
1
e_5
e_6
c = 0
c = 0
(b)

The Maximum Cost



(c)

Observations

- If the sum cost is used, the optimal solution tree is formed by the nodes S, B, D, E, e_5 and e_6
- If the maximum cost is used, the optimal solution tree is formed by the nodes S, A, e_1, e_2 and e_3
- The function c(S) associated to an optimal solution tree is the real cost, similarly to the function f*(S) from the state space informed search

Definition

- The **most promising solution tree** T in a weighted AND/OR graph is defined by:
  - 1. The initial problem node S_i is in T
  - 2. If the AND/OR search tree contains an AND node, then all the successors of the node are in T
  - 3. If the AND/OR search tree contains an OR node with the successors S_1, S_2,…, S_k then the node S_j, j = 1, k for which the sum cost_arc(S,S_j) +f(S_j) is minimum belongs to T

Observations

- In the search algorithm, the cost of the most promising solution tree $f(S_i)$ is computed from leaves towards the root
- Because at a certain moment the tree is partially built, the function $f(S_j)$ must estimate heuristically the cost of still unexpanded nodes S_j
- When such a node is expanded, a reevaluation of the total cost of the tree $f(S_i)$ is performed, based on the new cost obtained for the node S_j

Definition

- The AO* algorithm is **admissible**, so it finds the optimal solution tree if:
  - 1. $f(S) \le c(S)$ for any node S
  - 2. $cost(S_k, S_{k+1}) > 0$ and is finite, for any nodes S_k, S_{k+1} with S_{k+1} the direct successor of S_k