Bloque 1 – Representación de Conocimiento y Búsqueda

Tema 3:
Búsqueda heurística con memoria limitada

Bloque 1, Tema 3

1. Métodos A* con memoria limitada
2. Algoritmo A* de Profundización Iterativa (IDA*)
3. Algoritmo Recursive-Best First Search (RBFS)
4. Algoritmo Simplified Memory-bounded A* (SMA*)

Bibliografía

- S. Russell, P. Norvig. ***Artificial Intelligence. A modern approach.*** Prentice Hall, 3rd edición, 2010 (Capítulo 3) http://aima.cs.berkeley.edu/
- S. Russell, P. Norvig. ***Inteligencia artificial . Una aproximación moderna.*** Prentice Hall, 2ª edición, 2004 (Capítulos 3 y 4) http://aima.cs.berkeley.edu/2nd-ed/

1. Métodos A* con memoria limitada

Objetivo: reducir el alto coste de memoria de la búsqueda A* que necesita mantener todos los nodos generados en memoria

Algoritmos:

- Algoritmo A* de profundidad iterativa
  Iterative Deepening A (IDA*)*
- Búsqueda Recursiva de primero el mejor
  Recursive best-first search (RBFS)
- Algoritmo A* con Memoria Acotada Simplificada
  Simplified Memory-bounded A (SMA*)*

2. Algoritmo A* de profundización iterativa (IDA*)

IDA* proporciona los beneficios de A* sin necesidad de mantener todos los nodos en memoria aunque necesite visitar algunos estados varias veces.

Características de IDA*:

- El valor del límite es el valor-f o coste-f (g+h) en lugar del nivel de profundidad como en ID.
- Utiliza la información de valor-f para determinar el límite de la búsqueda en la siguiente iteración; es decir, utiliza valor-f para determinar los nodos que hay que explorar y el límite de profundidad en el que hay que parar.
- En cada iteración i, el límite es el valor-f más pequeño de cualquier nodo que haya excedido el límite de coste-f de la iteración anterior i-1.

2. Algoritmo IDA*

```
1.  lim-f = f(nodo_raiz)
2.  while true      ;; diferentes iteraciones
3.          OPEN ={nodo_raíz}; lim-f-sig = ∞; PATH={}     ;; comienza una iteración
4.          while true      ;; recorrido en profundidad
5.                  if OPEN vacía
                        then if lim-f-sig = ∞
                                    then return FALLO
                                    else   lim-f = lim-f-sig
                                           break     ;; volvemos al punto 2
                        else  n = pop (OPEN); insert n in PATH      ;; expandimos el nodo n
6.                  if n es objetivo
                        then return ÉXITO
7.                  for each hijo n' of n:
                        if f(n') <= lim-f
                              then    insertar n' en OPEN   ;; en PROFUNDIDAD
                              else   lim-f-sig = min(lim-f-sig, f(n'))  ;;nodo no válido (se ignora)
8.                  if n no tiene hijos insertados en OPEN
                        then Backtracking (n)    ;; Backtracking de  PROFUNDIDAD
```

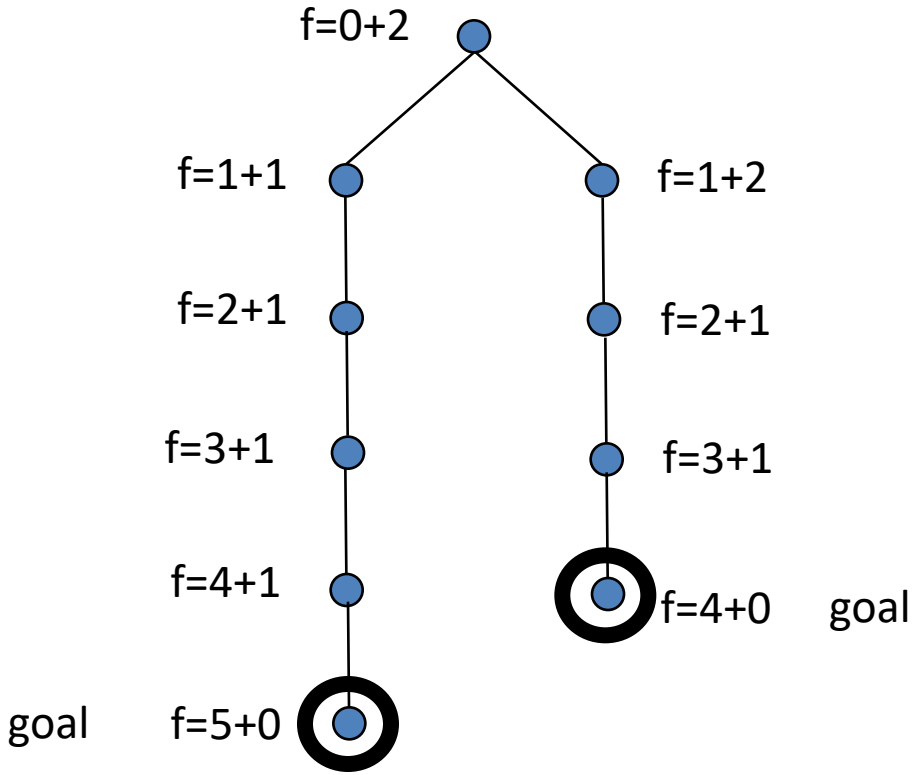
2. Algoritmo IDA*

Claves del algoritmo IDA*:

- Se realiza una serie de búsquedas siguiendo el orden de expansión en PROFUNDIDAD. Cada iteración del algoritmo se ejecuta hasta donde permita el límite de valor-f de la iteración anterior (lim-f). Cuando el valor f(n) de un nodo n supera el valor lim-f de la iteración, el nodo se ignora y se pasa al siguiente nodo según criterio de Profundidad con Backtracking.

- En cada iteración de búsqueda en PROFUNDIDAD, el valor de lim-f será igual al menor valor de los nodos cuyo valor-f fue superior al lim-f de la iteración anterior; es decir, el menor f(n) de los nodos que se ignoraron en la iteración anterior.

2. Algoritmo IDA*: ejemplo 1

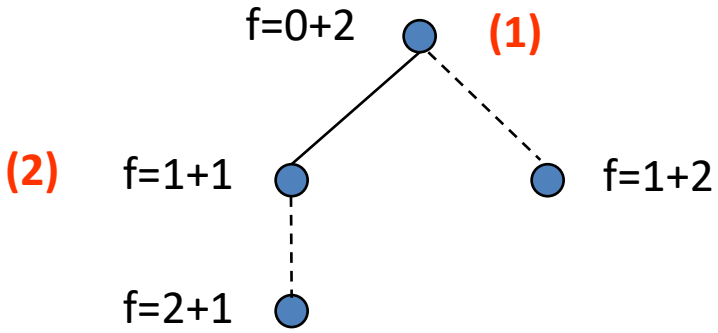


2. Algoritmo IDA*: ejemplo 1

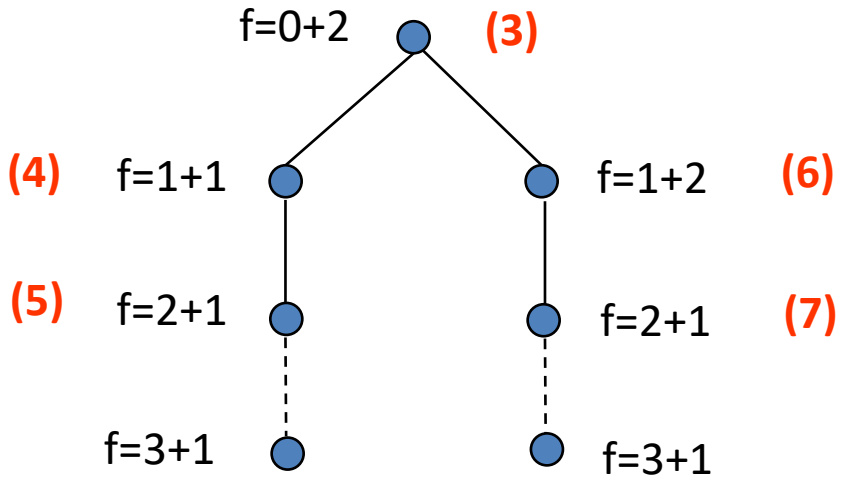
(x): orden de expansion o extracción de OPEN

lim-f= f(nodo_raiz)=2

lim-f=2
lim-f-sig=∞
lim-f-sig=3

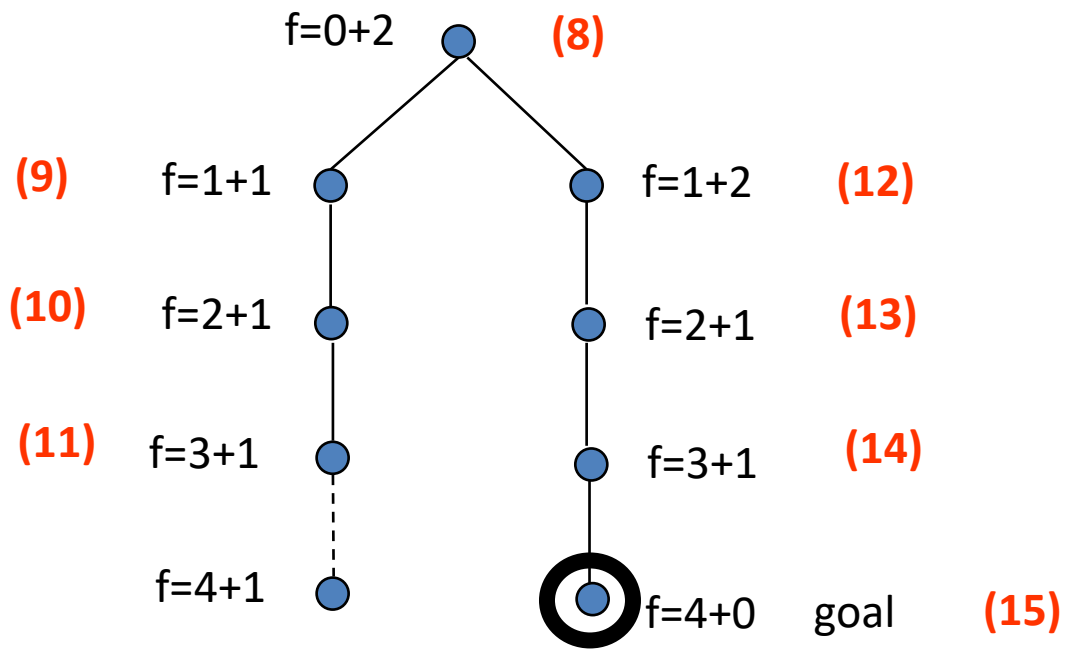


lim-f=3
lim-f-sig=∞
lim-f-sig=4



2. Algoritmo IDA*: ejemplo 1

lim-f=4
lim-f-sig=∞
lim-f-sig=5



*A** necesitaría 9 nodos en memoria (OPEN + CLOSED)
*IDA** necesitaría 5 nodos en memoria (OPEN + PATH)

2. Algoritmo IDA*: complejidad

- Optimalidad:
  - IDA* encuentra la solución óptima bajo las mismas condiciones que A* ($\forall n$, $\mathbf{h(n) \leq h^*(n)}$). IDA* expande nodos con coste creciente, por tanto, la primera solución es la de menor coste.
- Complejidad espacial:
  - Lineal $O(b \cdot d)$ como profundidad y Profundización Iterativa (ID)
  - Poca memoria: solo nodos del camino actual y los valores *lim-f* y *lim-f-sig*
- Complejidad temporal:
  - La complejidad temporal asintótica de IDA* es la misma que A* ($O(b^d)$)
  - Pero IDA* genera muchos más nodos que A*
    - En el peor de los casos la última iteración de IDA* expande los mismos nodos que A* (k nodos)
    - Si los k nodos tienen distinto valor de coste-f, se visita un nuevo nodo en cada iteración: $1 + 2 + 3 + \ldots + k\text{-}1 + k = k(k\text{-}1)/2 \rightarrow O(k^2) \equiv O(b^{2d})$ (mucho más lento que A*)
    - En un problema donde cada nodo tiene un valor de coste-f diferente, cada nueva iteración podría contener solo un nuevo nodo, y el número de iteraciones sería igual al número de estados.
    - En general, IDA* funciona bien cuando hay pocos valores distintos de coste-f.

2. Algoritmo IDA*: propiedades

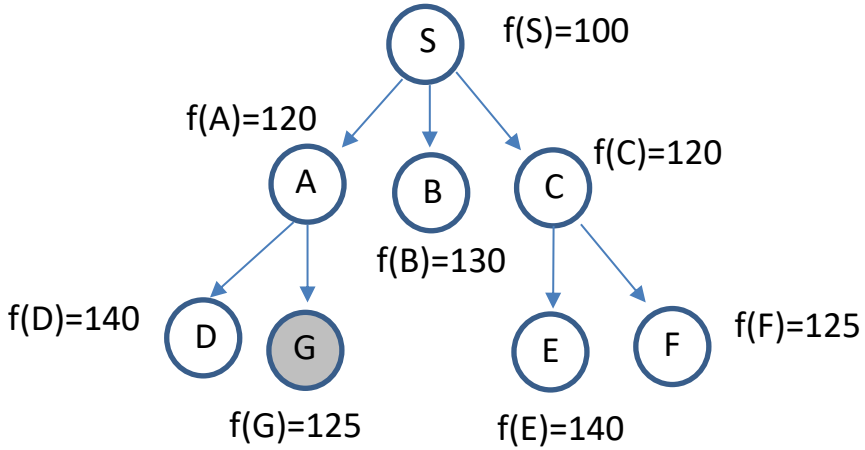
Ventajas:

- Uso limitado de memoria comparado con A* porque usa Profundidad en la expansión
- Evita el extra-coste de mantener una lista de nodos ordenados
- Expande solo los nodos expandidos por A*
- Encuentra solución óptima si se usa una heurística admisible
- Algoritmo práctico para muchos problemas con costes unitarios (como el 8-puzzle) porque típicamente coste-f se incrementa solo un número pequeño de veces.

Desventajas:

- No comprueba estados repetidos, solo la ruta actual, por lo que puede explorar el mismo estado múltiples veces.
- Si todos los valores $f(n)$ de los nodos del espacio de búsqueda son distintos, cada iteración podría contener solo un nodo nuevo. En este caso, el número de iteraciones sería igual al número de nodos con distintos valores $f(n)$ menores que el coste del camino óptimo → proceso lento.

2. Algoritmo IDA*: ejercicio a resolver



1) ¿Cuántas iteraciones necesita IDA* para encontrar la solución?
2) Asumiendo que a igualdad de criterio de profundidad se expande el nodo más a la izquierda, ¿cuál es el máximo número de nodos que IDA* necesita almacenar en memoria?

3. Algoritmo Recursive Best First Search (RBFS)

RBFS expande nodos siguiendo el criterio de menor valor-f de los nodos (como A*) pero la utilización de memoria es lineal, es decir, mantiene una rama del árbol en memoria.

Funciona de un modo similar a Profundidad (TREE-SEARCH) con *backtracking* pero en lugar de aplicar *backtracking* cronológico, recuerda el valor-f del mejor camino alternativo disponible desde cualquier nodo antepasado del nodo actual.

Si el valor-f del nodo actual excede el valor-f del mejor camino alternativo, el proceso recursivo *olvida el sub-árbol actual* y vuelve al camino alternativo (backtracking).

Cuando se aplica backtracking, RBFS reemplaza el valor-f de cada nodo del camino que se abandona con el mejor $f(n)$ de sus hijos. De este modo, RBFS recuerda el valor-f del mejor nodo hoja del sub-árbol podado y puede decidir más adelante si volver a dicho sub-arbol.

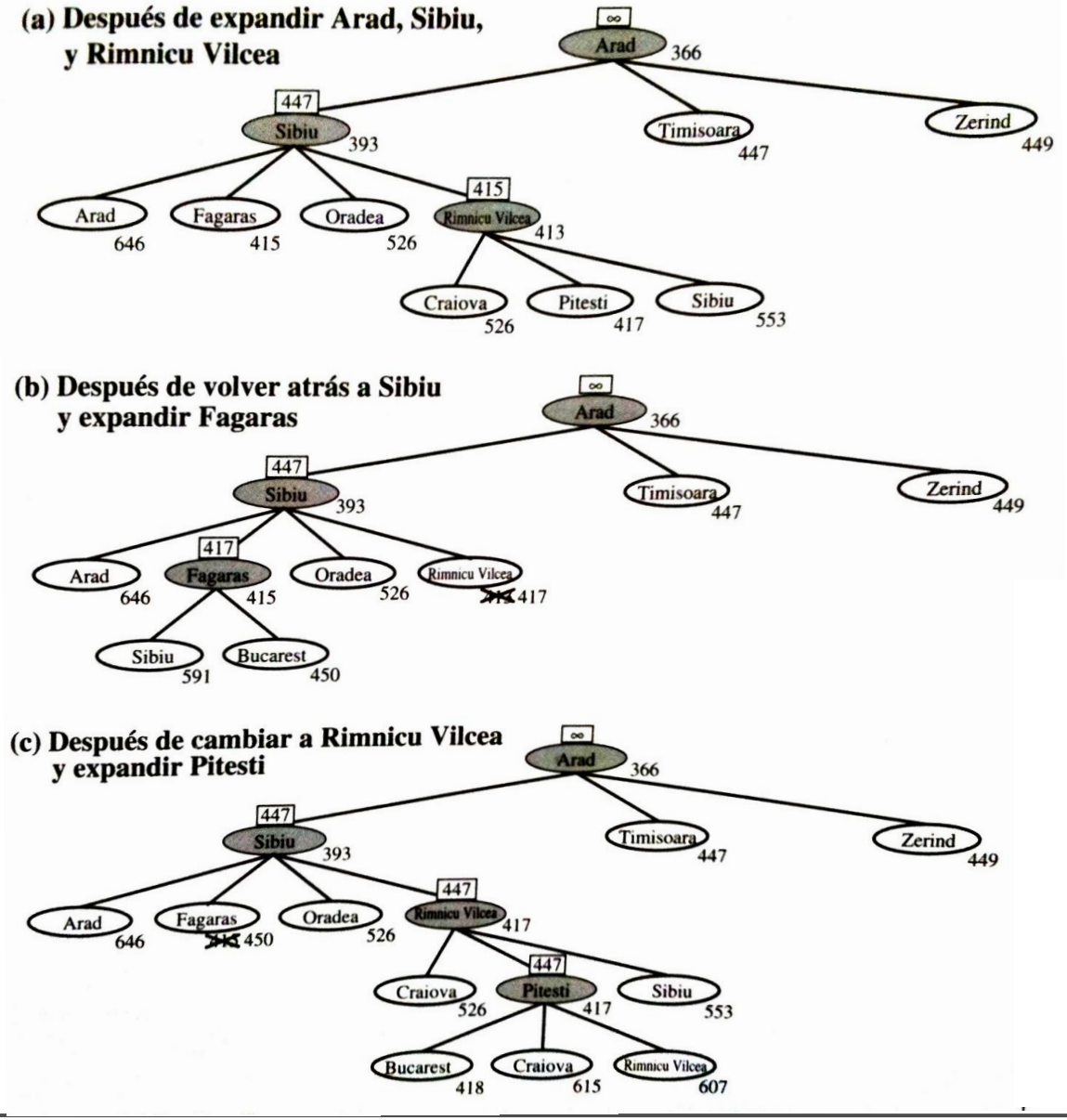
3. Algoritmo Recursive Best First Search (RBFS)

Ejemplo búsqueda RBFS aplicado al problema de obtener el mejor camino de Arad a Bucarest

b es una cota, el valor-f del mejor camino alternativo

Dado un nodo **n** y una cota **b**, RBFS explora el subárbol con raíz **n** mientras existan nodos con valor-f <= **b**.

RBFS actualiza el valor-f de un nodo al menor valor-f entre los nodos hijos que han excedido la cota **b**.



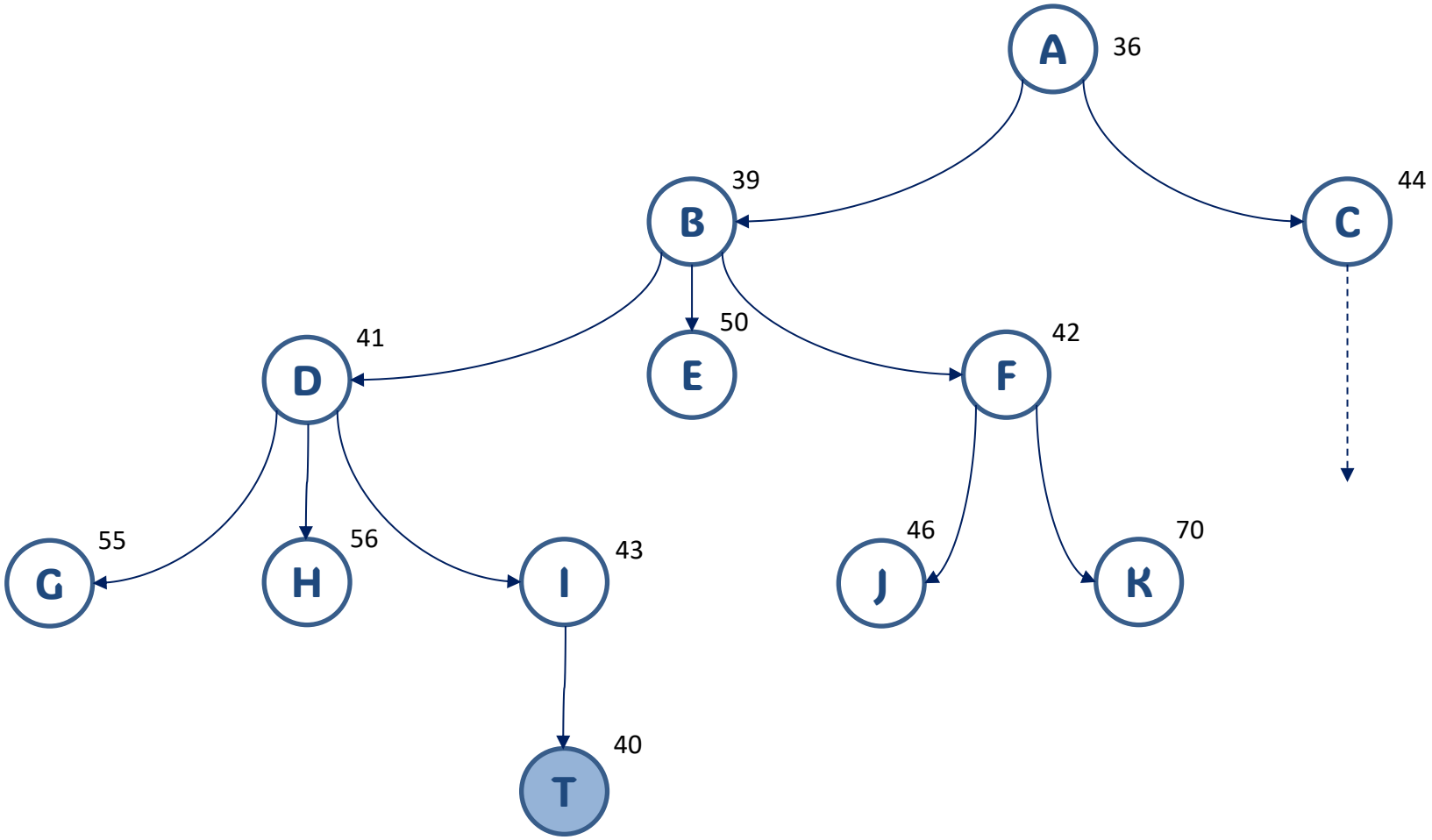
3. Algoritmo Recursive Best First Search (RBFS)

Cuando se expande un nodo 'n', se calcula su valor de la cota, b(n). Solo los nodos expandidos del camino actual (lista PATH en búsqueda en profundidad) tienen asociado el valor b(n).

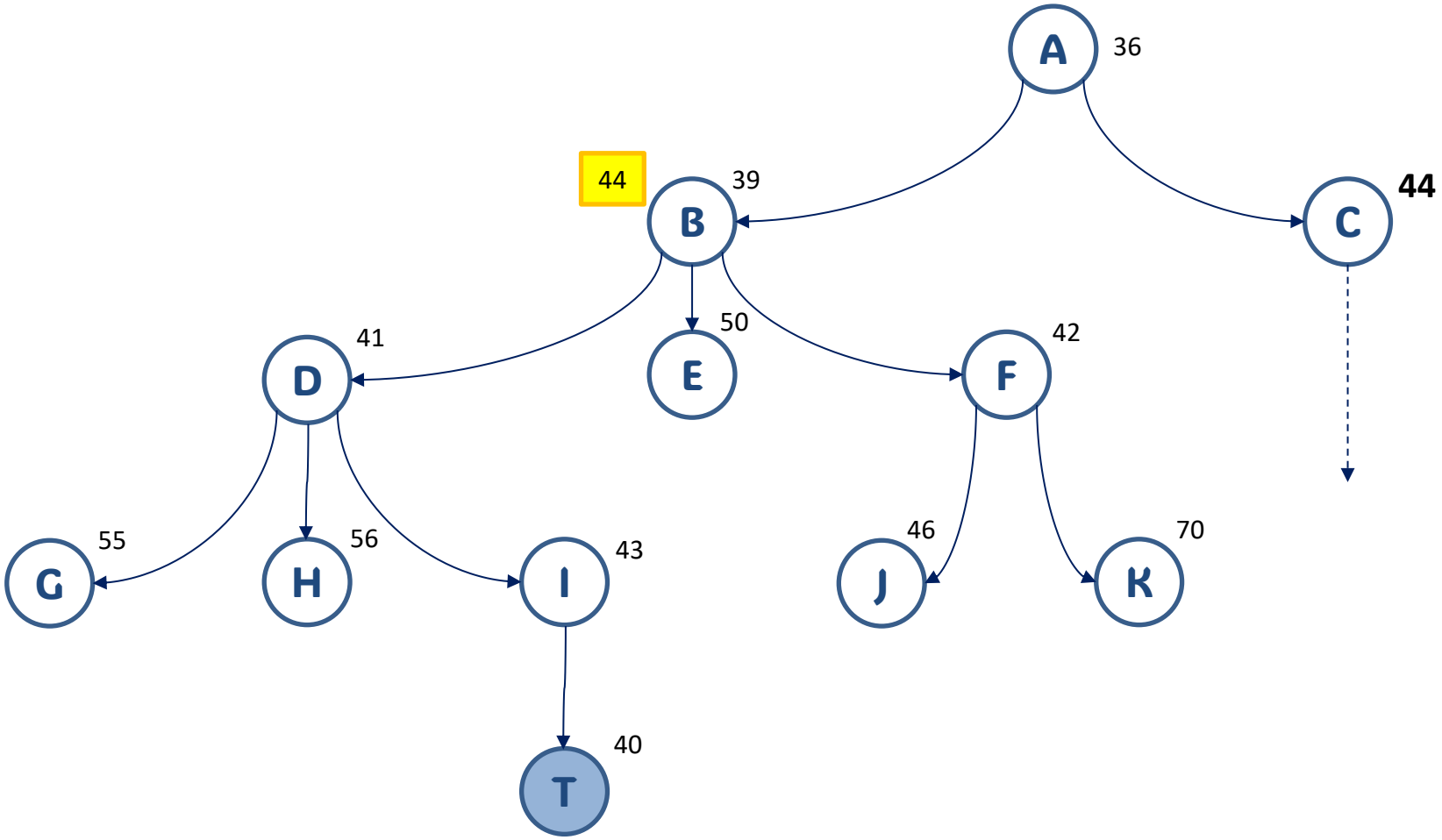
Pseudo-algoritmo para la expansión del algoritmo RBFS cuyo comportamiento es equivalente al algoritmo recursivo.

1. if n = objetivo then return ÉXITO *;; n es el nodo que se va a expandir*
2. if n = nodo_raíz
   then b(n) = ∞
   else b(n) = valor-f del mejor camino alternativo entre todos los nodos abiertos
3. if n no tiene hijos
   then f(n) = ∞
   eliminar de memoria el nodo n
   if n = nodo_raíz then return FALLO
   n = padre (n)
   go to paso 1
4. if ∀hijo of n, f(hijo) > b(n)
   then actualizar f(n) al mínimo del valor-f de los hijos *;; finalización de la recursión*
   eliminar de memoria los hijos del nodo n
   if n = nodo_raíz then return FALLO
   n = padre(n)
   go to paso 4
   else n = nodo hijo con menor valor-f
   go to paso 1 *;; llamada recursiva*

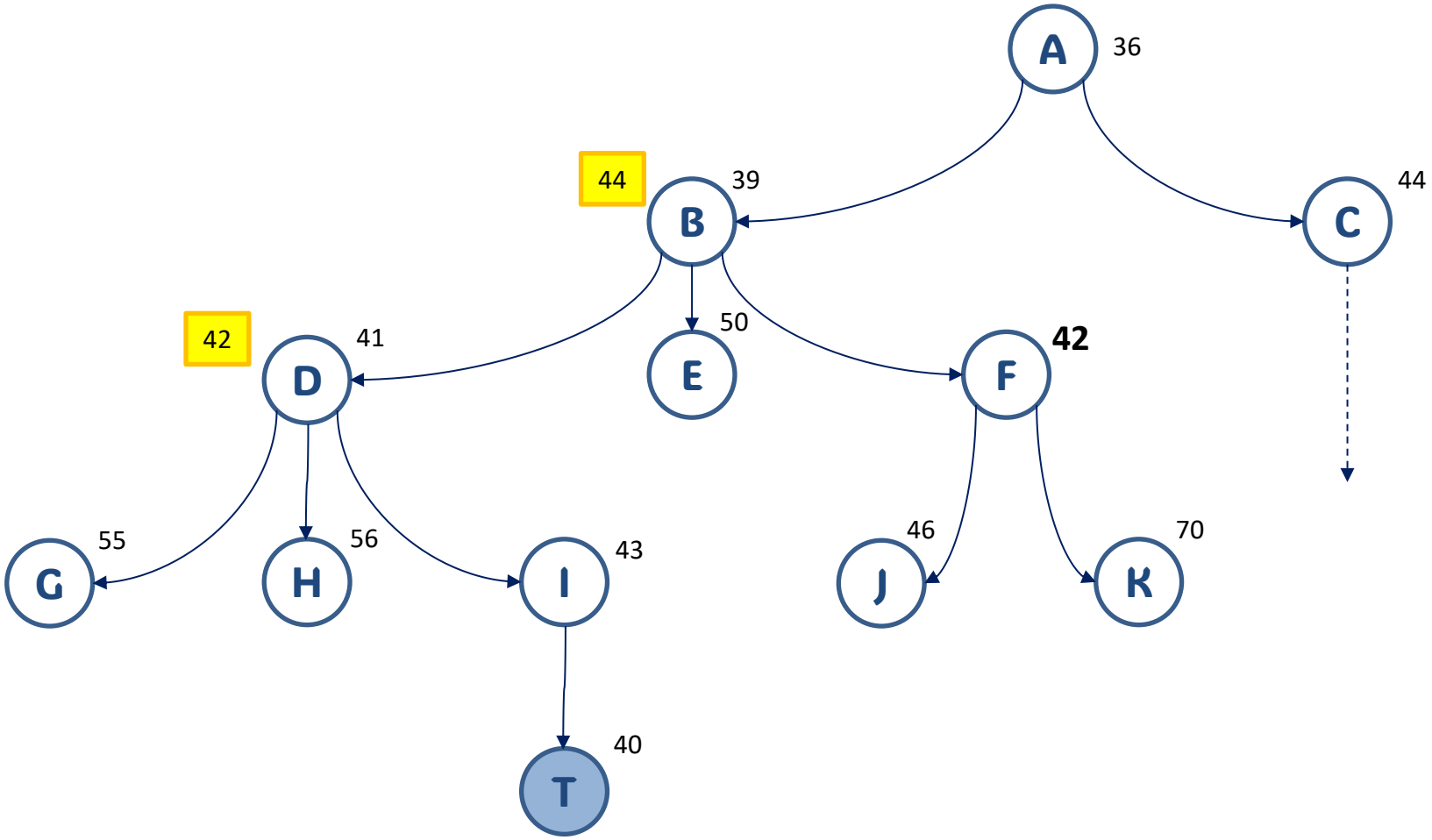
3. Algoritmo RBFS



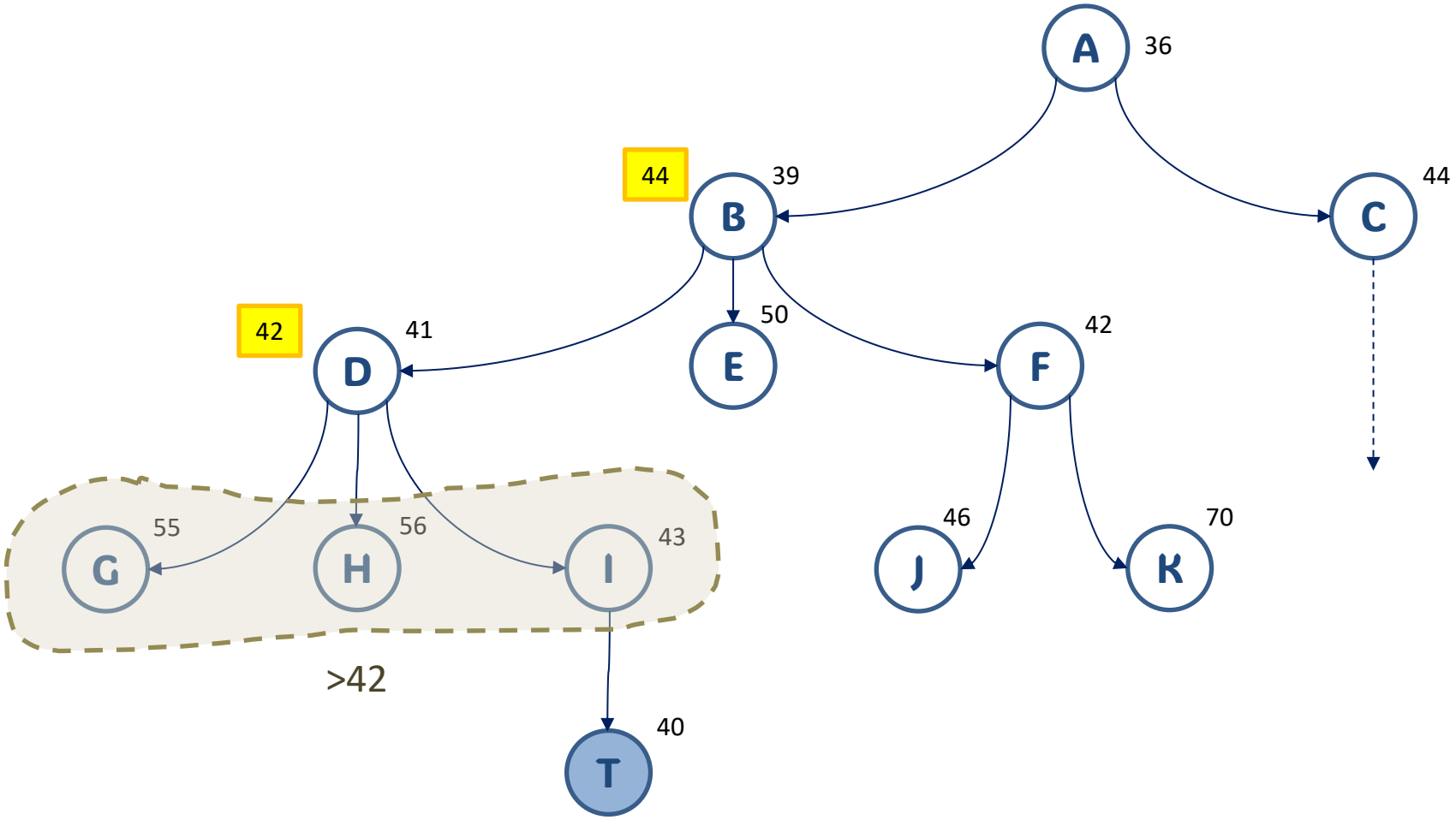
3. Algoritmo RBFS



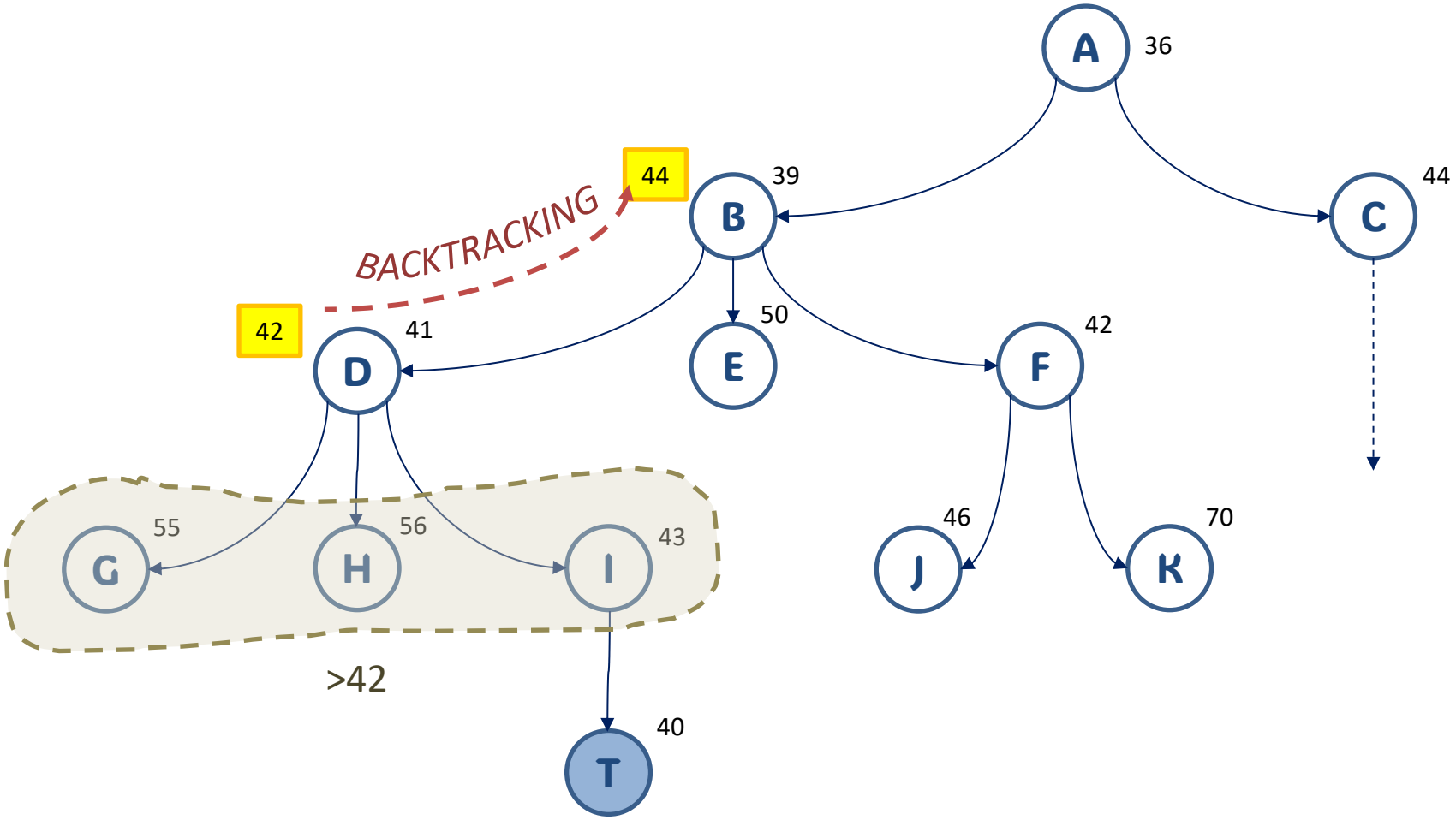
3. Algoritmo RBFS



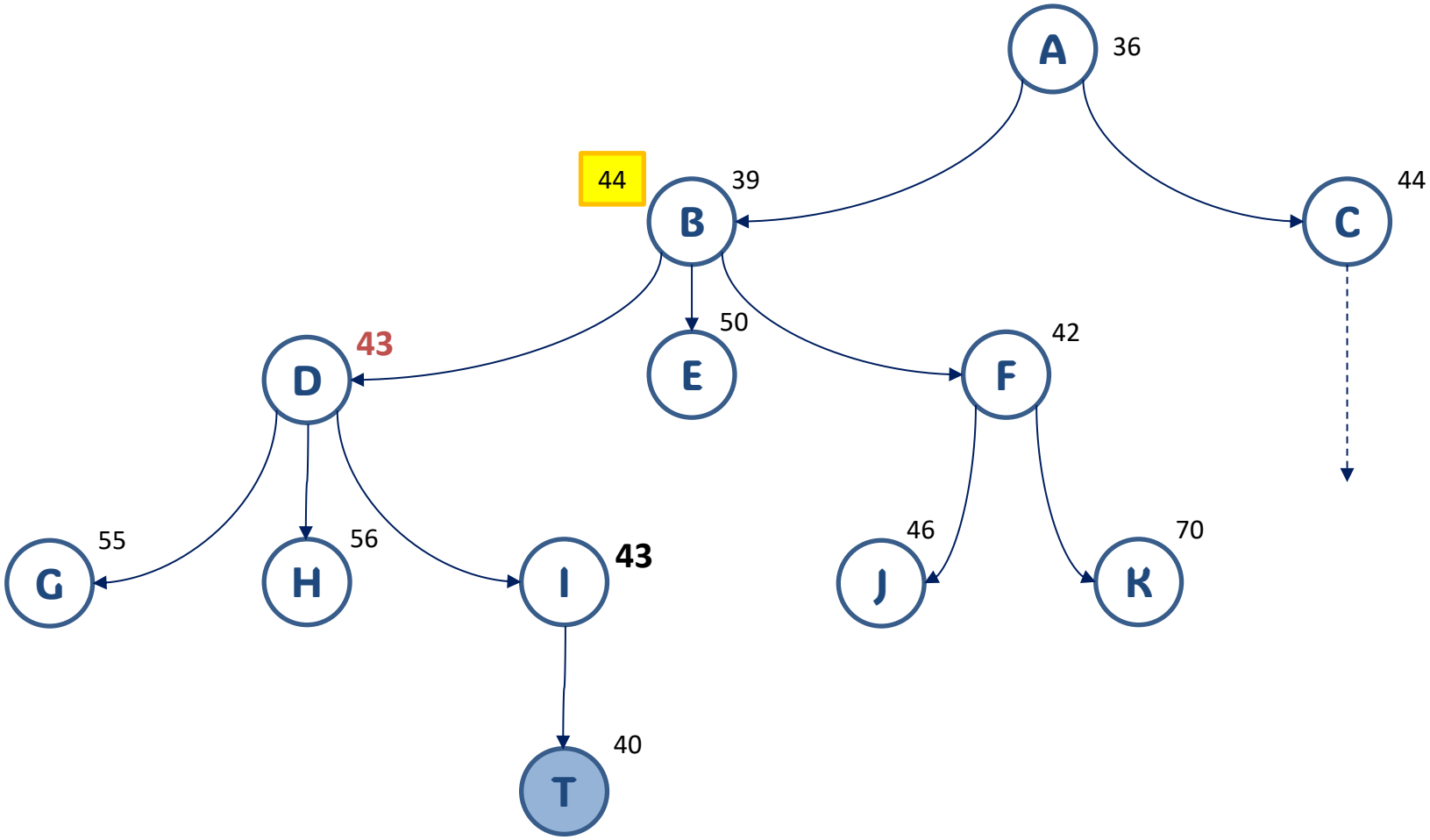
3. Algoritmo RBFS



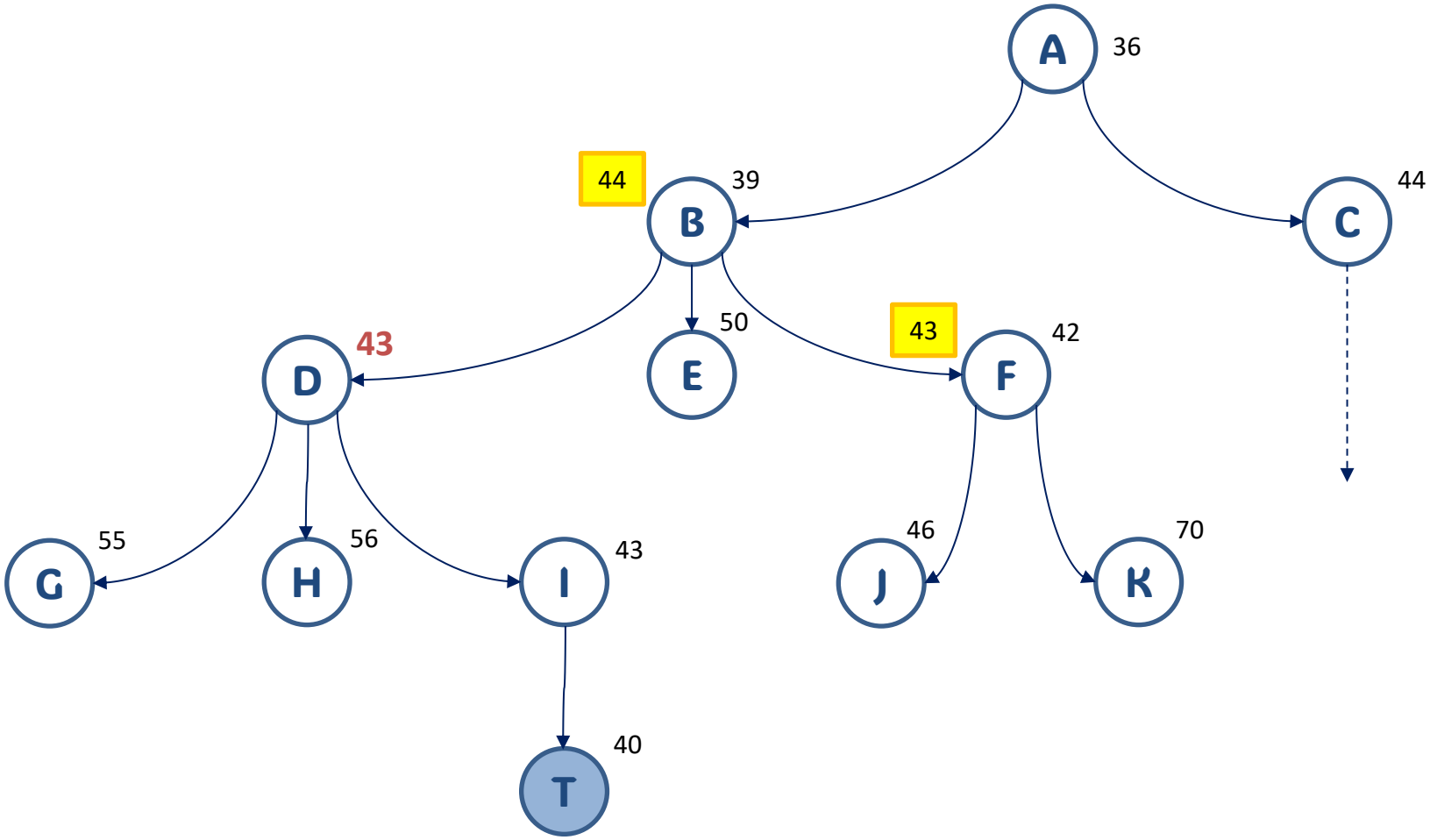
3. Algoritmo RBFS



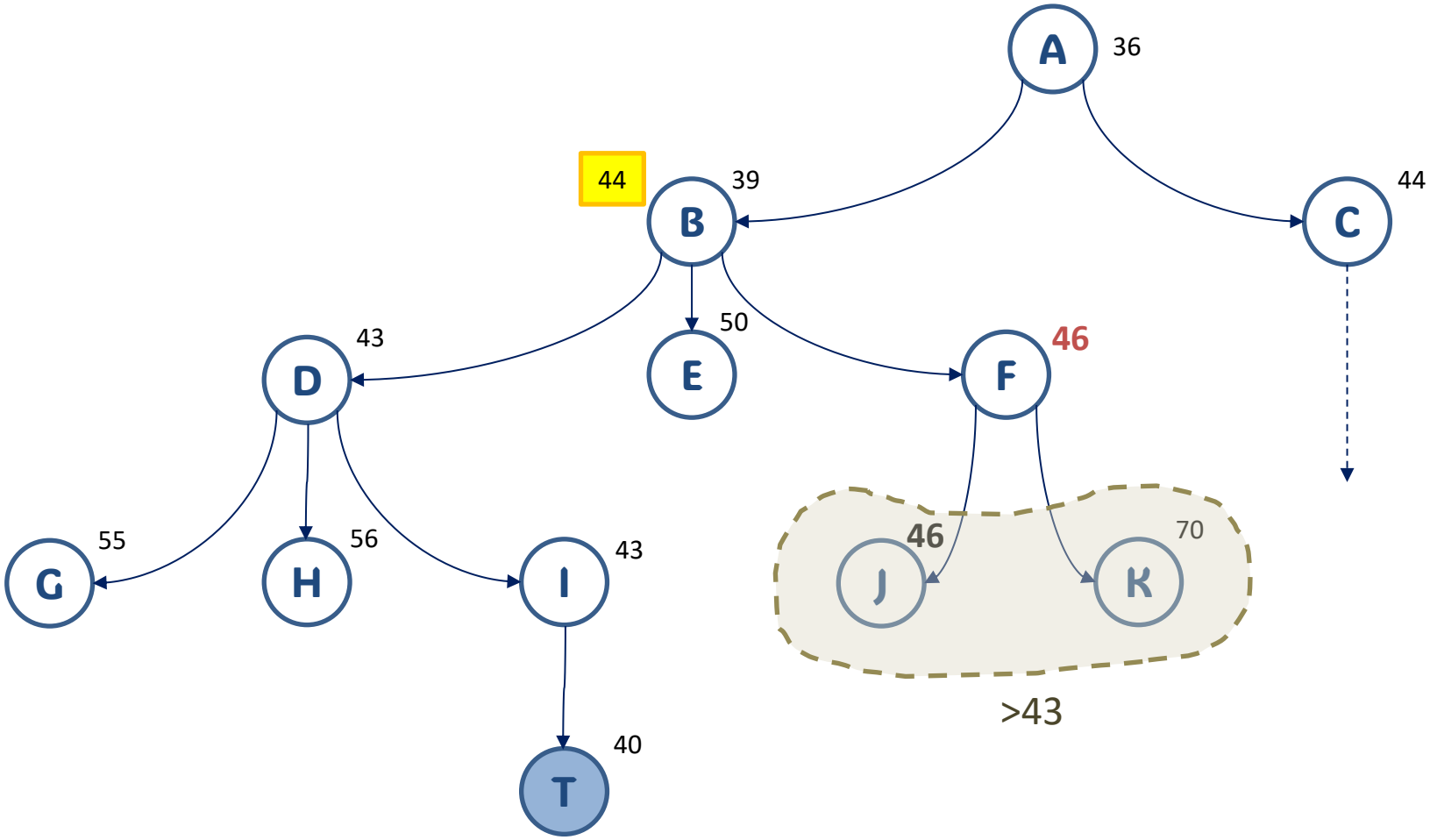
3. Algoritmo RBFS



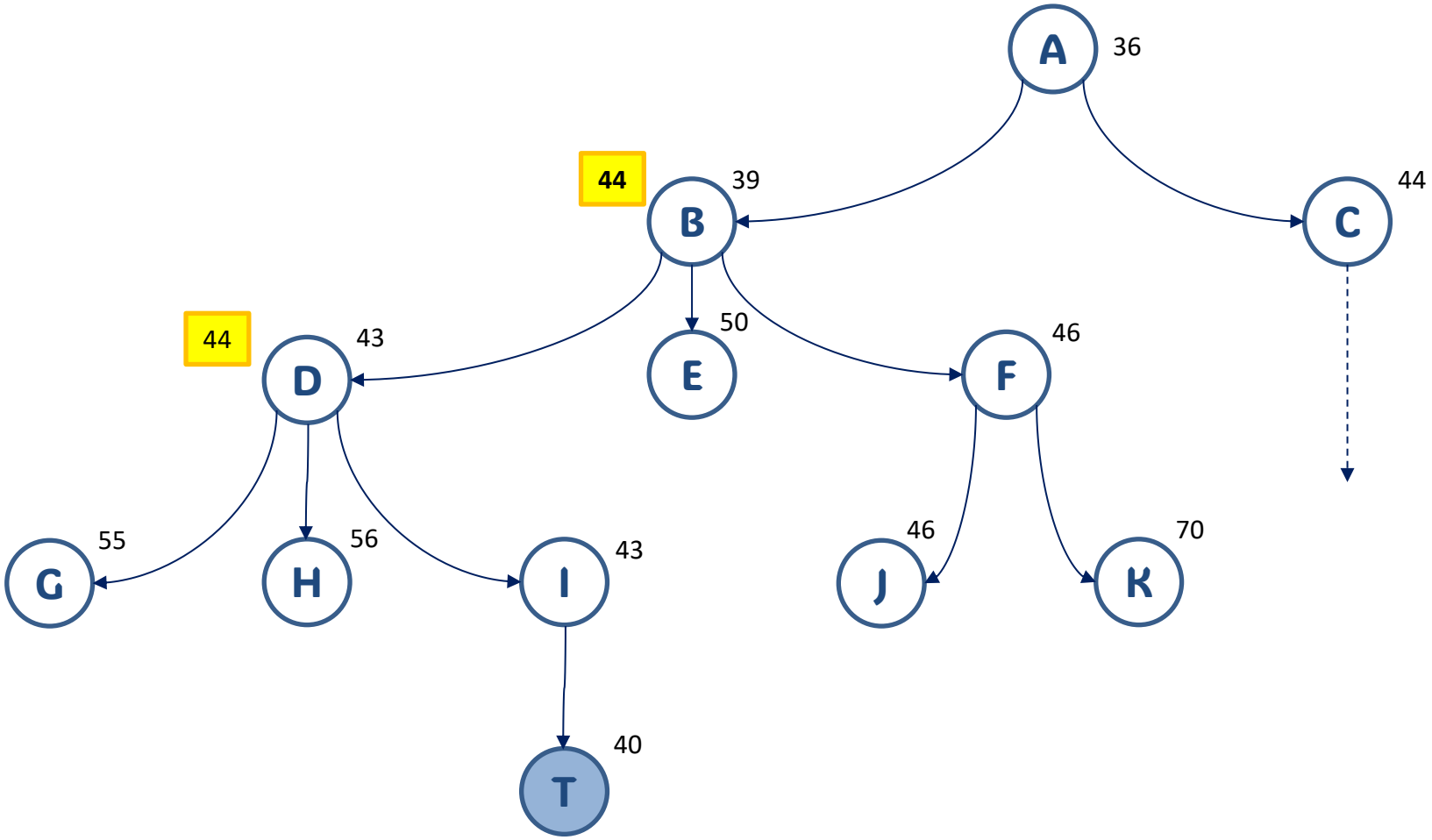
3. Algoritmo RBFS



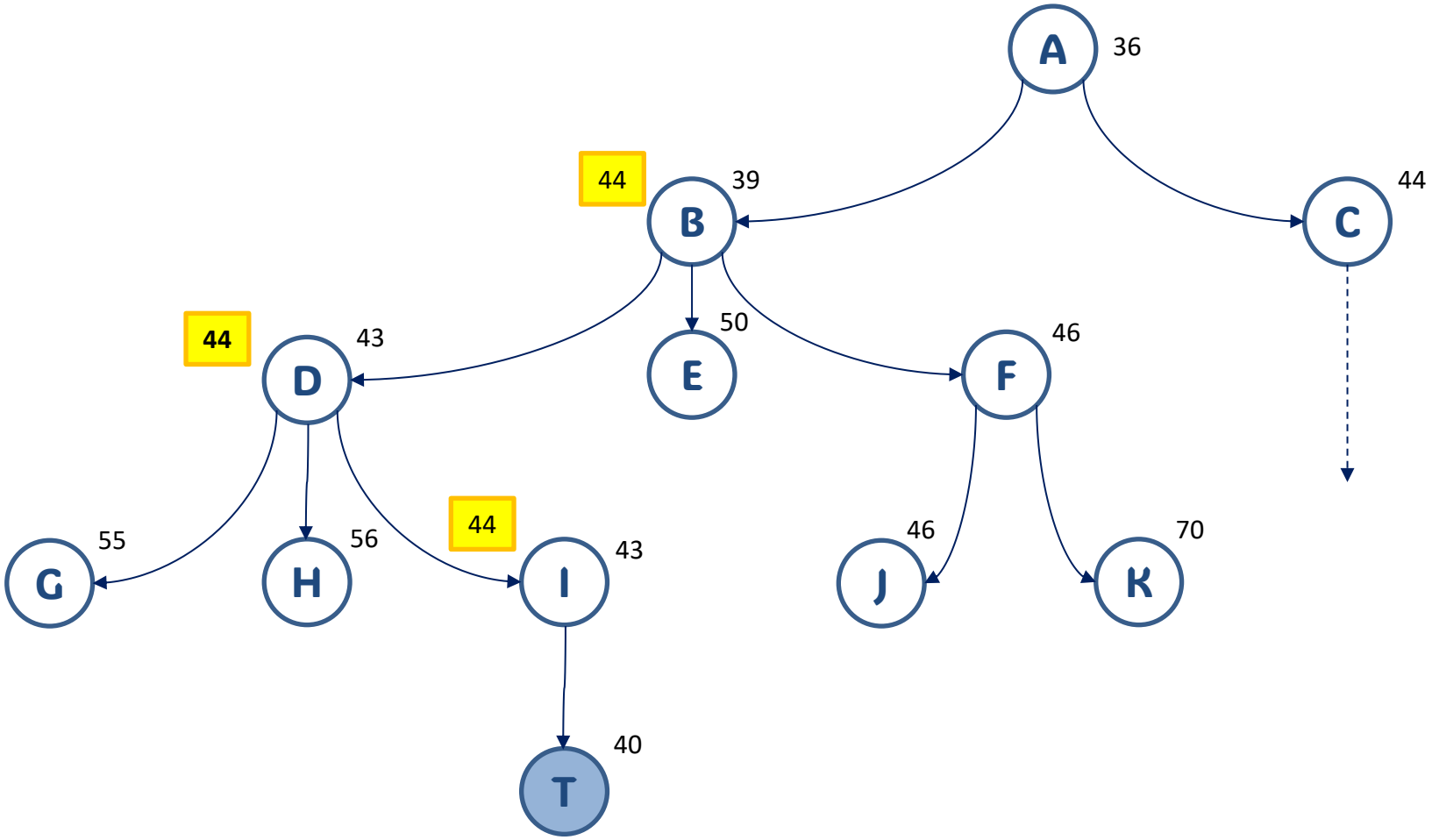
3. Algoritmo RBFS



3. Algoritmo RBFS



3. Algoritmo RBFS



3. Algoritmo RBFS

Algoritmo original diseñado por **Richard Korf:**

Linear-Space Best-First Seach: Summary of Results, AAAI-92
Linear-Space Best-First Search, Artificial Intelligence 1993
1992-Linear-Space Best-First Search: Summary of Results (aaai.org) (link donde aparece el algoritmo)

Algoritmo en libro S. Russell, P. Norvig. ***Artificial Intelligence. A modern approach.***

Propiedades:

- RBFS es óptimo bajo las mismas condiciones que A* ($\forall n, h(n) \leq h^*(n)$)
- La complejidad espacial es lineal con la profundidad del nodo solución más profundo (nodos del camino actual y nodos hermanos).
- La complejidad temporal es más difícil de caracterizar: depende de la precisión de la heurística y de cuántas veces cambia al camino alternativo.
- Expande nodos en orden creciente de valor-f incluso si la función $f(n)$ no es monotónica.

3. Comparación RBFS-IDA*

- Tanto IDA* como RBFS no pueden comprobar si hay estados repetidos, salvo los contenidos en la ruta actual. Por tanto, pueden explorar el mismo estado muchas veces (complejidad temporal exponencial).

- Cada backtracking del RBFS corresponde a una iteración del IDA* y podría tener que volver a generar muchos nodos para recrear el mejor camino.

- RBFS mantiene un valor-f volcado diferente para cada nodo interior mientras que IDA* mantiene un único límite de valor-f para todo el árbol.

- Entre iteraciones, IDA* solo retiene el valor de lim-f. RBFS mantiene más información en memoria pero aun teniendo más espacio disponible, RBFS no tendría modo de hacer uso de ello (los dos algoritmos 'olvidan lo que han hecho' y terminan explorando los mismos estados muchas veces)

- RBFS es algo más eficiente que IDA* porque no tiene que regenerar todo el árbol sino que hace backtracking al segundo camino mejor disponible. Aun así, sigue teniendo el problema de la excesiva regeneración de nodos.

4. Simplified memory-bounded A*: SMA*

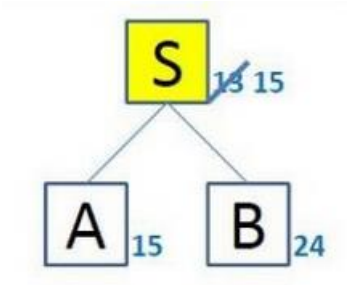
- Expande nodos como A* hasta que se 'llena' la memoria (hasta que el número de nodos almacenado alcanza un valor prefijado).

- Cuando se llena la memoria no se puede añadir un nuevo nodo a menos que se elimine un nodo de esta lista (*nodo olvid*ado), que es el peor nodo hoja (el de mayor valor-f).

- Al igual que RBFS, SMA* vuelca el valor del nodo olvidado a su padre. De este modo:
  - El antecesor de un sub-árbol olvidado guarda información sobre la calidad del mejor camino en ese sub-árbol
  - SMA* regenera el sub-árbol olvidado solo cuando todos los otros caminos son peores que el valor-f del camino del sub-árbol

- A medida que se va explorando el espacio de búsqueda, SMA* almacena en cada nodo el valor-f del mejor camino alcanzable a partir de dicho nodo.

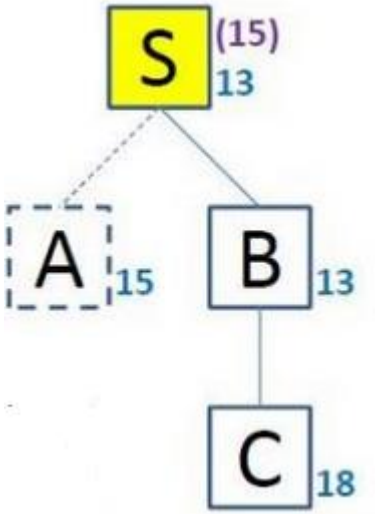
4. Características del funcionamiento de SMA*



Actualización del valor-f

Cuando se han generado todos los hijos de un nodo S, se actualiza f(S) al mejor valor-f de sus hijos para recordar el mejor valor alcanzable a partir de S.

Esta actualización se propaga de hijos a padres: si tras generar hijos de A se actualiza f(A)=20, se actualiza también f(S)=20.

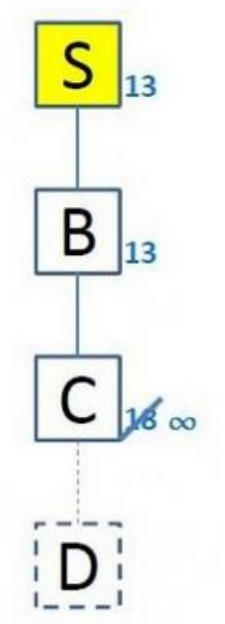


Memoria llena: (ejemplo: max num. nodos = 3)

Para poder generar el nodo C hay que eliminar previamente un nodo:

- se elimina el nodo hoja con mayor valor-f (nodo A)
- se recuerda el valor del nodo olvidado en el padre

4. Características del funcionamiento de SMA*

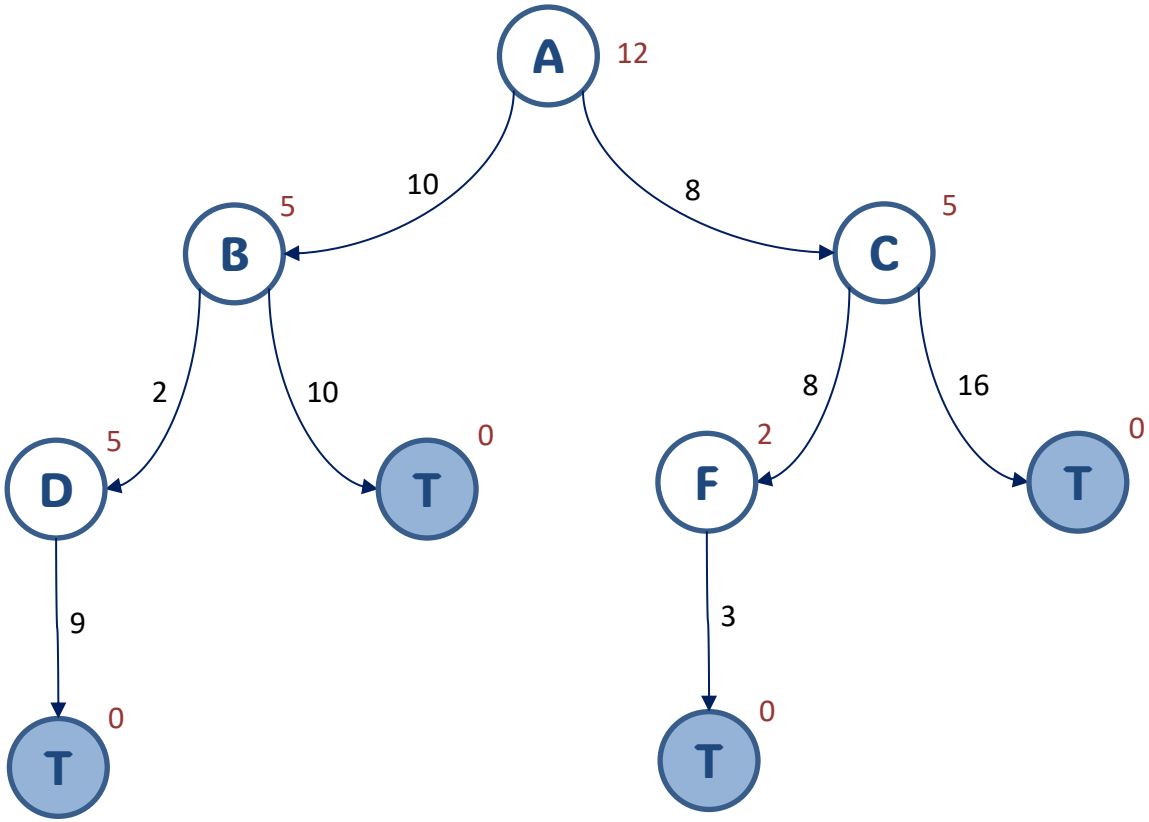


Caminos demasiado largos: (ejemplo: max num. nodos = 3)

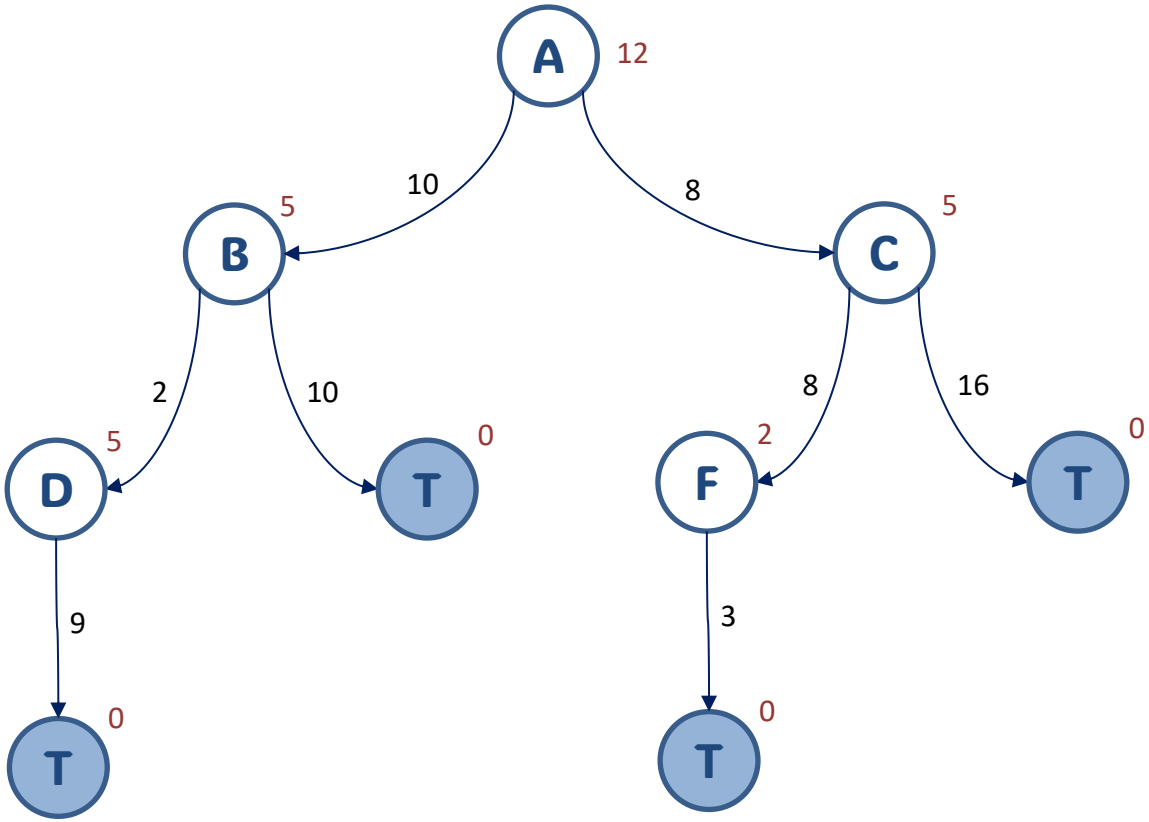
No se puede seguir expandiendo el camino por el nodo C porque ya no caben más nodos en memoria, y C no es solución:

- se actualiza f(C)= ∞ para reflejar que no se puede encontrar solución a través del nodo C

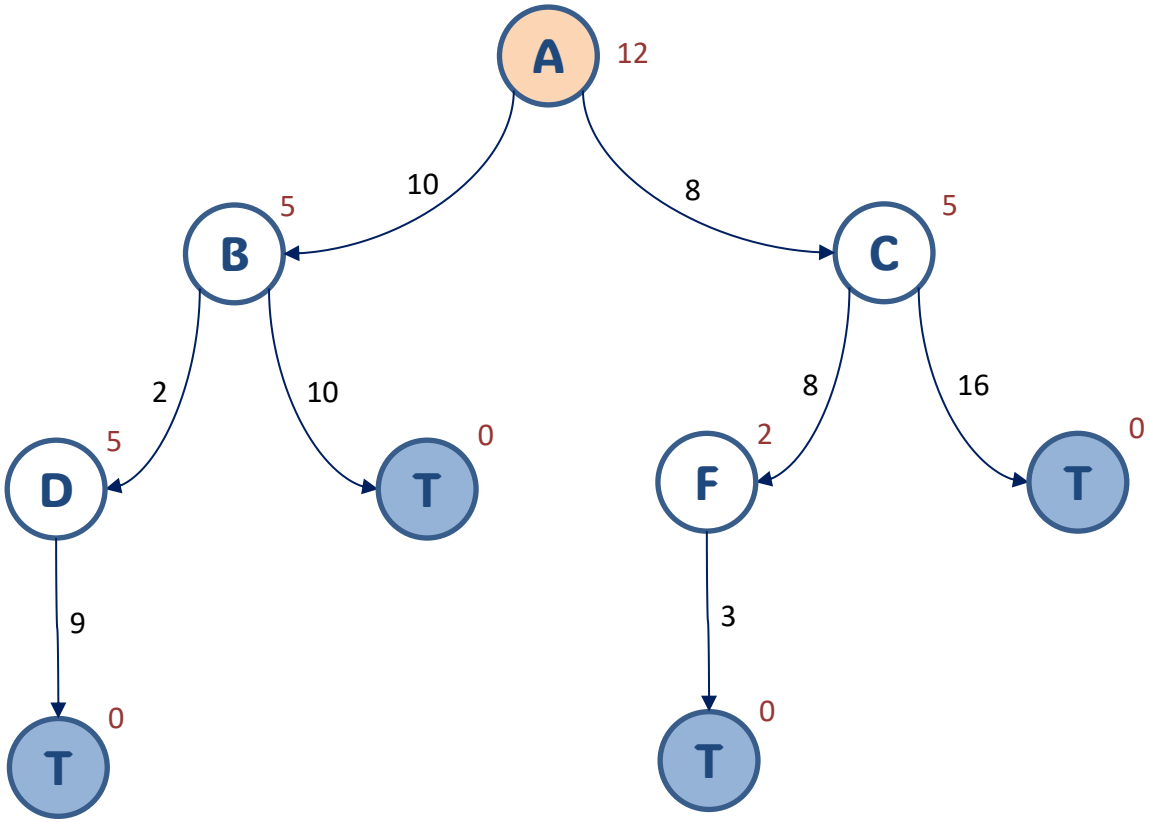
3. Algoritmo SMA*

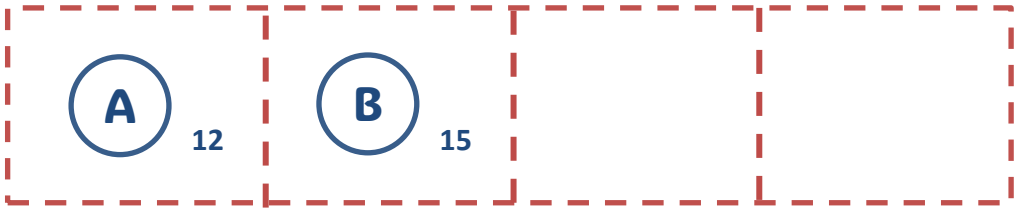


3. Algoritmo SMA*

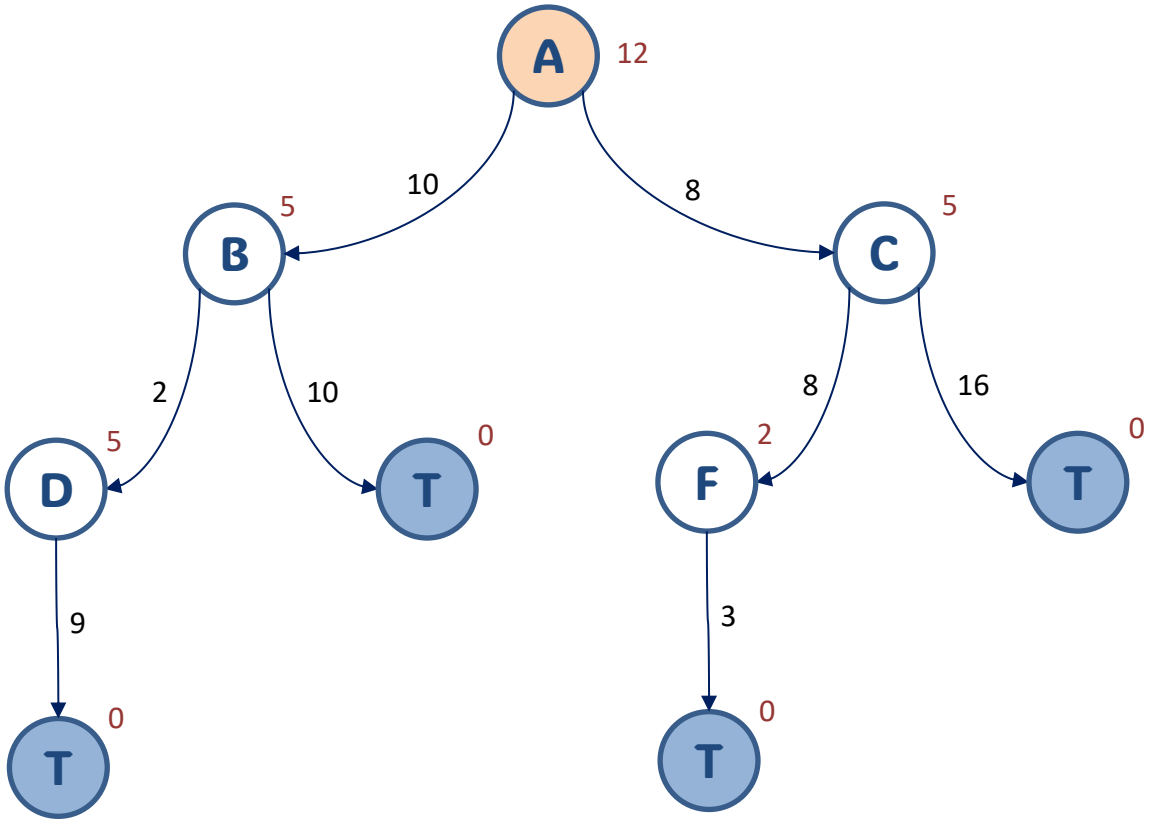


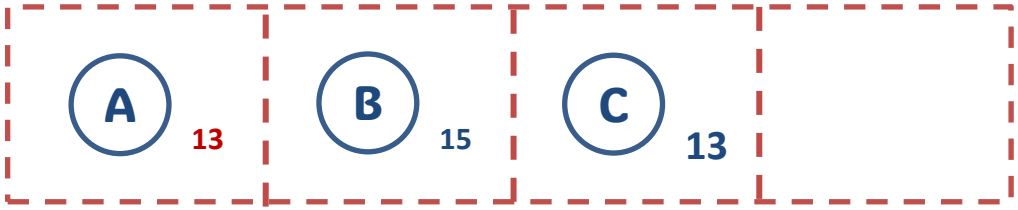
3. Algoritmo SMA*



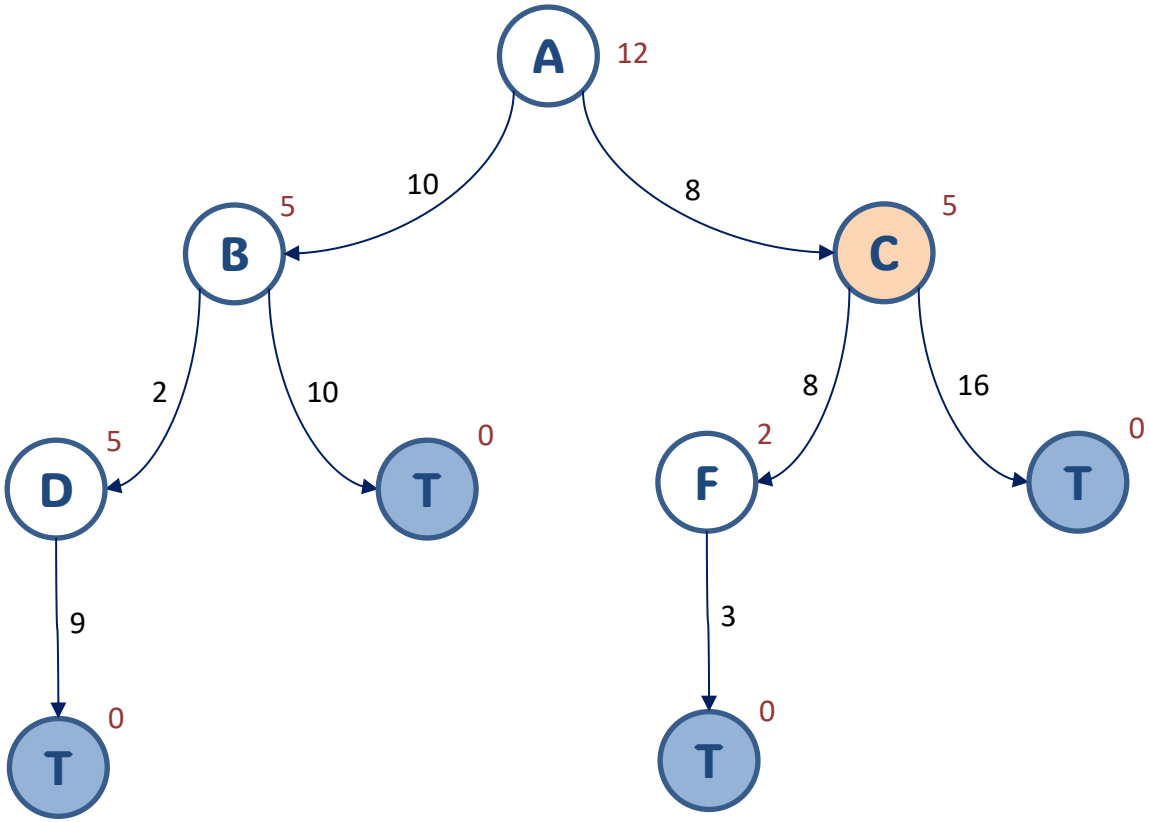


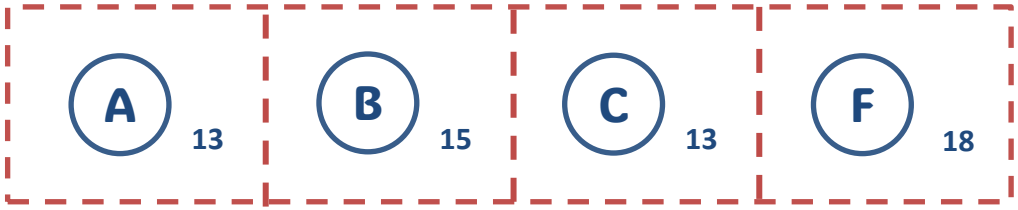
3. Algoritmo SMA*



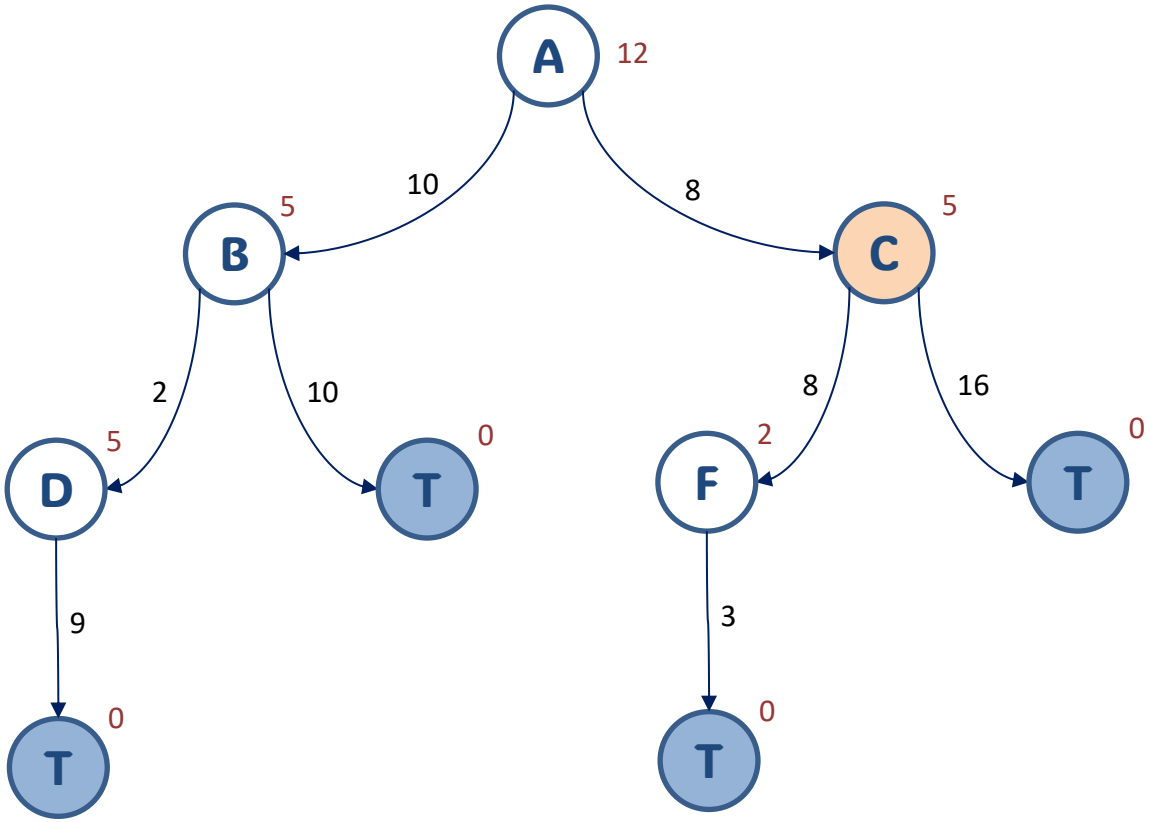


3. Algoritmo SMA*

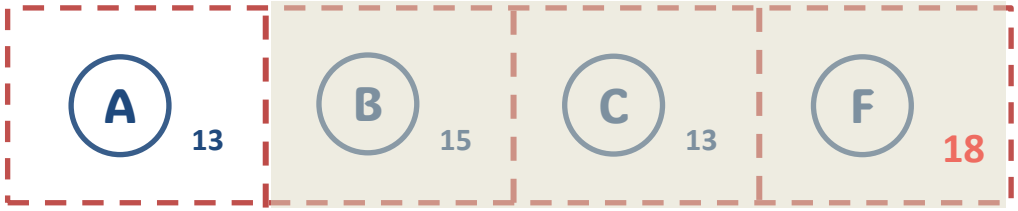




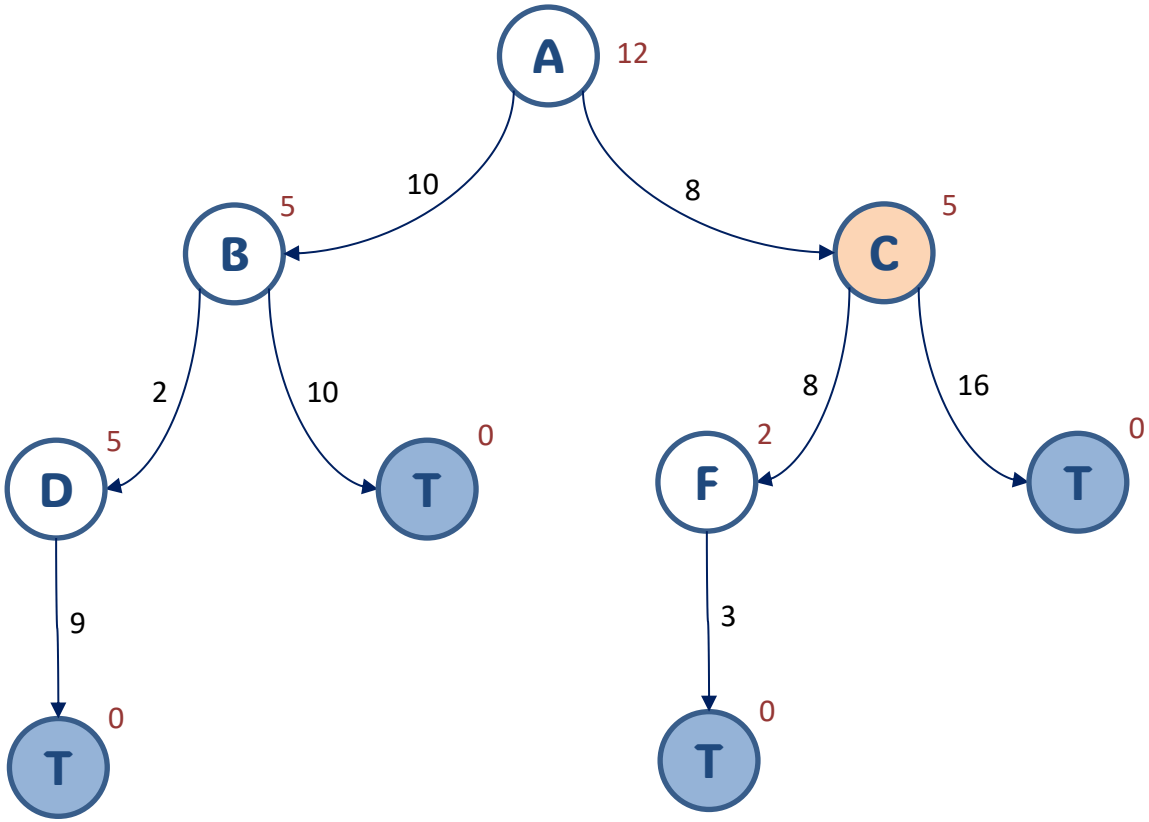
3. Algoritmo SMA*

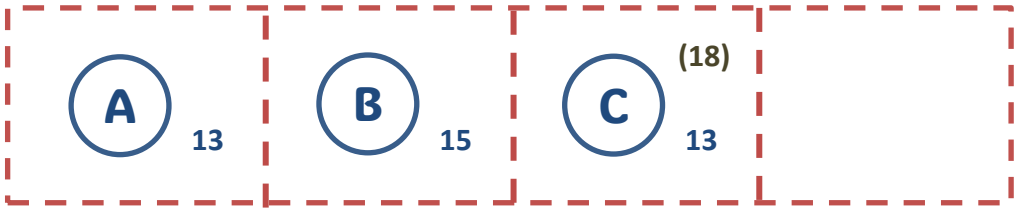


Max is Deleted

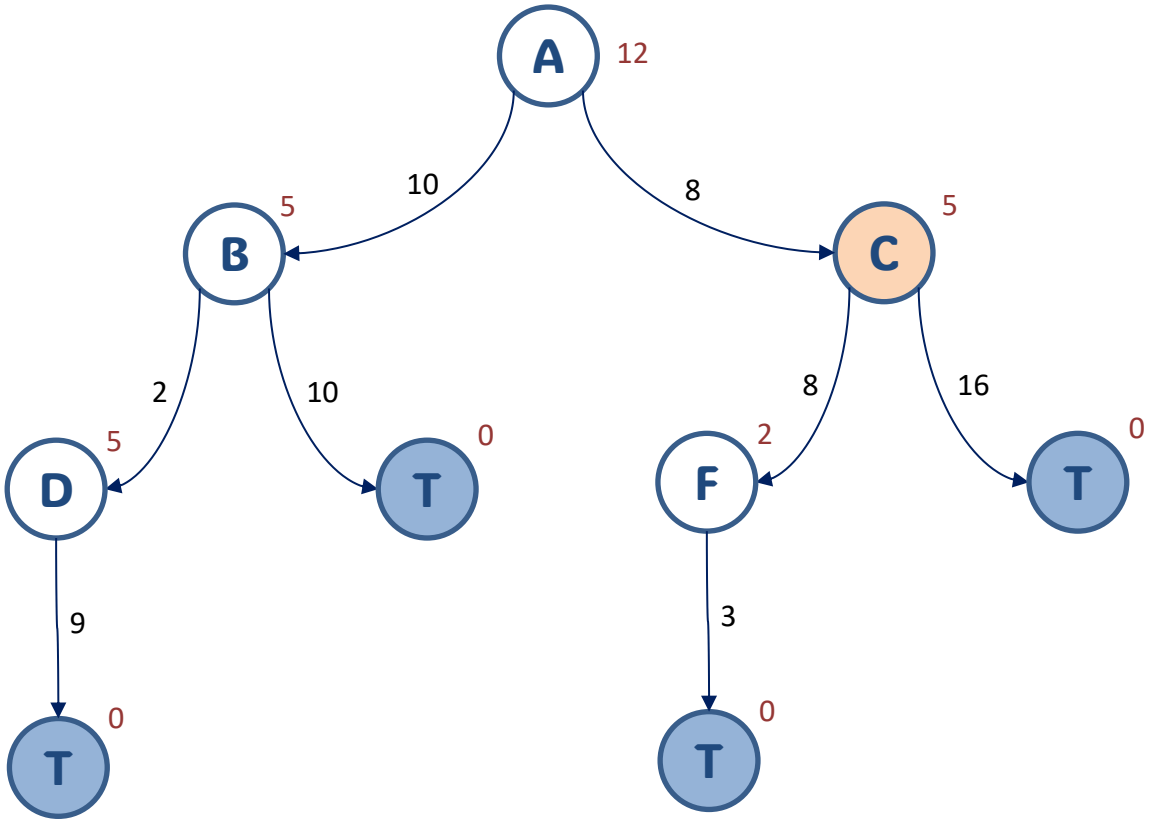


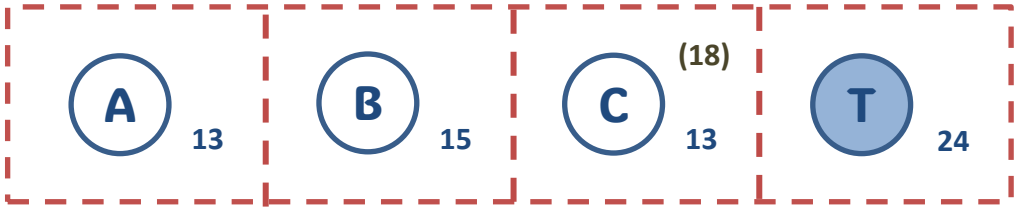
3. Algoritmo SMA*



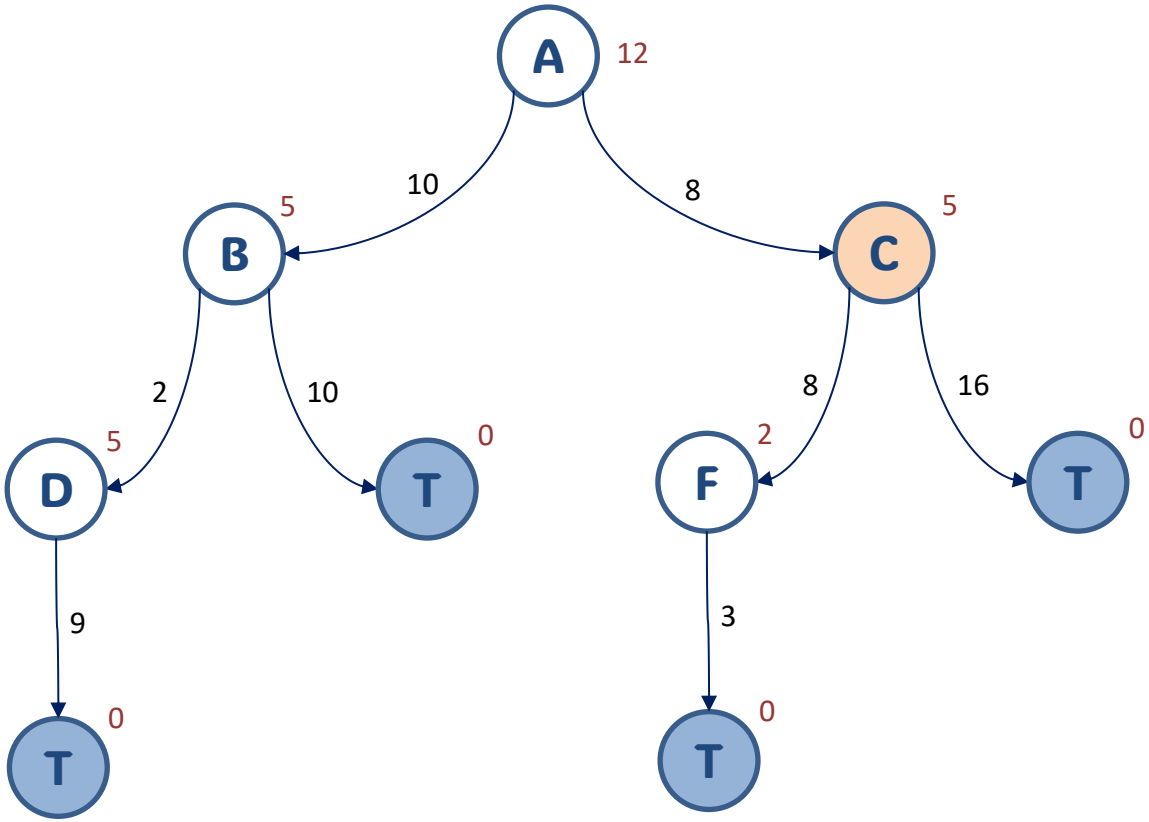


3. Algoritmo SMA*

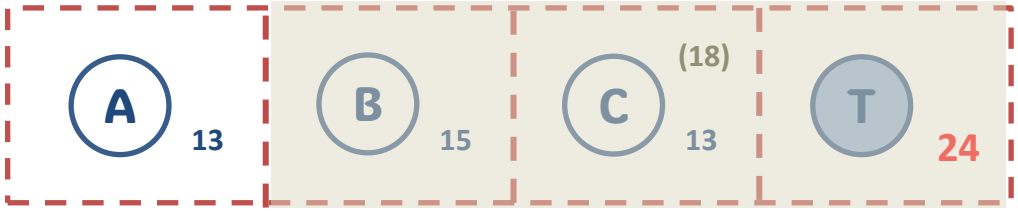




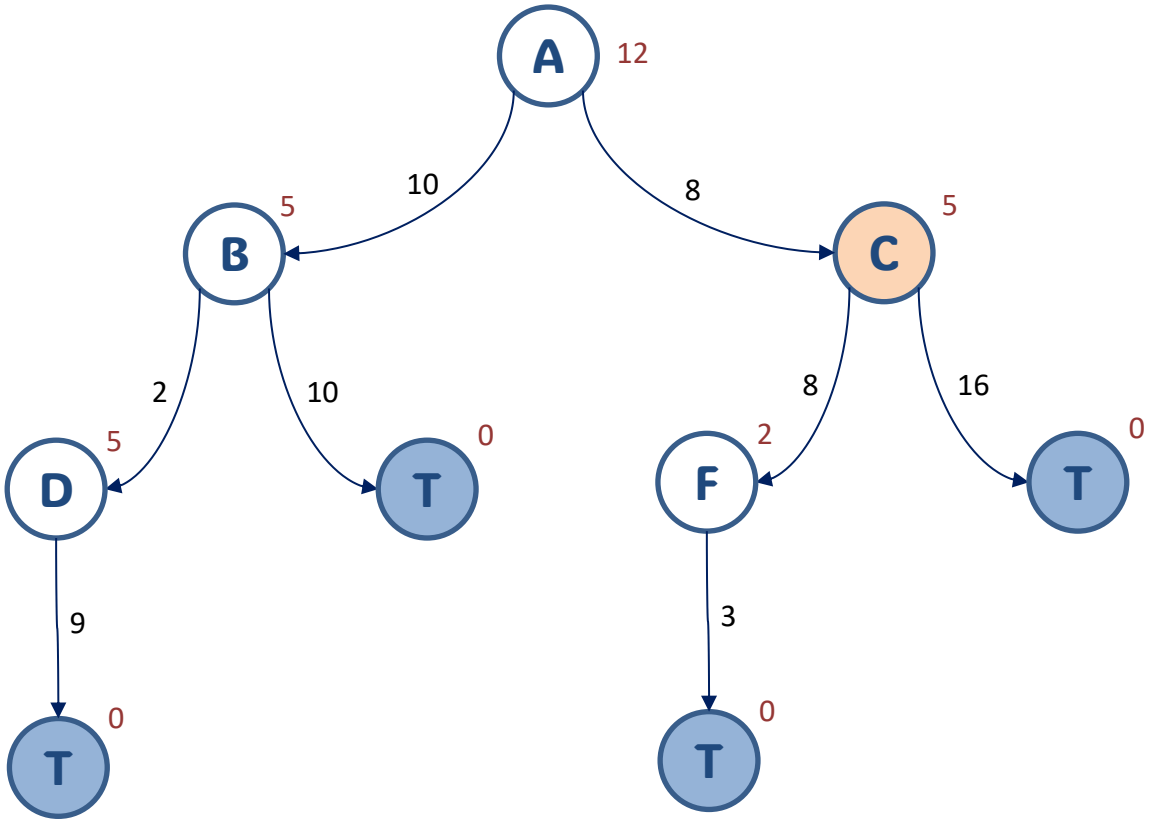
3. Algoritmo SMA*



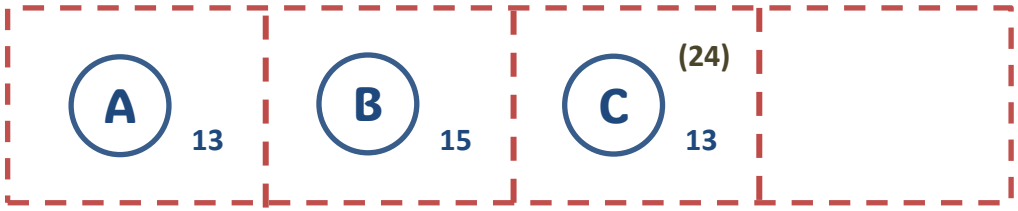
Max is Deleted



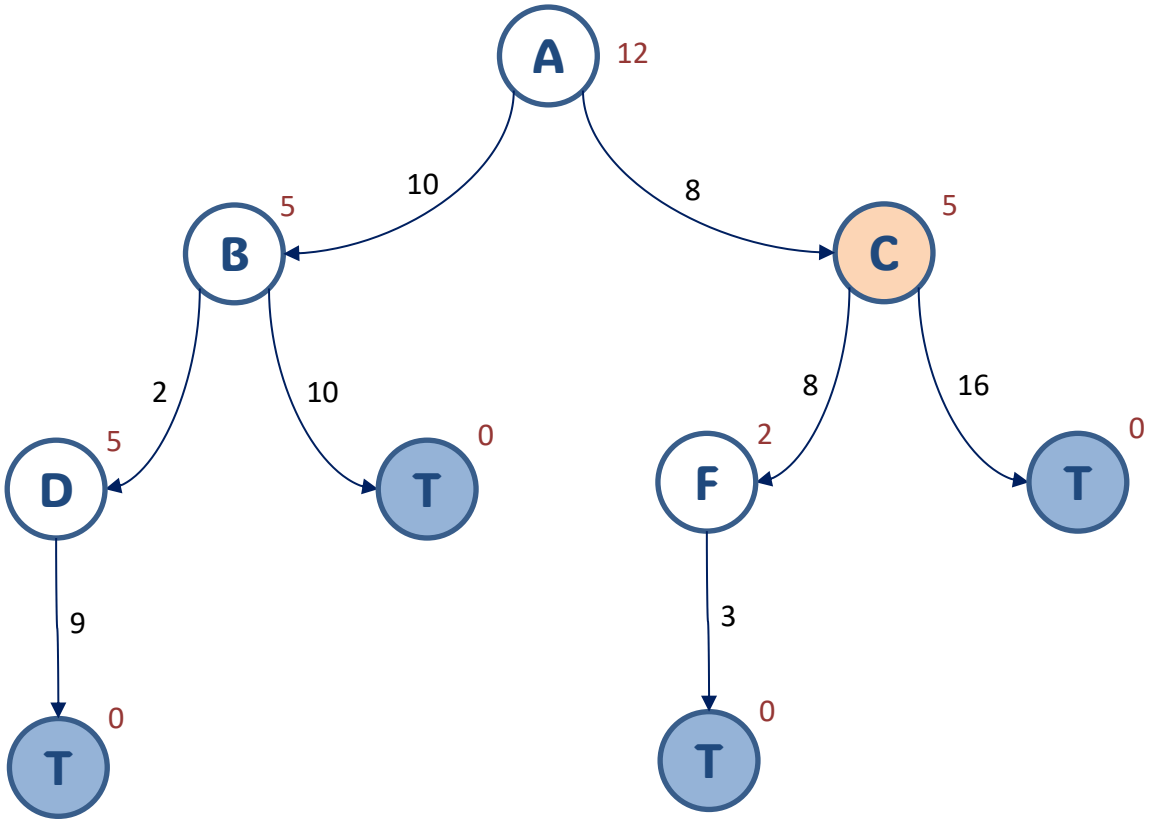
3. Algoritmo SMA*

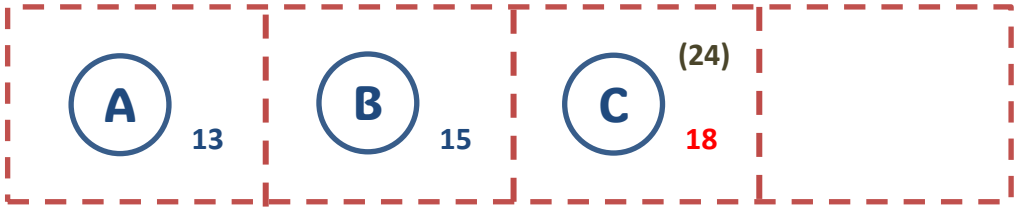


(18)<24

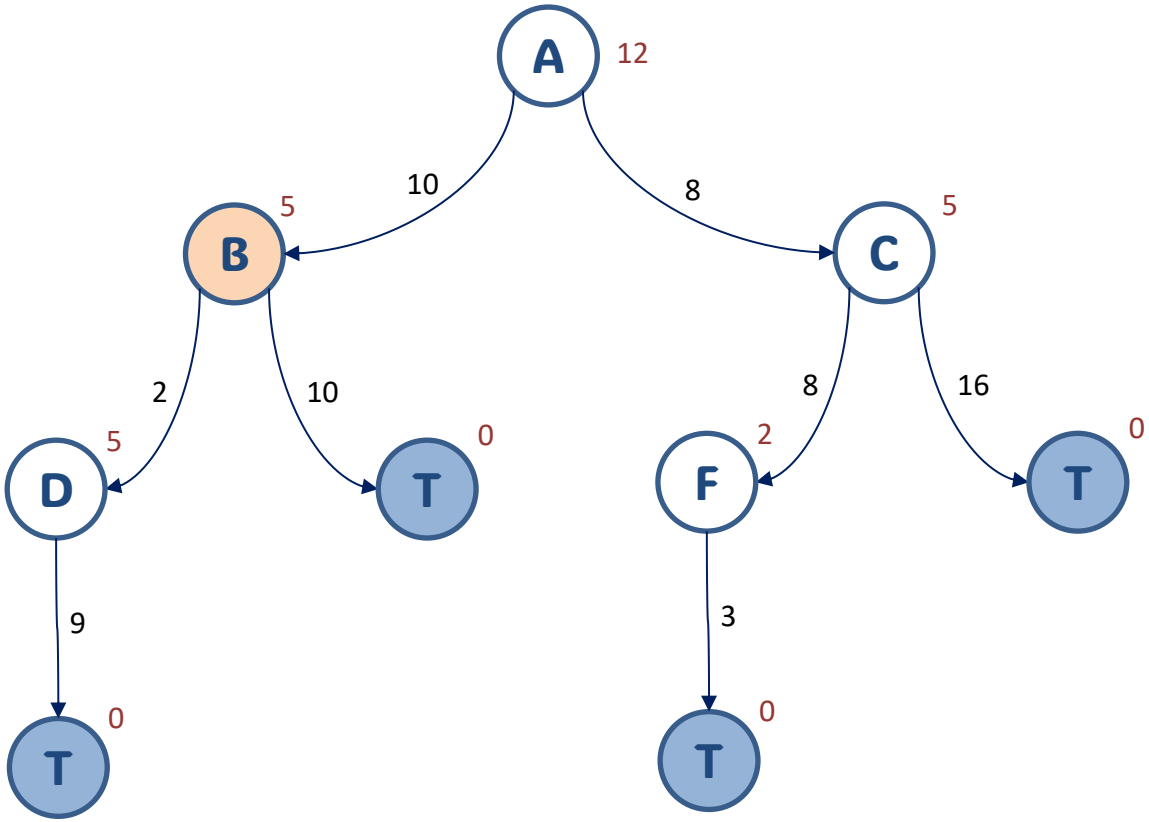


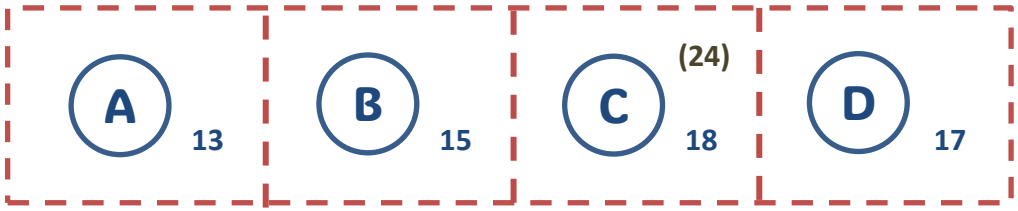
3. Algoritmo SMA*



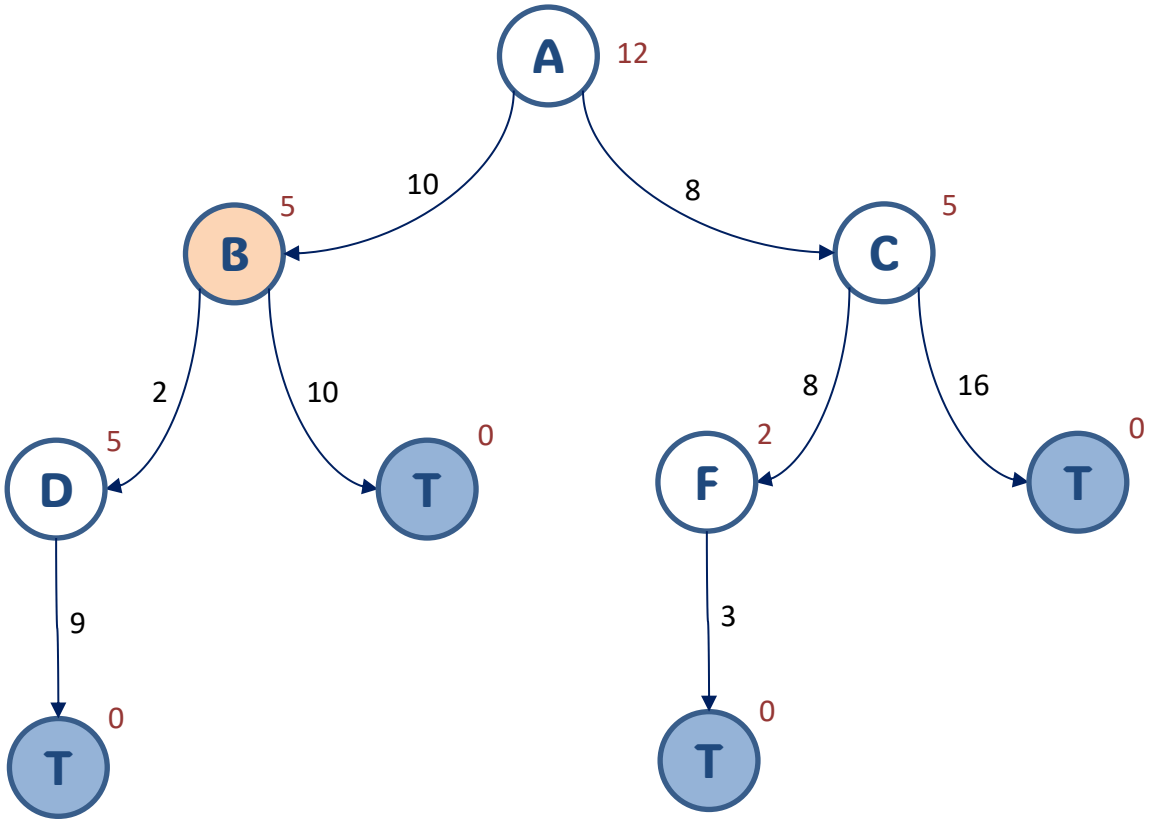


3. Algoritmo SMA*

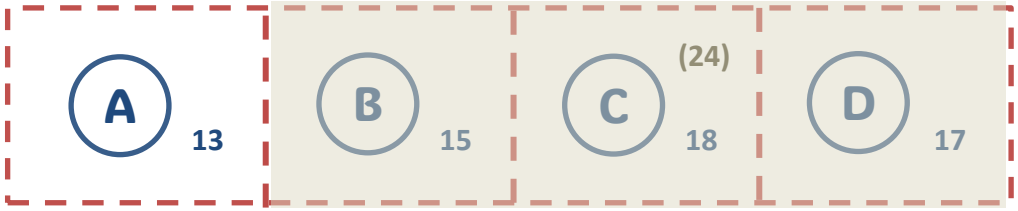




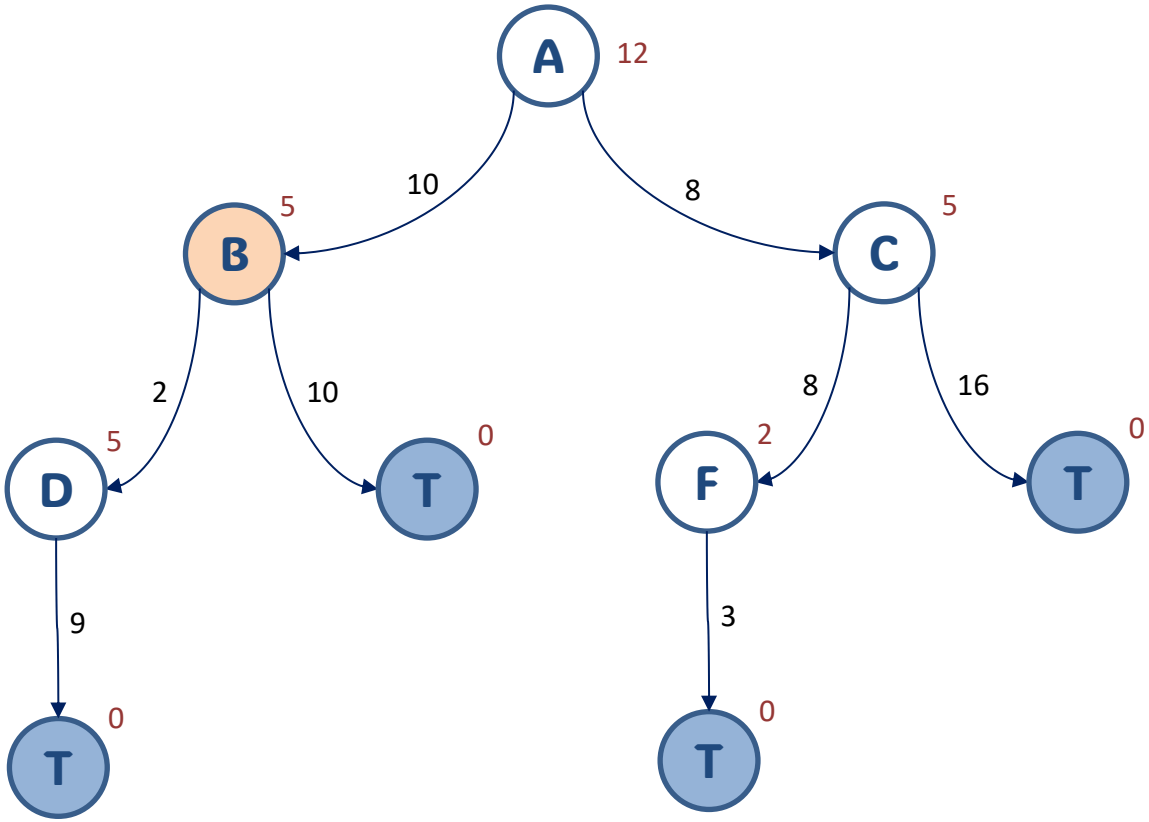
3. Algoritmo SMA*

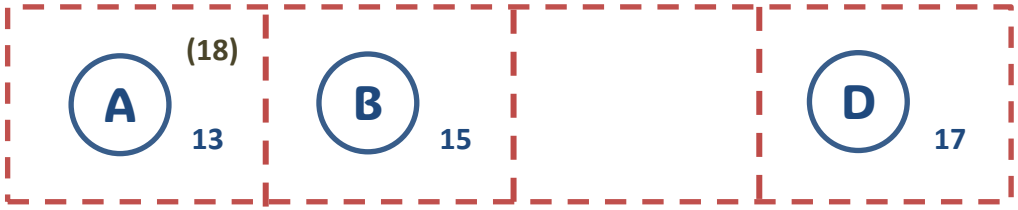


Max is Deleted

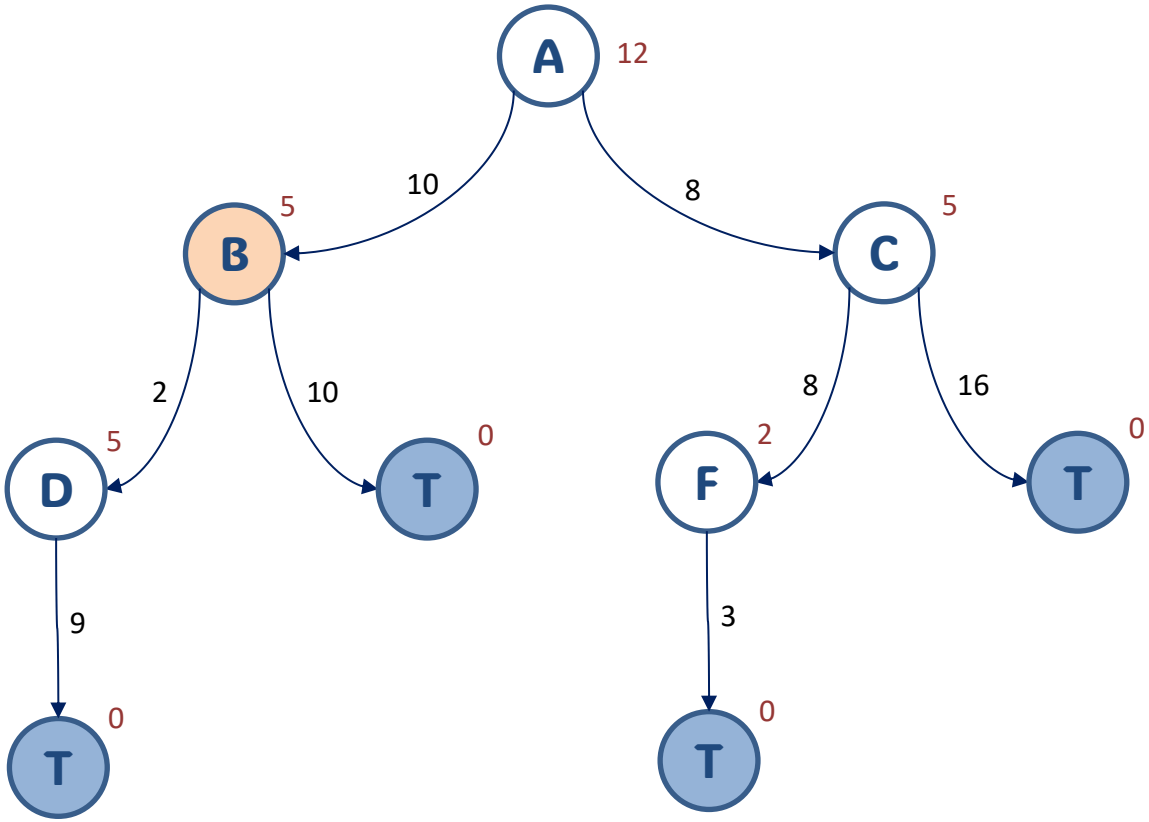


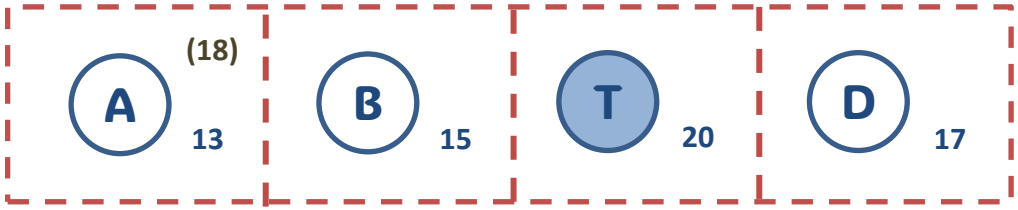
3. Algoritmo SMA*



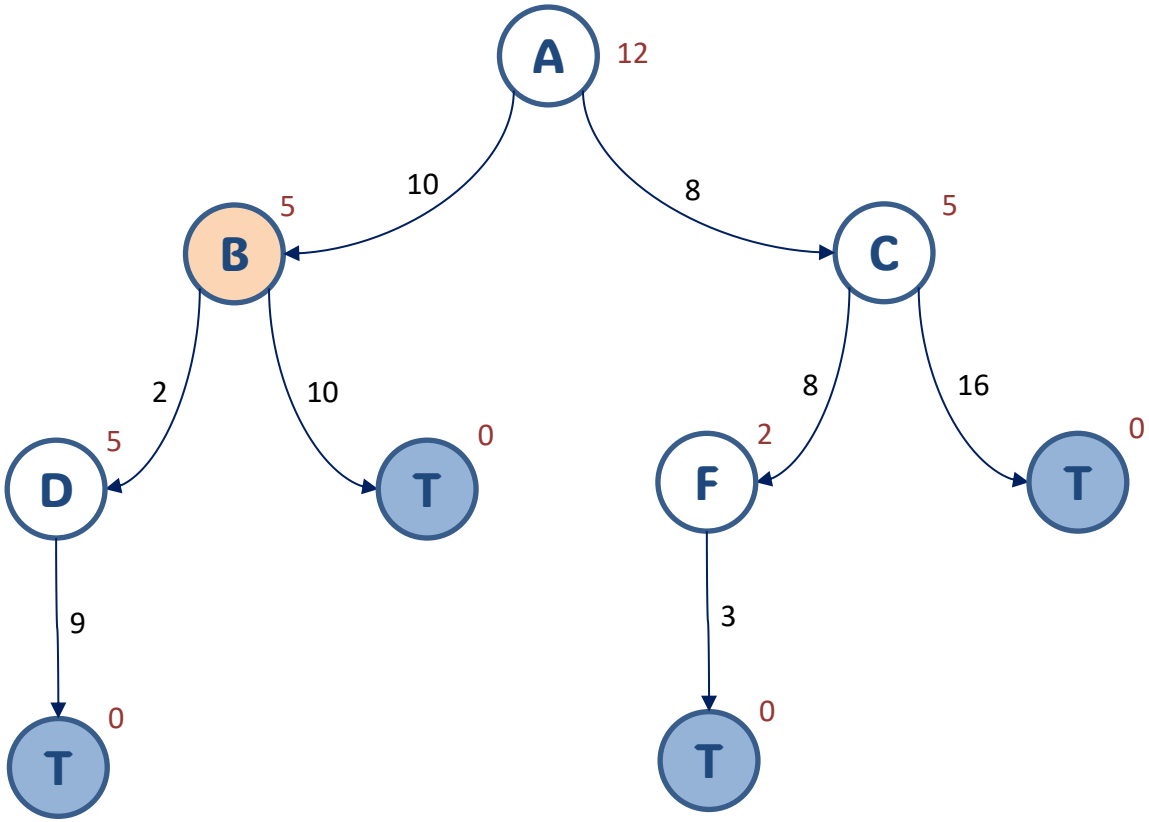


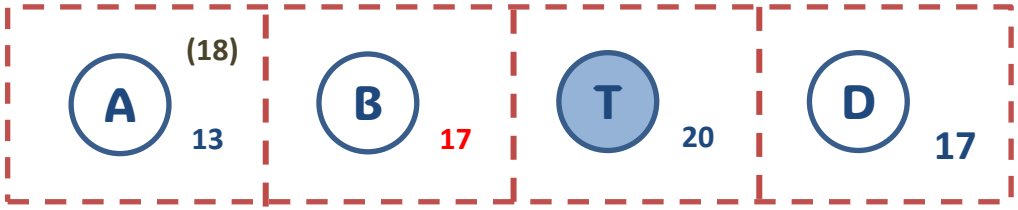
3. Algoritmo SMA*



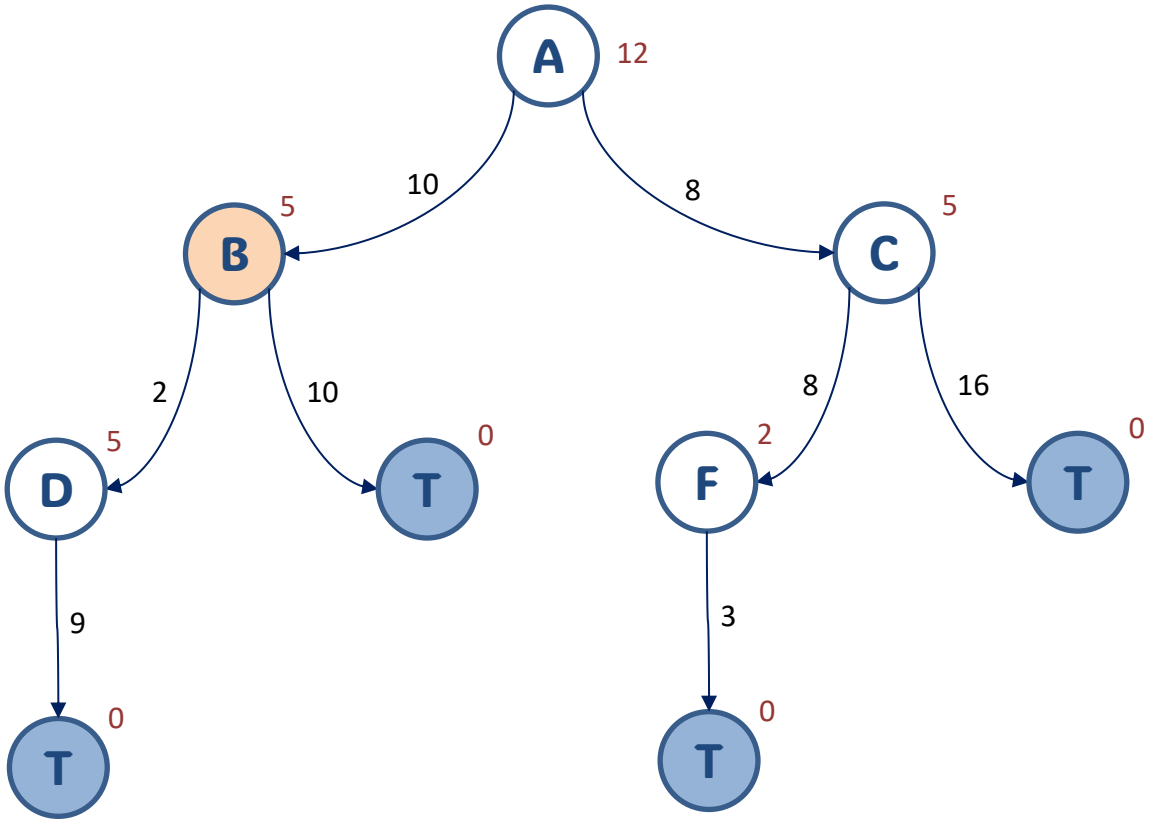


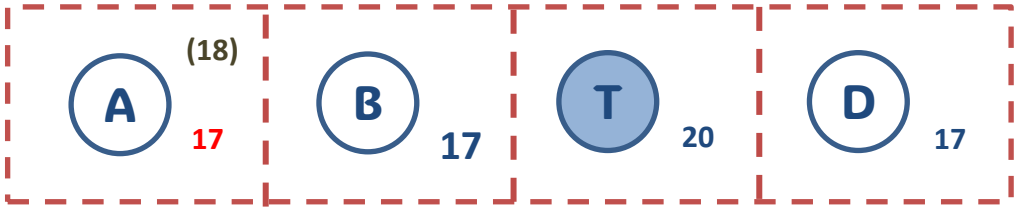
3. Algoritmo SMA*





3. Algoritmo SMA*





4. Propiedades SMA*

- El mejor camino alternativo en RBFS puede ser cualquier nodo del árbol, en SMA* se almacena el mejor hijo olvidado.

- SMA* es completo si hay alguna solución alcanzable, es decir si d, la profundidad del nodo meta de menor profundidad es menor que el tamaño de la memoria QUEUE expresado en número de nodos (complejidad temporal $O(b^d)$).

- Es admisible si dispone de suficiente memoria para guardar la ruta de la solución óptima de menor profundidad. En caso contrario encontrará la 'mejor' solución que pueda encontrar con la memoria disponible.

- En la práctica, SMA* es un algoritmo de memoria limitada de uso general bastante robusto para encontrar soluciones óptimas, especialmente:
  - para problemas difíciles donde el espacio de estados es un grafo (espacios de estados altamente conectados)
  - cuando los costes no son uniformes
  - la generación de un nodo es costosa comparado con el coste adicional de mantener las listas de abiertos y cerrados.

- Sin embargo, en problemas muy difíciles, generalmente SMA* tiene que saltar continuamente de un nodo a otro entre muchos caminos solución candidatos, y solo un subconjunto de estos caminos puede caber en memoria.