ST. FRANCIS XAVIER UNIVERSITY

CSCI-564
CONSTRAINT PROCESSING AND HEURISTIC SEARCH

LECTURE 10 – LINEAR-SPACE SEARCH

Dr. Jean-Alexis Delamer

Recap

- The A* algorithm always terminates with an optimal solution
- It can be applied to general state space problems.
  - You can transform a problem to use a distance metric.

Linear-Space search

- Suppose that for a given problem:
  - You need 100 bytes to store one state.
  - A* generates 100,000 new states every second.
  - It represents ~10 MB per second.

- After 10 minutes you are using 5 GB of memory.

In a matter of minutes, you can run out of memory.

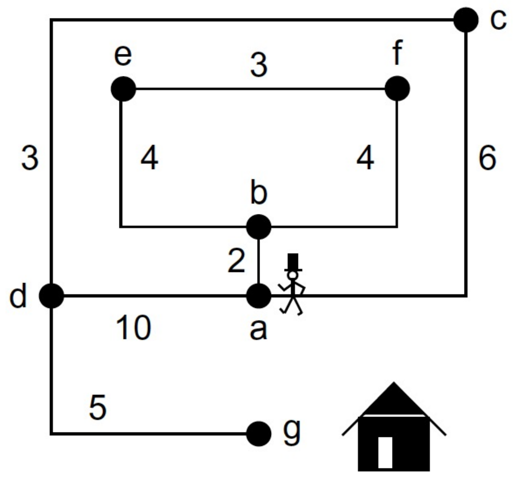
Linear-Space search

- Several search techniques can scale linearly with the search depth.
  - A major trade-off is an increase in time (possibly exponential).

- Example:
  - The lower bound is a search with logarithmic space.
  - However, the time increase makes it intractable.

Linear-Space search

- Example:
  - The lower bound is a search with logarithmic space.
  - However, the time increase makes it intractable.



- Divide-and Conquer BFS
  - A graph with n nodes.
  - Call $Exists\text{−}Path$ that return if a path exists between u and v with l edges.
    - If $l = 1$, return true
    - Otherwise, for each intermediate nodes index j, $1 \leq j \leq n$, it calls recursively *Exists-Path*($u, j, \lceil l/2 \rceil$) and *Exists-Path*($j, b, \lfloor l/2 \rfloor$).

Linear-Space search

- Divide-and Conquer BFS
  - A graph with n nodes.
  - Call $Exists\text{—}Path$ that return if a path exists between u and v with l edges.
    - If $l = 1$, return true
    - Otherwise, for each intermediate nodes index j, $1 \leq j \leq n$, it calls recursively *Exists-Path(*$u, j, \lfloor l/2 \rfloor$*)* and *Exists-Path(j,* v*,* $\lfloor l/2 \rfloor$*).*
    - The recursion stack must store at most $O(\log n)$ states.
  - Let $T(n, l)$ be the time to determine if there is a path of l edges, where n is the number of nodes.
  - We have the recurrence relation:
    - $T(n, 1) = 1$
    - $T(n, l) = 2n \times T(n, \frac{l}{2})$
  - Resulting in $T(n, n) = (2n)^{\log n} = n^{1+\log n}$ for one test.
  - Because v is varying and we iterate l on the range $\{1, \ldots, n\}$, we have an overall performance of $O(n^{3+\log n})$.

Linear-Space Search

Procedure DAC-BFS
Input: Explicit problem graph G with n nodes and start node s
Output: Level of every node

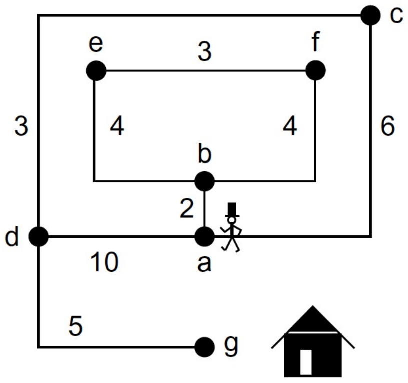
```
for each i in {1,...,n}                          ;; For all nodes i
   for each l in {1,...,n}                       ;; For all distances l
      if (Exists-Path(s,i,l))                    ;; If path of length l exists
         print (s,i,l); break                    ;; Output level and terminate
```

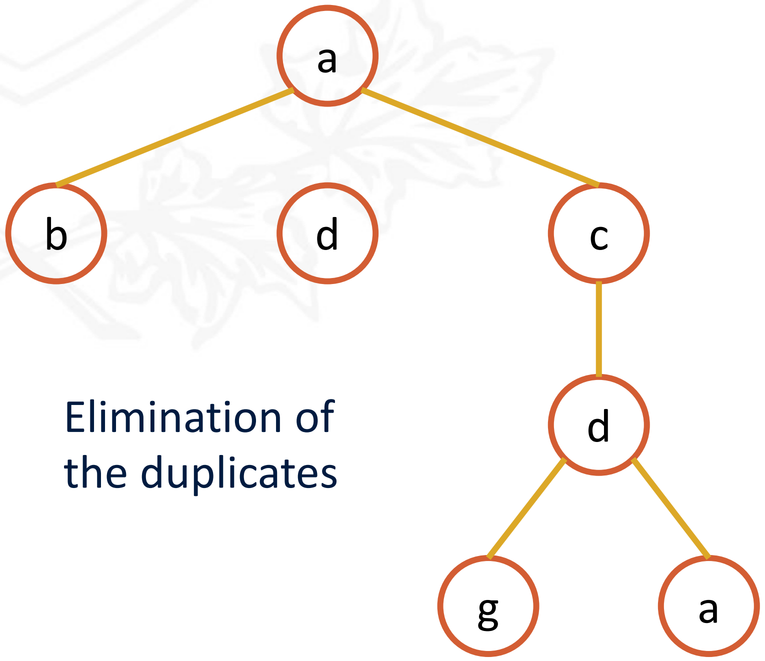
Procedure Exists-Path
Input: Nodes a and b, expected distance l between a and b
Output: Boolean, denoting if path of this length does exist

```
if (l = 1)                                       ;; If path has come down to one edge
   return ((a,b) ∈ E)                            ;; Feedback if edge between a and b exists
for each j in {1,...,n−1}                        ;; For all intermediate values
   if (Exists-Path(a,j,⌈l/2⌉) and Exists-Path(j,b,⌊l/2⌋))   ;; Recursive check
      return true                                ;; If both calls are successful, a path exists
return false                                     ;; No path possible
```

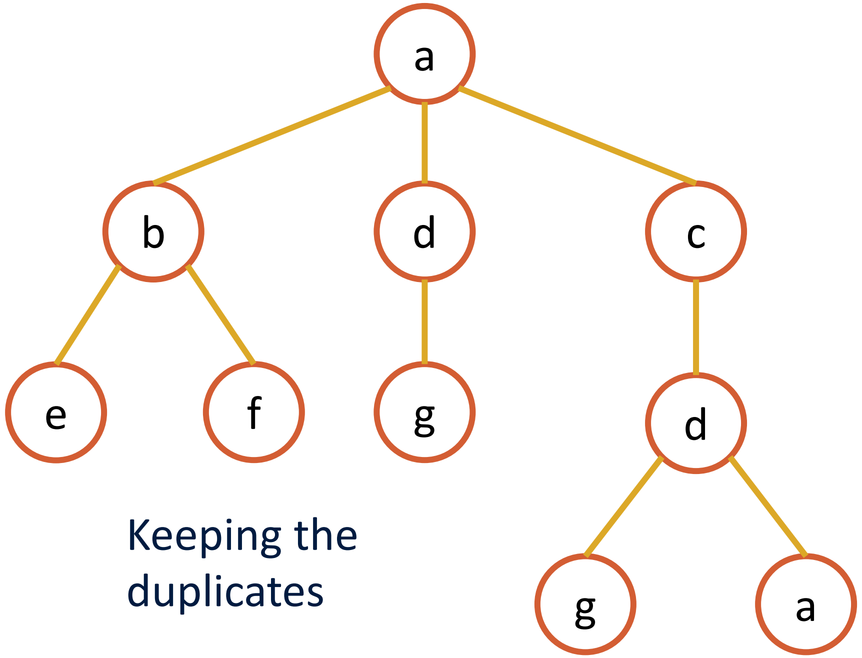
Search Tree

- We can transform the state space of a problem into a graph.
- The algorithms find the shortest path in a graph.
  - We are updating the search tree when we find a shortest path (duplicates).
- However, not every search algorithms eliminates duplicates.





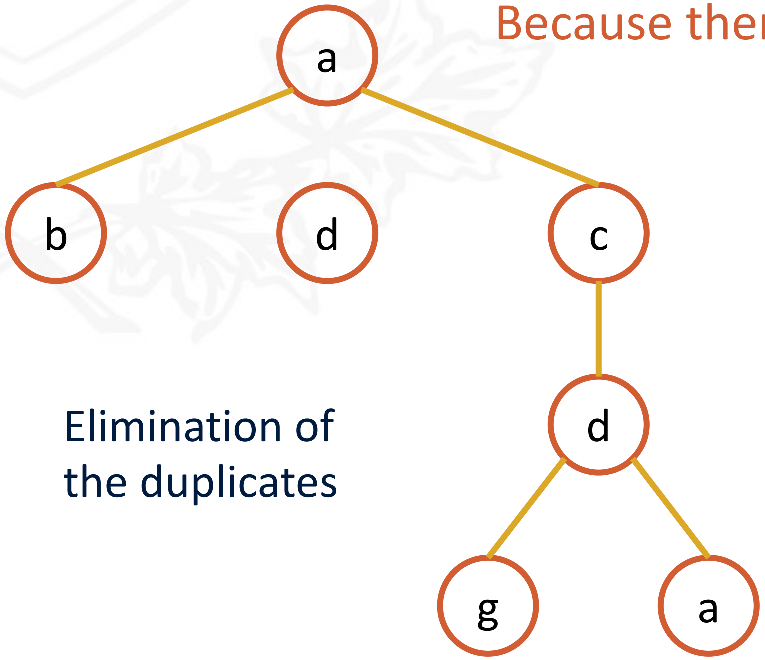
Elimination of the duplicates



Keeping the duplicates

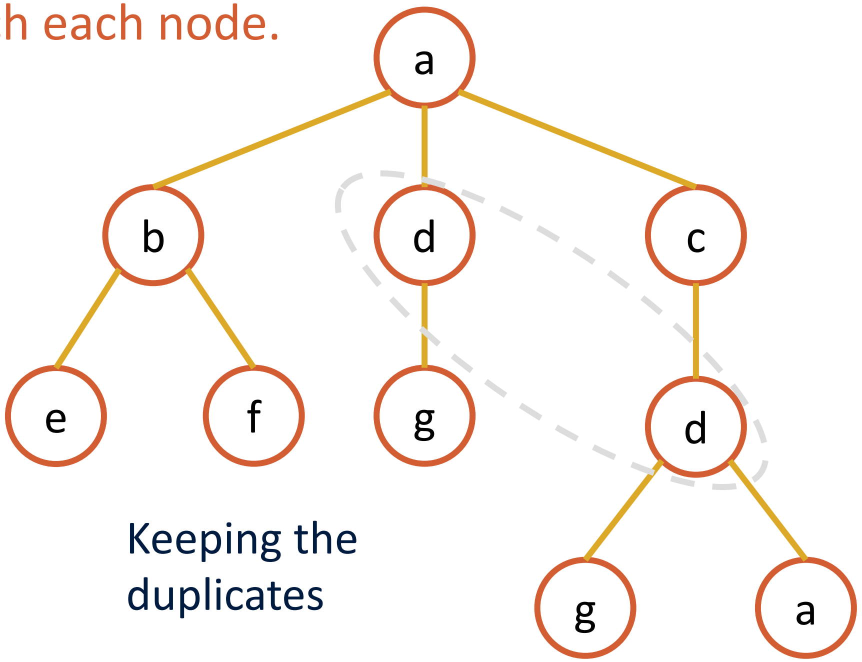
Search Tree

- We need to consider each nodes individually.
- It’s easier to consider a search tree starting from a root s than a graph.
  - Why?



Elimination of the duplicates

Because there is a unique path to reach each node.



Keeping the duplicates

Search Tree

- The element in the search space are paths.
- Recall that for A* the admissibility condition is:
  - $\delta(u, T) = \min\{\delta(u, t) | t \in T\} \geq 0, \forall u \in S$
- In search trees it becomes:
  - $\min\{w(q) | (p, q) \in \mathbf{T}\} \geq 0$
  - $w: \mathbf{S} \rightarrow \mathbf{T}$, where $\mathbf{S}$ is the search tree problem state space starting in s.
  - $\mathbf{T}$ the subset of paths that end in a goal node.

Search Tree

- Because we are considering paths the search tree grows very quickly.
- How can we solve this issue?
  - Pruning path that cannot be better than the current best solution.

- Branch-And-Bound is this type of algorithm.
  - Branching: expands interesting subproblems
  - Bounding: Ignore solutions that are outside some values.

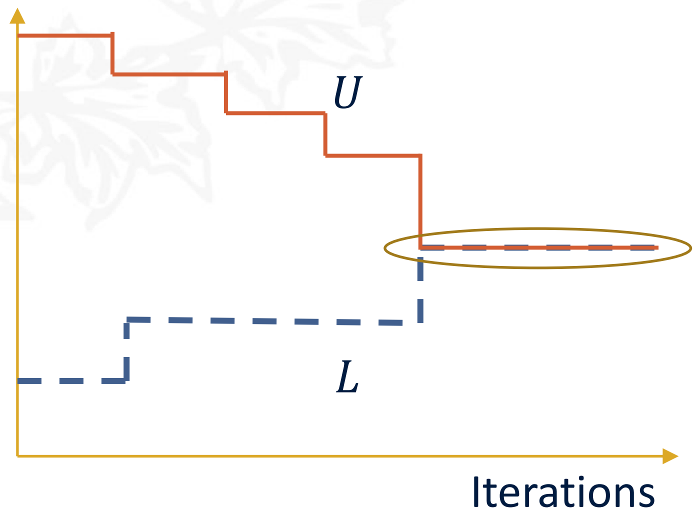
Branch-And-Bound

- Branch-And-Bound:
  - We extend DFS by applying Branch-And-Bound.
  - Maintain lower and upper bounds (L and U).
  - DFS will expand only the branch (partial path) that are inside the bounds.

- How would you calculate the lower bound?
  - By applying an admissible heuristic h
    - $L(u) = g(u) + h(u)$
- And for the upper bound?
  - Calculate a first solution (it can be greedy).

Branch-And-Bound

- Is the first solution obtained optimal?
  - No, like DFS the first solution is not optimal.
- Each time a goal is found we update the upper bound.



We stop when $L(u)$ is equal to U.

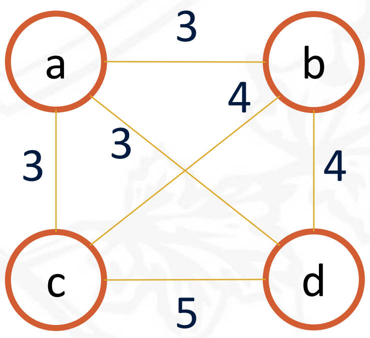
Branch and Bound

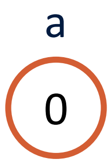
```
Function DFBNB(u, g, U):                                                    //Goal found
        if (Goal(u))
                if (g < U)                            // Improvement to currently shortest path
                        bestPath←Path(u)                          // Record solution path
                        U←g                                         // Update upper bound
        else                                                           // Nongoal node
                Succ(u)←Expand(u)                              // Generate successor set
                Let {v_0,…,v_n} be Succ(u), sorted according to h    // Optimize search order
                for each j in {1,…,n}
                        if (g + h(v_j) < U)                     // Apply upper bound pruning
                                DFBnB(v, g + w(u,v), U)
End Function

Initialize upper bound U;
bestPath ←∅;
DFBnB(s,0,U);
return bestPath;
```

Branch and Bound

- Example:

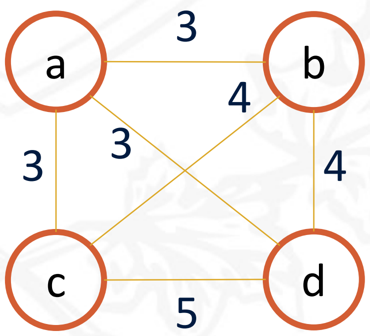


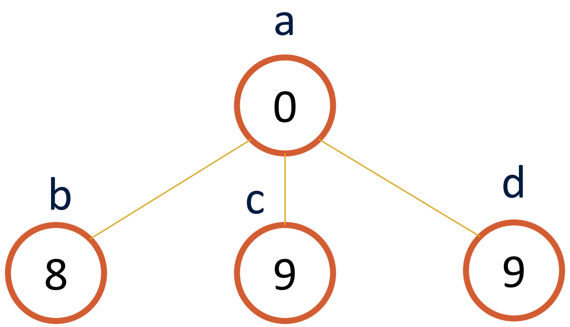


$U = \infty$

Branch and Bound

- Example:

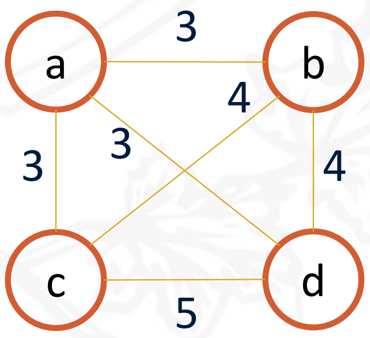


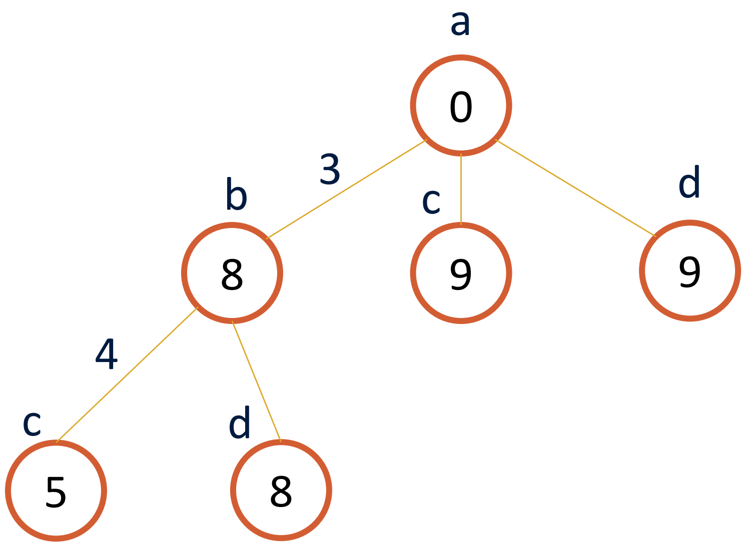


$$U = \infty$$

Branch and Bound

- Example:

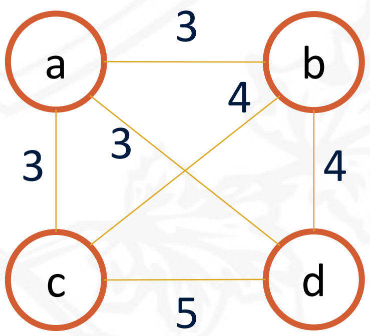


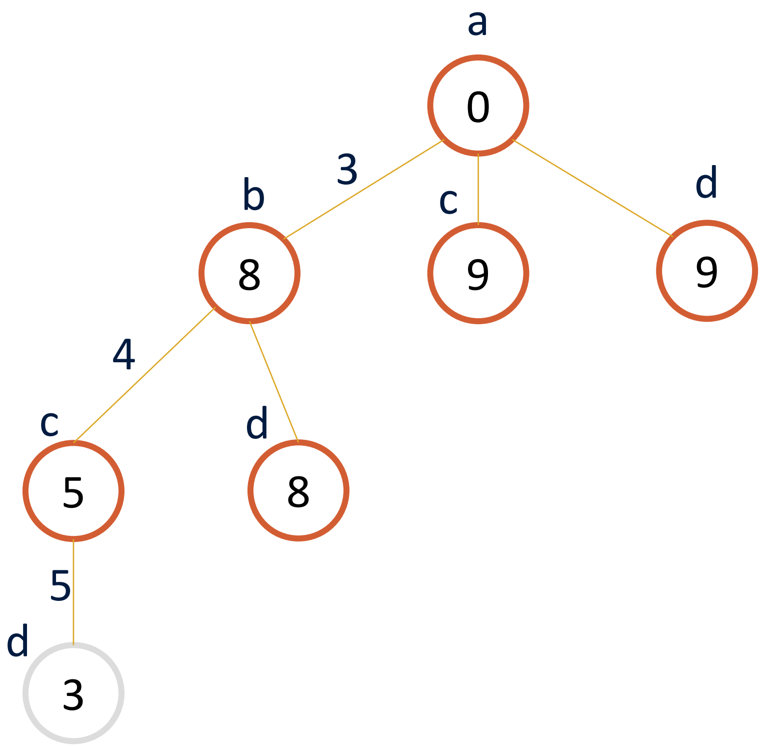


$U = \infty$

Branch and Bound

- Example:

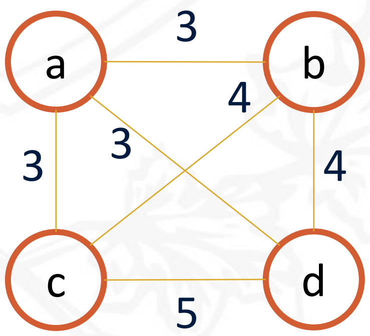


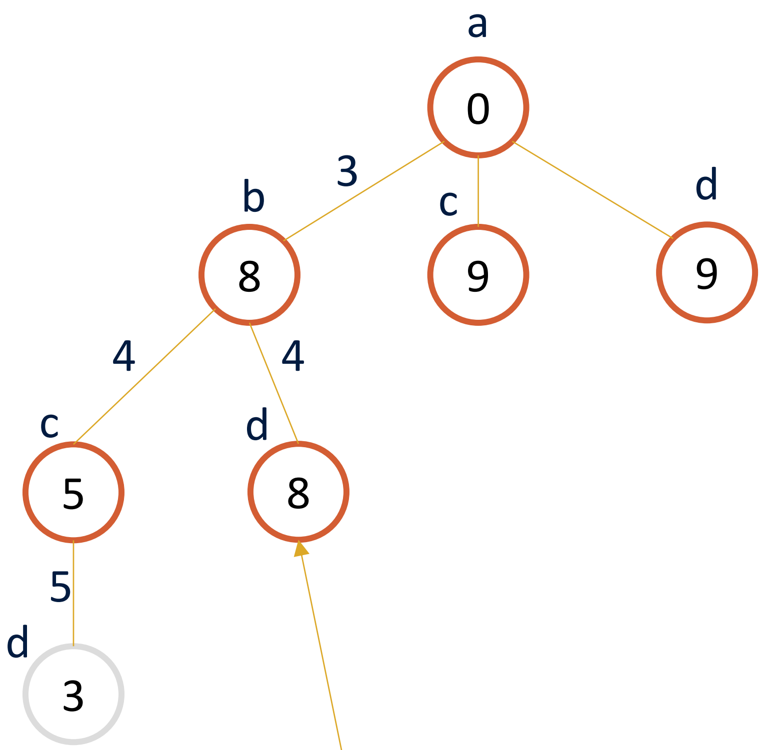


$U = 15$

Branch and Bound

- Example:



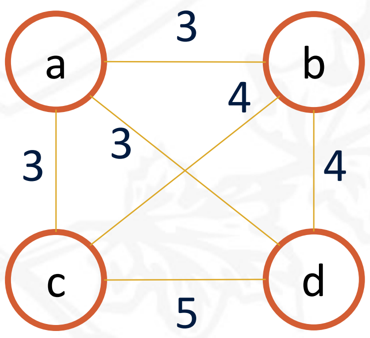


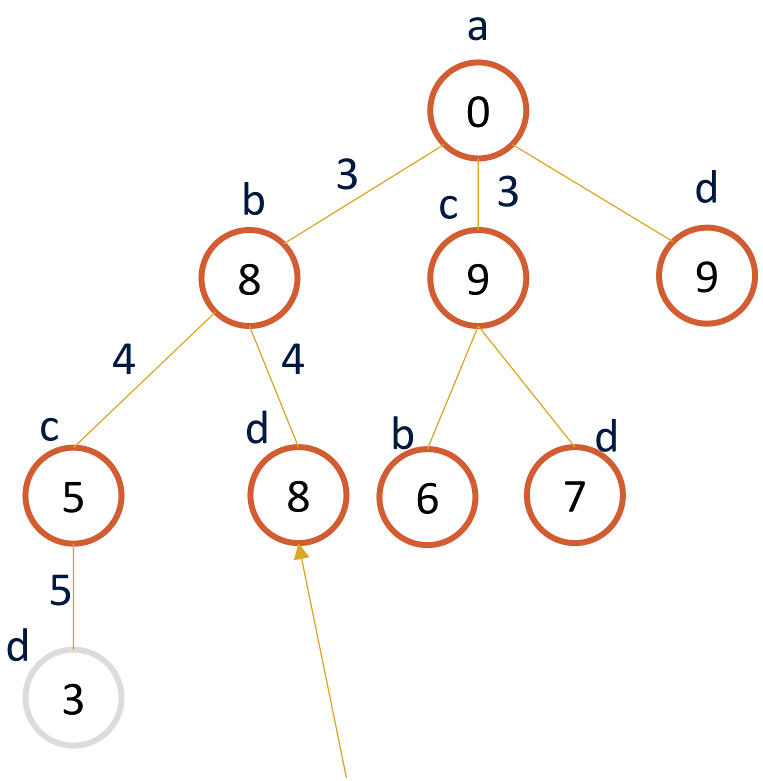
$U = 15$

$g + h(abd) = 15$
So, we don't explore.

Branch and Bound

- Example:



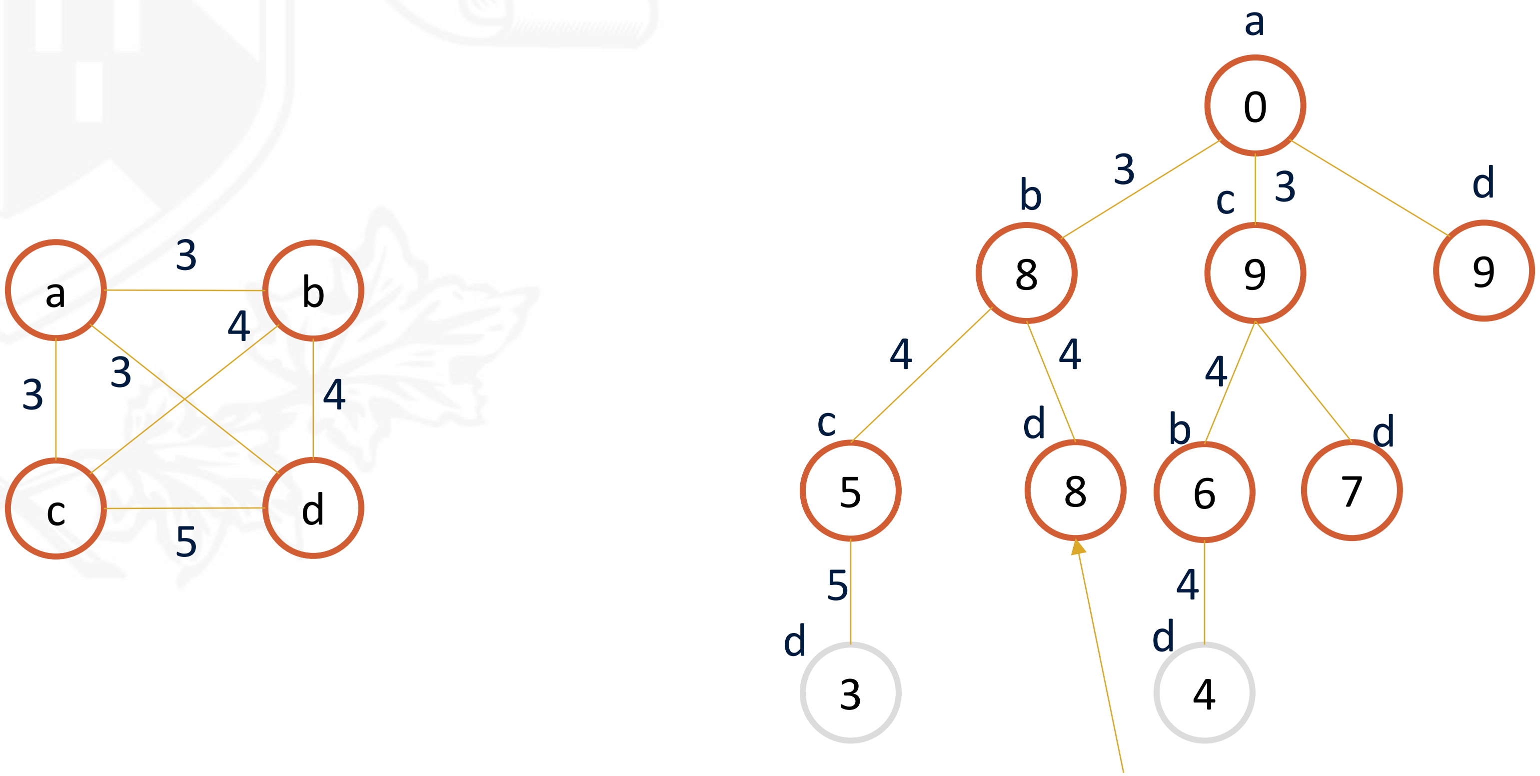


$U = 15$

$g + h(abd) = 15$
So, we don't explore.

Branch and Bound

- Example:

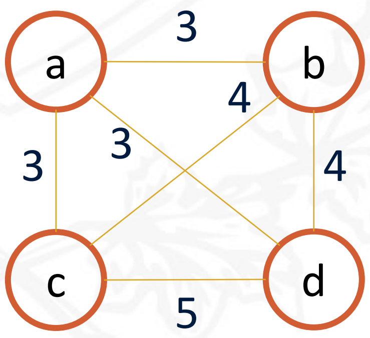


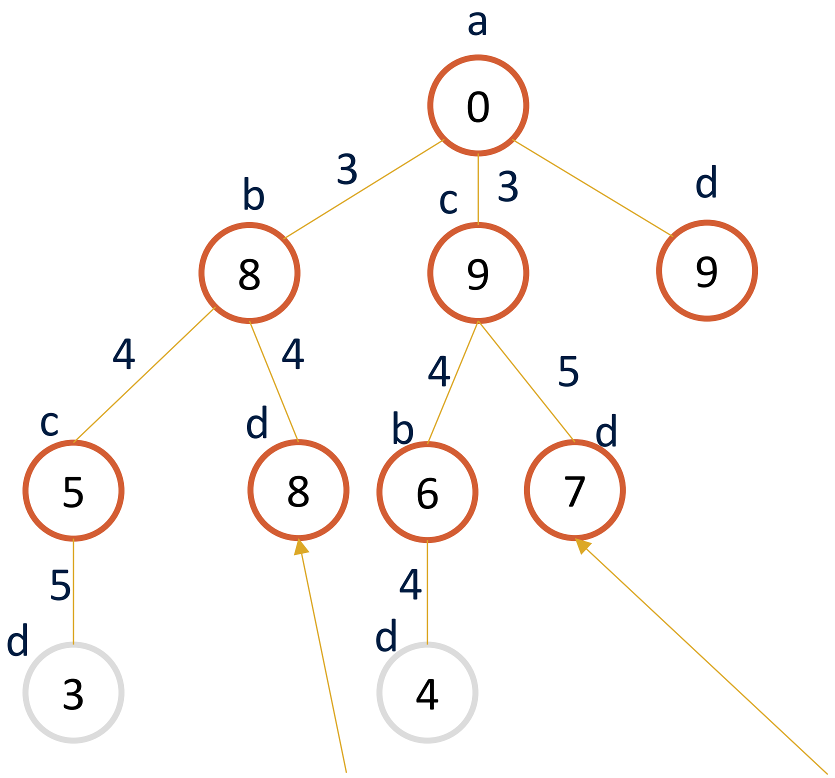
$U = 14$

$g + h(abd) = 15$
So, we don't explore.

Branch and Bound

- Example:





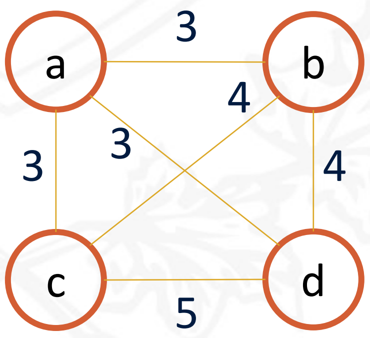
$U = 14$

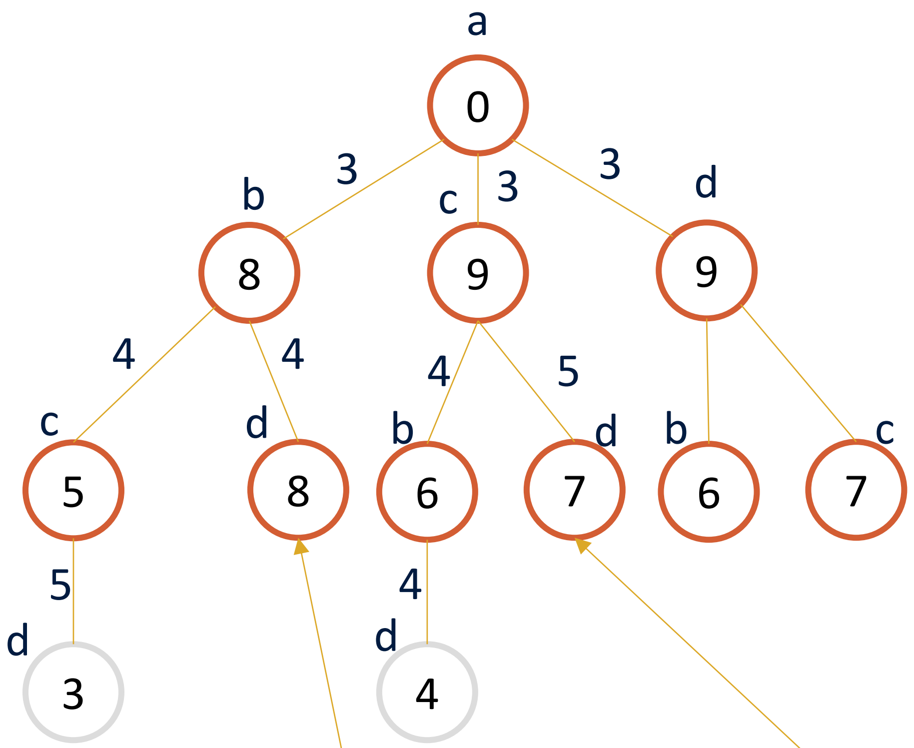
$g + h(abd) = 15$
So, we don't explore.

$g + h(acd) = 15$
So, we don't explore.

Branch and Bound

- Example:





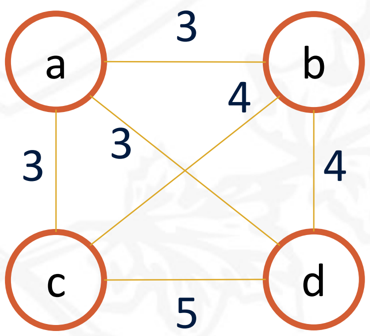
$U = 14$

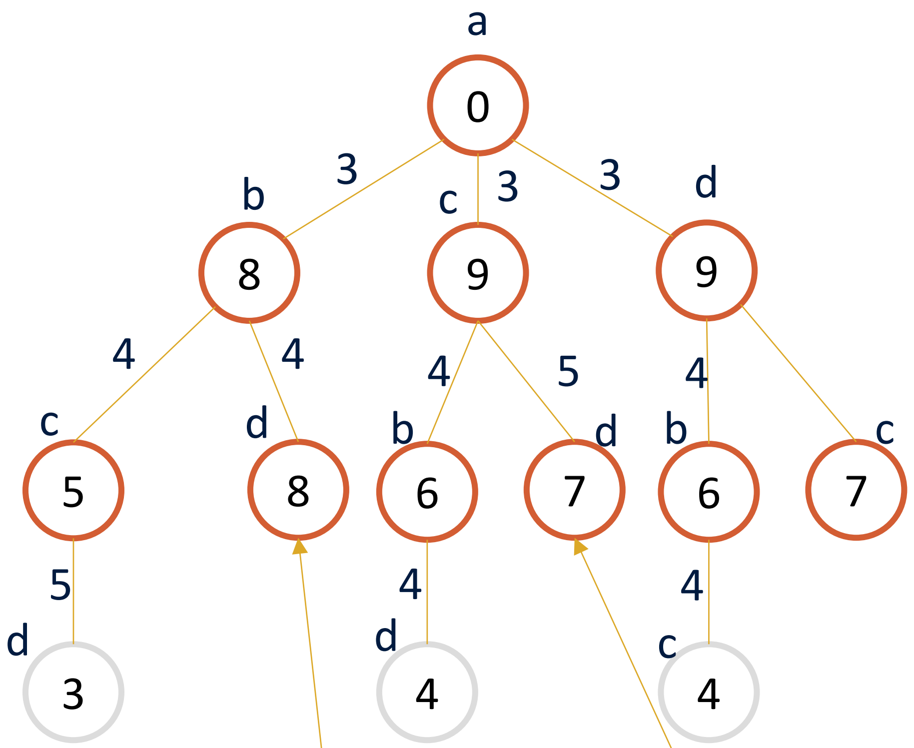
$g + h(abd) = 15$
So, we don't explore.

$g + h(acd) = 15$
So, we don't explore.

Branch and Bound

- Example:





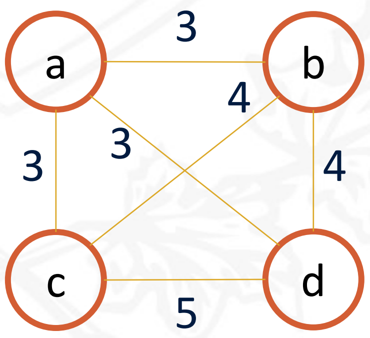
$U = 14$

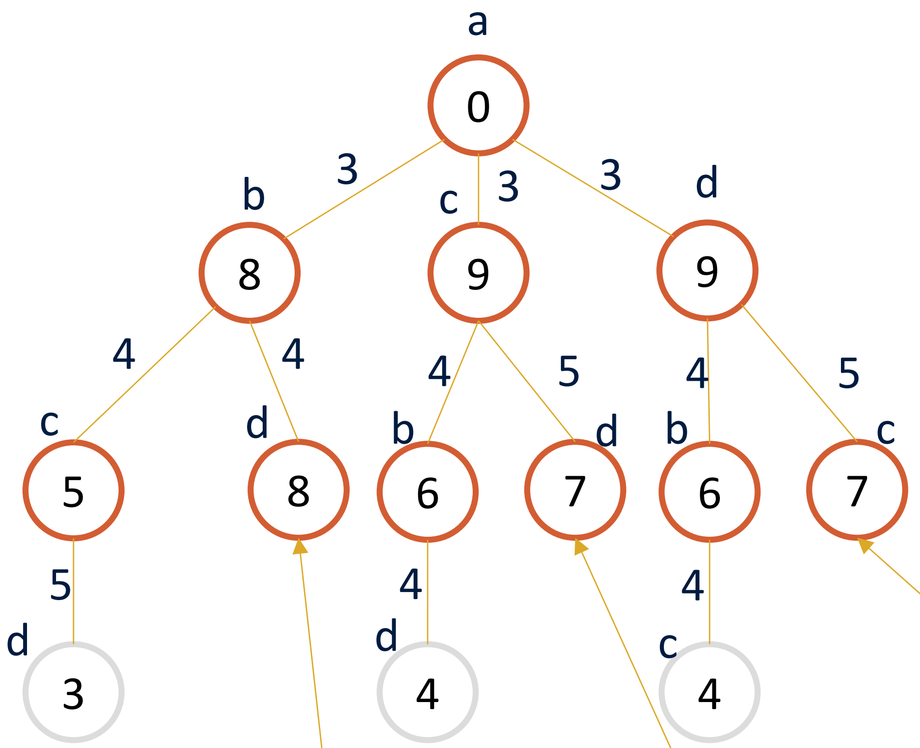
$g + h(abd) = 15$
So, we don’t explore.

$g + h(acd) = 15$
So, we don’t explore.

Branch and Bound

- Example:





$U = 14$

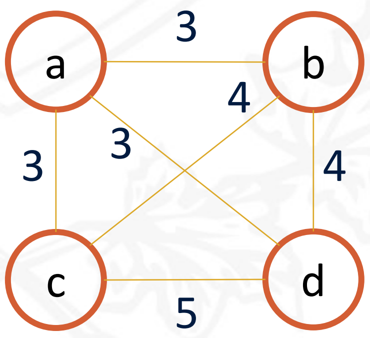
$g + h(abd) = 15$
So, we don't explore.

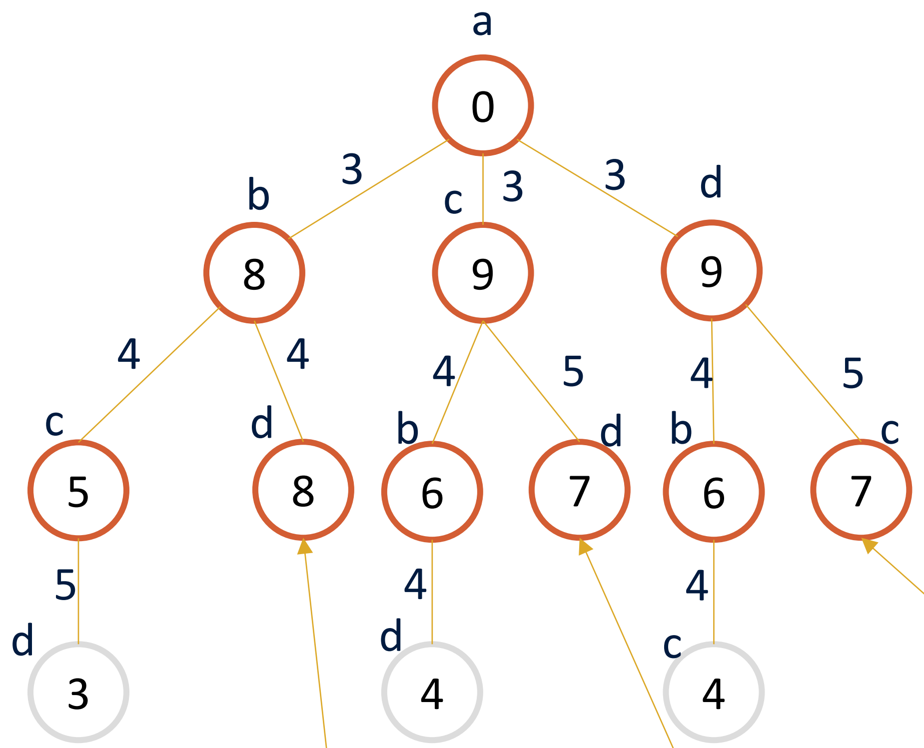
$g + h(acd) = 15$
So, we don't explore.

$g + h(adc) = 15$
So, we don't explore.

Branch and Bound

- Example:





$U = 14$

We don't explore every branch.

$g + h(abd) = 15$

So, we don't explore.

$g + h(acd) = 15$

So, we don't explore.

$g + h(adc) = 15$

So, we don't explore.

Branch and Bound

- Theorem (Optimality Depth-First Branch-and-Bound):
  - Algorithm depth-first branch-and-bound is optimal for admissible weight functions

- Proof:
  - If there is no pruning, every path will be explored, thus the optimal solution will be found.
  - Condition $L(v_j) < U$ confirms that the node's lower bound is smaller than the upper bound.
  - Otherwise, the branch is pruned, since admissible weight functions exploring the subtree cannot lead to better solutions.

Depth-First Iterative-Deepening

- The first solution of Depth-First Branch and Bound is not always optimal.
- If the heuristic bounds are weak it can lead to a complete search of the tree (DFS).
- To control these two points, we can use Depth-First Iterative-Deepening.
  - It combines BFS with a series of DFS
  - Optimality of BFS and the space complexity of DFS.

Depth-First Iterative-Deepening

```
Function DFID(u,g,U):
        if (Goal(u))    //Goal found
                return (u)
        Succ(u) ← Expand(u)
        call={}
        foreach v in Succ(u)
                if (g + w(u,v) ≤ U)
                        call ← (v, g + w(u,v))
                else if g + w(u,v) < U'
                        U' ← g + w(u,v)
         foreach (v,g) in call
                DFID(v,g,U)
End Function

U' ← 0
bestPath ← ∅;
While(bestPath=∅ and U' ≠ ∞)
        U ← U'
         U' ← ∞
        bestPath ← DFID(s,0,U)
return bestPath;
```

Depth-First Iterative-Deepening

- Example: