CS 6212 DESIGN AND ANALYSIS OF ALGORITHMS

LECTURE: BRANCH AND BOUND

Instructor: Abdou Youssef

OBJECTIVES OF THIS LECTURE

By the end of this lecture, you will be able to:

- Describe the all-powerful optimization design technique: Branch & Bound (B&B)
- Apply B&B to designing optimality-guaranteeing algorithms for new optimization problems by being able to
  - Derive valid approximate cost functions for new optimization problems, and
  - Prove the validity of approximate cost functions
- Prove why B&B guarantees optimality when the approximate cost function used by the algorithm satisfies certain conditions

OUTLINE

- Introduction: Universality of B&B, and when to use it or not use it
- Laying the ground word: Illustration on the Job Assignment Problem
- The general Branch and Bound algorithm
- Criteria for the choice of the approximate cost functions (ACF)
    - Proof of why criteria-satisfying ACFs guarantee optimality of B&B solutions
- Implementation of the B&B Job Assignment algorithm
- General rules of thumb for deriving valid ACFs

INTRODUCTION

- B&B is a systematic method for solving optimization problems
- B&B is a rather general optimization technique that applies where the greedy method and dynamic programming fail
- But it is much slower: Often takes exponential time in the worst case
  - So use it only if the greedy method and DP both fail to give you optimality
- However, if applied carefully, it runs <u>reasonably fast</u> <u>on average</u>.
- The general idea of B&B:
  - It is a **BFS-like search** for the optimal solution, in the solution space
  - But not all nodes get expanded (i.e., their children generated)
  - Rather, a carefully selected criterion determines <u>which</u> node to expand and <u>when</u>
  - And another criterion tells the algorithm when an optimal solution has been found

LAYING THE GROUND WORK
-- ILLUSTRATION ON THE JOB ASSIGNMENT PROBLEM --

- The Job Assignment Problem
- **Input**: n jobs, n employees, and an $n \times n$ matrix A where A_{ij} is the cost incurred if person i performs job j
- **Output**: A one-to-one matching f of the n employees to the n jobs so that the total cost is minimized.
  - Recall that a matching is a permutation.
- Cost C of a solution f is: $C(f) = \sum_{i=1}^{n} A_{i,f(i)}$
- That is, $C(f) = A_{1,f(1)} + A_{2,f(2)} + \cdots + A_{n,f(n)}$

THE JOB ASSIGNMENT PROBLEM
-- AN EXAMPLE --

- Example: $n = 3$ (i.e., 3 employees and 3 jobs) and the cost matrix

$$A = \begin{pmatrix} 2 & 4 & 5 \\ 2 & 7 & 10 \\ 5 & 3 & 7 \end{pmatrix}$$

(Recall the matrix notation:, $A_{31} = 5, A_{23} = 10$ ...)

- A solution (i.e., permutation) $f = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{pmatrix}$ means assigning

employee 1 -> job 2, employee 2 -> job 1, and employee 3 -> job 3.

- A solution is like selecting in A 3 numbers:
  - One from each row
  - No two numbers are in same column

$$A = \begin{pmatrix} 2 & 4 & 5 \\ 2 & 7 & 10 \\ 5 & 3 & 7 \end{pmatrix}$$

$C(f)=4+2+7=13$

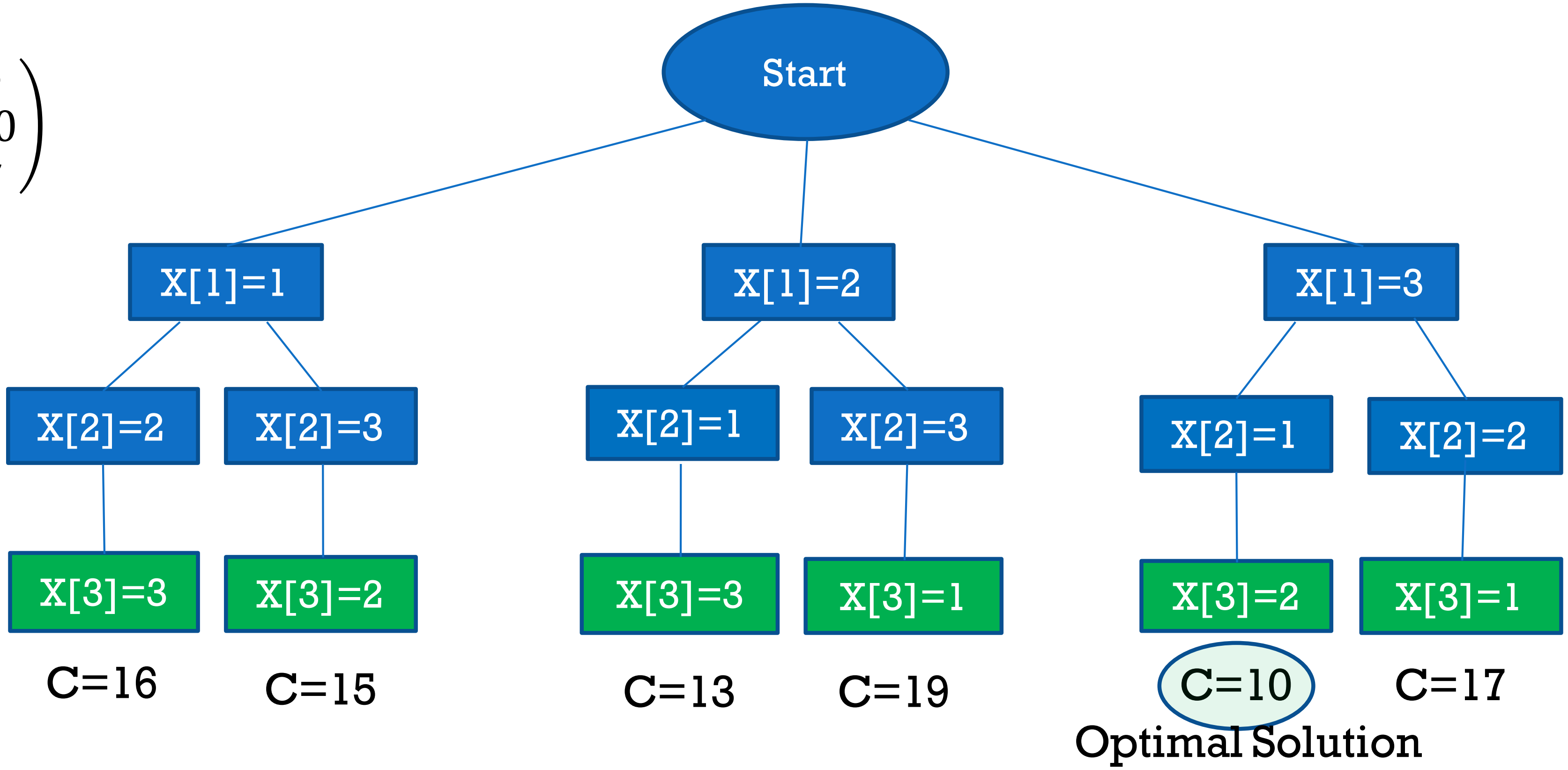
THE JOB ASSIGNMENT PROBLEM
-- BRUTE-FORCE METHOD --

- To start, we will develop a brute-force method:
  - Which generates the whole solution tree, where every path from the root to any leaf is a solution,
  - Then we will evaluate the Cost C of each solution, and
  - Finally choose the path with the minimum cost.

THE JOB ASSIGNMENT PROBLEM
-- BRUTE-FORCE METHOD ON THE EXAMPLE -

$n = 3$

$$A = \begin{pmatrix} 2 & 4 & 5 \\ 2 & 7 & 10 \\ 5 & 3 & 7 \end{pmatrix}$$



- Represent each permutation as X[1:3] like in Backtracking
- Generate the same solution tree (as in backtracking) but in BFS order
- Don’t show dead-ends or invalid solutions

WHAT IS WRONG WITH BRUTE FORCE?

- Too costly
- n! solutions
- For large n, n! is too huge and will take too much time
- B&B eliminates unpromising solutions early on
- We'll see how

MAIN IDEA
-- USING A PREDICTOR --

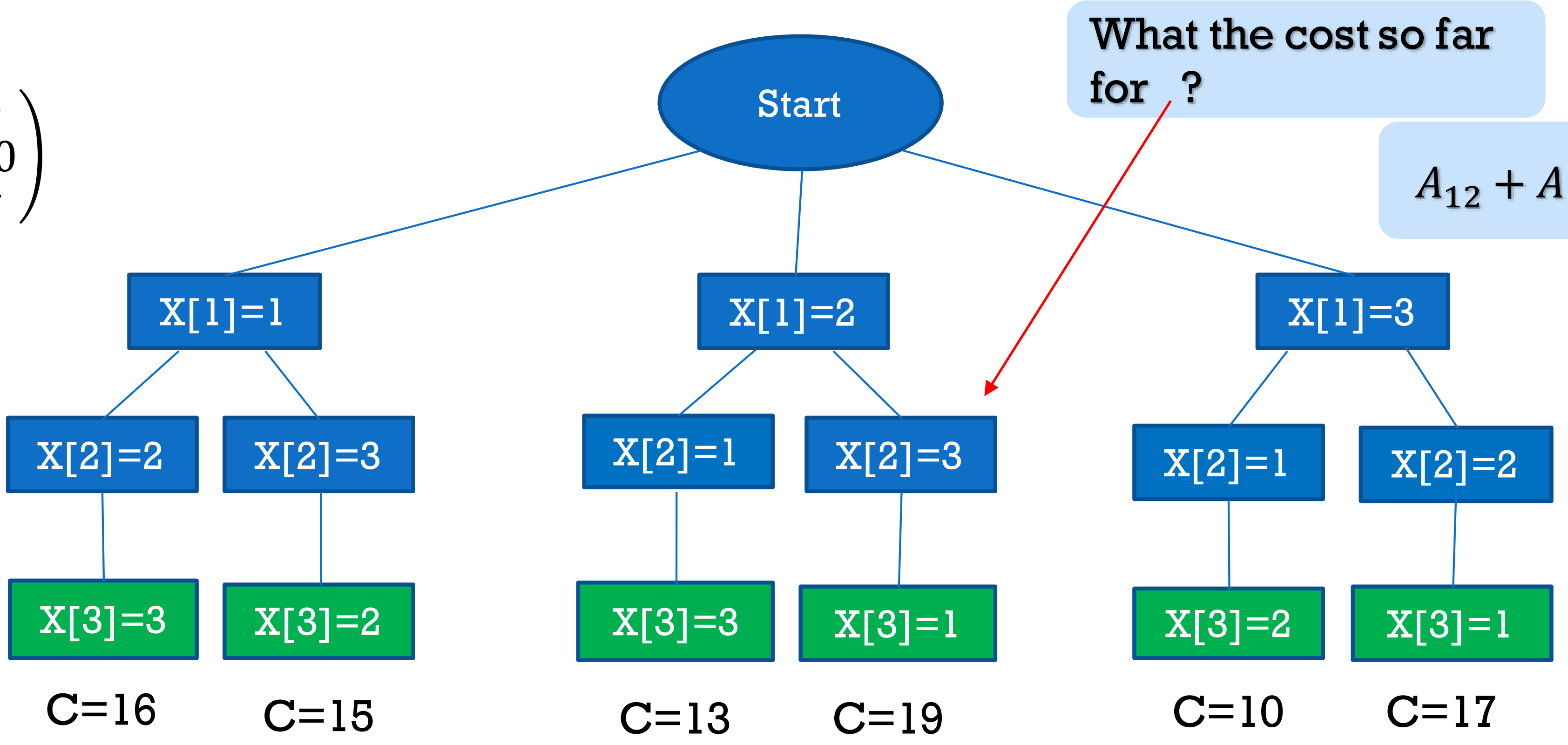
- The first idea of B&B is to develop a quantitative "predictor" of the likelihood of a node (in the solution tree) that it will lead to an optimal solution
- We denote the predictor for a node N as $\hat{C}(N)$
- What could that predictor be? (in **minimization** problems)
  - One candidate predictor is: **the cost so far**
  - Each tree node corresponds to a (partial) solution (from the root to that node)
  - The cost-so-far predictor is the cost of the partial solution so far

THE JOB ASSIGNMENT PROBLEM

-- PREDICTOR ILLUSTRATION: COST SO FAR --

$n = 3$

$$A = \begin{pmatrix} 2 & 4 & 5 \\ 2 & 7 & 10 \\ 5 & 3 & 7 \end{pmatrix}$$



$$A_{12} + A_{23} = 4 + 10 = 14$$

MAIN IDEA
-- HOW DOES B&B USE THE PREDICTOR --

- Instead of a blind bread-first search order of generating nodes
  - B&B chooses the *live node* with the best predictor value
  - B&B simply expands that node (i.e., generate all its children)

Live node = temporary leaf

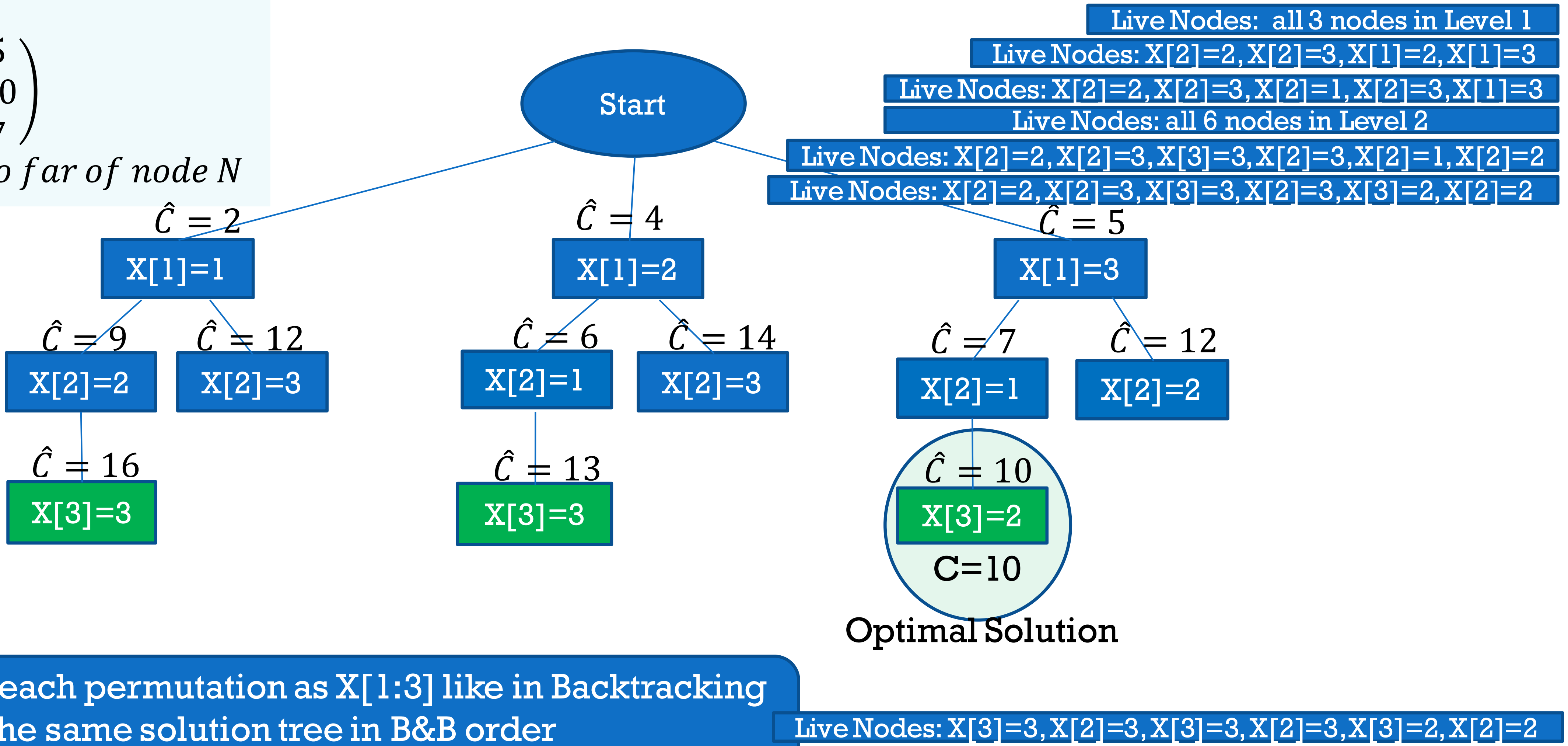
- The predictor value of each newly generated node is computed
- Termination criterion:
  - When the best live node chosen for expansion turns out to be a final leaf (i.e., at level n), the algorithm terminates
  - That node corresponds to the optimal solution.
- The proof of optimality will be presented later on.

BB APPLIED ON THE EXAMPLE

$n = 3$

$$A = \begin{pmatrix} 2 & 4 & 5 \\ 2 & 7 & 10 \\ 5 & 3 & 7 \end{pmatrix}$$

$\hat{C}(N) = cost\ so\ far\ of\ node\ N$



- Represent each permutation as X[1:3] like in Backtracking
- Generate the same solution tree in B&B order
- Show $\hat{C}(N)$ for each generated node N

TERMINOLOGY

- A **live node**: a temporary leaf, i.e., a node whose children have not been generated
- A **dead node**: a node whose children have been generated
- **Expanding node** (or **E-node**): The node selected to be expanded, i.e., the live node with best predictor value
- An **answer node**: a node that corresponds to a complete solution (i.e., a node at the bottom level)
- A predictor is referred to as ***approximate cost function***, and denoted $\hat{C}$

LESSONS LEARNED SO FAR

- B&B searches the solution space (tree) in a BSF-like order
- To speed up the search, it uses a predictor (the approximate cost function) to estimate how likely a tree node will lead to an optimal solution, and uses the predictor to know which node to expand next
- The cost so far predictor is OK, leads to some savings, but better predictors are needed
- More lessons to come

OBSERVATIONS

- B&B "seems" to find the optimal solution
- B&B with the cost-so-far $\hat{C}$ does save on the solution tree
- But the savings are not impressive
- Can it be made faster (i.e., more savings/bounding) if we use a better $\hat{C}$?
- Other questions that will be addressed a little later:
  - Would all $\hat{C}$ work (i.e., lead to an optimal solution)?
  - If not, how would we know which $\hat{C}$ works?
  - How would we know which $\hat{C}$ is better than another?

A BETTER $\hat{C}$

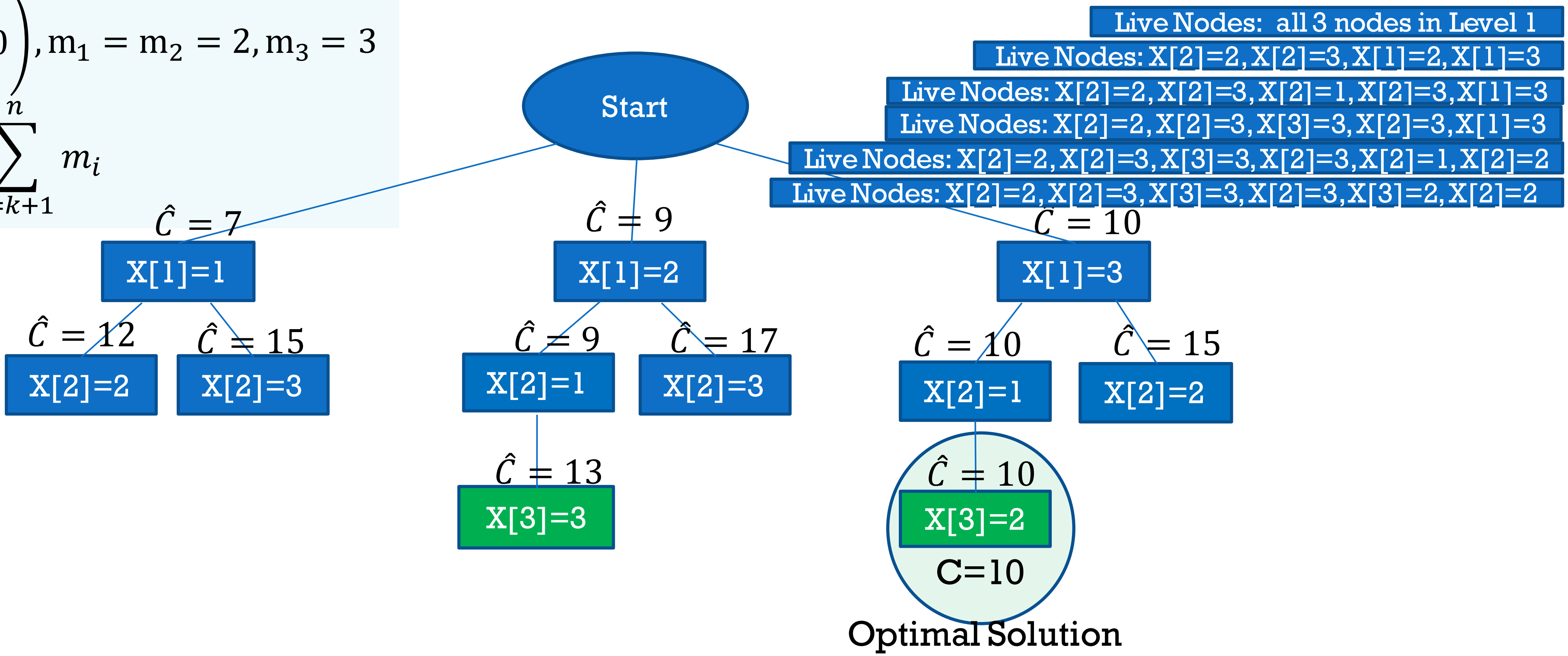
- The previous $\hat{C}$ relies entirely on "past" performance to predict future performance
- While the past is usually a good indicator, using additional knowledge about the situation can improve predictions (make fewer errors)
- Take $\hat{C}(N) =$ cost so far $+\sum_{i=k+1}^{n} m_i$, where k is the level of node N, and m_i is the minimum of row i
- In the example, $m_1 = 2, m_2 = 2, m_3 = 3$

BB APPLIED ON THE EXAMPLE

-- USING THE SECOND $\hat{C}$ --

$$A = \begin{pmatrix} 2 & 4 & 5 \\ 2 & 7 & 10 \\ 5 & 3 & 7 \end{pmatrix}, m_1 = m_2 = 2, m_3 = 3$$

$$\hat{C}(N) = \text{csf} + \sum_{i=k+1}^{n} m_i$$



- Represent each permutation as X[1:3] like in Backtracking
- Generate the same solution tree in B&B order
- Show $\hat{C}(N)$ for each generated node N

OBSERVATIONS

- B&B found the optimal solution faster with the 2nd $\hat{C}$
- That indicates that there can be better $\hat{C}$'s
- Can it be made faster with a better $\hat{C}$?
- Yes: Observe that the 2nd $\hat{C}$ ignored column conflicts in its m_i's
- Let's create a 3rd $\hat{C}$ that doesn't ignore column conflicts:
- Take $\hat{C}(N) = \text{cost so far} + \sum_{i=k+1}^{n} p_i$, where
  - k is the level of node N,
  - p_i is the minimum of row i such that p_i is not in the column of any of the terms chosen in the partial solution up to node N

BB APPLIED ON THE EXAMPLE

-- USING THE THIRD $\hat{C}$ --

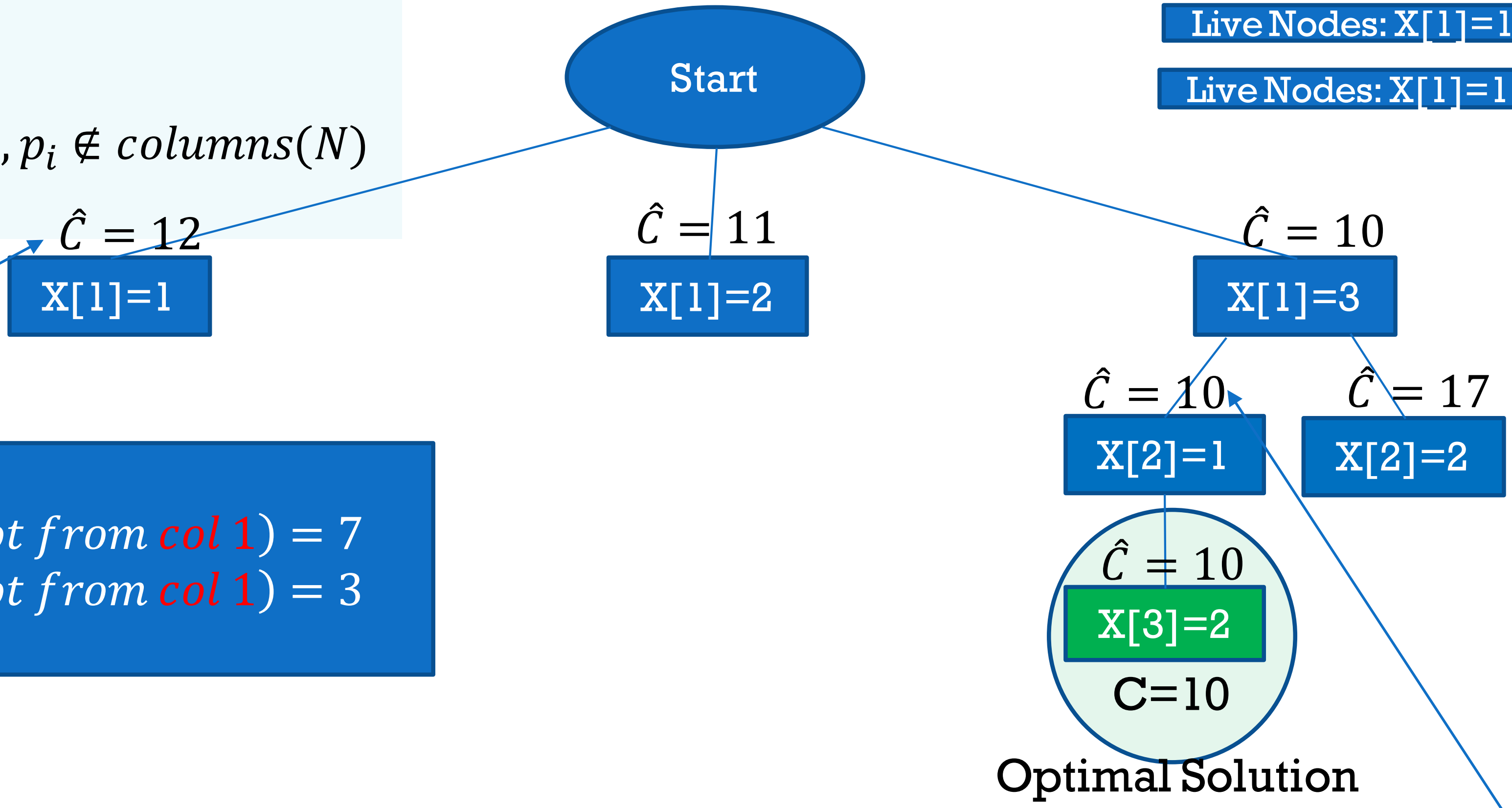
$$A = \begin{pmatrix} 2 & 4 & 5 \\ 2 & 7 & 10 \\ 5 & 3 & 7 \end{pmatrix},$$

$$\hat{C}(N) = \text{csf} + \sum_{i=k+1}^{n} p_i, p_i \notin columns(N)$$

Live Nodes: all 3 nodes in Level 1

Live Nodes: X[1]=1, X[1]=2, X[2]=1, X[2]=2

Live Nodes: X[1]=1, X[1]=2, X[3]=2, X[2]=2



$\hat{C} = A_{11} + p_2 + p_3$
$p_2 = \min(row\ 2\ not\ from\ col\ 1) = 7$
$p_3 = \min(row\ 3\ not\ from\ col\ 1) = 3$
$\hat{C}$ =2+7+3=12

$\hat{C} = A_{13} + A_{21} + p_3$
$p_3 = \min(row\ 3\ not\ from\ cols\ 3\ and\ 1) = 3$
$\hat{C}$ =5+2+3=10

- Represent each permutation as X[1:3] like in Backtracking
- Generate the same solution tree in B&B order
- Show $\hat{C}(N)$ for each generated node N

OBSERVATIONS

- B&B found the optimal solution much faster with the 3rd $\hat{C}$
- That is further evidence that better $\hat{C}$'s get us to optimal solution faster
- The more "compliant" with the constraints $\hat{C}$ is, the better it seems to be

LESSONS LEARNED SO FAR

- B&B searches the solution space (tree) in a BSF-like order
- To speed up the search, it uses a predictor to estimate how likely a tree node will lead to an optimal solution, and uses the predictor to know which node to expand next
- The cost-so-far predictor is OK
- There can be many predictors: some better than others
- Predictors need not comply with the constraints of the solution
- But the closer a predictor is to compliance, the faster it gets us to the optimal solution
- More lessons to come

THE GENERAL B&B ALGORITHM
-- SOME POINTS TO KEEP IN MIND FIRST --

- Each solution of the problem is assumed to be expressible as an array X[1:n] (as was seen in Backtracking).

- An approximate cost function $\hat{C}$ is assumed to have been defined

THE GENERAL B&B ALGORITHM
-- THE PSEUDO-CODE--

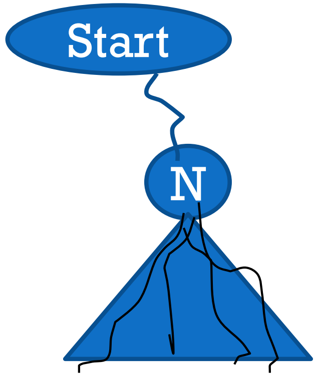
```
Procedure B&B()
begin
    E: nodepointer;
    E := new (node); // the root (start) node
    H: heap;  //A heap for all the live nodes
    while (true) do
        if (E is a final leaf) then
            // E is an optimal solution
            print out the path from E to root;
            return;
        endif
        Expand(E); // E is not an answer node
        if (H is empty) then
            report that there is no solution;
            return;
        endif
        E := delete-min(H); // next E-node
    endwhile
end
```

```
Procedure Expand(E)
begin
    1.  Generate all the children of E;
    2.  Compute the Ĉ of each child;
    3.  Insert each child into the heap H;
end
```

- The specifics of Expand(E) vary from problem to problem, and depend on your choice of $\hat{C}$
- The heap is a min-heap for minimization problems, but a max-heap for maximization problems (more on maximization later)

CRITERIA FOR THE CHOICE OF THE APPROXIMATE COST FUNCTIONS $\hat{C}$ (1/2)

- **Definition of the cost function** C: For every node N in the solution tree, the ***cost function*** $C(N)$ is the cost of the best solution that goes through node N.



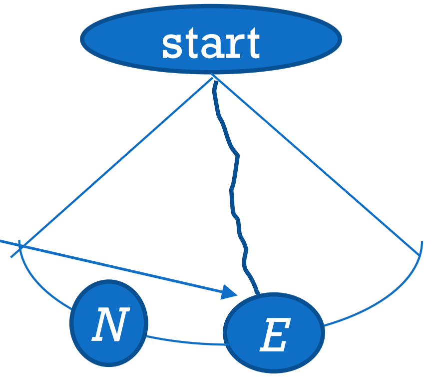
- Notes:
  - Be careful to distinguish between the "cost function" C and the "approximate cost function" $\hat{C}$
  - C(N) will not be computed. It is only a theoretical, mathematical quantity to be used for analysis, proof, and "inspiration" for deriving good $\hat{C}$'s

CRITERIA FOR THE CHOICE OF THE APPROXIMATE COST FUNCTIONS $\hat{C}$ (2/2)

- **Theorem**: In the case of minimization problems, if $\hat{C}$ satisfies the following two validity criteria:
  1) $\hat{C}(N) \le C(N)$ for every node N, and
  2) $\hat{C}(N) = C(N)$ for every answer node (final leaf) node N,

  then the first expanding node (best-$\hat{C}$ node) that happens to be a final leaf corresponds to an optimal solution.

Since $\hat{C}(N) \le C(N)$, we say that $\hat{C}$ is an ***underestimate*** of C



- **Proof**:
  a. Assume $\hat{C}$ satisfies the two conditions of the theorem
  b. Let E be the E-node that happens to be a final leaf (where B&B algorithm stops)
  c. Need to prove that $\boldsymbol{C(E) \le C(N)}$ for every live node N
  d. $C(E) = \hat{C}(E)$ by condition (2) of the theorem and because E is a final leaf
  e. $\hat{C}(E) \le \hat{C}(N)$ for every live node N, because E is the expanding node, that is, the minimum-$\hat{C}$ node at the moment it is chosen (recall that the algorithm chooses E to be the the minimum-$\hat{C}$ node)
  f. $\hat{C}(N) \le C(N)$ for every node N (and thus for every live node) by condition (1) of the theorem
  g. Therefore, $C(E) = \hat{C}(E) \le \hat{C}(N) \le C(N)$ for every live node N, (by *d, e* and *f*, respectively)
  h. Therefore, $\boldsymbol{C(E) \le C(N)}$. Q.E.D.

OPTIMALITY OF THE B&B SOLUTION

- If $\hat{C}$ satisfies the following two validity criteria, then the B&B algorithm solution is optimal
- That is because it returns the solution corresponding to the first expanding node (best-$\hat{C}$ node) that happens to be a final leaf
  - By last theorem, that solution is optimal
- Therefore, to design an optimality-guaranteeing B&B algorithm, simply derive and use in the B&B algorithm a $\hat{C}$ that satisfies the two validity criteria
- So, the crux of the design process is
  - the derivation of a $\hat{C}$, and
  - the proof that it satisfies the two validity criteria

THE COST FUNCTION *C(N)* OF THE JOB ASSIGNMENT PROBLEM

- Let N be a node at level k, and $\mathrm{X}[1{:}\,k]$ be the first k entries assigned on the path from the root to node N in the solution tree
- $C(N) =$ the cost of the best (min) solution that goes through node N
- $C(N) =$ cost so far + the cost of the best (min) continuation from N
- $C(N) = \sum_{i=1}^{k} A_{i,X[i]} +$

  $$\min\{\sum_{i=k+1}^{n} A_{i,X[i]} \mid \text{for all possible valid fillings of } \mathrm{X}[k+1{:}\,n]\}$$
- Computing the min in the 2nd part of $C(N)$ is very costly because the number of entities to minimize over is exponential
- But, in any case, the B&B algorithm doesn't use $C(N)$

EXERCISES

- **Exercise 1**: Prove that the first $\hat{C}$ we defined for the Job Assignment Problem satisfies the two conditions of the theorem.
- **Exercise 2**: Prove that the second $\hat{C}$ we defined for the Job Assignment Problem satisfies the two conditions of the theorem.
- **Exercise 3**: Prove that the third $\hat{C}$ we defined for the Job Assignment Problem satisfies the two conditions of the theorem.
- Hint: Use the expression of $C(N)$ in the previous slide

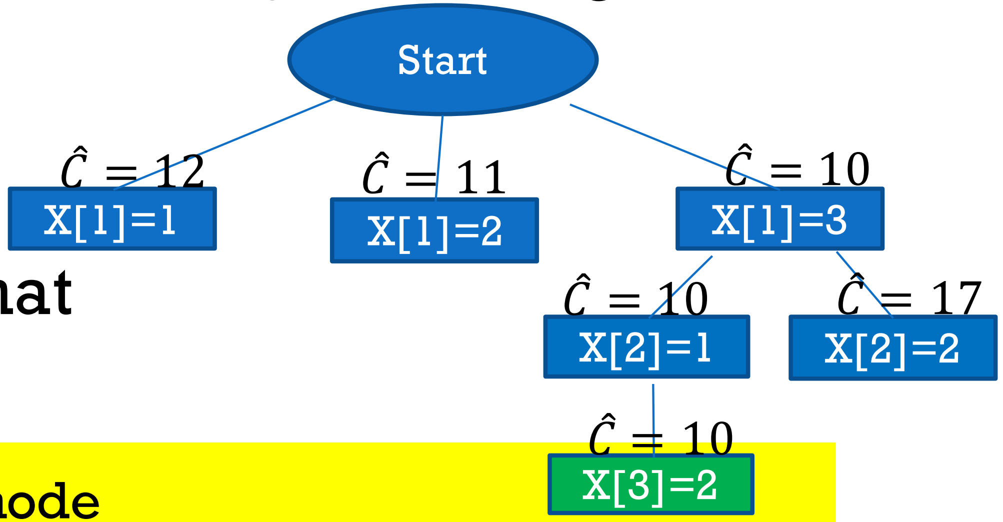
IMPLEMENTATION OF THE B&B JOB ASSIGNMENT ALGORITHM (1)

- We saw that the B&B algorithm needs an Expand procedure that depends on the problem and on the choice of the $\hat{C}$
- We need to
  - define the full record of a node, and
  - fully implement the Expand procedure

```
Procedure Expand(E)
begin
    1. Generate all the children of E;
    2. Compute the Ĉ of each child;
    3. Insert each child into the heap H;
end
```

IMPLEMENTATION OF THE B&B JOB ASSIGNMENT ALGORITHM (2)

- Node record:
  - Every node corresponds to something like X[i]=j, which signifies that employee i is assigned to job j
  - Every node must store its $\hat{C}$ value
  - Every node must point to its parent so that
    - when an optimal leaf is found, the path from that leaf to the root can be traced and printed out as the optimal solution




```
Record node
begin
      parent: nodepointer;
      i: integer; // employee i;  X[i]=j
      j: integer; // job j assigned to person i
      cHat: real;
end
```

IMPLEMENTATION OF THE B&B JOB ASSIGNMENT ALGORITHM (3)

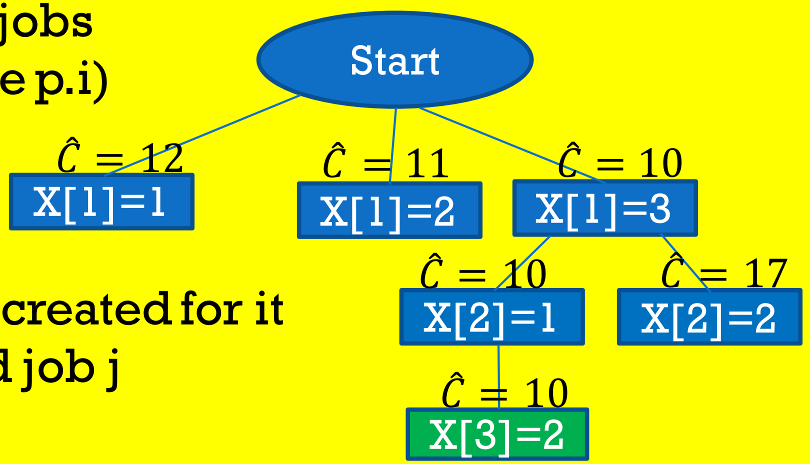
- Let's use the second $\hat{C}$
  - $\hat{C}(N) =$ cost so far $+\sum_{i=k+1}^{n} m_i$, where k is the level of node N, and m_i is the minimum of row i of the cost matrix A
- Observe that if N is a pointer to a node, then

$$N.cHat = N.parent.cHat + A[N.i, N.j] - m_{N.i}$$

- It should be easy to write a piece of code that finds the minimum m_i for row i, for all i (that is a little exercise for you)
- So assume we have the m_i's

IMPLEMENTATION OF THE B&B JOB ASSIGNMENT ALGORITHM (4)

```
Procedure Expand(in: E, A[1:n, 1:n], m[1:n]; in/out: H)   // m is assumed computed; H is the heap
begin
    S[1:n]: Boolean;   // S is a bitmap set initialized to 0. It'll contain all the jobs that have been
                       // assigned by the partial path from the root to E
    N,  p: nodepointer;
    p := E;
    while (p is not the root) do  // walk from E to root finding the assigned jobs
        S[p.j] := 1;           // job p.j has already been assigned (to employee p.i)
        p := p.parent;
    endwhile
    for j=1 to n do
        if S[j] = 0 then  // job j unassigned, and so a new child of E will be created for it
            N := new (node);       // a new child node for person E.i+1 and job j
            N.i := E.i + 1;           N.j := j;           N.parent:= E;
            N.cHat = E.cHat + A[N.i,N.j]  -  m[N.i];
            Insert(N,H); // insert N into heap H
        endif
    endfor
end
```



EXERCISES

- **Exercise 4**: Give an Expand procedure for the third $\hat{C}$ of the Job Assignment problem

HOW TO FIND A GOOD $\hat{C}$

-- A RULE OF THUMB --

1. Express $C(N)$ mathematically (recall that $C(N)$ is the cost of the min-cost constraints-compliant solution that goes through N)
2. Relax some of the constraints (as little as possible) for the continuation of the solution from node N, and compute a new $C(N)$ under the new relaxed constraints
3. Take $\hat{C}$ to be the new $C(N)$

- Notes:
  - You need to relax the constraints enough so that the new $C(N)$ can be computed fast
  - But don't relax too much because the closer $\hat{C}$ is to the original C, the fewer nodes need to be generated in the solution tree, and vice versa
  - Because $C(N)$ is the min-cost under tighter constraints, and $\hat{C}(N)$ is the min-cost under loser constraints, $\hat{C}(N) \leq C(N)$, satisfying validity criterion (1)
  - Because at answer nodes (i.e., final leaves) E there is only one solution that goes through E, $\hat{C}(E) = C(E)$, satisfying validity criterion (2)
  - Therefore, the rule of thumb given above guarantees a valid $\hat{C}$

B&B FOR MAXIMIZATION PROBLEMS

- In the case of maximization problems
  - there is a profit (instead of cost) associated with each solution, and
  - we are interested in finding the solution corresponding to the maximum profit
- Instead of cost function C and approximate cost function $\hat{C}$, we talk about *profit function* P and *approximate profit function* $\hat{P}$
- The theorem will have to be slightly modified so the validity criteria become
  1) $\hat{P}(N) \geq P(N)$ for every node N, and
  2) $\hat{P}(N) = P(N)$ for every answer node (final leaf) node N

$\hat{P}(N)$ is called an *overestimate* of $P(N)$

- The same rule of thumb applies for deriving $\hat{P}(N)$ from $P(N)$, by relaxing the constraints as little as possible to make the computation of $P(N)$ fast

LESSONS LEARNED SO FAR

- B&B searches the solution space (tree) in a modified BSF order
- To speed up the search, B&B uses a predictor $\hat{C}$ to estimate the promise of a node, and always selects the min-$\hat{C}$ live node to expand next
- The cost-so-far $\hat{C}$ is OK, and many predictors $\hat{C}$ exist: some better than others
- $\hat{C}$ need not comply with the constraints of the solution
- The closer $\hat{C}$ is to C, the faster it gets to the optimal solution

1) $\hat{C}(N) \leq C(N)\ \forall$ node N,
2) $\hat{C}(N) = C(N)\ \forall$ final leaf N

- If $\hat{C}$ satisfies two validity criteria, then B&B yields an optimal solution
- To derive a provably valid $\hat{C}$, set $\hat{C}$ = the cost function C under relaxed constraints
- B&B is an algorithm, not a template

Relax the constraints just enough to make $\hat{C}$ computation fast enough (polynomial)

- To design a B&B algorithm for a problem, express C, derive from it a $\hat{C}$, prove the two validity criteria for your $\hat{C}$, and implement the Expand procedure
- Maximization B&B works similarly: Replace cost by profit, $\leq$ by $\geq$, min by max, and min-heap by max-heap

A MAXIMIZATION APPLICATION OF B&B
-- THE 0/1 KNAPSACK PROBLEM --

- **Input**:
  - Items: 1, 2, 3, … , n
  - Weights: W_1 W_2 W_3 … , W_n
  - Prices: P_1 P_2 P_3 … , P_n
  - Capacity: C

P_i is the price of the whole item i, not the price per pound

- **Output**: Which items to take (in whole) such that the total of the taken weights is $\leq C$, and the total of the prices of the taken items is maximized.

  More formally:
  - $\forall i$, let $x_i = 1$ if item i is taken, 0 0therwise.
  - Output: Find $x_1, x_2, \dots, x_n$ to maximize $\Sigma_{i=1}^{n} x_i P_i$ such that $\Sigma_{i=1}^{n} x_i W_i \leq C$
- **Task**: Write a B&B algorithm for solving this problem

A MAXIMIZATION APPLICATION OF B&B
-- THE 0/1 KNAPSACK PROBLEM --

- The *profit function* $P(N) \stackrel{\text{def}}{=}$ the profit of the best solution that goes thru node N
- $P(N)$ = (the profit so far) + (the best 0/1 profit that can be gained from the remaining items $k+1, k+2, \dots, n$)
- To derive an *approximate profit function* $\hat{P}(N)$, **relax the 0/1 constraint** in

  (the best 0/1 profit that can be gained from the remaining items $k+1, k+2, \dots, n$)

  into the **regular knapsack** constraint (where you can take fractions of items)
- Then, the *relaxed* $P(N)$ = (the profit so far) + (the best regular knapsack profit that can be gained from the remaining items $k+1, k+2, \dots, n$)
- Take $\hat{P}(N)$ = the *relaxed* $P(N)$ = (profit so far) + (the greedy solution of the regular knapsack problem for the remaining items $k+1, k+2, \dots, n$ where the capacity is the remaining capacity), where N is a node at level k.
- Since $\hat{P}$ is derived using the relaxation rule, it is easy to prove that it satisfies the 2 validity criteria for maximization

The greedy solution is fast to compute, so it is fast to compute $\hat{P}(N)$

OTHER APPLICATIONS OF B&B
-- IN AI, AND IN NEAR-OPTIMIZATION --

- B&B is used heavily in classical Artificial Intelligence, under a different name: the A* algorithm
- When an optimal solution is costly to find, *near-optimal* solutions may be adequate. Different methods can be used to find near-optimal (or sub-optimal) solutions:
  - The Greedy method (fast but solution may not be good enough)
  - B&B, stopping when solution is good enough or when a pre-set time limit expires
    - This solution can be better (closer to optimal) than the greedy solution
    - This approach allows for progressively better solutions with more execution time