CS 6212 DESIGN AND ANALYSIS OF ALGORITHMS

LECTURE: BACKTRACKING

Instructor: Abdou Youssef

OBJECTIVES OF THIS LECTURE

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

- Systematically generate all objects of a finite family, called combinatorial objects (e.g., graphs, strings, permutations, cliques, cycles, etc.)
- Describe the Backtracking algorithm for generating combinatorial objects
- Specify and represent combinatorial objects of new combinatorial families in a generic, uniform way
- Leverage the common components of Backtracking, and devbelop the problem-specific part of the code for each separate (new) combinatorial family

OUTLINE

- Background
- Combinatorial families and Combinatorial objects
- Definition and purpose of Backtracking
- Uniform representation of combinatorial objects:
  - General format
  - Specifics for each of 8 combinatorial families
- Backtracking algorithm
- Implementation for each of the 8 combinatorial families

BACKTRACKING
-- BACKGROUND AND DEFINITION --

- So far, we have focused on computing **just one solution** to a given problem
- In certain situations, users may need to have all the solutions, like all graphs of a given size
- That is, a user may need all *objects* in a given *finite* family
- Finite objects of finite-size families are called *combinatorial objects*

EXAMPLES OF COMBINATORIAL FAMILIES/OBJECTS

Combinatorial Family/Objects	Size of the Family
All binary strings of n bits	2^n
All subsets of a given set E of n elements	2^n
All directed graphs of n nodes (self-loops ok)	$2^{(n^2)}$
All undirected graphs of n nodes (no self-loops)	$2^{\frac{n(n-1)}{2}}$
All Permutations of a size n	$n!$
All Hamiltonian cycles of a given graph	It depends on the graph. For a complete graph, it is $n!$
All k-cliques of a given graph	It depends on the graph. For a complete graph, it is $\binom{n}{k}$
All k-colorings of a given graph	It depends on the graph

FINE POINTS
-- NON-COMBINATORIAL FAMILIES/OBJECTS --

- Would the family of weighted graphs be considered a finite combinatorial family? Why or why not?
- Would the family of continuous curves be considered combinatorial? Why or why not?
- Can you think of other examples of non-combinatorial families/objects?

BACKTRACKING
-- DEFINITION AND PURPOSE --

- **Definition**: Backtracking is a systematic method for generating all objects of a given combinatorial family

- Typical application: Testing
  - If you design an algorithm whose input is a combinatorial object of a certain family, and
  - you want to test the algorithm,
  - Then you need a fairly large sample of inputs to test your algorithm

NOTE ON BACKTRACKING TIME COMPLEXITY

- As we will see, generating a single object is fairly fast
- But generating all the objects is prohibitively expensive
- That is because in most combinatorial families, the number of objects is huge (exponential)
- Therefore, in many Backtracking applications, only a subset of the objects is generated
  - Like a random sample of objects
  - Or the first N objects generated by Backtracking
- In this lecture, we ignore time complexity, and focus on how to generate **all** the objects in a given combinatorial family

ALGORITHM, NOT TEMPLATE

- We will give an actual Backtracking algorithm that can apply to a large collection of combinatorial families
  - Not a template, not a sequence of steps,
  - But an **actual algorithm**!
- To be able to have such a generic algorithm, we have to have a uniform representation of the combinatorial objects across many combinatorial families
- We'll present that next

UNIFORM REPRESENTATION OF COMBINATORIAL OBJECTS

- In most of the combinatorial families we deal with:
  - **Each object** in the family is represented by **an array X[1:*N*]** for some fixed N
  - Each element of the array takes values from a finite domain $S = \{a_1, a_2, \ldots, a_m\}$, for some fixed positive integer value m
    - Often, S consists of successive integers
    - Examples: $S = \{0,1\}$, or $S = \{1,2, \ldots, n\}$
  - The values of array X must satisfy some constraints *C* so that X represents a legitimate object of the family in question
- Each *C*-compliant instance of the whole array X[1:*N*] represents a <u>single, separate, full object</u>
- Each combinatorial family can be thus modeled as **(X[1:*N*], *S*, *C*)**, where X[1:*N*] is meant to represent any single object of the family
- We will see what (X[1:*N*], *S*, *C*) is for each of the 8 aforementioned families

BINARY STRINGS

For a given positive integer n:

- Every n-bit binary string is represented by an array X[1:n], where **X[i] is the i^th bit of the binary string**. So $N = n$.
  - Example: For string=0110, X=[0,1,1,0]
- $S = \{0,1\}$: X[i] takes its values from {0,1} for each i.
- $C = \phi$: The constraints C are empty because the values of the individual bits in a binary string are independent of one another

N=n, S={0,1}, C= ϕ, X[i] represents the i^{th} bit of the string

SUBSETS OF A GIVEN SET

Given a set $E = \{1,2,\ldots,n\}$

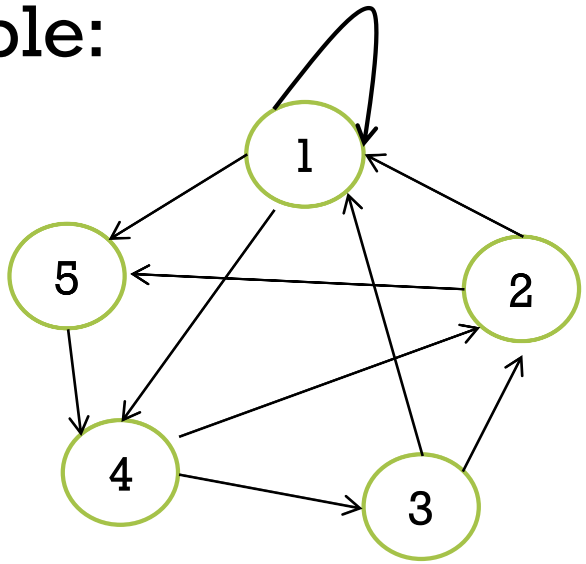
- Every subset is represented by the bitmap (i.e., Boolean array) X[1:n];
  - $X[i] = \begin{cases} 1 \text{ if i is in the subset being represented} \\ 0 \text{ if i is not in the subset being represented} \end{cases}$
- Example: n=4, $E = \{1,2,3,4\}$. Take subset A={2,4}.
  - It is represented by array X=[0,1,0,1].
  - X[1]=0 because $1 \notin A$, X[2]=1 because $2 \in A$, etc.
- $S = \{0,1\}$: As just seen, every X[i] is 0 or 1.
- $C = \phi$: The constraints are empty because whether i is an element of the subset has no bearing on whether j is an element of the subset.
- Note: Abstractly, this problem is identical to the binary strings problem

N=n, S={0,1}, C= ϕ, X[i] represents if i is in the subset

DIRECTED GRAPHS

Given a positive integer n

- Every digraph of n nodes is representable by a 2D array A[1:n,1:n], which is the well-known adjacency matrix
  - A[i,j]=1 if(i,j) is an edge; A[i,j]=0 if (i,j) is not an edge
- Example:



$$A = \begin{pmatrix} 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

DIRECTED GRAPHS

Given a positive integer n

- Every digraph of n nodes is representable by a 2D array A[1:n,1:n], which is the well-known adjacency matrix
- The values of the entries in the array are binary and independent of one another (why)
- The 2D array A can be represented by a 1D binary array X[1:*N*] where $N = n^2$
  - Each X[i] is 0 or 1: 1 represents that the corresponding edge exists, 0 otherwise
- $S = \{0,1\}$: As just seen, every X[i] is 0 or 1.
- $C = \phi$: Because the values of entries of X (which are the entries of A) are independent
- Mapping from A[1:n,1:n] to X[1:n^2]: Stack the rows of A one after another.
  - Example: $A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \Rightarrow X = [a\ b\ c\ d\ e\ f\ g\ h\ i]$
  - X[(i-1)n + j] = A[i,j]

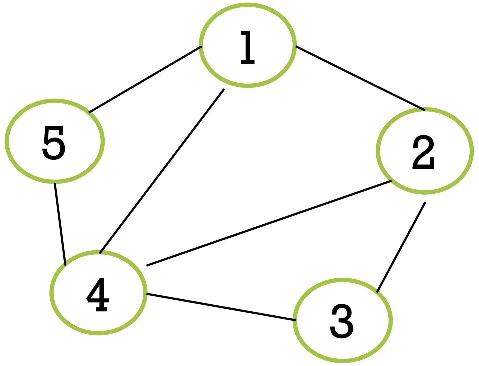
N=n^2, S={0,1}, C= ϕ,
X [(i-1)n + j] represents if (i,j) is an edge

Note: Abstractly, this problem is identical to the previous problem: binary strings, and subsets!!

UNDIRECTED GRAPHS

Given a positive integer n

- Every graph of n nodes is representable by a 2D adjacency matrix A[1:n,1:n], which is symmetric (i.e., A[i,j]=A[j,i] for all i and j)



$$A = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix}$$

- N=n^2, S={0,1},
- C: $\forall i,j$, $A[i,j] = A[j,i]$; and $A[i,i] = 0\ \forall i$
- X [(i-1)n + j] represents whether (i,j) is an edge or not
- C: $\forall i,j$, X [(i-1)n + j] = X[(j-1)n+i], and X[(i-1)n+i]=0

- We can use the same 1D array representation X[1,N] where N=n^2
- $C = \forall i,j\ , A[i,j] = A[j,i]$; also, if no self-loops are allowed, $A[i,i] = 0\ \forall i$
- But the simpler the constraints, the simpler and faster the algorithm.
- So, can we have a better representation (with simpler constraints)?

UNDIRECTED GRAPHS
-- A CONSTRAINT-FREE REPRESENTATION --

Given a positive integer n

- Every graph of n nodes is representable by a 2D binary adjacency matrix A[1:n,1:n], which is symmetric (i.e., A[i,j]=A[j,i] for all i and j)
- Since the top triangle is identical to the bottom triangle, and the diagonal is all zeros, capture only the top triangle, i.e., a graph can be represented by the top triangle only
- $A = \begin{bmatrix} 0 & a & b & c \\ a & 0 & d & e \\ b & d & 0 & f \\ c & e & f & 0 \end{bmatrix}$

$$A = \begin{bmatrix} 0 & a & b & c \\ a & 0 & d & e \\ b & d & 0 & f \\ c & e & f & 0 \end{bmatrix} \Rightarrow X = [a\ b\ c\ d\ e\ f]$$

- To represent a graph with a 1D array X, map the top triangle to a linear array row-wise
- The 1D array representation: X[1,N] where N=$(n-1)+(n-2)+\cdots+2+1=\frac{n(n-1)}{2}$
- $C=\phi$: Because whether an undirected pair of nodes is an edge has no bearing on whether another undirected pair of nodes is an edge.

N=$\frac{n(n-1)}{2}$, S={0,1}, $C=\phi$
X [k] represents if (i,j) is an edge. What is k in terms of (i,j)?

PERMUTATIONS

Given a positive integer n (i.e., a set E={1,2,…,n})

- **Definition**: A permutation is a ***one-to-one and onto mapping*** (function) f from E to E. The mapping of element i is denoted f(i)
- **Another definition**: A permutation is a re-ordering (***re-arrangement***) of the elements 1,2,…,n
- **A third definition**: A permutation is a **one-to-one matching**.
  - i is said to be matched with f(i).
- Math representation of a permutation: $f = \begin{pmatrix} 1\ 2\ 3 & \ldots & i & \ldots & n \\ 2\ 4\ 1 & \ldots & f(i) & \ldots & f(n) \end{pmatrix}$

  where the top row is 1, 2, … , n; and the value under i is f(i)

PERMUTATION REPRESENTATION

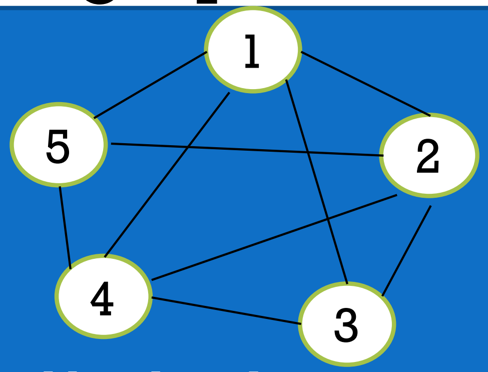
- A permutation f can be represented by a 1D array X[1:n] where **X[i]=f(i).**

N=n; S={1,2,…,n}; $C: \forall i \neq j, X[i] \neq X[j]$
X [i] represents f(i)

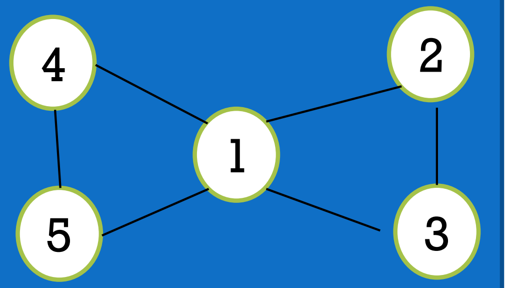
- Example:
  - n=4, $f = \begin{pmatrix} 1\ 2\ 3\ 4 \\ 2\ 4\ 1\ 3 \end{pmatrix}$, $i.e., f(1) = 2, f(2) = 4, f(3) = 1, f(4) = 3,$
  - X=[2 4 1 3], i.e., the bottom row.
- $S = \{1,2, \dots, n\}$: X[i] can be any value 1, 2, … , or n.
- $C: \forall i \neq j, X[i] \neq X[j]$: By def, the bottom row of f is a re-arrangement of the top row => no two values in bottom row can be equal.

HAMILTONIAN CYCLES
-- DEFINITION AND EXAMPLES --

- Given an undirected graph G (of n nodes)
- Definition: a Hamiltonian cycle of G is any cycle that goes through every node of G exactly once. Thus a HC has all the n nodes, in some arrangement.
- Note that not all graphs have Hamiltonian cycles



HC 1: 1, 2, 3, 4, 5, and back to 1
HC 2: 1, 3, 5, 2, 4, and back to 1
There are many more HCs



There are no Hamiltonian cycles in this graph. Why?

HAMILTONIAN CYCLES
-- UNIFORM REPRESENTATION --

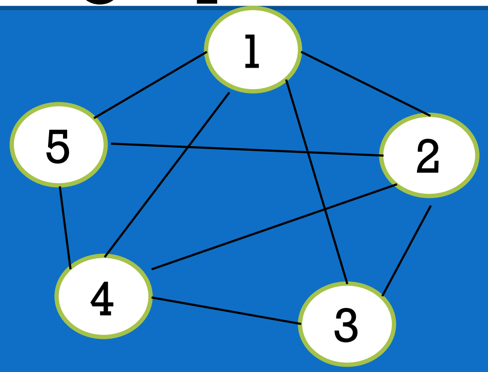
Given a graph G(V,E) of n nodes:

- A Hamiltonian cycle (of n nodes) can be represented by X[1:n] where **X[i] is the ith node of the cycle**
- $S = \{1,2,\dots,n\}$: X[i] can be any of the nodes 1, 2, … , n.
- $C: \forall i \neq j, X[i] \neq X[j]$, and $\forall i\ (X[i], X[i+1]) \in E$, and $(X[n], X[1]) \in E$

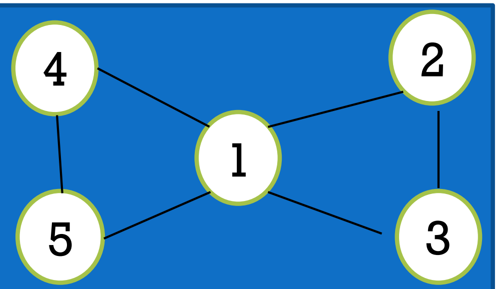
N=n; S={1,2,…,n};
$C: \forall i \neq j, X[i] \neq X[j]; \forall i\ (X[i], X[i+1]) \in E; (X[n], X[1]) \in E$
X [i] represents the ith node of the cycle

K-CLIQUES
-- DEFINITION AND EXAMPLES --

- Given an undirected graph G (of n nodes) and a positive integer k $\leq n$
- Definition: A k-clique of G is a subset of k nodes where every pair of those nodes are adjacent in G.
- Note that not all graphs have k-cliques



3-cliques: {1,2,3}, {5,3,4}, etc.
4-cliques: {1,2,3,4}, {1,2,3,5}, {2,3,4,5], …
5-clique: {1,2,3,4,5}



2-cliques: {1,2}, {1,3}, {4,5}, …
3-cliques: {1,2,3}, {1,4,5}
4-cliques: None
5-cliques: None

K-CLIQUES
-- UNIFORM REPRESENTATION --

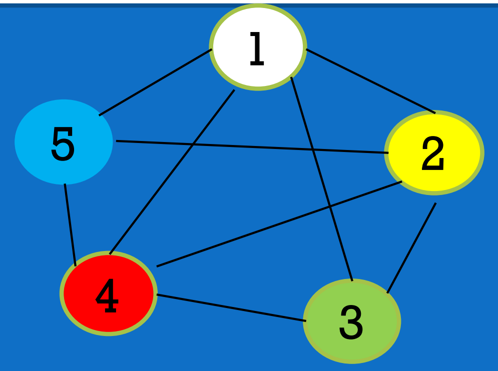
Given a graph G(V,E) of n nodes, and a positive integer k $\leq n$

- A k-clique (of k nodes) can be represented by X[1:k] where **X[i] is the ith node of the clique**
- $S = \{1,2,\ldots,n\}$: X[i] can be any of the nodes 1, 2, … , n.
- $C: \forall i \neq j, X[i] \neq X[j]$ and $(X[i], X[j]) \in E$

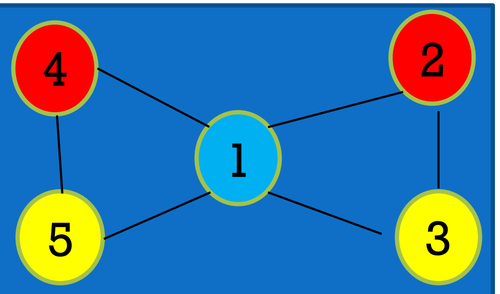
N=k; S={1,2,…,n};
$C: \forall i \neq j X[i] \neq X[j]$ and $(X[i], X[j]) \in E$
X [i] represents the ith node of the clique

K-COLORING
-- DEFINITION AND EXAMPLES --

- Given an undirected graph G (of n nodes) and a positive integer $k \leq n$ of colors
- Definition: A k-coloring of G is an assignment of a color to each node in such a way that every two neighboring nodes have distinct colors, and the total number of colors used is $\leq k$.



This graph has no 3-coloring, no 4-coloring, but has a 5-coloring (as shown in the figure)



2-coloring: None
3-coloring: As shown in the picture

K-COLORING
-- UNIFORM REPRESENTATION --

Given a graph G(V,E) of n nodes, and a positive integer $k \leq n$ of colors (we can label the colors $1, 2, \ldots, k$)

- A k-coloring of G can be represented by X[1:n] where **X[i] is the color of node i**
- $S = \{1,2,\ldots,k\}$: X[i] can be any of the color $1, 2, \ldots, k$.
- $C: \forall (i,j) \in E$, $X[i] \neq X[j]$

N=n; S={1,2,…,k}; $C: \forall (i,j) \in E$, $X[i] \neq X[j]$
X [i] represents the color of node i

LESSONS LEARNED SO FAR

- Backtracking is for generating combinatorial objects in finite families
- Backtracking is more than a template/technique: it is an algorithm
- For many combinatorial families, we can represent the objects with a uniform representation: (X[1:N], S, C). This allows having a shared Backtracking algorithm
- Even when the natural representation is not 1D arrays, one can map the natural representation to a 1D array so the Backtracking algorithm can apply with minimum effort
- Constraints can be simplified if we reduce/eliminate representation-redundancy (and thus inter-dependencies)
- More lessons to come

BACKTRACKING ALGORITHM
-- PRELIMINARIES (1/3) --

- The algorithm will generate all valid arrays X[1:N] whose elements come from the domain $S = \{a_1, a_2, \ldots, a_m\}$ of successive integers, such that the constraints C are satisfied.
- The algorithm is a depth-first search like traversal (or generation) of the tree that represents the entire solution space.
- In the tree:
  - the root designates the starting point
  - every path from the root to a leaf is of length N, where the node in level i specifies a value for element X[i]
  - The whole path corresponds to the whole array and represents a single solution, that is, a single object.

BACKTRACKING ALGORITHM
-- PRELIMINARIES (2/3) --

- During the generation of the tree, when we are to create a new node corresponding to X[i], we try to assign X[i] the next domain value (given the current value of X[i] as reference).
  - If that value does not violate the constraints *C*, it is assigned.
  - If, on the other hand, that value violates *C*, the next value after that is tried, and so on until either a C-compliant value is found or all remaining values are exhausted.
  - If a C-compliant value is found and assigned to X[i], we move to the next level to find a value for X[i+1].
  - If no C-compliant remaining value is found for X[i], we *backtrack* (i.e., go back) to the previous level to find a new value for X[i-1].

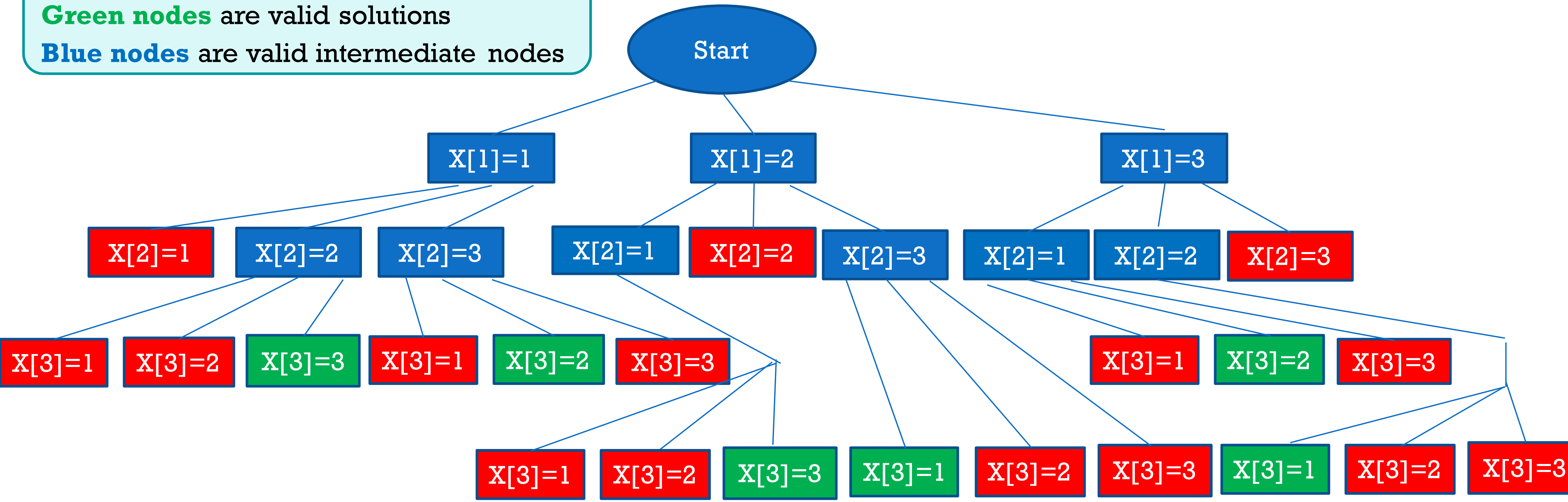
BACKTRACKING ALGORITHM
-- PRELIMINARIES (3/3) --

- When we backtrack to the root, the whole tree has been fully generated, and the algorithm stops
- Whenever a *C*-compliant value for X[n] is found, a complete new object has been generated, and the path from the root to that node corresponding to X[n] is printed out as the object
- Recall that in the algorithm, when a new node for a new value for X[i] is being generated, the values that are tried are the "next" values (in *S*) from a *reference value*, which is the current value of X[i]
- To be consistent with the previous bullet, let the reference value at the outset be initialized be a value $a_0 \stackrel{\text{def}}{=} a_1 - 1$
- That way, the next value is always the reference (current) value + 1.

ILLUSTRATION OF THE ALGORITHM

-- ON PERMUTATIONS OF SIZE 3 --

Red nodes are dead-ends (violate *C*)
Green nodes are valid solutions
Blue nodes are valid intermediate nodes



BACKTRACKING ALGORITHM

-- THE PSEUDO-CODE--

```
Procedure Backtrack()
begin
    int r := 1; //r is the tree level, index of X
    int X[1:N];
    for i=1 to N do: X[i] := a_0; endfor // initialize X
    while r > 0 do
        getnext(X,r);
        // assigns to X[r] its next C-compliant value,
        // if available; else, it re-initlizes X[r] to a_0
        if (X[r] == a_0) then
            r := r-1; //backtrack to the previous level
        elseif r==N then
            print(X[1:N]); //a new complete solution
        else
         r := r+1; //move to the next level for X[r+1]
        endif
    endwhile
end // Backtrack
```

```
Procedure getnext(in/out: X[1:N]; in: r)
Begin
    X[r] := X[r] + 1; // next tentative value
    while (X[r] <= a_m) do
        if (Bound(X[1:N],r) is true) then
            return; // new value for X[r] is found
        else   // try the next value in S
            X[r] := X[r] + 1;
        endif
    endwhile
    // if getnext has not returned, that
    // means no C-compliant remaining
    // value was found. Re-initialize X[r]
    X[r] := a_0;
end
```

- Bound(X[1:N],r) checks if the value of X[r] is C-compliant, and if so, returns true
- Bound assumes X[1:r-1] are C-compliant
- The code for Bound varies from problem to problem

WHAT YOU NEED TO DO WHEN SOLVING A BACKTRACKING PROBLEM

1. Derive the uniform representation **(X[1:*N*], *S*, *C*)**
   - Give the value of **N**
   - Specify the domain ***S*** (i.e., give $\{a_1, a_2, \dots, a_m\}$), and a_0
   - Describe what every **X[i] means/signifies**
   - Present the constraints ***C*** in a logical manner
2. Give the pseudocode for **Bound**(…)
3. Copy the code for Backtrack() and getnext(…), **replacing** the values of N, a_0, and a_m by their appropriate values

THE BOUND FUNCTIONS FOR THE 8 PROBLEMS

- For each of our 8 combinatorial families,
  - The model (X[1:*N*], *S, C*) has been given
  - What remains is to give the Bound function

- We will do so next

BOUND FOR THE FIRST 4 FAMILIS

- For the first 4 families (binary strings, subsets of a given set, directed graphs, undirected graphs):
  - $C = \phi \Rightarrow$ there are no constraints to comply with
  - Therefore, the Bound function should allows return true:

```
Function Bound(X[1:N]; r)
begin
    return true;
end Bound
```

- Binary Strings: N=n
- Subsets: N=n
- Directed Graphs: N=n^2
- Undirected graphs: N=$N = \frac{n(n-1)}{2}$
- For all four families: S={0,1}, $a_0 = -1, a_1 = 0, a_2 = 1$

BOUND FOR PERMUTATIONS

N=n; S={1,2,…,n}; X [i] represents f(i)
$C: \forall i \neq j, X[i] \neq X[j]$
$a_0 = 0, m = n, a_m = n$

```
Function Bound(X[1:n],r)
begin
    // X[1:r-1] have C-compliant values. Bound checks to see if X[r] is C-compliant.
    for i=1 to r-1 do
        if X[r] == X[i] then        // violates C: no two X values can be equal
            return(false);
        endif
    endfor
    // If we reach here without returning, the value of X[r] doesn't violate C
    return(true);
end Bound
```

BOUND FOR HAMILTONIAN CYCLES

N=n; S={1,2,…,n};
X [i] represents the i^th node in the HC
$C: \forall i \neq j, X[i] \neq X[j]; \forall i\ (X[i], X[i+1]) \in E; (X[n], X[1]) \in E$
$a_0 = 0, m = n, a_m = n$

```
Function Bound(X[1:n],r)
begin
    // X[1:r-1] have C-compliant values. Bound checks to see if X[r] is C-compliant.
    // next, check for violations that could be incurred by X[r]
    for i=1 to r-1 do
        if X[r] == X[i] then     return(false);   endif
    endfor
    if (r > 1 and (X[r-1],X[r])  is not an edge) then      return(false);  endif
    if (r==n and (X[n],X[1])  is not an edge) then      return(false);  endif
    return(true) ;
end Bound
```

BOUND FOR K-CLIQUES

N=n; S={1,2,…,n};
X [i] represents the ith node in the HC
$C: \forall i \neq j, X[i] \neq X[j]$ and $(X[i], X[j]) \in E$;
$a_0 = 0, m = n, a_m = n$

```
Function Bound(X[1:n],r)
begin
    // X[1:r-1] have C-compliant values. Bound checks to see if X[r] is C-compliant.
    // next, check for violations that could be incurred by X[r]
    for i=1 to r-1 do
        if (X[r] == X[i] or (X[r],X[i]) is not an edge) then
            return(false);
        endif
    endfor
    return(true) ;
end Bound
```

BOUND FOR K-COLORING

N=n; S={1,2,…,k};
X [i] represents the color of node i
$C: \forall (i,j) \in$ E, X[i] $\neq$ X[j]
$a_0 = 0, m = k, a_m = k$

```
Function Bound(X[1:n],r)
begin
    // X[1:r-1] have C-compliant values. Bound checks to see if X[r] is C-compliant.
    // next, check for violations that could be incurred by X[r]
    for i=1 to r-1 do
        if ((r,i) is an edge and X[r] == X[i] ) then
            return(false);
        endif
    endfor
    return(true) ;
end Bound
```

LESSONS LEARNED SO FAR

- Backtracking is for generating combinatorial objects in finite families
- Backtracking is more than a template/technique: it is an algorithm
- For many combinatorial families, we can represent the objects with a uniform representation: (X[1:N], S, C). This allows having a shared Backtracking algorithm
- Even when the natural representation is not 1D arrays, one can map the natural representation to a 1D array so the Backtracking algorithm can apply with minimum effort
- Constraints can be simplified if we reduce/eliminate representation-redundancy (and thus inter-dependencies)
- The uniform representation and the Bound function are all you need to do for a new Backtracking problem
- The simpler the constraints are, the easier and the simpler the Bound function is

OTHER BACKTRACKING PROBLEMS

- Generate all directed graphs of n nodes and p edges
- Generate all regular undirected n-node graphs of degree d (where regular means that all the nodes have the same degree), for a given n and d
- Generate all undirected n-node graphs where the degree of every node is $\leq d$, for a given n and d
- Generate k-letter strings, for a given k, where each letter is any of the 26 lower case English letters
- Generate k-digit numbers where the k digits are in strictly increasing order (meaning that the i^{th} digit is < then the digit after it, for all i)

NEXT LECTURE

- Branch and Bound
  - A last-resort optimization technique