Optimization problems

An optimization problem generally has two parts:

- An objective function that is to be maximized or minimized.
  - For example, find the route that has the cheapest airfare between Boston and Istanbul.
- A set of constraints (possibly empty) that must be honored.
  - For example, don’t exceed an upper bound on the travel time, or on the number of legs in the trip.

Examples of classic optimization problems

- Shortest path
- Traveling salesman
- Bin packing
- Sequence alignment
- Knapsack

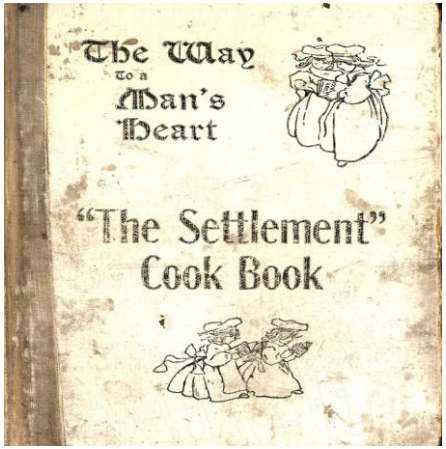
Valuable to know about these kinds of problems, as can use **problem reduction** to reduce a new problem to a variant of a known problem type for which others have created efficient solutions

Efficiency of optimization problems

- We've seen some very efficient classes of algorithms
  - Sublinear (binary search), linear, polynomial
- Many optimization problems are much worse in efficiency if we want the best answer
- So may want to settle for a "good enough" solution that can be found quickly

A classic example of an optimization problem

- The Knapsack Problem



Knapsack

Some attributes of the “loot”

- Burglar can only carry 20 pounds of loot
- Given this information what should he do?

	Value	Weight	Value/Weight
Clock	175	10	17.5
Painting	90	9	10
Radio	20	4	5
Vase	50	2	25
Book	10	1	10
Computer	200	20	10

A greedy algorithm approach

- Take the best item
- Then take next best
- Continue until cannot fit any more items (e.g. because of weight)

- Still have to decide what is “best”
  - Most valuable? Lightest? Best value/weight ratio? Something else?

A greedy approach to the knapsack problem

```
class Item(object):
    def __init__(self, n, v, w):
        self.name = n
        self.value = float(v)
        self.weight = float(w)
    def getName(self):
        return self.name
    def getValue(self):
        return self.value
    def getWeight(self):
        return self.weight
    def __str__(self):
        result = '<' + self.name + ', ' + str(self.value)\
                 + ', ' + str(self.weight) + '>'
        return result
```

A greedy approach to the knapsack problem

```
def buildItems():
    names = ['clock', 'painting', 'radio',
             'vase', 'book' 'computer']
    vals = [175,90,20,50,10,200]
    weights = [10,9,4,2,1,20]
    Items = []
    for i in range(len(vals)):
        Items.append(Item(names[i],
                          vals[i], weights[i]))
    return Items
```

Being greedy

```
def greedy(Items, maxWeight, keyFcn):
    assert type(Items) == list and maxWeight >= 0
    ItemsCopy = sorted(Items, key=keyFcn, reverse = True)
    result = []
    totalVal = 0.0
    totalWeight = 0.0
    i = 0
    while totalWeight < maxWeight and i < len(Items):
        if (totalWeight + ItemsCopy[i].getWeight()) <= maxWeight:
            result.append((ItemsCopy[i]))
            totalWeight += ItemsCopy[i].getWeight()
            totalVal += ItemsCopy[i].getValue()
        i += 1
    return (result, totalVal)
```

Remember a function is an object

- `ItemsCopy = sorted(Items, key=keyFcn, reverse = True)`
- Using `keyFcn` parameter lets us generalize one procedure to use different measures of goodness
- Just requires that `keyFcn` defines an ordering on the list of elements
- Then use this to create an ordered list

- Use `sorted` to create a copy of the list

So let's get greedy

```
def value(item):
    return item.getValue()

def weightInverse(item):
    return 1.0/item.getWeight()

def density(item):
    return item.getValue()/item.getWeight()

def testGreedy(Items, constraint, getKey):
    taken, val = greedy(Items, constraint, getKey)
    print ('Total value of items taken = ' + str(val))
    for item in taken:
        print '  ', item
```

So lets get greedy

```
def testGreedys(maxWeight = 20):
    Items = buildItems()
    print('Items to choose from:')
    for item in Items:
        print '  ', item
    print 'Use greedy by value to fill a
knapsack of size', maxWeight
    testGreedy(Items, maxWeight, value)
    print 'Use greedy by weight to fill a
knapsack of size', maxWeight
    testGreedy(Items, maxWeight, weightInverse)
    print 'Use greedy by density to fill a
knapsack of size', maxWeight
    testGreedy(Items, maxWeight, density)
```

And if we are greedy?

```
>>> testGreedys()
…
Use greedy by value to fill a knapsack of size 20
Total value of items taken = 200.0
   <computer, 200.0, 20.0>
Use greedy by weight to fill a knapsack of size 20
Total value of items taken = 170.0
   <book, 10.0, 1.0>
   <vase, 50.0, 2.0>
   <radio, 20.0, 4.0>
   <painting, 90.0, 9.0>
Use greedy by density to fill a knapsack of size 20
Total value of items taken = 255.0
   <vase, 50.0, 2.0>
   <clock, 175.0, 10.0>
   <book, 10.0, 1.0>
   <radio, 20.0, 4.0>
```

No guarantee that any greedy algorithm will find the optimal solution

Efficiency of the greedy approach

- Two factors to consider
  - Complexity of sorted
  - Number of times through the while loop

- Latter is bounded by number of items in list (hence linear)
- But sorted is O(n log n)
- So overall algorithm is O(n log n), where n is length of list of items

The 0/1 Knapsack problem – finding an optimal solution

- Each item is represented by a pair, <value, weight>.
- The knapsack can accommodate items with a total weight of no more than w.
- A vector, I, of length n, represents the set of available items. Each element of the vector is an item.
- A vector, V, of length n, is used to indicate whether or not each item is taken by the burglar. If V[i] = 1, item I[i] is to be taken. If V[i] = 0, item I[i] is not taken.
- Find a V that maximizes the sum of V[i]*I[i].value over all values of i, subject to the constraint that the sum of V[i]*I[i].weight over all values of i is no more than w.

An approach to solving this problem

- Enumerate all possible combinations of items, this is called the **power set**,
- Remove all of the combinations whose weight exceeds the allowed weight,
- From the remaining combinations choose any one whose value is at least as large as the value of the other combinations.

But this is going to be slow?

- How big is a power set?
  - Suppose we have two elements {a, b}
  - Then the power set is {}, {a}, {b}, {a, b}
  - Now suppose we have three elements {a, b, c}
  - Then the power set is {}, {a}, {b}, {c}, {a, b}, {a, c}, {b, c}, {a, b, c}

  - And if we have four elements?

Representing the power set

- We can represent any combination of items by a vector of 0's and 1's.
- The combination containing no items would be represented by a vector of all 0's.
- The combination containing all of the items would be represented by a vector of all 1's.
- The combination containing only the first and last elements would be represented by 100…001.
- And so on
- If we look at four items, the possible choices are:

a	b	c	d	combos
0	0	0	0	{}
0	0	0	1	{d}
0	0	1	0	{c}
0	0	1	1	{c,d}
0	1	0	0	{b}
0	1	0	1	{b,d}
0	1	1	0	{b,c}
0	1	1	1	{b,c,e}
1	0	0	0	{a}
1	0	0	1	{a,d}
1	0	1	0	{a,c}
1	0	1	1	{a,c,d}
1	1	0	0	{a,b}
1	1	0	1	{a,b,d}
1	1	1	0	{a,b,c}
1	1	1	1	{a,b,c,d}

Capturing the power set

- Just looking at the right hand column, this may seem confusing
- But looking at the left side columns, there is a clear method to generating the power set
  - We are just enumerating all possible four digit binary numbers

A brute force approach

```
def dToB(n, numDigits):
    """requires: n is a natural number less than 2**numDigits
      returns a binary string of length numDigits representing
      the decimal number n."""
    assert type(n)==int and type(numDigits)==int
           and n >=0 and n < 2**numDigits
    bStr = ''
    while n > 0:
        bStr = str(n % 2) + bStr
        n = n//2
    while numDigits - len(bStr) > 0:
        bStr = '0' + bStr
    return bStr
```

A brute force approach

```
def genPset(Items):
    """Generate a list of lists representing
       the power set of Items"""
    numSubsets = 2**len(Items)
    templates = []
    for i in range(numSubsets):
        templates.append(dToB(i, len(Items)))
    pset = []
    for t in templates:
        elem = []
        for j in range(len(t)):
            if t[j] == '1':
                elem.append(Items[j])
        pset.append(elem)
    return pset
```

A brute force approach

```
def chooseBest(pset, constraint, getVal, getWeight):
    bestVal = 0.0
    bestSet = None
    for Items in pset:
        ItemsVal = 0.0
        ItemsWeight = 0.0
        for item in Items:
            ItemsVal += getVal(item)
            ItemsWeight += getWeight(item)
        if ItemsWeight <= constraint and ItemsVal > bestVal:
            bestVal = ItemsVal
            bestSet = Items
    return (bestSet, bestVal)
```

And the best is…

```
def testBest():
    Items = buildItems()
    pset = genPset(Items)
    taken, val = chooseBest(pset, 20, Item.getValue,
                                Item.getWeight)
    print ('Total value of items taken = ' + str(val))
    for item in taken:
        print '  ', item

>>> testBest()
Total value of items taken = 275.0
   <clock, 175.0, 10.0>
   <painting, 90.0, 9.0>
   <book, 10.0, 1.0>
```

Complexity?

- Note that this will in principle be slow since we generate the entire power set, then check each possible subset of elements to determine whether that subset meets the constraints
- This will find the best solution, since we check all options, but this is exponential in the number of items because we create the entier power set

A different approach to optimal solutions

- A rooted binary tree is an acyclic directed graph in which:
  - There is exactly one node (the root) with no parents
  - Each non-root node has exactly one parent
  - Each node has at most two children. A childless node is called a **leaf**

A different approach to optimal solutions

Each node is labeled with a quadruple that denotes a partial solution to the knapsack problem:

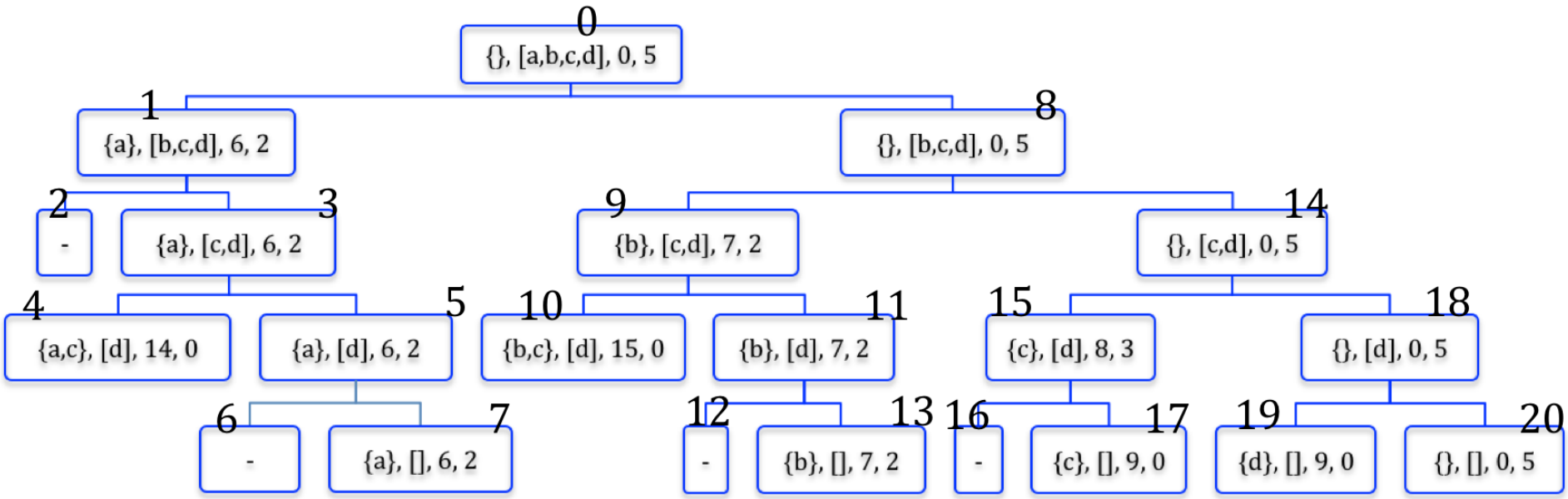
- A (perhaps partial) set of items to be taken.
- The list of items for which a decision about whether or not to take each item in the list has not been made.
- The total value of the items in the set of items to be taken. This is merely an optimization, since the value could be computed from the set.
- The remaining space in the knapsack. Again, this is an optimization since this is merely the difference between the weight allowed and the weight of all the items taken so far.

A different approach to optimal solutions

- Build the tree top down starting with the root.
- Select an element from the still to be considered items.
- If there is room for that item in the knapsack, a node is constructed that reflects the consequence of choosing to take that item. By convention, we draw that as the left child. The right child shows the consequences of choosing not to take that item.
- The process is then applied recursively to non-leaf children. Because each edge represents a decision (to take or not to take an item), such trees are called **decision trees**.

A simple example

Name	Value	Weight
A	6	3
B	7	3
C	8	2
D	9	5



- Numbers above nodes indicate order of search
- Depth first

Recursive implementation

```
def maxVal(toConsider, avail):
    if toConsider == [] or avail == 0:
        result = (0, ())
    elif toConsider[0].getWeight() > avail:
        result = maxVal(toConsider[1:], avail)
    else:
        nextItem = toConsider[0]
        #Explore left branch
        withVal, withToTake = maxVal(toConsider[1:],
                                     avail - nextItem.getWeight())
        withVal += nextItem.getValue()
        #Explore right branch
        withoutVal, withoutToTake = maxVal(toConsider[1:], avail)
        #Choose better branch
        if withVal > withoutVal:
            result = (withVal, withToTake + (nextItem,))
        else:
            result = (withoutVal, withoutToTake)
    return result
```

Implementation and testing

```
def smallTest():
    Items = buildItems()
    val, taken = maxVal(Items, 20)
    for item in taken:
        print(item)
    print ('Total value of items taken = ' + str(val))
```

This gives us the same solution we saw earlier

But it gives us a different way of thinking about finding the solution

And it doesn’t explore the entire powerset, since it truncates search

Was it worth it?

- This gives a different, and better, answer than any of the greedy methods.
- But how long does it take to find the answer?
- If we have n items, then the power set is the same size as the number of binary numbers with n digits – or 2^n
- Exponential in size if we look at everything
- Even with decision tree approach, still probably exponential

Is it really that bad?

- Suppose we have 50 items
- Suppose we can generate and check each potential combination of these items in a microsecond (one millionth of a second)
- How long will it take to check all combinations?

- About 36 years

The tradeoff

- A greedy algorithm is making the best (as defined by some metric) local choice at each step
- It is making a choice that is **locally optimal**
- But does not mean the solution found is **globally optimal**
- The tradeoff is that an approximation algorithm may find a good, but not best, solution, but take far less time than needed to find the best solution

Can we do better?

- There are few optimizations that might reduce the typical running time. For example, no need to generate overweight solutions.
-
- However, in a theoretical sense, the problem is hopeless. The 0/1 knapsack problem is **inherently exponential** in the number of items.
- In a practical sense, however, the problem is far from hopeless. For example, you might look up the idea of **dynamic programming**.

Saving redundant work

- We won't look at dynamic programming in this course
- But we can show a simpler version of the underlying idea, which is still valuable to know and utilize

- Note that in exploring tree, we may compute an answer to the same subproblem several times
  - What is the best answer using items {b,c,d} before deciding to include item {a}; similarly for {c,d} and so on

Saving redundant work

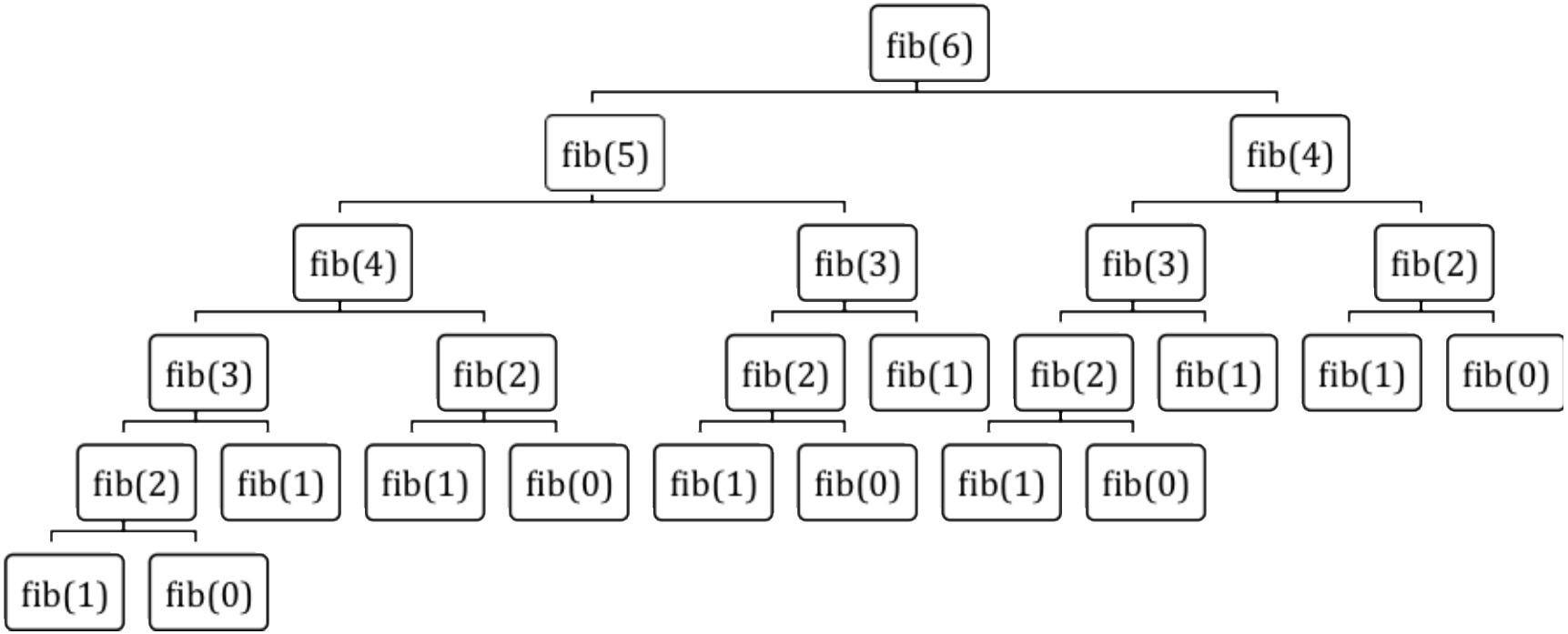
- We might be much more efficient if we could keep track of partial solutions, and then just look up the answer somewhere when we need to use this computation again
- Let's illustrate this idea with a simpler example – our old friend, Fibonacci

Standard Fibonacci

```
def fib(x):
    assert type(x) == int and x >= 0
    if x == 0 or x == 1:
        return 1
    else:
        return fib(x-1) + fib(x-2)

def testFib(n):
    assert type(n) == int and n >=  0
    for i in range(n):
        print ('fib of', i, '=', fib(i))
```

Improving Fibonacci

- This can be very slow
- It is repeating computations multiple times


fib(6)
fib(5)
fib(4)
fib(4)
fib(3)
fib(3)
fib(2)
fib(3)
fib(2)
fib(2)
fib(1)
fib(2)
fib(1)
fib(1)
fib(0)
fib(2)
fib(1)
fib(1)
fib(0)
fib(1)
fib(0)
fib(1)
fib(0)
fib(1)
fib(0)

Saving redundant work

- Note how many different times we compute fib(3) or fib(2)
- Since we know that fib does not do any side-effects, i.e., it doesn't mutate an internal variable, the answer is not going to be different
- So can we save this extra effort?
- **Memoization** – the first time we compute a function, keep track of the value; any subsequent time, just look up the value

```
def fastFib(x, memo):
    assert type(x) == int and x >= 0 and type(memo) == dict
    if x == 0 or x == 1:
        return 1
    if x in memo:
        return memo[x]
    result = fastFib(x-1, memo) + fastFib(x-2, memo)
    memo[x] = result
    return result

def testFastFib(n):
    assert type(n) == int and n >=  0
    for i in range(n):
        print ('fib of', i, '=', fastFib(i, {}))
```

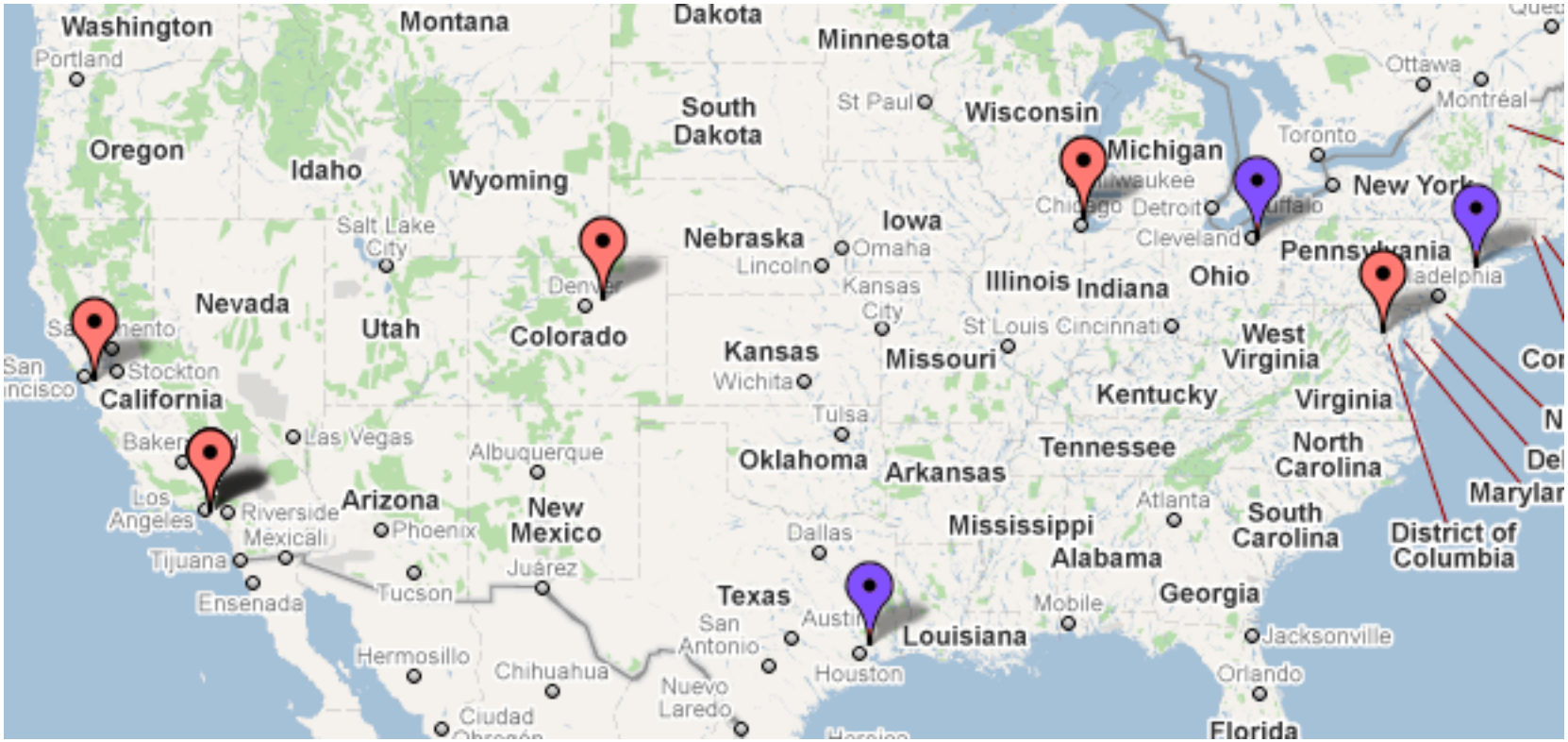
Memoization saves time

- Under this method, we only compute fib(n) once, thus saving a great deal of additional effort
- This idea can be used to improve other methods (check out dynamic programming)
- Memoization only applies if there is no mutation of internal variables

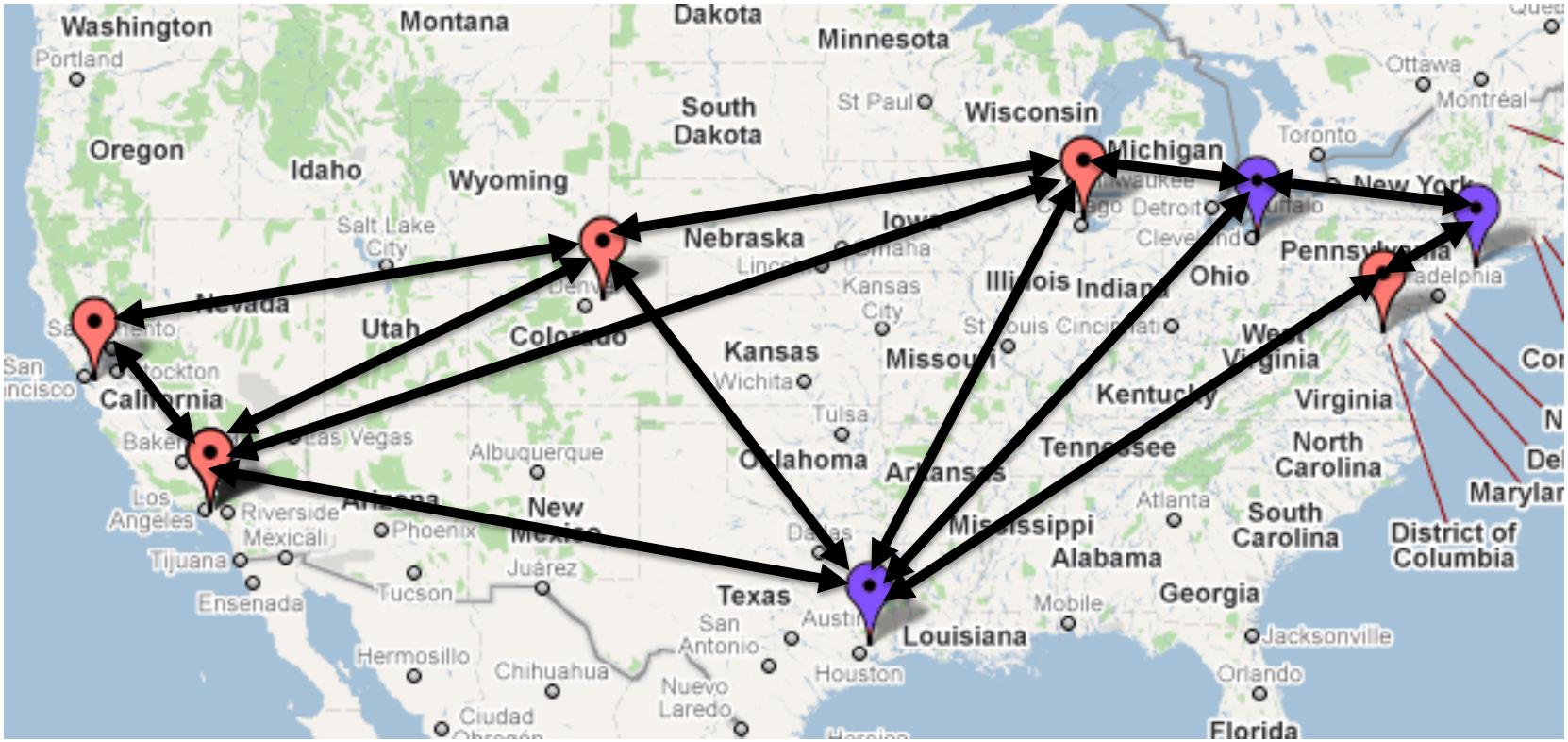
Graphs and Search

- Suppose you had data on flights between cities in the US, with prices
  - And you assume that for all cities, A, B and C, the cost of flying from A to C by way of B is the same as the cost of flying from A to B plus the cost from B to C
- You might like to know
  - Fewest stops between two cities
  - Least expensive airfare between two cities
  - Least expensive airfare between two cities, with no more than two stops

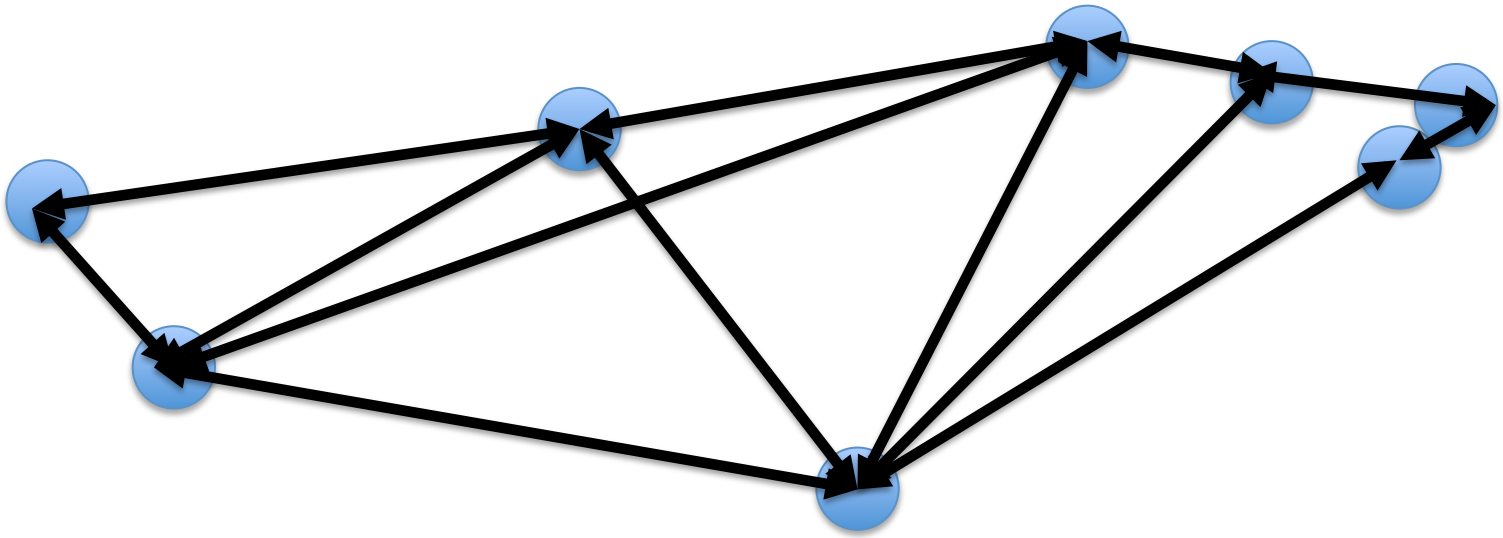
An example


Washington
Portland
Montana
Dakota
Minnesota
Ottawa
Montréal
South Dakota
St Paul
Wisconsin
Michigan
Toronto
Oregon
Idaho
Wyoming
Chicago
Detroit
Cleveland
New York
Salt Lake City
Nebraska
Omaha
Lincoln
Iowa
Pennsylvania
Nevada
Denver
Kansas City
Illinois
Indiana
Ohio
Utah
Colorado
St Louis
Cincinnati
West Virginia
Kansas
Missouri
Stockton
California
Wichita
Kentucky
Virginia
Tulsa
Las Vegas
Albuquerque
Tennessee
North Carolina
Oklahoma
Arkansas
Arizona
New Mexico
Atlanta
Riverside
Los Angeles
Phoenix
Dallas
Mississippi
South Carolina
Tijuana
Mexicali
Juárez
Alabama
District of Columbia
Ensenada
Tucson
Texas
Georgia
Mobile
San Antonio
Austin
Louisiana
Jacksonville
Hermosillo
Chihuahua
Houston
Orlando
Nuevo Laredo
Ciudad
Florida

An example


Washington
Portland
Montana
Dakota
Minnesota
Oregon
Idaho
Wyoming
South Dakota
St Paul
Wisconsin
Michigan
Ottawa
Montréal
Toronto
Salt Lake City
Nebraska
Iowa
Detroit
Cleveland
Pennsylvania
Nevada
Utah
Colorado
Kansas City
Illinois
Indiana
Ohio
Kansas
Missouri
Wichita
Tulsa
Kentucky
Virginia
San Francisco
Stockton
California
Las Vegas
Albuquerque
Oklahoma
Arkansas
Tennessee
North Carolina
Baker
Los Angeles
Riverside
Arizona
New Mexico
Phoenix
Atlanta
South Carolina
Mexicali
Tijuana
Ensenada
Juárez
Tucson
Mississippi
Alabama
Georgia
District of Columbia
Texas
Austin
San Antonio
Houston
Louisiana
Mobile
Jacksonville
Hermosillo
Chihuahua
Nuevo Laredo
Ciudad
Orlando
Florida

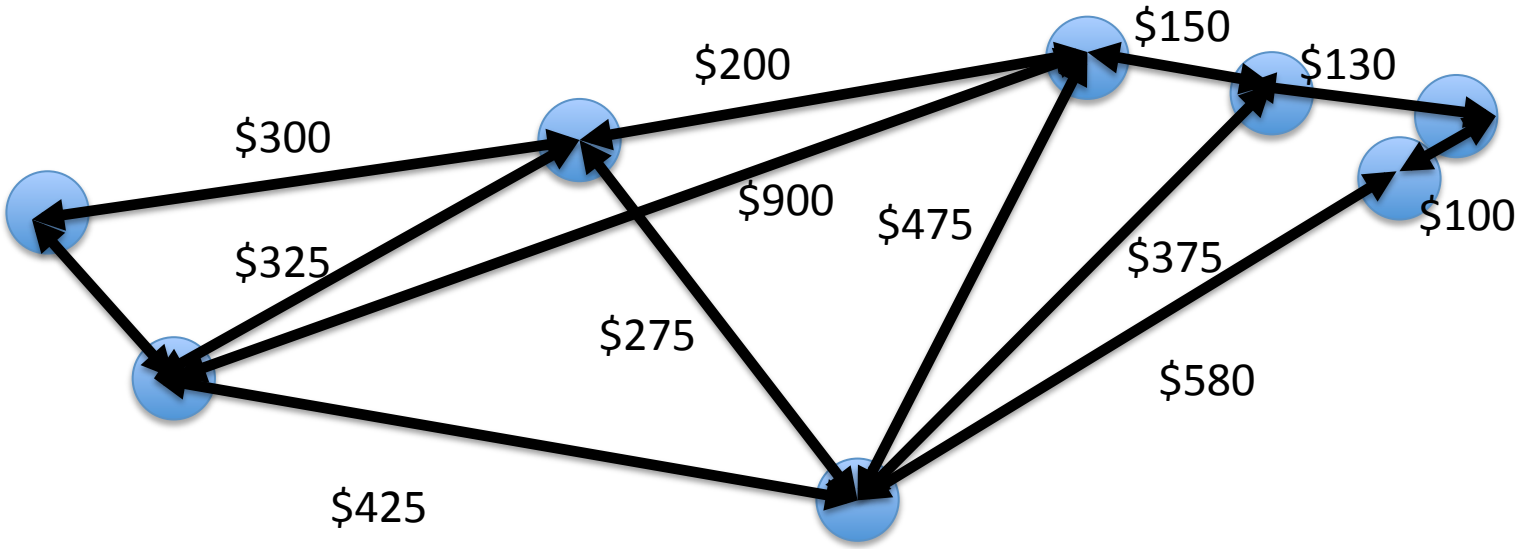
Graph abstraction



What is a graph

- Set of **nodes (or vertices)** 
  - connected by a set of **edges (or arcs)** 
- If edges are unidirectional, the graph is a **directed graph (or digraph)** 
- If we add a weight (or cost) to each edge, then the graph is a **weighted graph**

Weighted graph abstraction



Why are graphs interesting?

- Represent situations with interesting relationships among parts
- Naturally lead to questions that can be answered by search problems
- Can capture a wide range of problems
  - World Wide Web traffic (nodes are pages, edges are links, weights are how often links used)
  - Epidemiology (nodes are people with information about a disease, edges indicate interactions between people, weights indicate level of interaction)

Graph elements

```
class Node(object):
    def __init__(self, name):
        self.name = str(name)
    def getName(self):
        return self.name
    def __str__(self):
        return self.name

class Edge(object):
    def __init__(self, src, dest):
        self.src = src
        self.dest = dest
    def getSource(self):
        return self.src
    def getDestination(self):
        return self.dest
    def __str__(self):
        return str(self.src) + '->' + str(self.dest)
```

Graph elements

```
class WeightedEdge(Edge):
    def __init__(self, src, dest, weight = 1.0):
        self.src = src
        self.dest = dest
        self.weight = weight
    def getWeight(self):
        return self.weight
    def __str__(self):
        return str(self.src) + '->(‘\
            + str(self.weight) + ')'\
            + str(self.dest)
```

Graph definition

```
class Digraph(object):
    def __init__(self):
        self.nodes = set([])
        self.edges = {}
    def addNode(self, node):
        if node in self.nodes:
            raise ValueError('Duplicate node')
        else:
            self.nodes.add(node)
            self.edges[node] = []
    def addEdge(self, edge):
        src = edge.getSource()
        dest = edge.getDestination()
        if not(src in self.nodes and dest in self.nodes):
            raise ValueError('Node not in graph')
        self.edges[src].append(dest)
    # more later
```

Graph definition

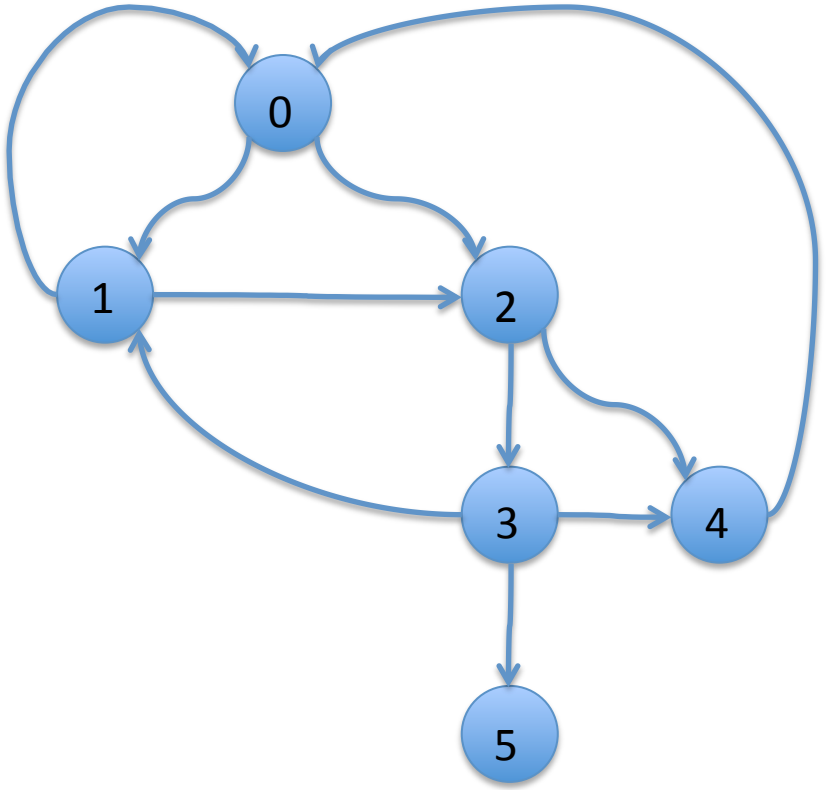
```
class Digraph(object):
    # more above
    def childrenOf(self, node):
        return self.edges[node]
    def hasNode(self, node):
        return node in self.nodes
    def __str__(self):
        res = ''
        for k in self.edges:
            for d in self.edges[k]:
                res = res + str(k) + '->' + str(d) + '\n'
        return res[:-1]

class Graph(Digraph):
    def addEdge(self, edge):
        Digraph.addEdge(self, edge)
        rev = Edge(edge.getDestination(), edge.getSource())
        Digraph.addEdge(self, rev)
```

Graph optimization problems

- **Shortest path**. For some pair of nodes, N1 and N2, find the shortest sequence of edges <sn, dn> (source node and destination node), such that
  - The source node in the first edge is N1
  - The destination node of the last edge is N2
  - For all a and b, if eb follows ea in the sequence, the source node of eb is the destination node of ea.
- **Shortest weighted path**. Like the shortest path, except instead of choosing the shortest sequence of edges that connects two nodes, define a function on the weights of the edges in the sequence (e.g., their sum) and minimize that value.
- **Cliques**. Find a set of nodes such that there is a path (or often a path with a maximum length) in the graph between each pair of nodes in the set.
- **Min cut**. Given two sets of nodes in a graph, a **cut** is a set of edges whose removal eliminates all paths from every node in one set to each node in the other. The minimum cut is the smallest set of edges whose removal accomplishes this.

Example

```
def testSP():
    nodes = []
    for name in range(6):
        nodes.append(Node(str(name)))
    g = Digraph()
    for n in nodes:
        g.addNode(n)
    g.addEdge(Edge(nodes[0],nodes[1]))
    g.addEdge(Edge(nodes[1],nodes[2]))
    g.addEdge(Edge(nodes[2],nodes[3]))
    g.addEdge(Edge(nodes[2],nodes[4]))
    g.addEdge(Edge(nodes[3],nodes[4]))
    g.addEdge(Edge(nodes[3],nodes[5]))
    g.addEdge(Edge(nodes[0],nodes[2]))
    g.addEdge(Edge(nodes[1],nodes[0]))
    g.addEdge(Edge(nodes[3],nodes[1]))
    g.addEdge(Edge(nodes[4],nodes[0]))
    sp = DFS(g, nodes[0], nodes[5])
    print 'Shortest path found by DFS:', printPath(sp)
```



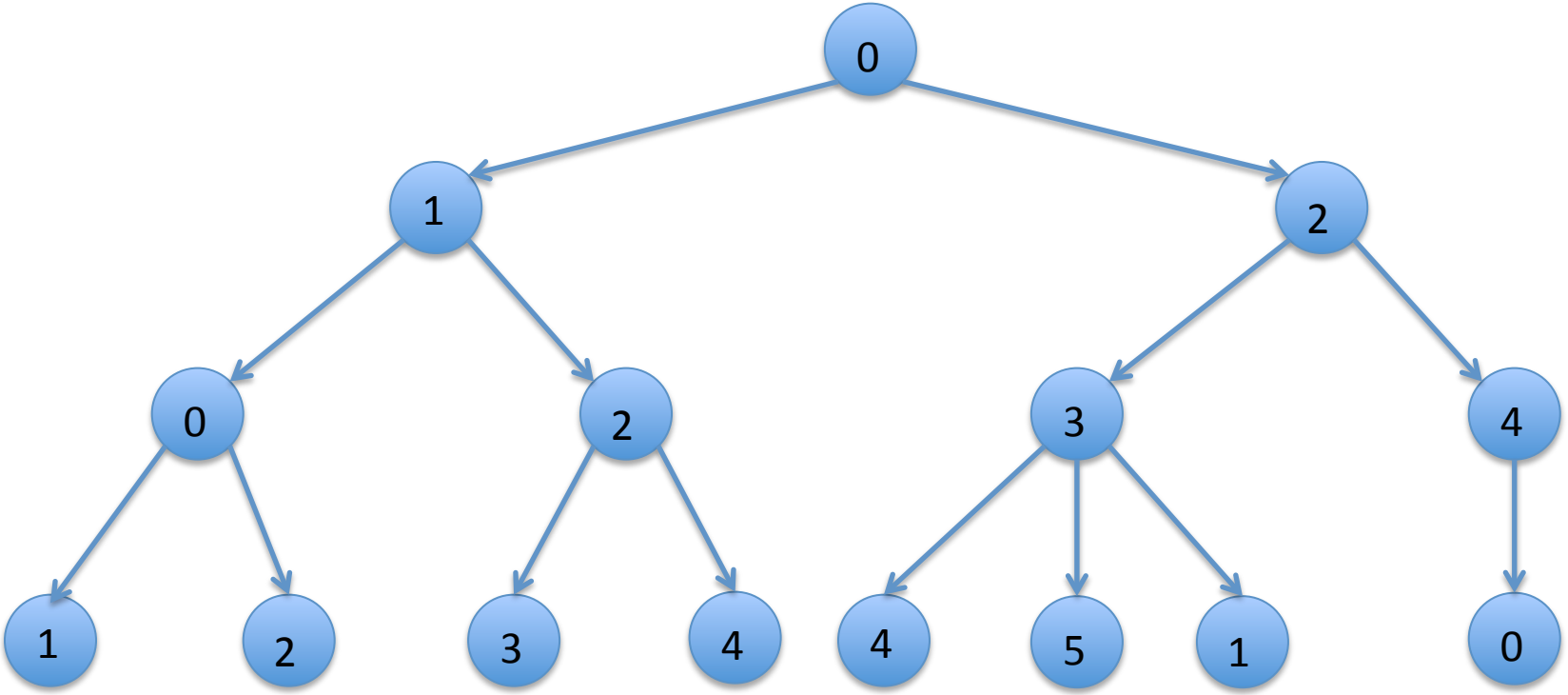
Search a graph

- Suppose we want to find the shortest path from node 0 to node 5
  - Just in terms of number of steps
  - Or might have weights on edges, and want to minimize total cost
- How might we find this path?

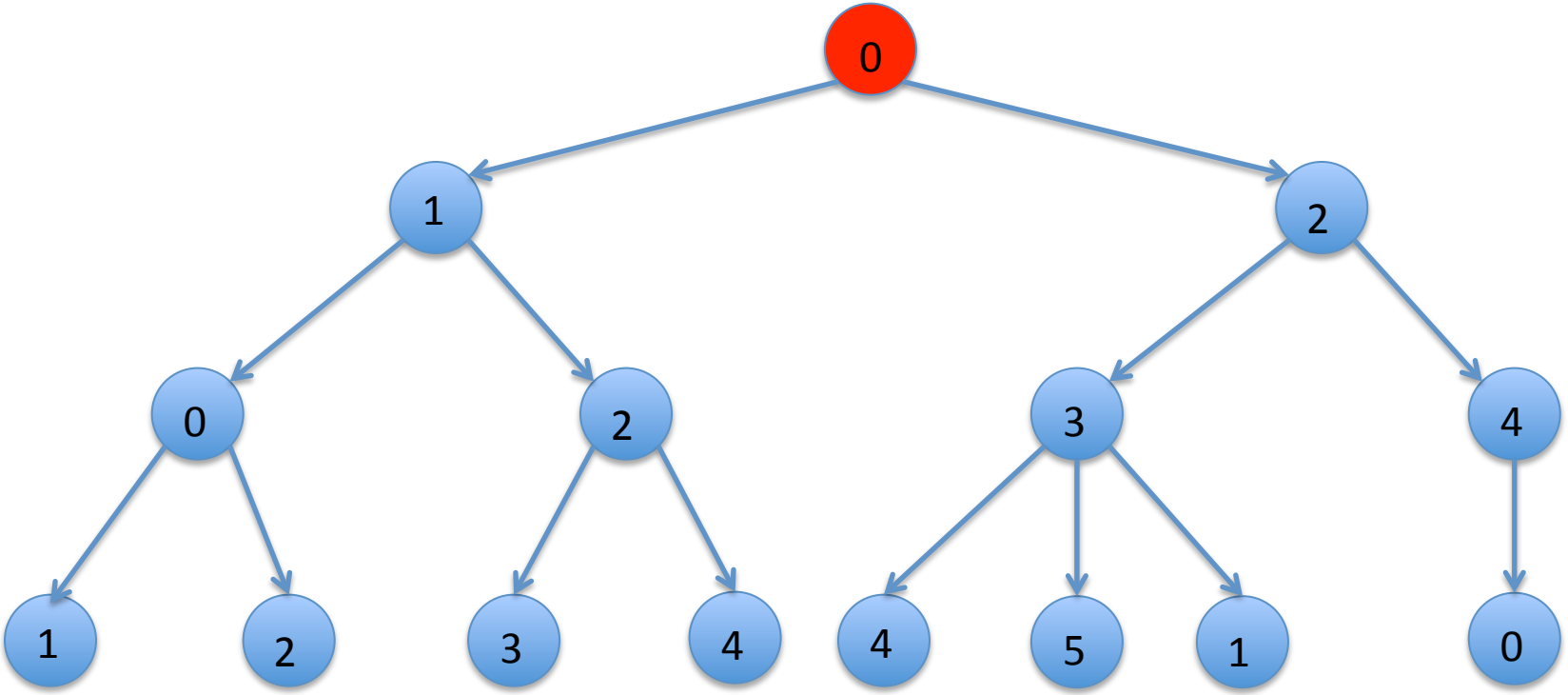
Depth first search

- Start at “root” node
  - Set of possible paths is just root node
- If not at “goal” node, then
  - Extend current path by adding each “child” of current node to path
  - Add these new paths to potential set of paths, at front of set
  - Select next path and recursively repeat
  - If current node has no “children”, then just go to next option
- Stop when reach “goal” node, or when no more paths to explore

A tree of solutions to explore

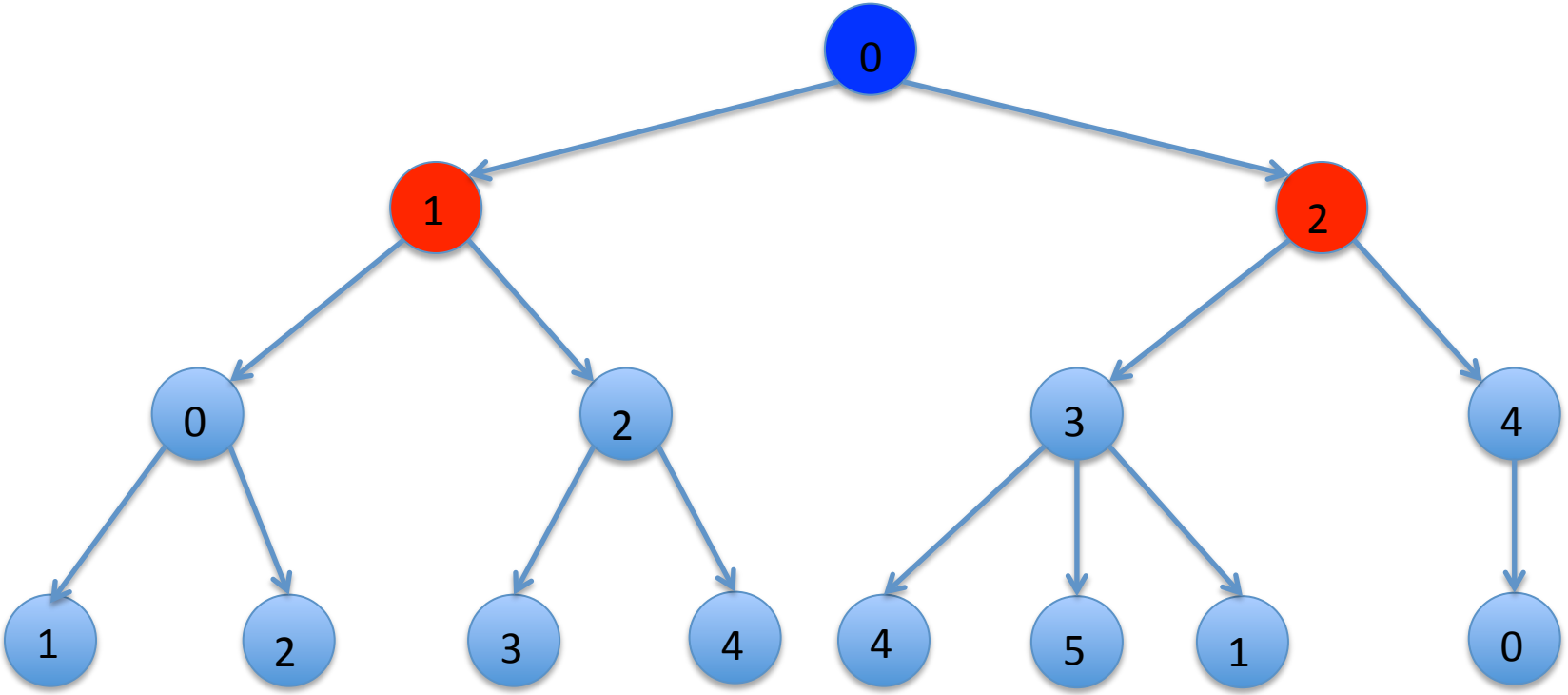


Simple depth first search



0

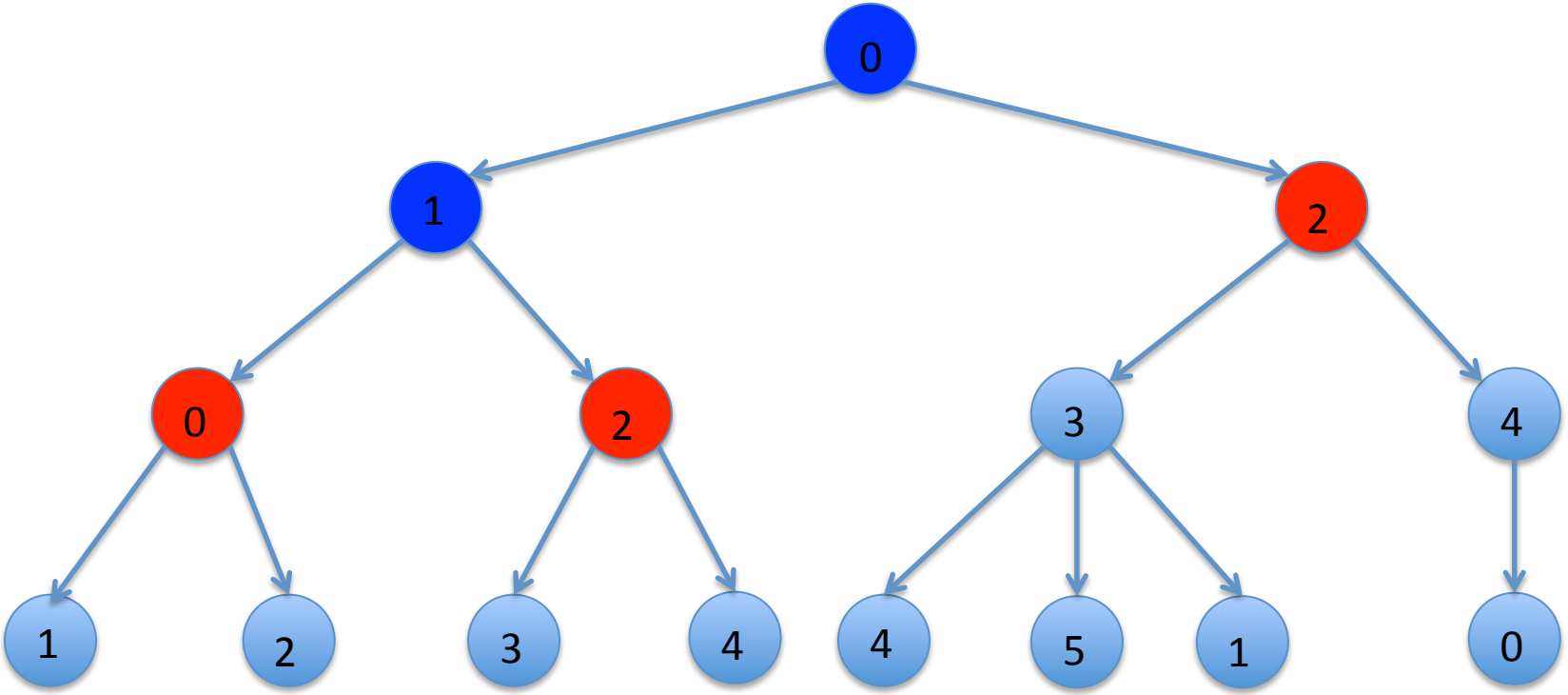
Simple depth first search



0

01 02

Simple depth first search

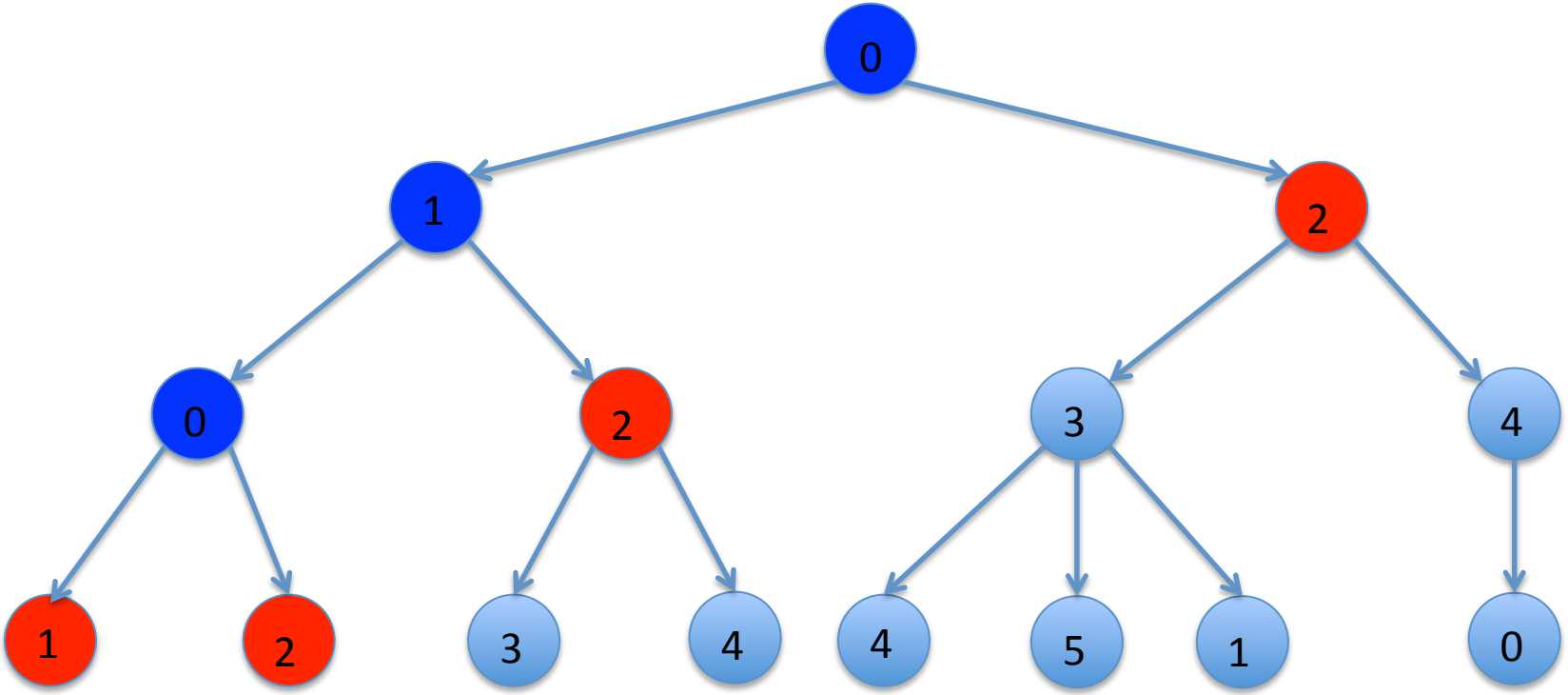


0

01 02

010 012 02

Simple depth first search



0

01 02

010 012 02

0101 0102 012 02

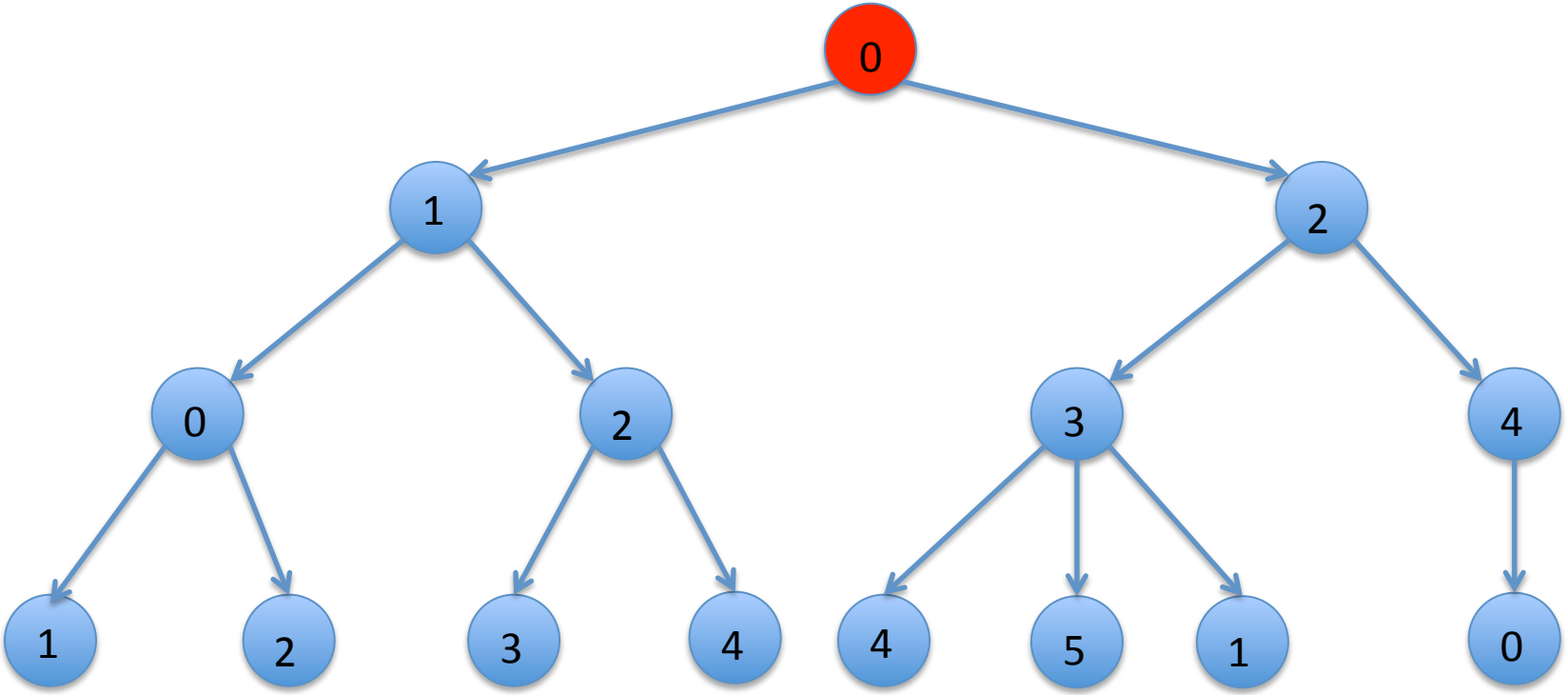
We don’t want to loop forever!

- If we keep going depth first in this tree, we will just keep cycling between node 0 and node 1
- Avoid visiting a node more than once

Depth first search

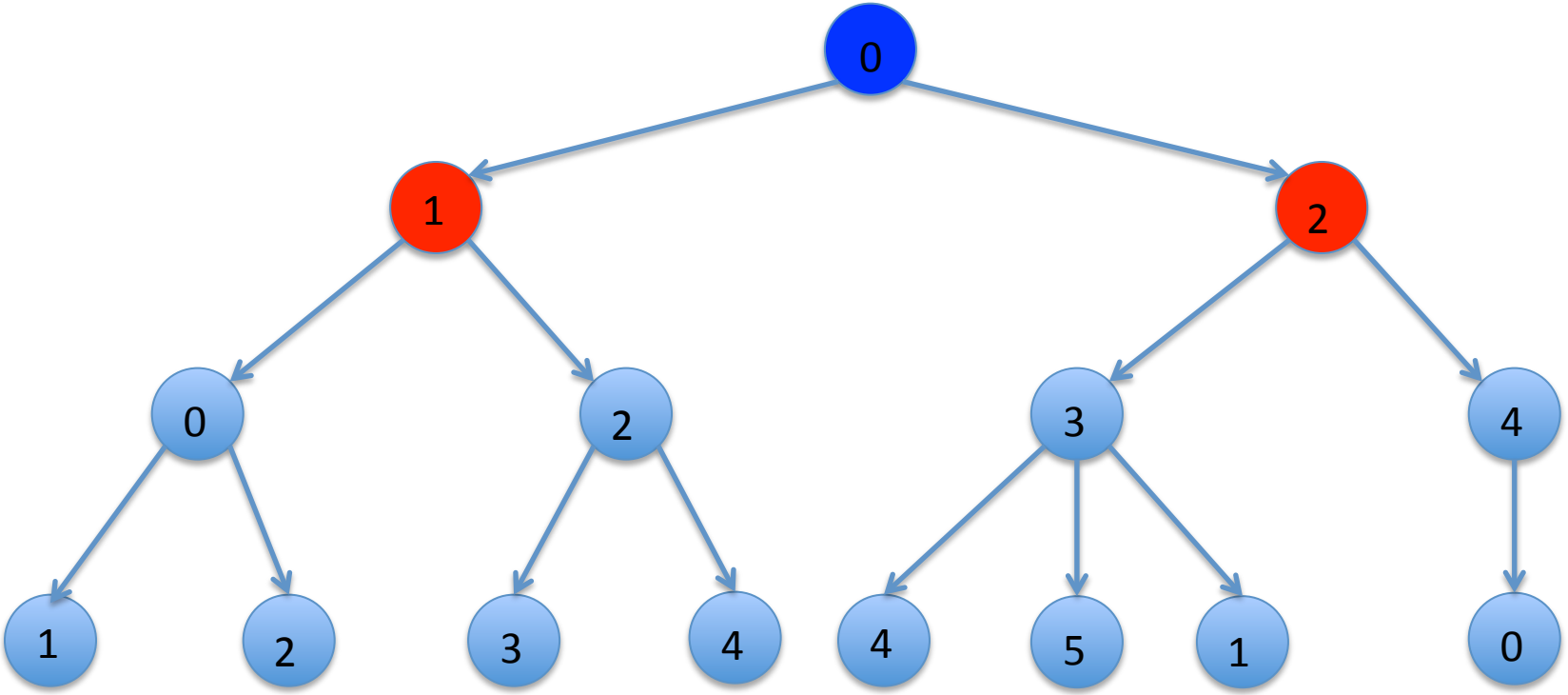
- Start at “root” node
  - Set of possible paths is just root node
- If not at “goal” node, then
  - Extend current path by adding each “child” of current node to path, unless child already in path
  - Add these new paths to potential set of paths, at front of set
  - Select next path and recursively repeat
  - If current node has no “children”, then just go to next option
- Stop when reach “goal” node, or when no more paths to explore

Better depth first search



0

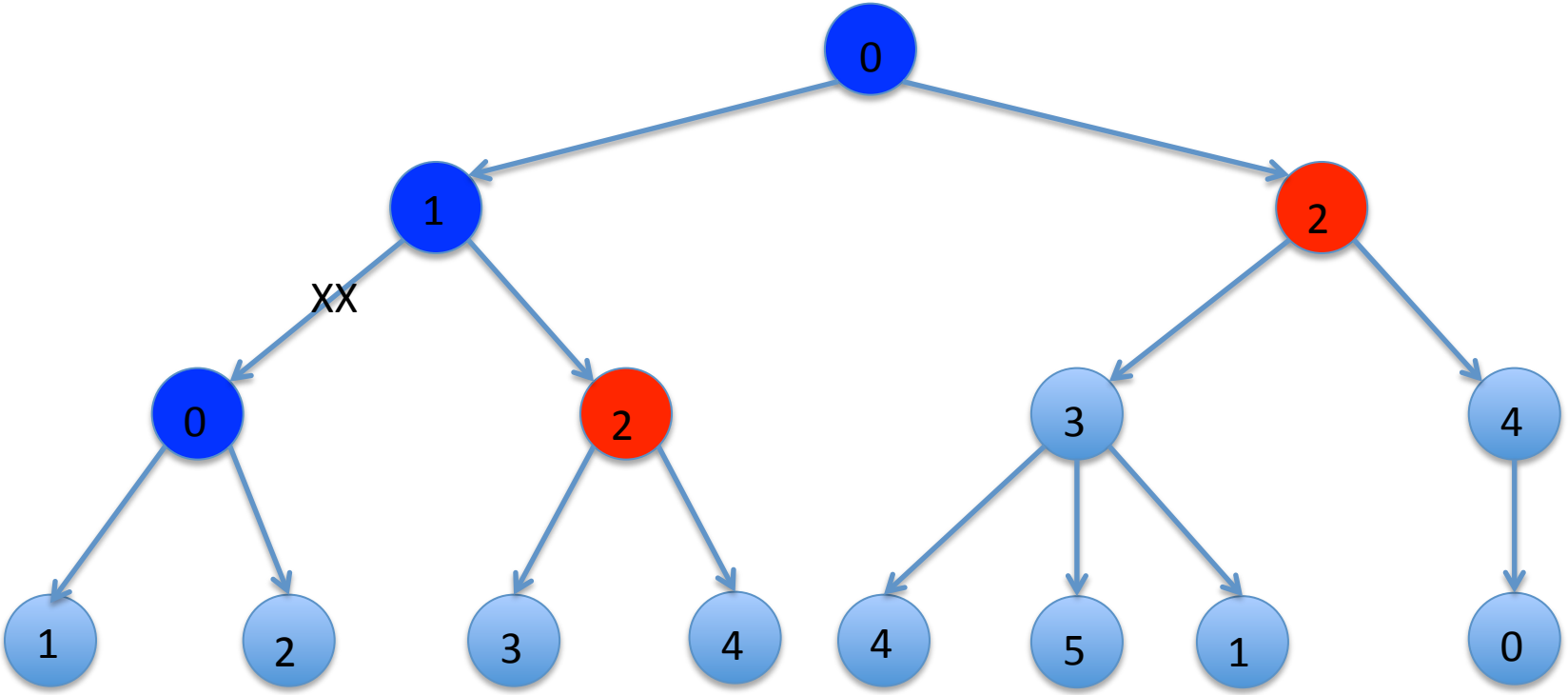
Better depth first search



0

01 02

Simple depth first search

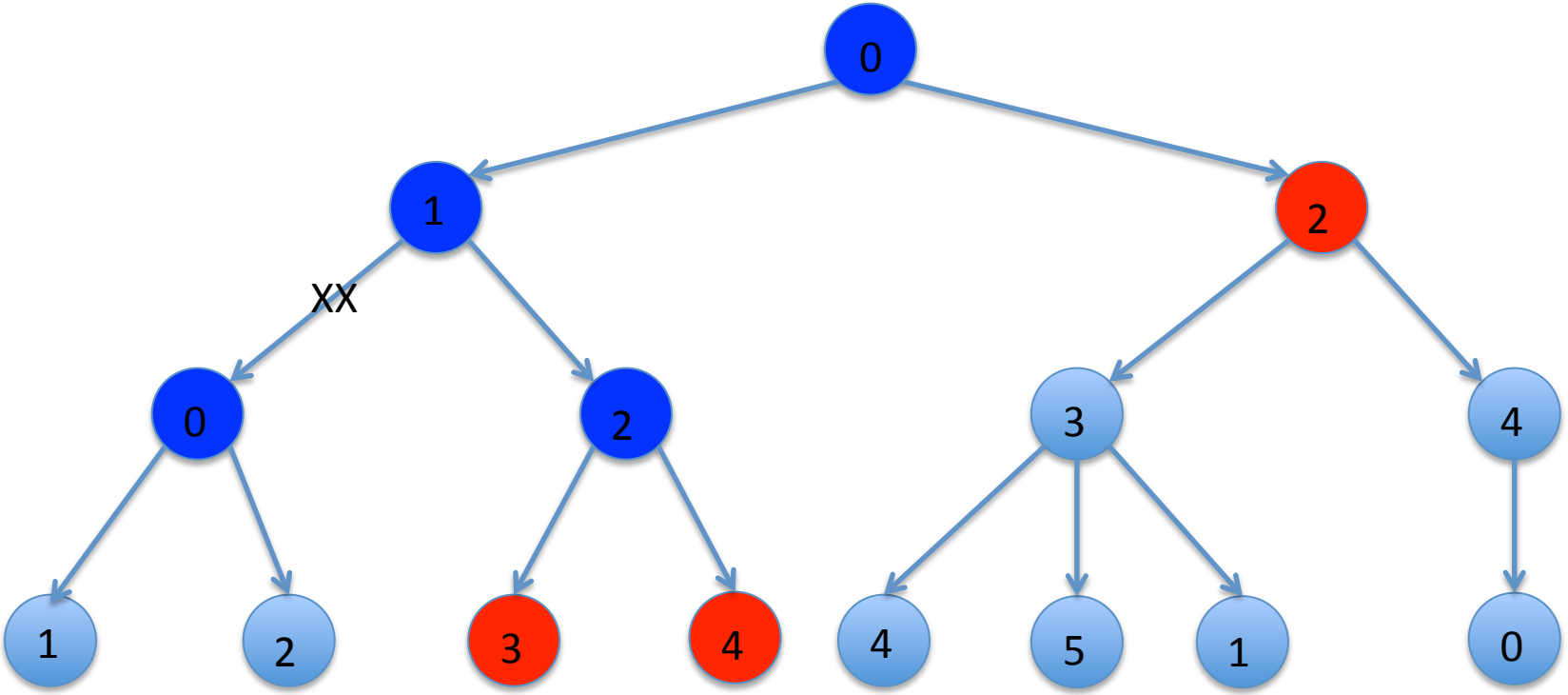


0

01 02

012 02

Simple depth first search



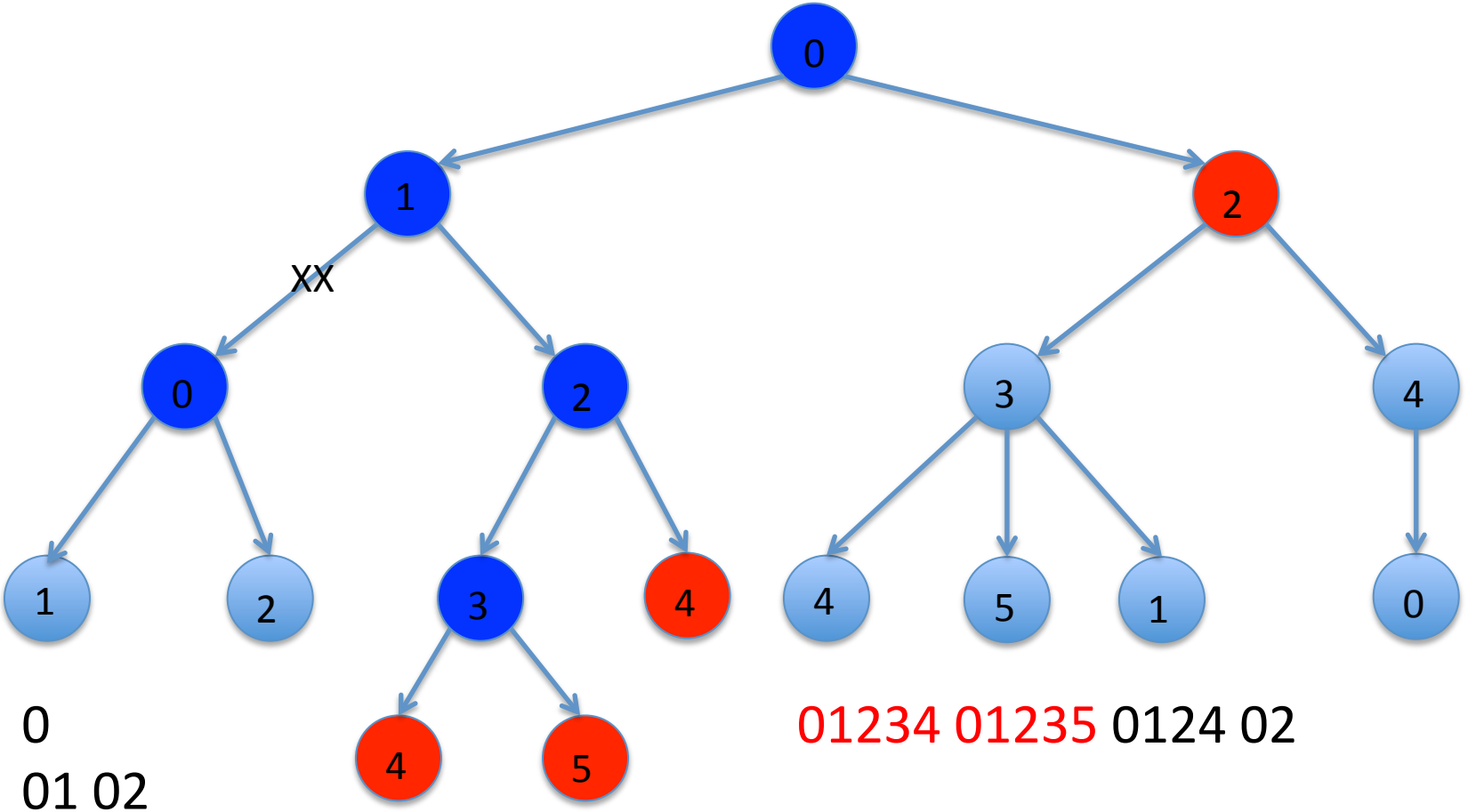
0

01 02

012 02

0123 0124 02

Simple depth first search



0

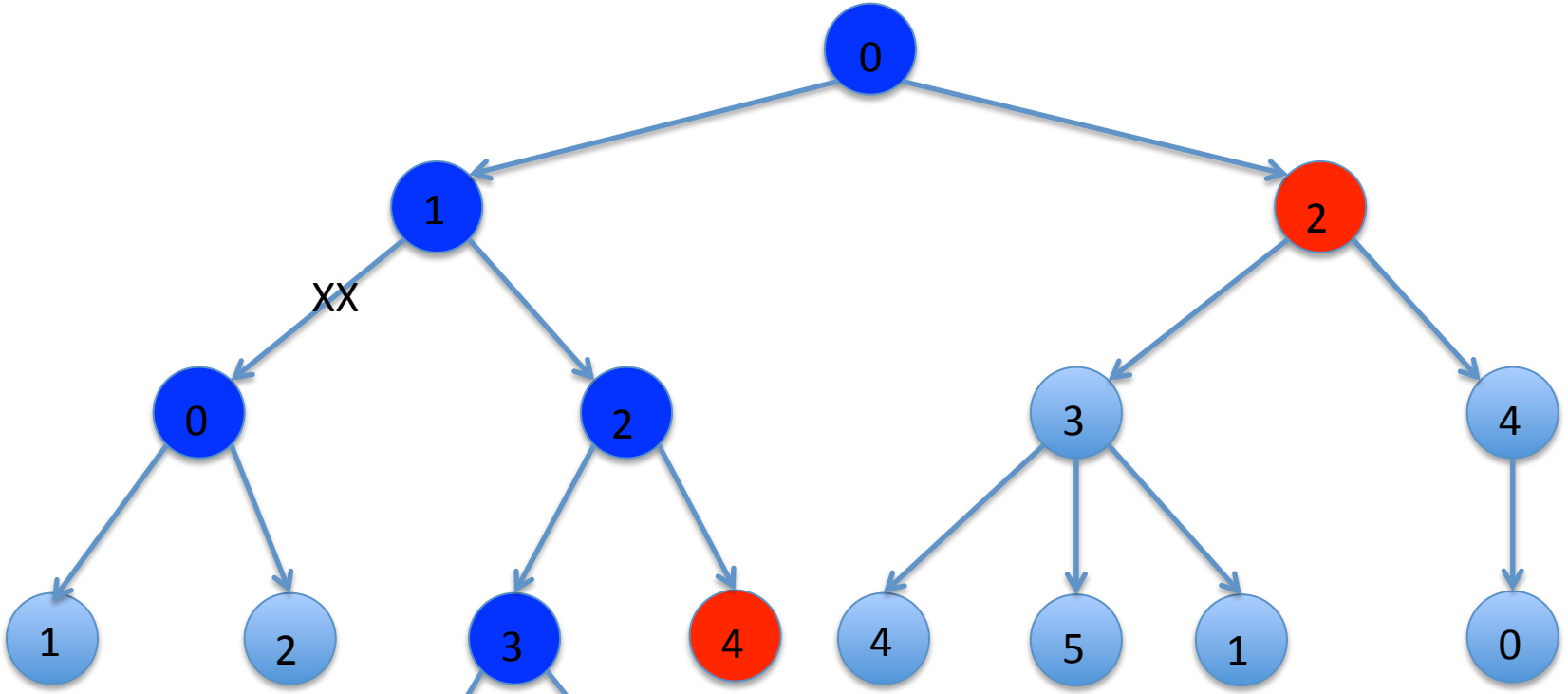
01 02

012 02

0123 0124 02

01234 01235 0124 02

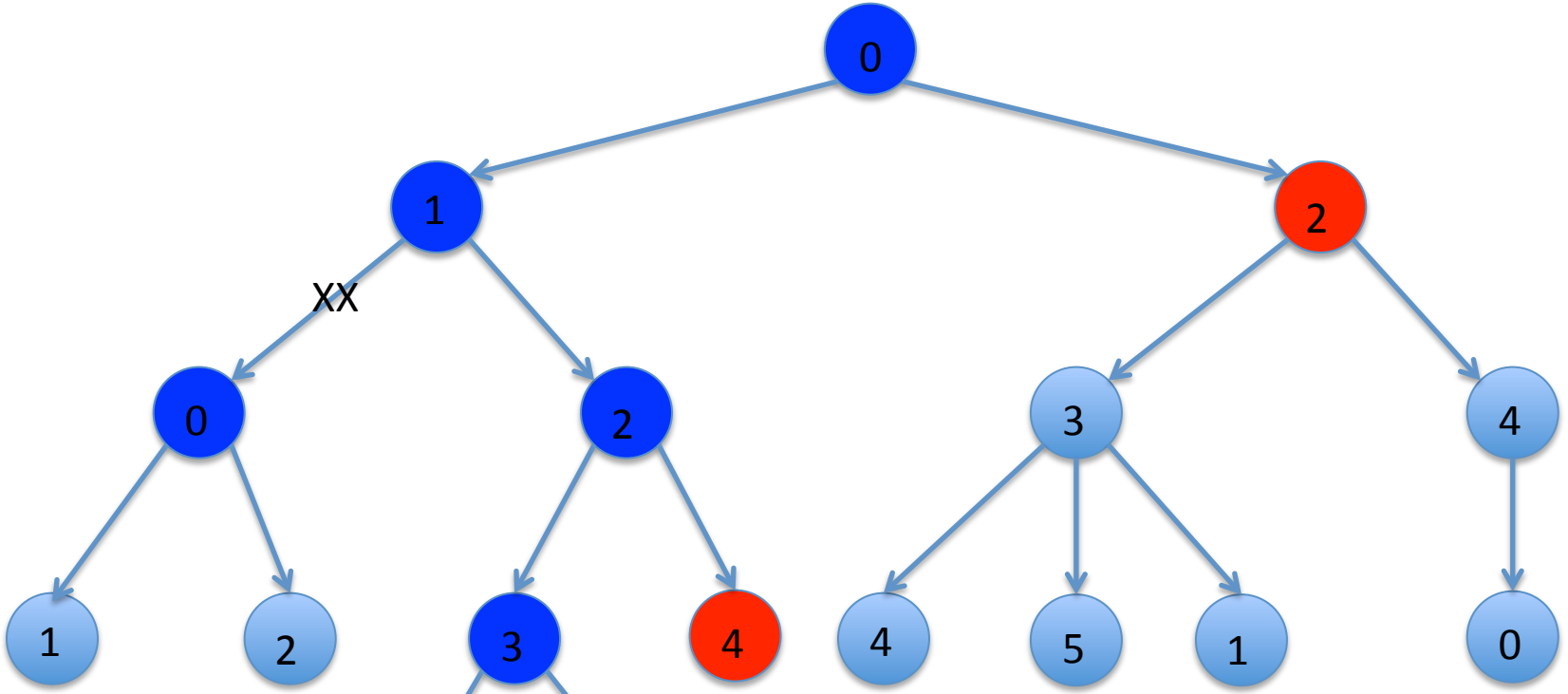
Simple depth first search



0
01 02
012 02
0123 0124 02

01234 01235 0124 02
01235 0124 02

Simple depth first search



0
01 02
012 02
0123 0124 02

01234 01235 0124 02
01235 0124 02

Some details

- Depth first search explores the first option in its set of possibilities
- It replaces the current first option by adding an ordered set of new options, extending the current path to each child node of the current node, and placing those options at the front of the set of possibilities (this is called a **stack**)
- It does not visit any child node already in the path
- As we have created the algorithm so far, it will stop once it finds a path

Sidebar: a stack

- A Stack is a data structure with a “last in, first out” behavior
  - We push items onto the top of the stack
  - We pop items off of the top of the stack
  - This maintains a set of items to be explored, where the top of the stack is the next item, and where new items are placed at top of the stack

Sidebar: a stack

- An example from our search
  - 0
  - 01 02 – pop 0 and push 01, 02
  - 012 02 – pop 01 and push 012
  - 0123 0124 02 – pop 012 and push 0123 and 0124
  - 01234 01235 0124 02 – pop 0123 and push 01234 and 01235

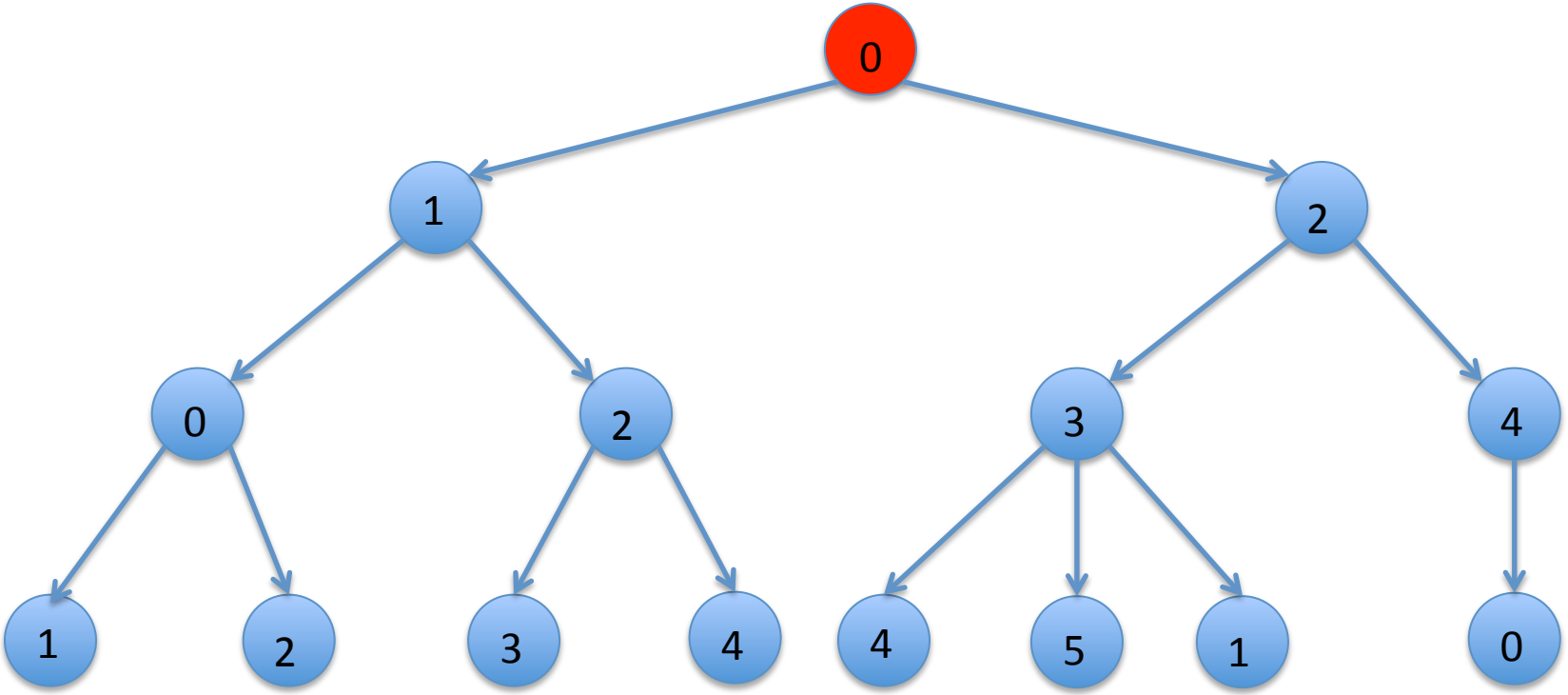
A simple DFS algorithm

```
def DFS(graph, start, end, path = []):
    # Assumes graph is a Digraph
    # Assumes start and end are nodes in graph
    path = path + [start]
    print 'Current dfs path:', printPath(path)
    if start == end:
        return path
    for node in graph.childrenOf(start):
        if node not in path: # Avoid cycles
            newPath = DFS(graph,node,end,path)
            if newPath != None:
                return newPath
    return None
```

To find shortest path

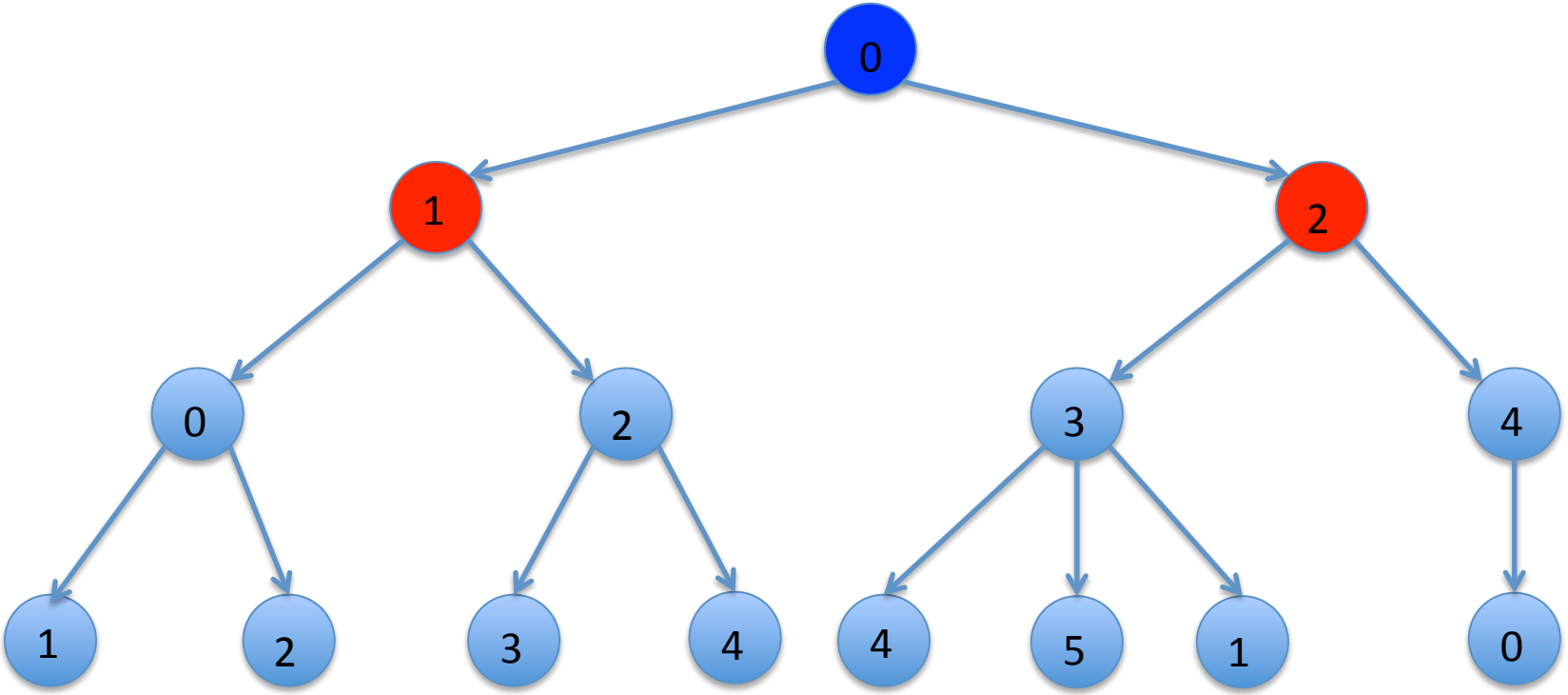
- We need to keep track of the best path found so far
- When we find a new path, keep going only if path still shorter than best seen so far

Better depth first search



0

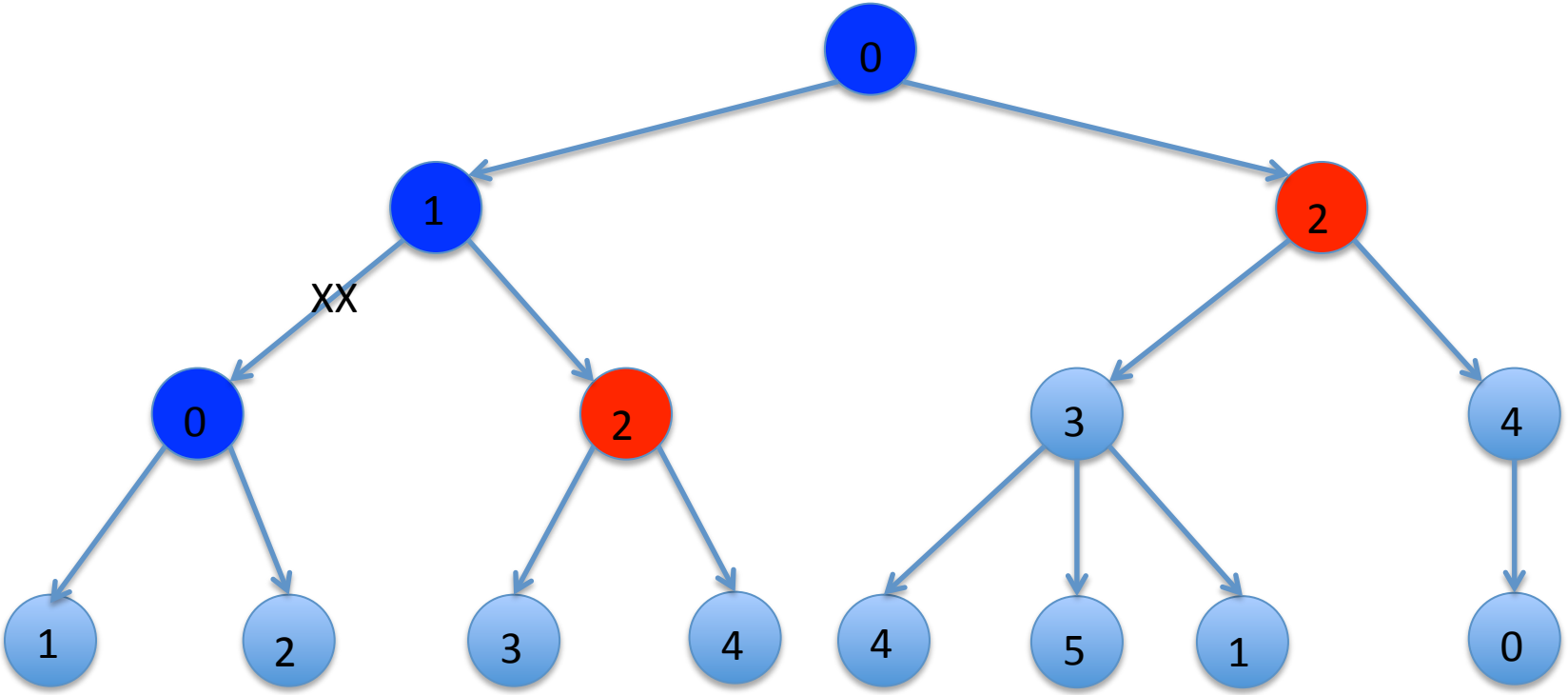
Better depth first search



0

01 02

Simple depth first search

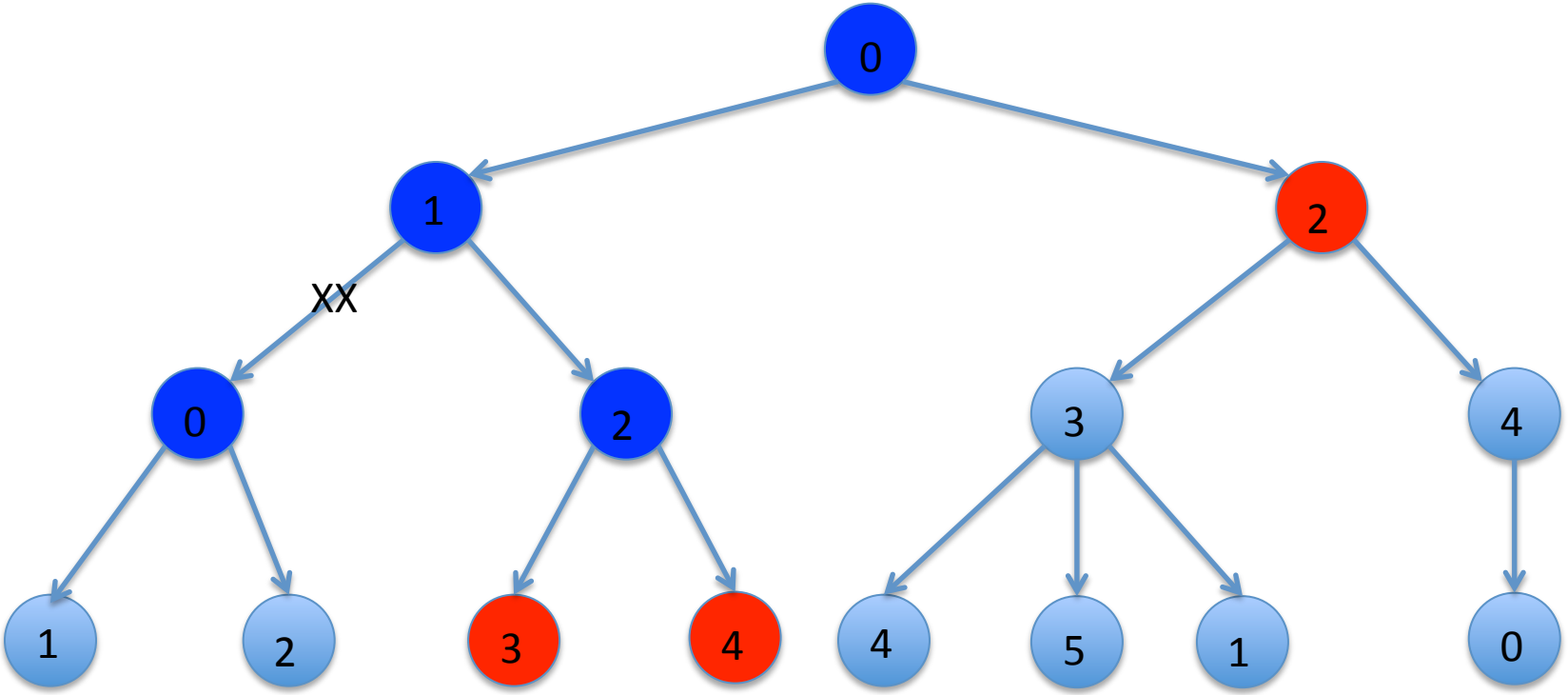


0

01 02

012 02

Simple depth first search



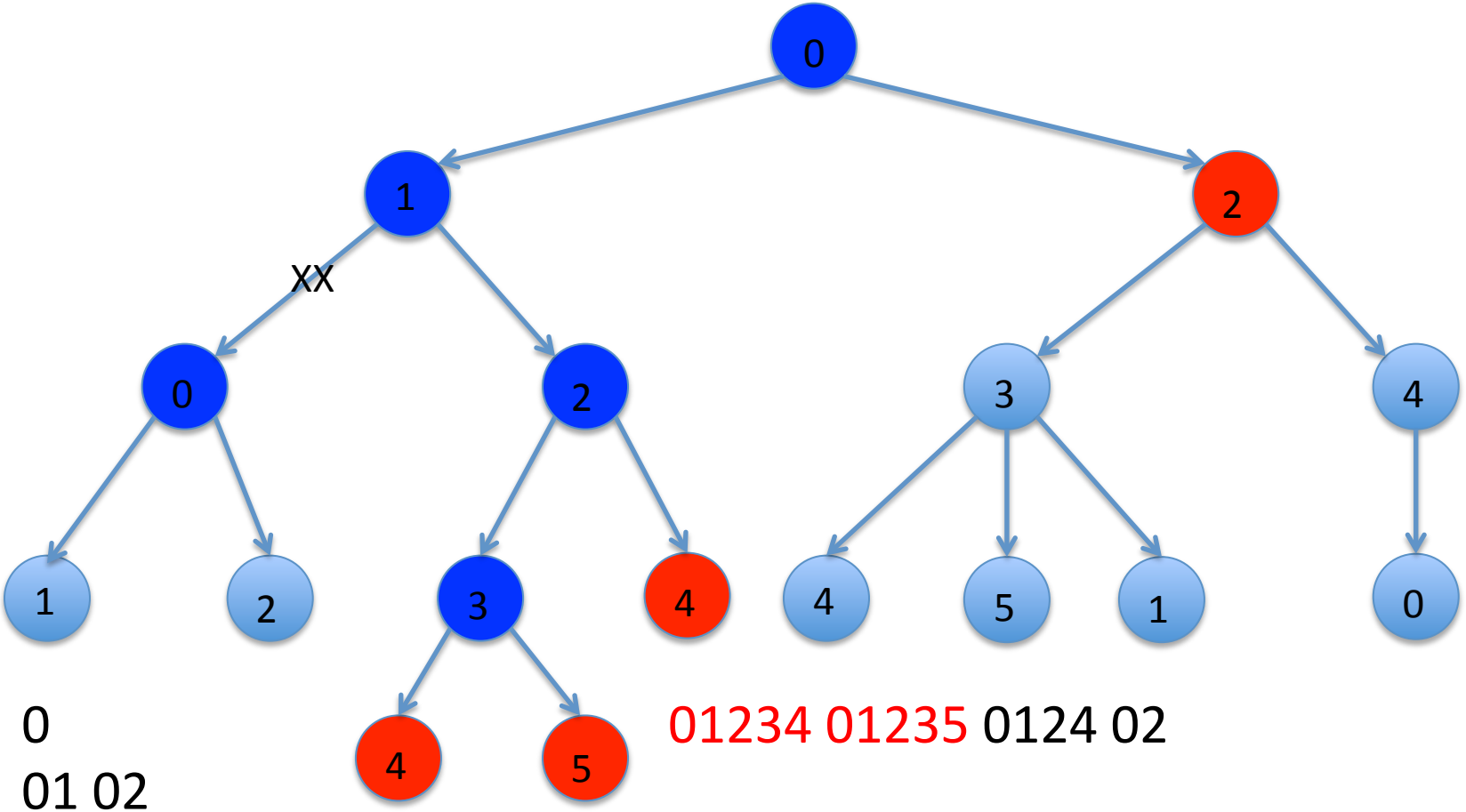
0

01 02

012 02

0123 0124 02

Simple depth first search



0

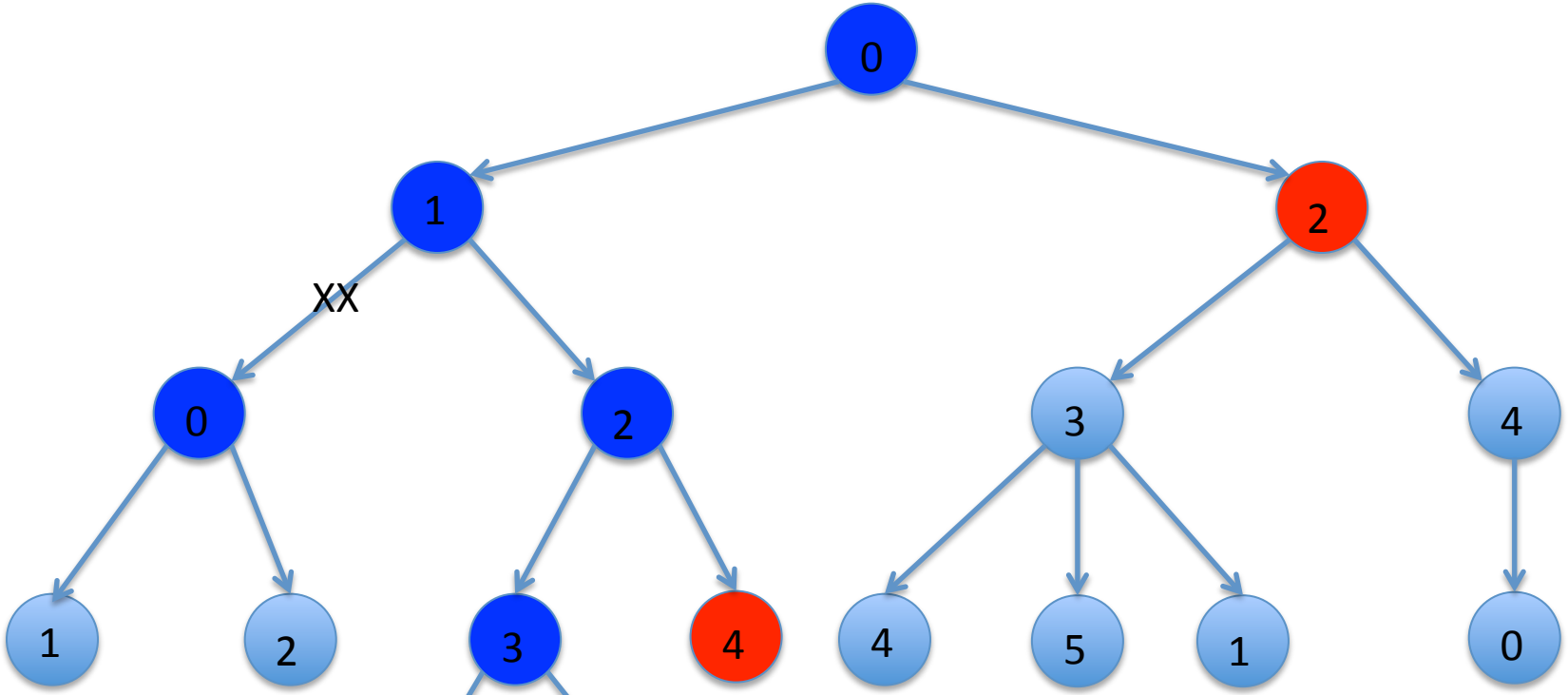
01 02

012 02

0123 0124 02

01234 01235 0124 02

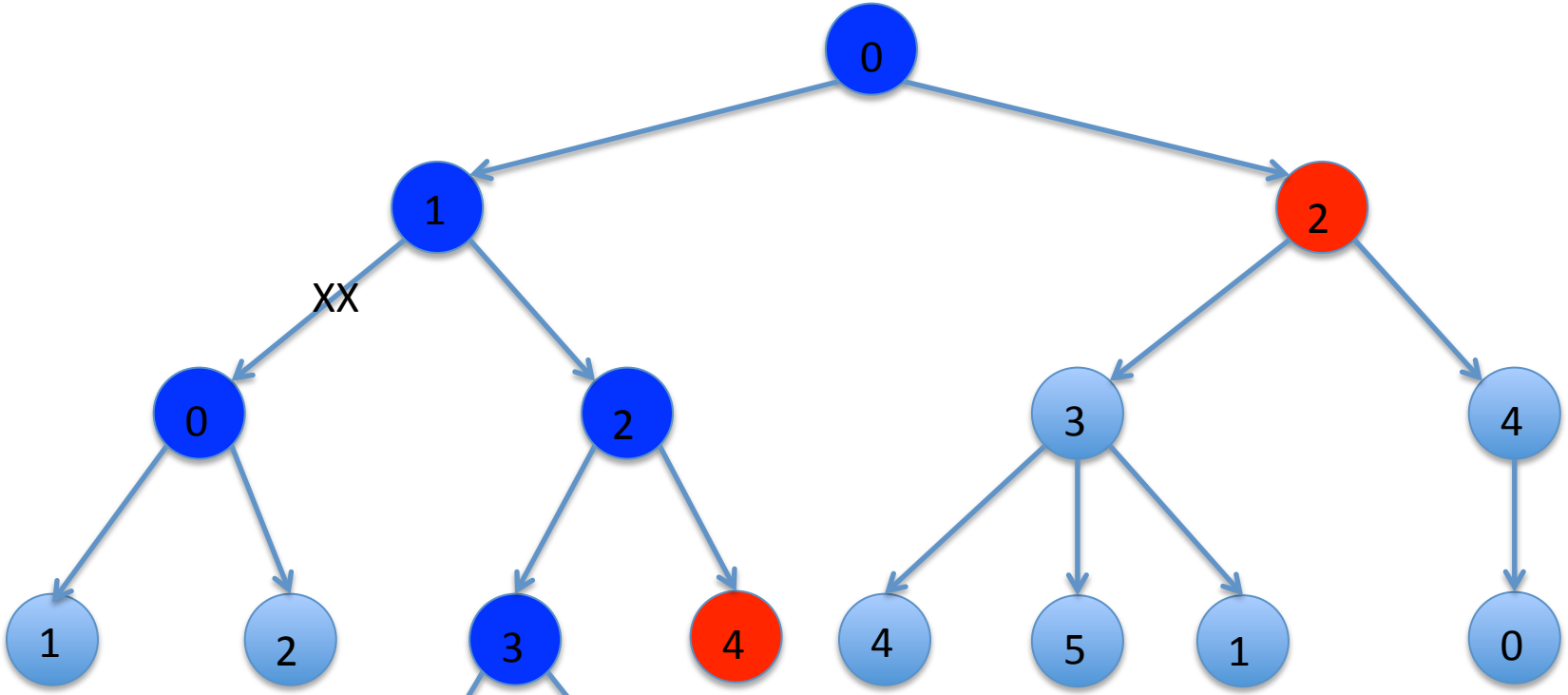
Simple depth first search



0
01 02
012 02
0123 0124 02

01234 01235 0124 02
01235 0124 02

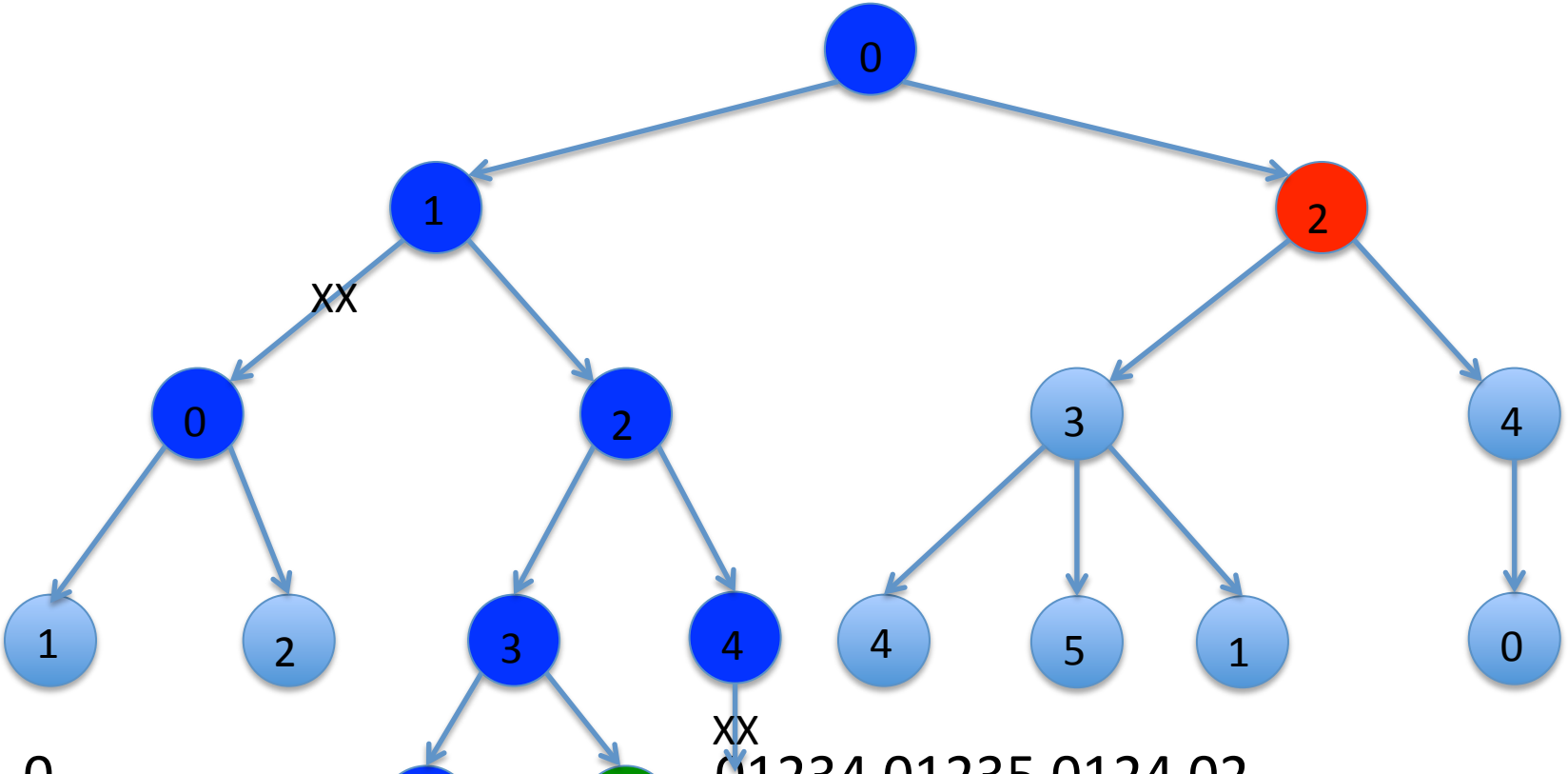
Simple depth first search



0
01 02
012 02
0123 0124 02

01234 01235 0124 02
01235 0124 02

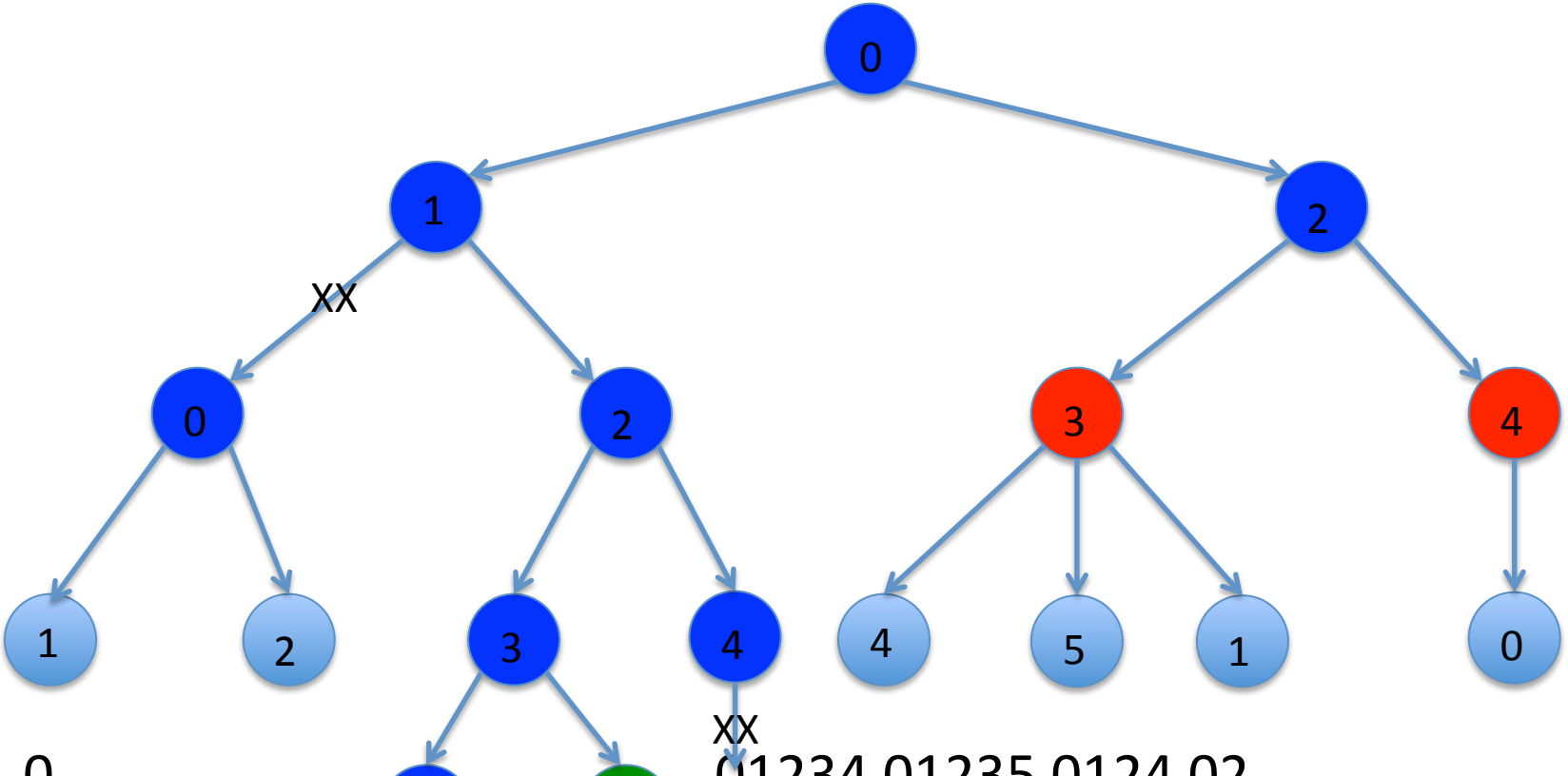
Simple depth first search



0
01 02
012 02
0123 0124 02

01234 01235 0124 02
01235 0124 02
02

Simple depth first search



0

01 02

012 02

0123 0124 02

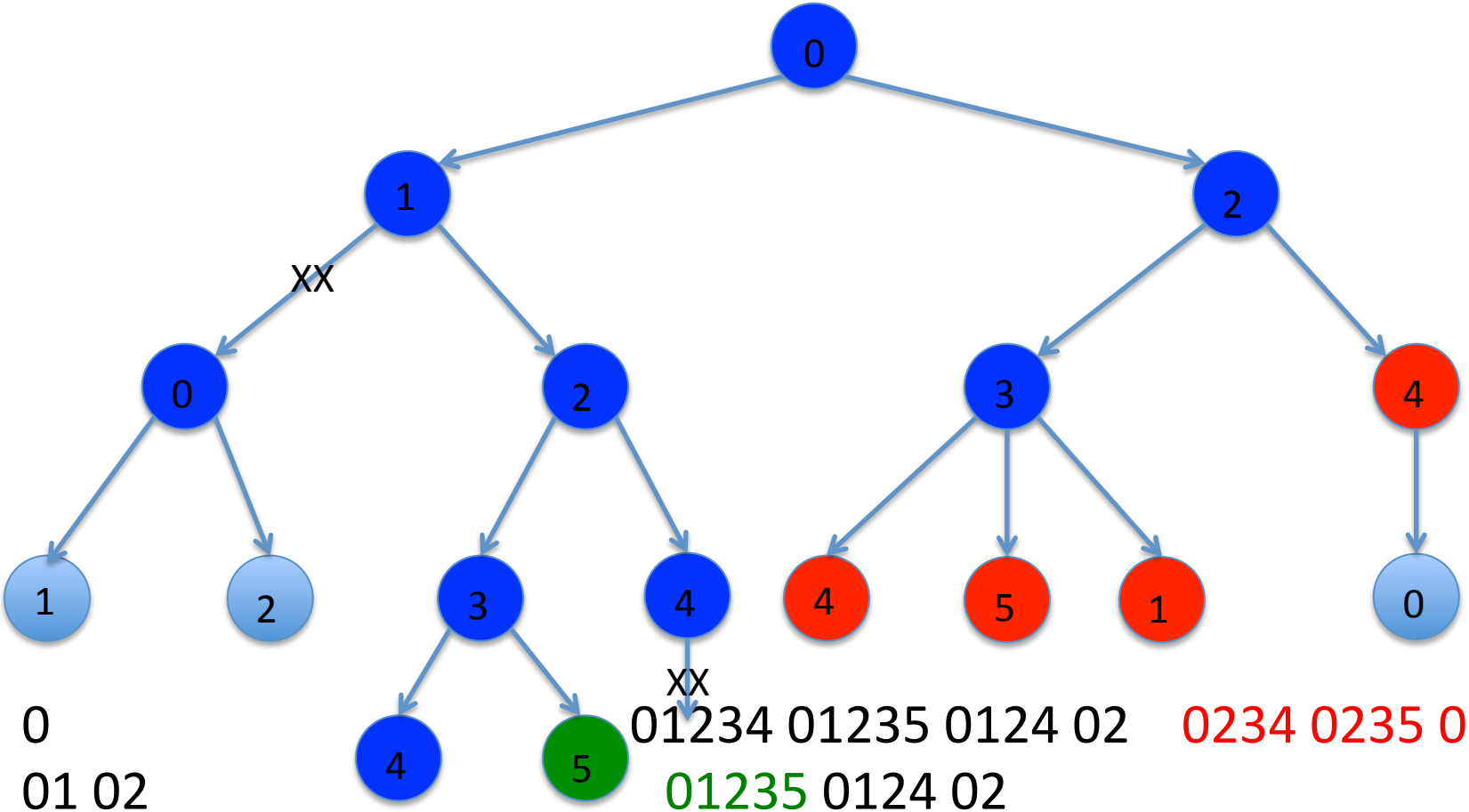
01234 01235 0124 02

01235 0124 02

02

023 024

Simple depth first search



0

01 02

012 02

0123 0124 02

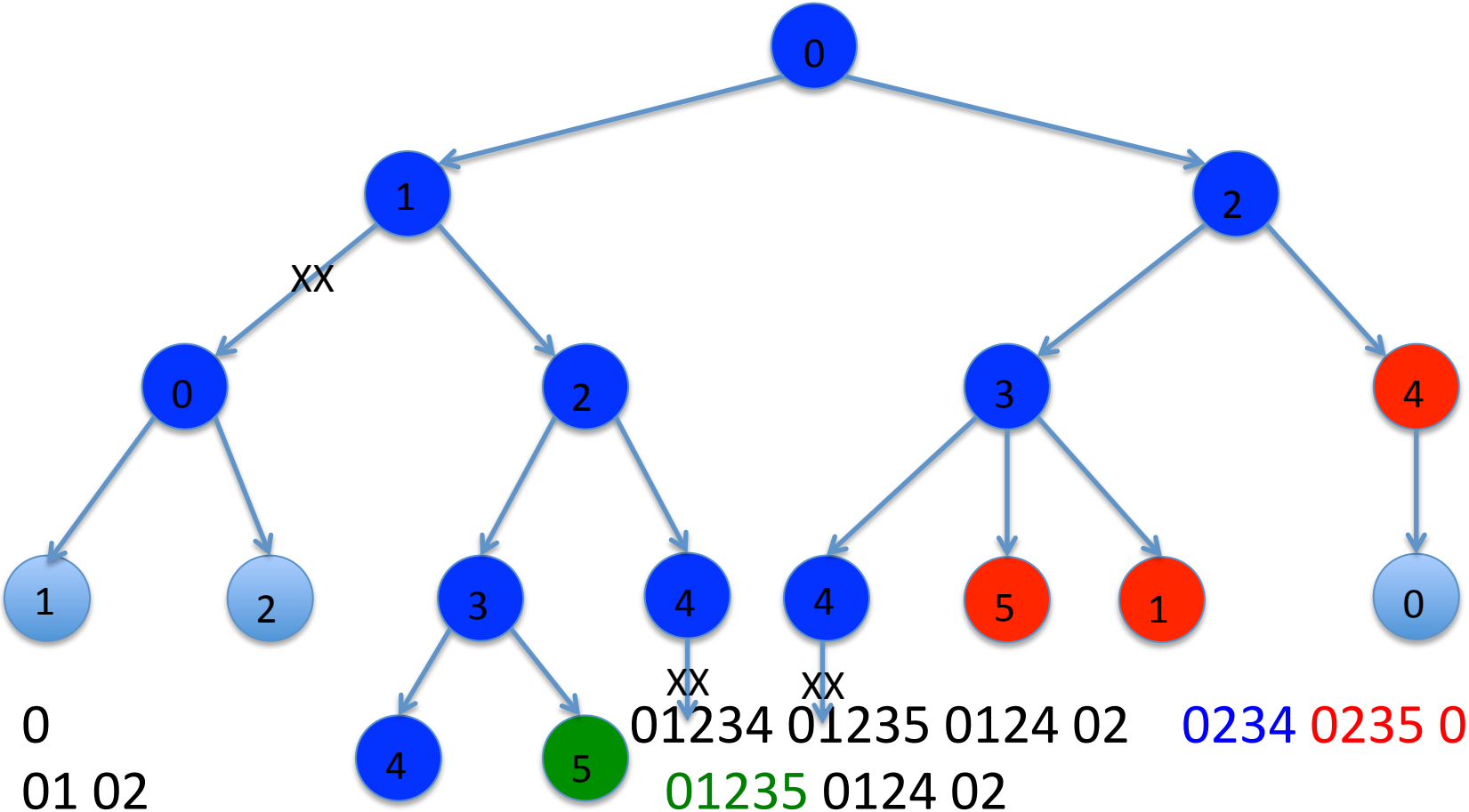
01234 01235 0124 02 0234 0235 0231 024

01235 0124 02

02

023 024

Simple depth first search



0

01 02

012 02

0123 0124 02

01234 01235 0124 02

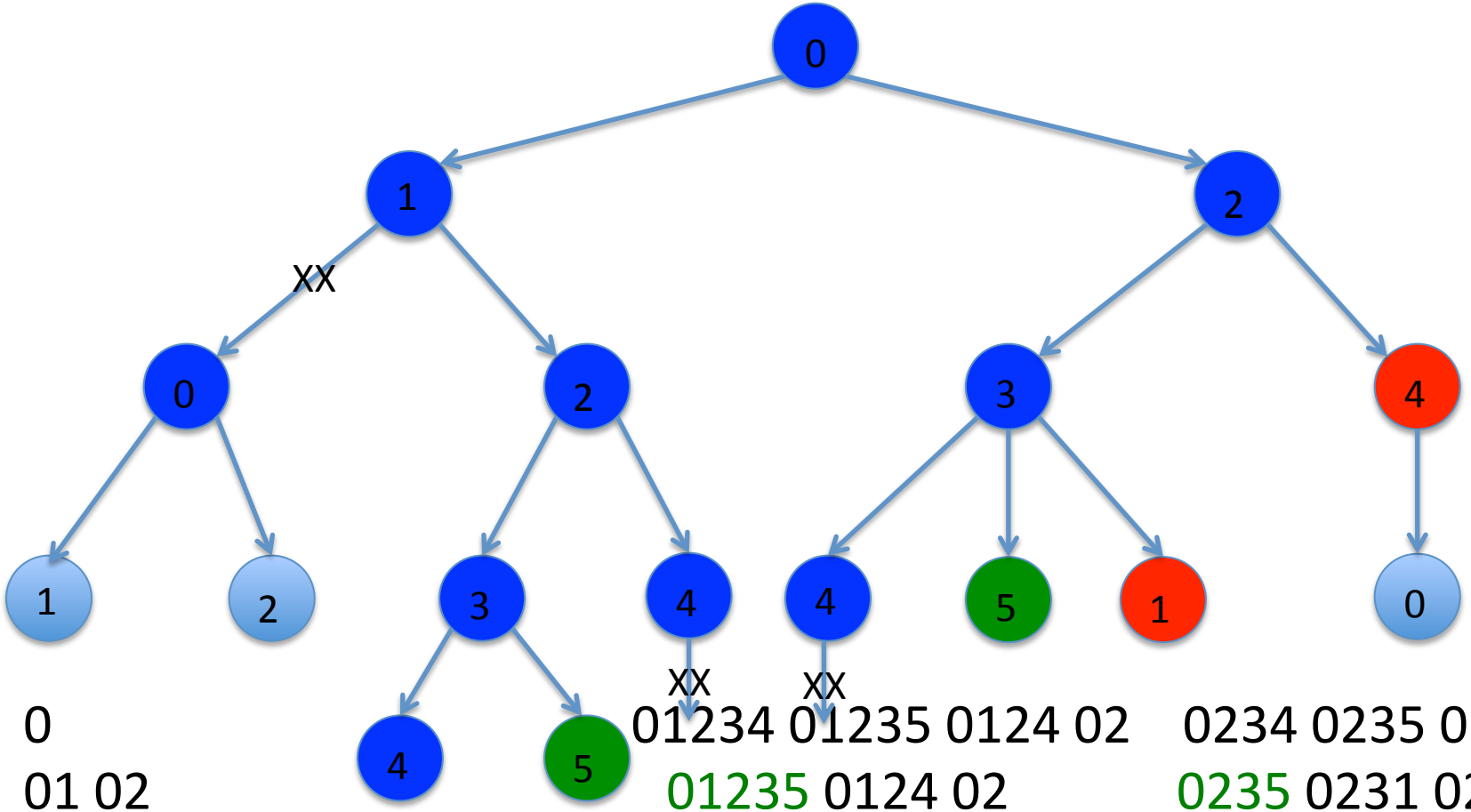
01235 0124 02

02

023 024

0234 0235 0231 024

Simple depth first search

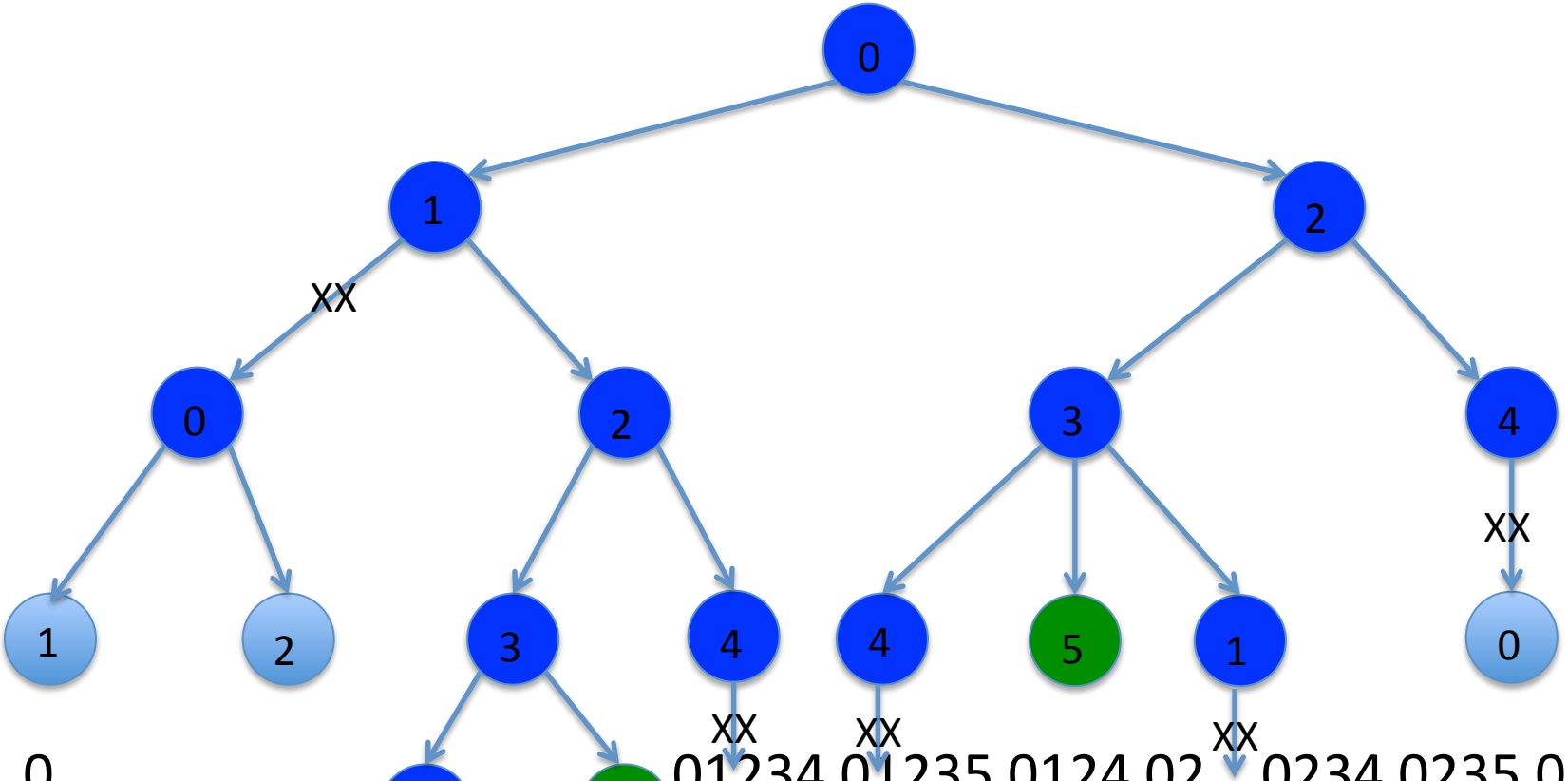


0
01 02
012 02
0123 0124 02

01234 01235 0124 02
01235 0124 02
02
023 024

0234 0235 0231 024
0235 0231 024

Simple depth first search

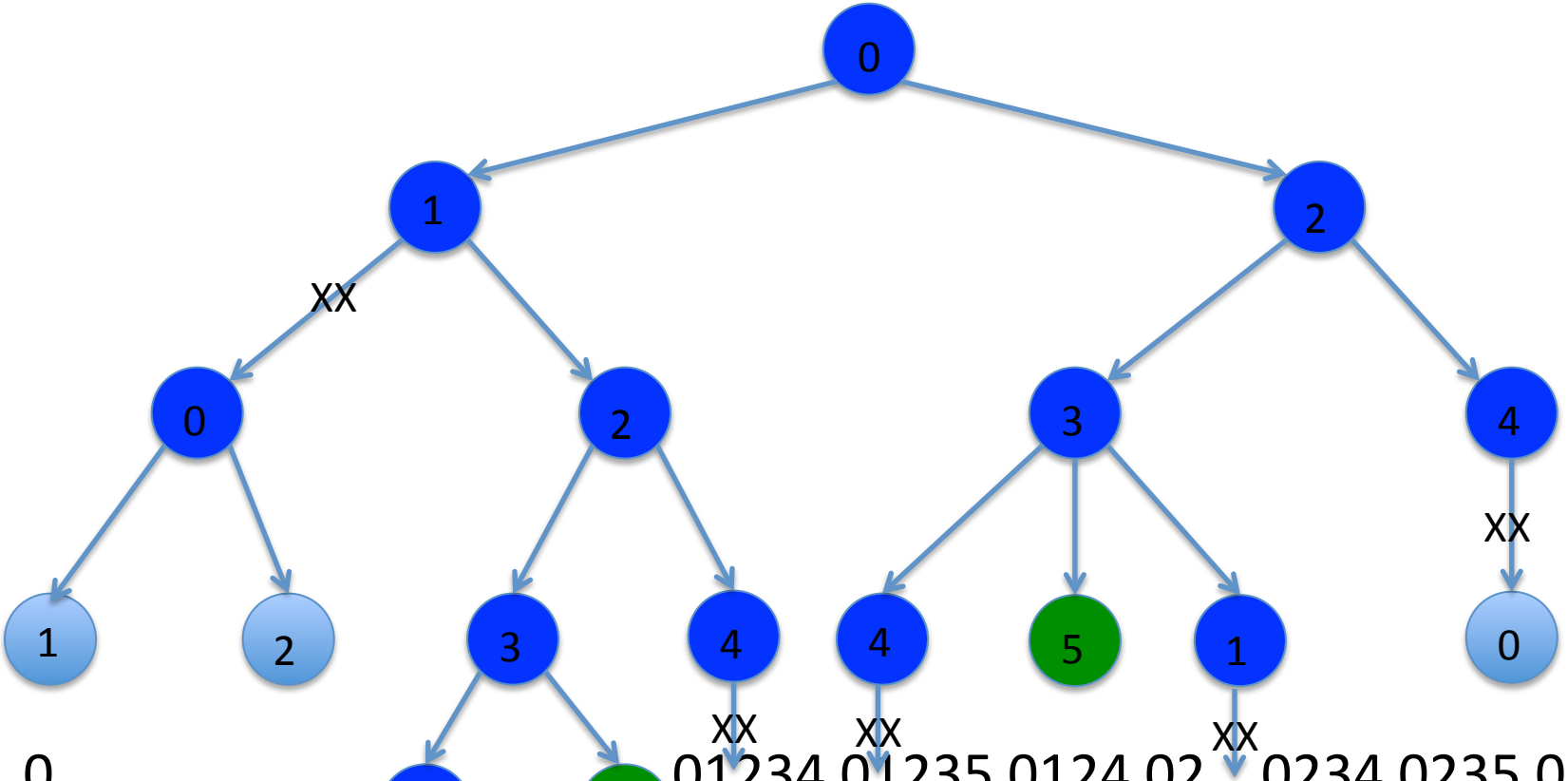


0
01 02
012 02
0123 0124 02

01234 01235 0124 02
01235 0124 02
02
023 024

0234 0235 0231 024
0235 0231 024
024

Simple depth first search



0
01 02
012 02
0123 0124 02

01234 01235 0124 02
01235 0124 02
02
023 024

0234 0235 0231 024
0235 0231 024
024

A shortest path DFS algorithm

```
def DFS(graph, start, end, path = [],shortest = None):
    # Assumes graph is a Digraph
    # Assumes start and end are nodes in graph
    path = path + [start]
    print 'Current dfs path:', printPath(path)
    if start == end:
        return path
    for node in graph.childrenOf(start):
        if node not in path: # Avoid cycles
            if shortest == None or len(path) < len(shortest):
                newPath = DFS(graph, node, end, path, shortest)
                if newPath != None:
                    shortest = newPath
    return shortest
```

Breadth first search

- Instead of going down the first branch of the tree, we could instead examine all children of a node first, before going deeper into tree
- In the simple case of no weights, we can stop as soon as we find a solution, since guaranteed to be shortest path

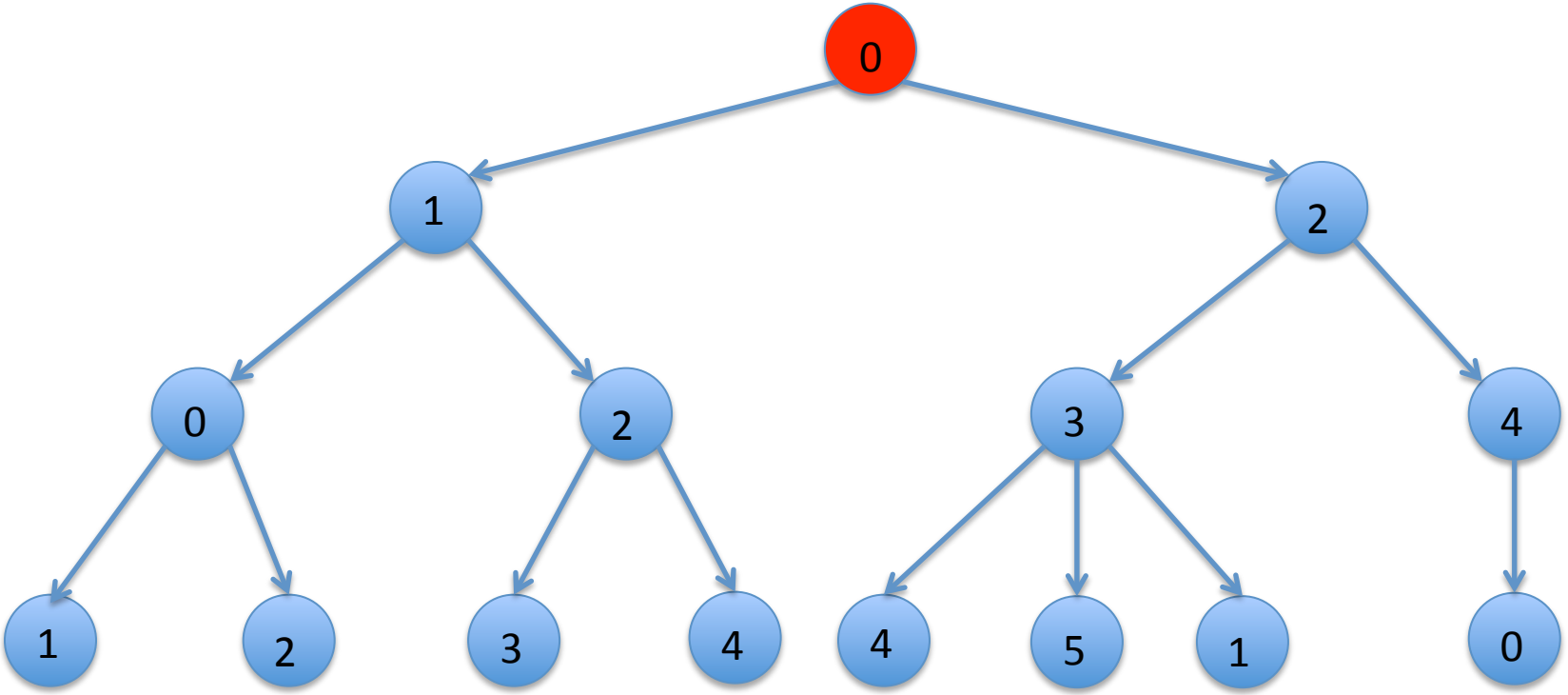
Breadth first search

- Start at “root” node
  - Set of possible paths is just root node
- If not at “goal” node, then
  - Extend current path by adding each “child” of current node to path, unless child already in path
  - Add these new paths to potential set of paths, but put at end of set (this uses a data structure called a **queue**)
  - Select next path and recursively repeat
  - If current node has no “children”, then just go to next option
- Stop when reach “goal” node, or when no more paths to explore

Sidebar: a queue

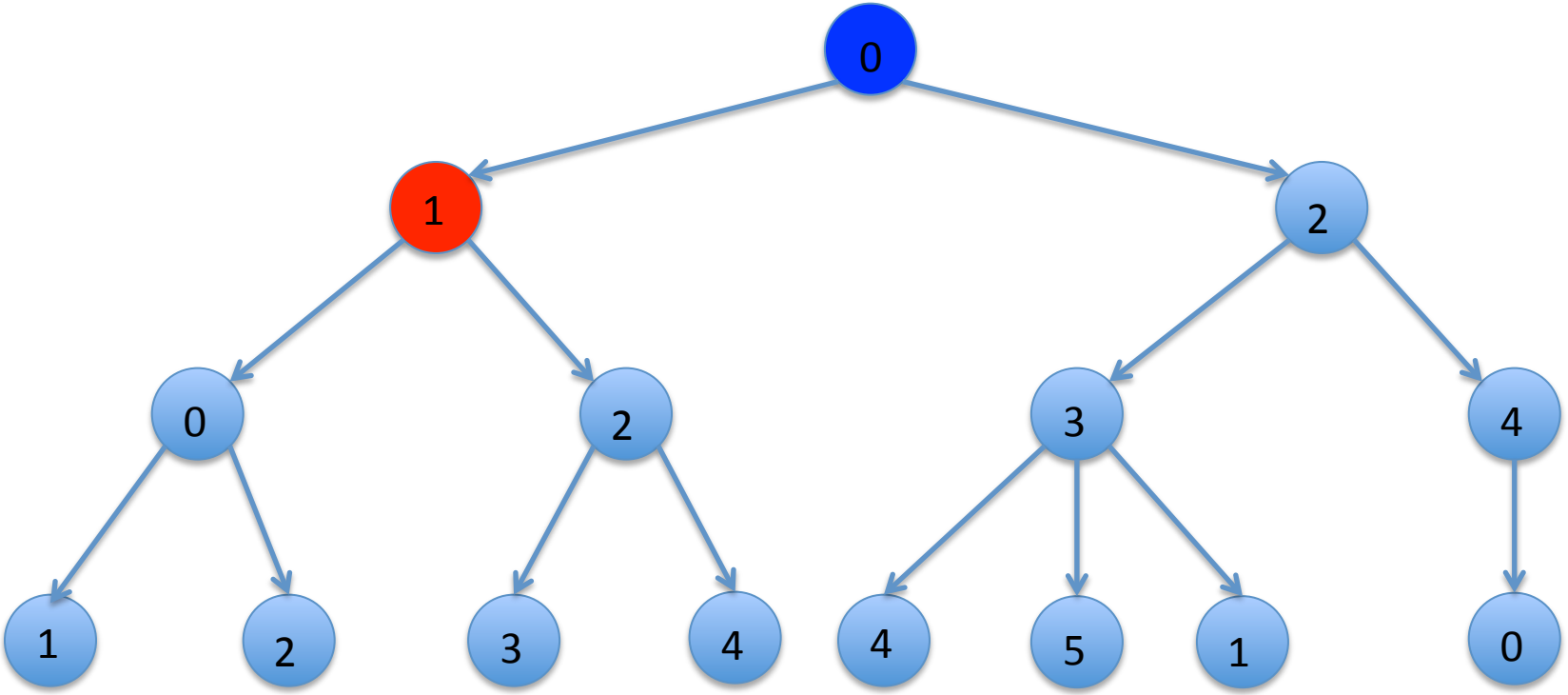
- A Queue is a data structure with a “first in, first out” behavior
  - We push items at the end of the queue
  - We pop items from the front of the queue
  - This maintains a set of items, where we explore each item in the order in which we create it

Breadth first search



0

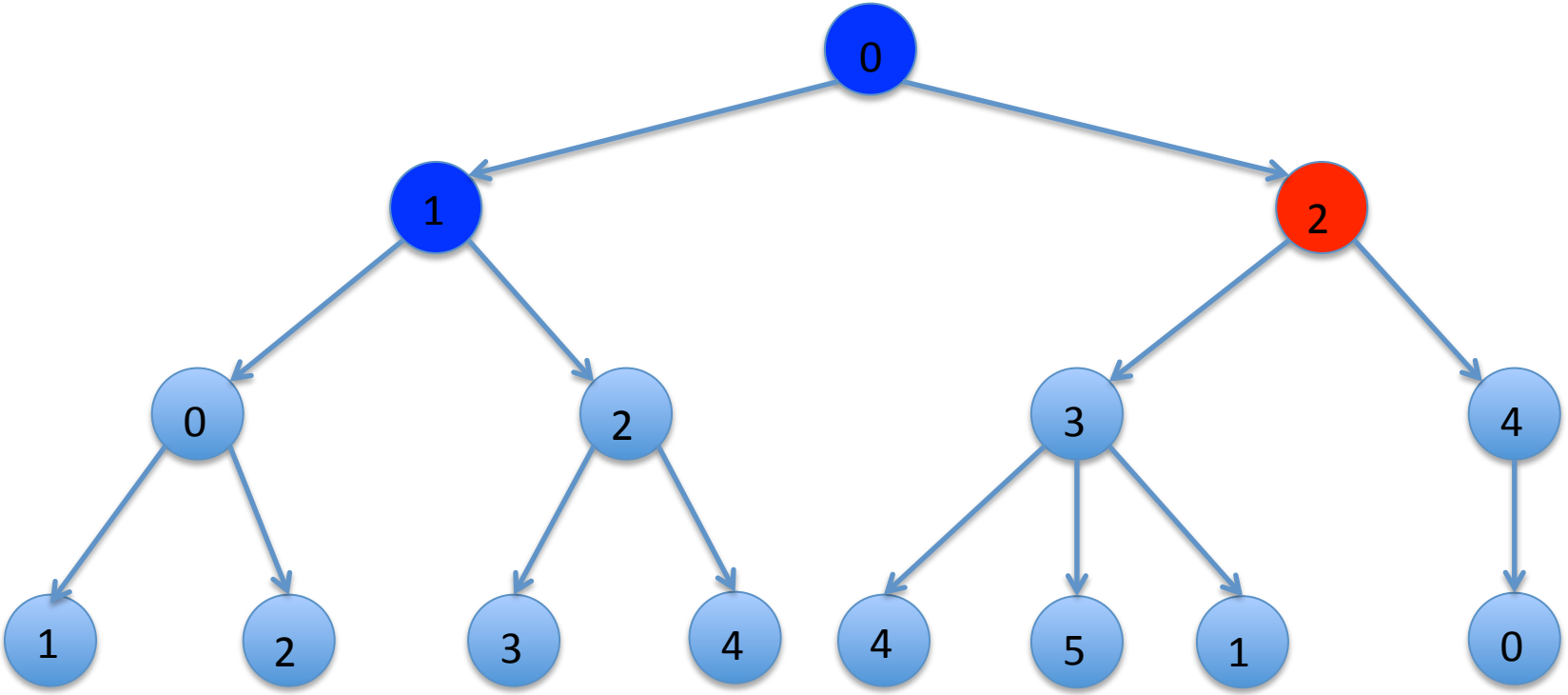
Breadth first search



0

01

Breadth first search

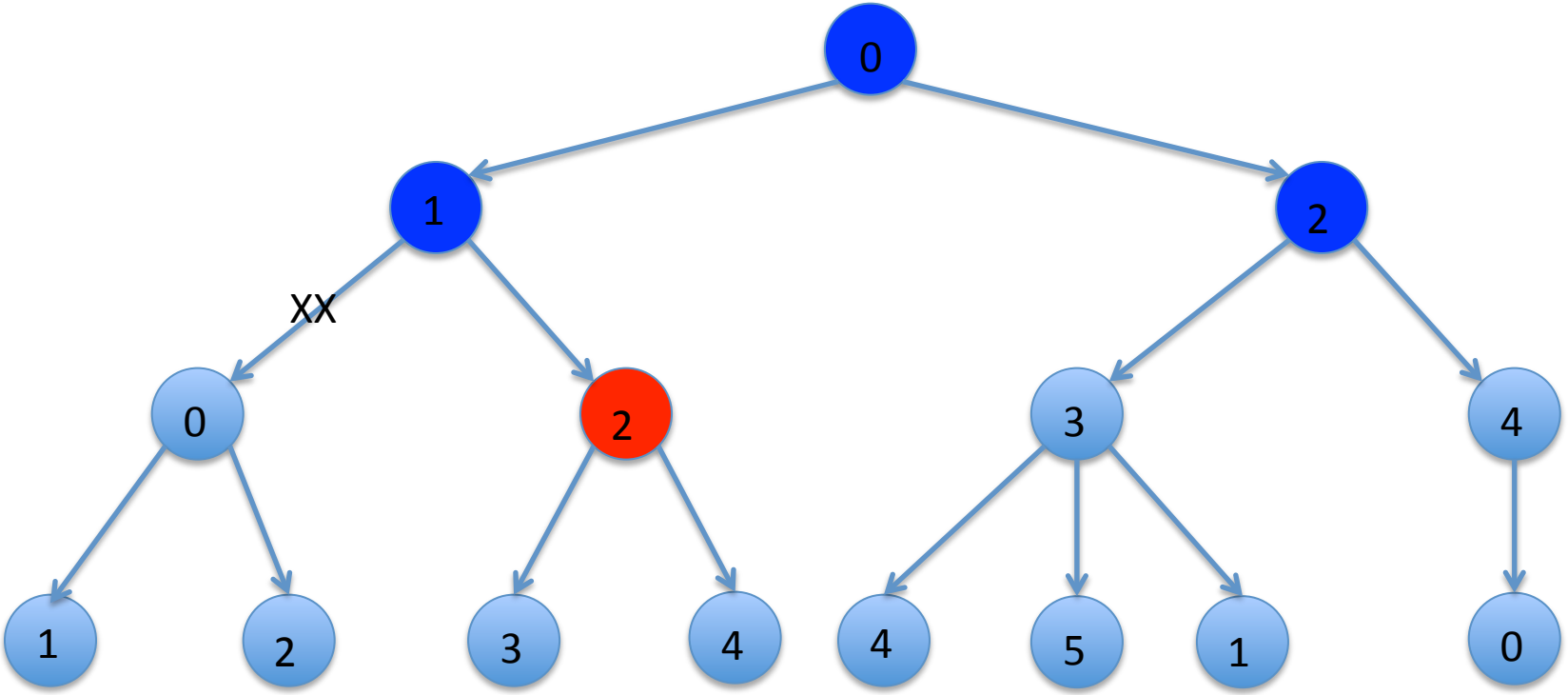


0

01

02

Breadth first search



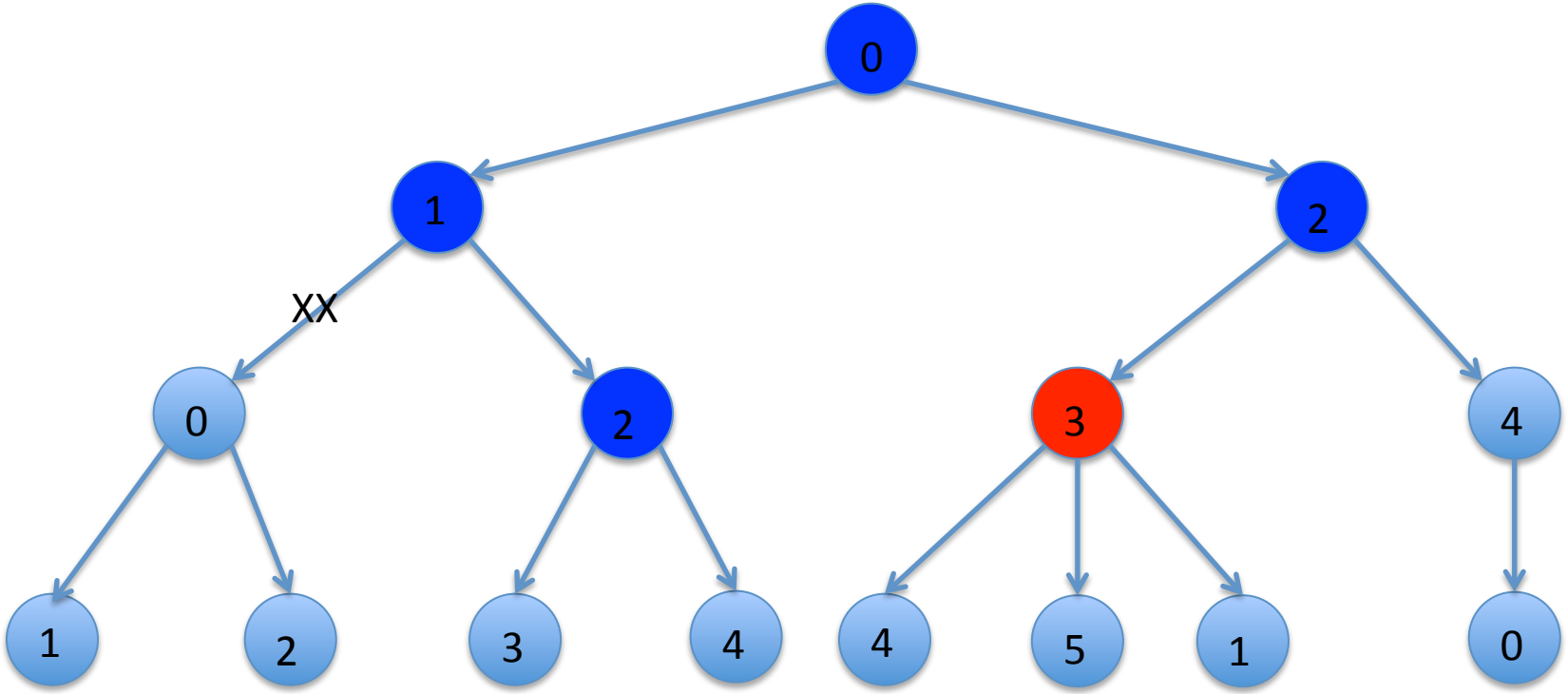
0

01

02

012

Breadth first search



0

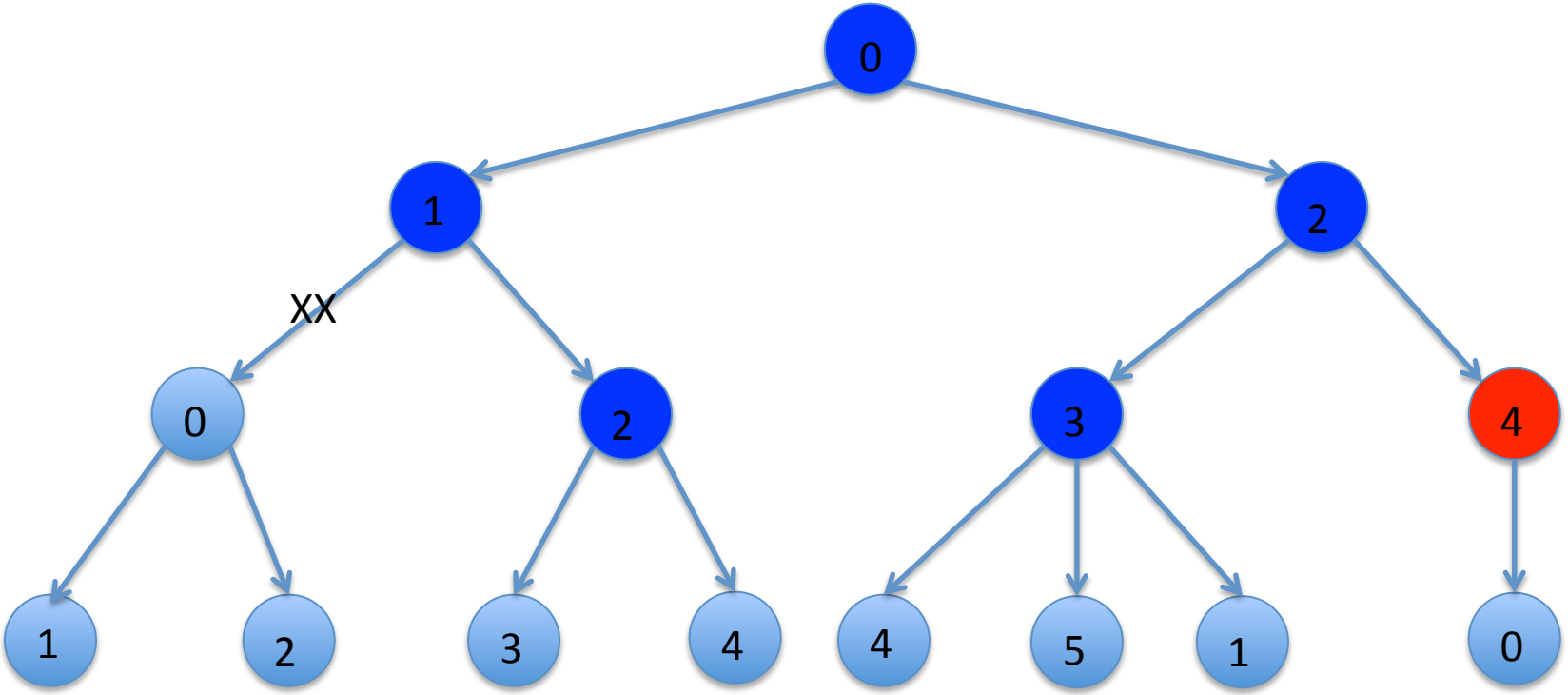
01

02

012

023

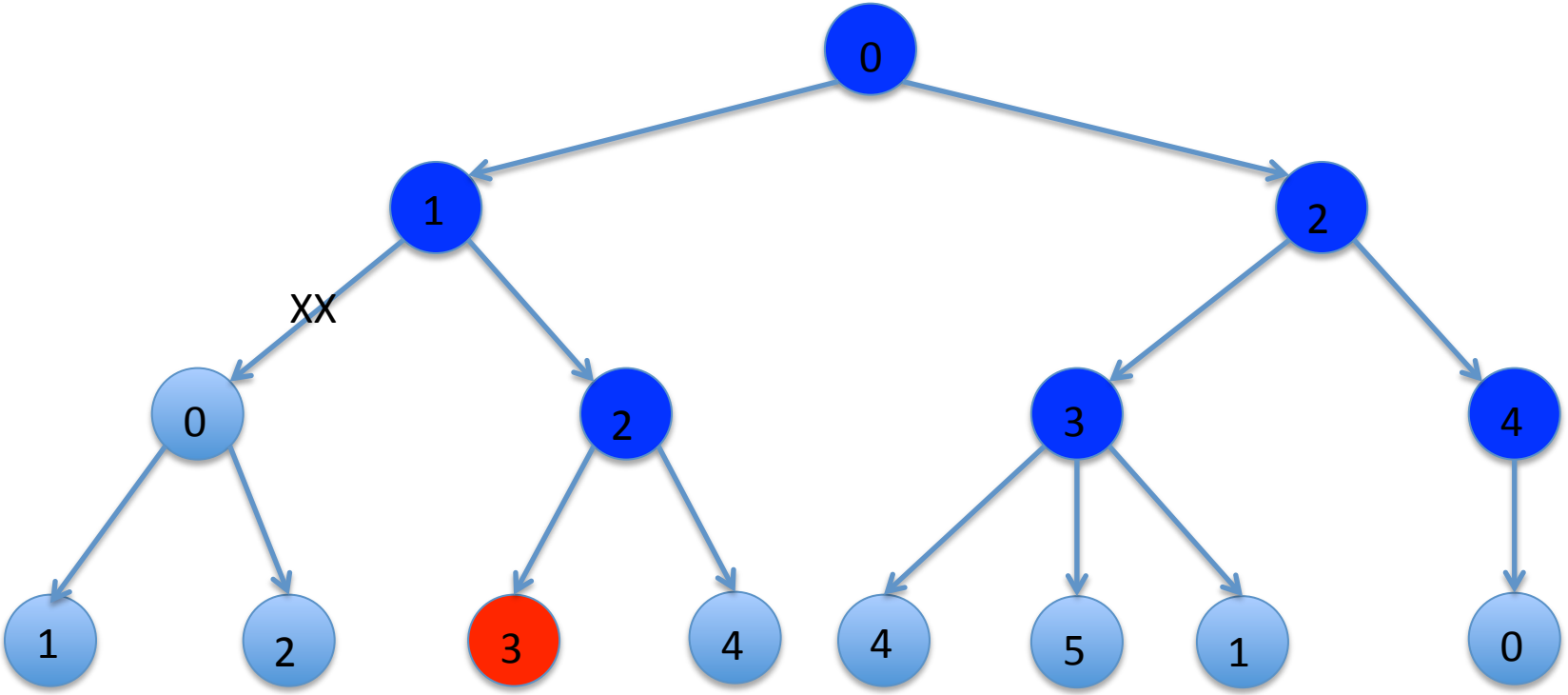
Breadth first search



0
01
02
012

023
024

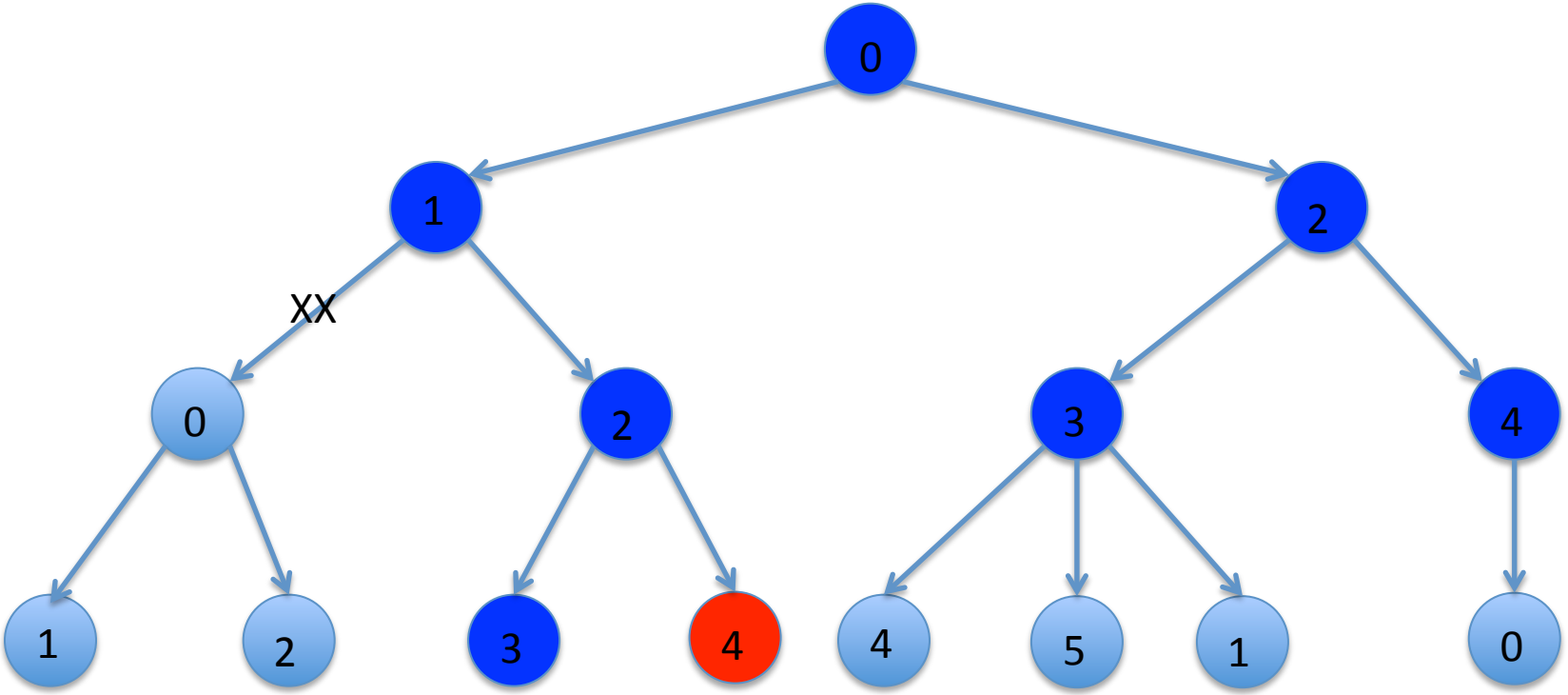
Breadth first search



0
01
02
012

023
024
0123

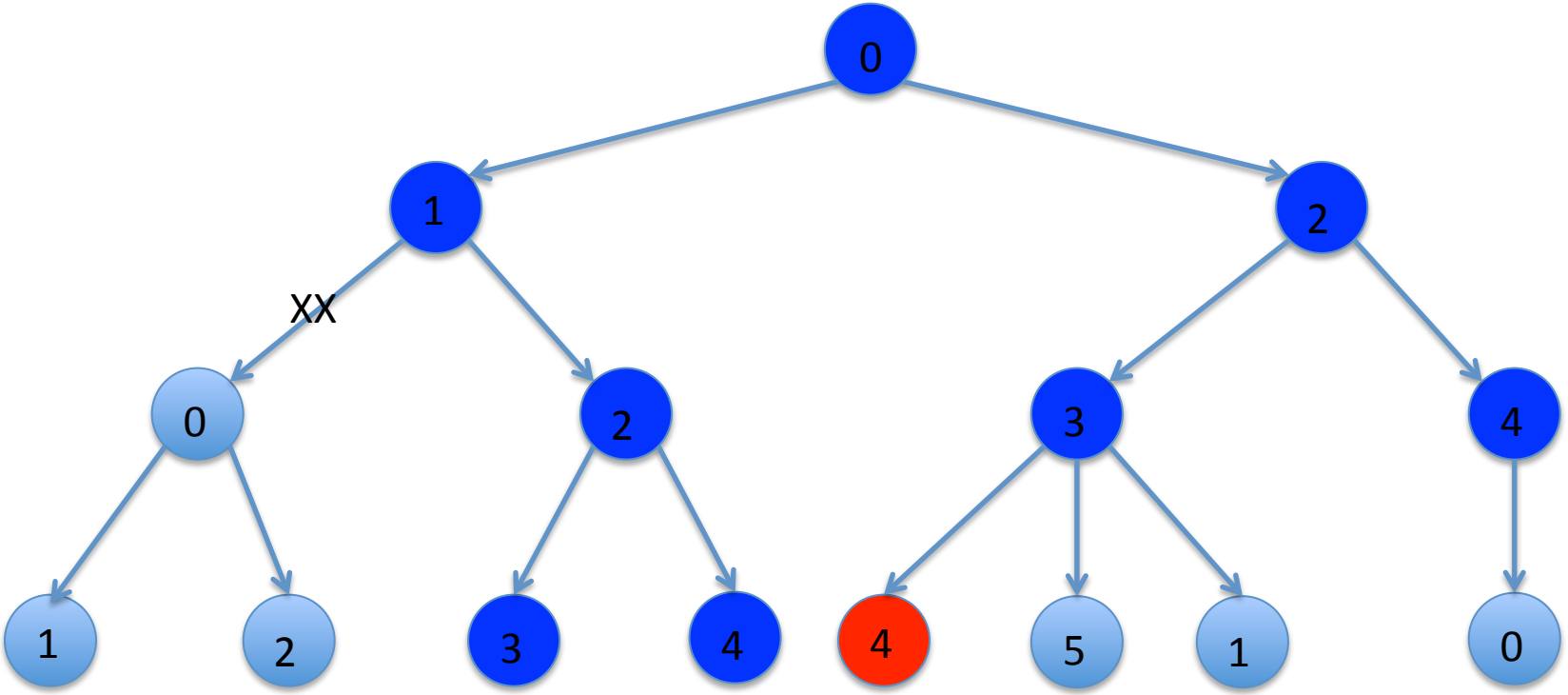
Breadth first search



0
01
02
012

023
024
0123
0124

Breadth first search

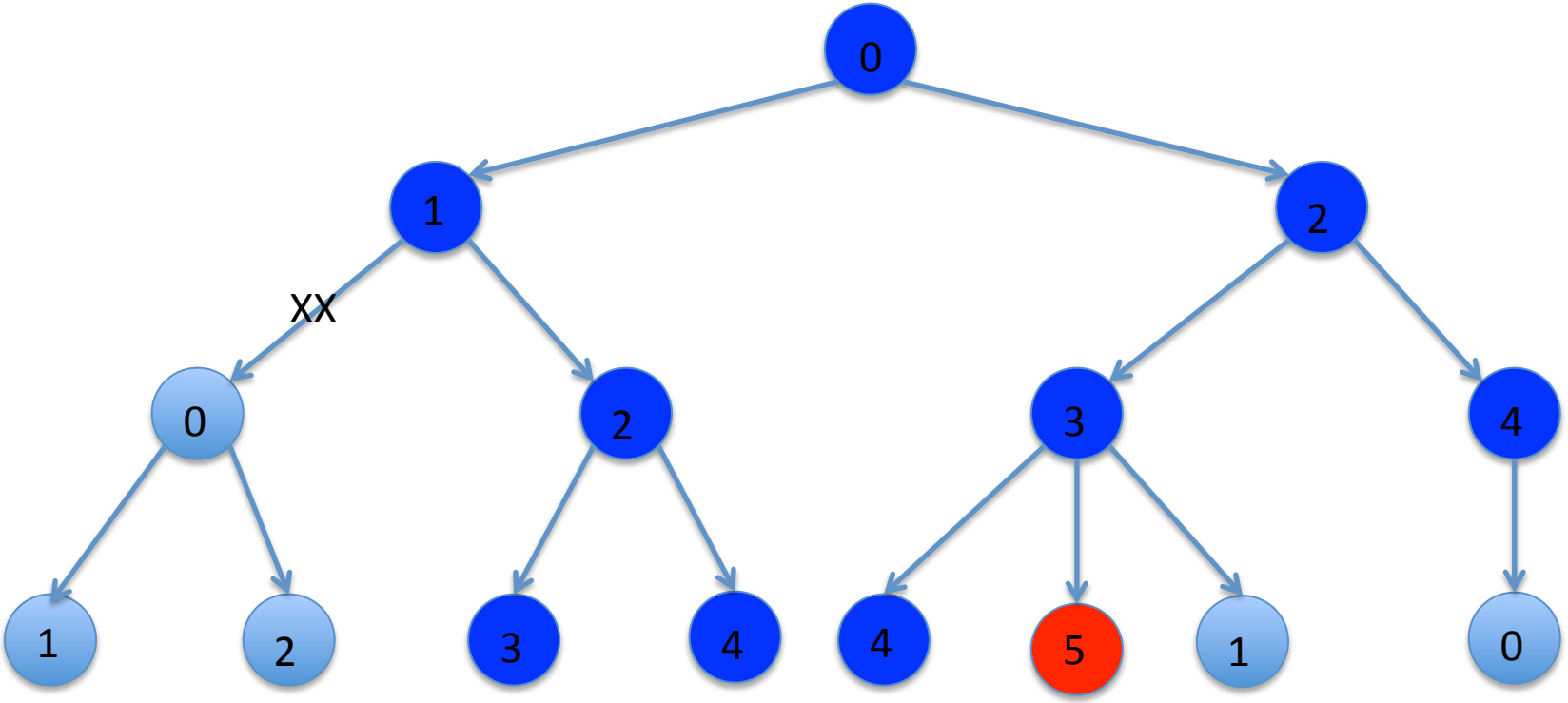


0
01
02
012

023
024
0123
0124

0234

Breadth first search



0
01
02
012

023
024
0123
0124

0234
0235

Sidebar: a queue

- An example from our search
  - 0
  - 01 02 – pop 0, insert 01, 02 at back of queue
  - 02 012 – pop 01, insert 012 at back of queue
  - 012 023 024 – pop 02, insert 023, 024
  - 023 024 0123 0124 – pop 012, insert 0123, 0124

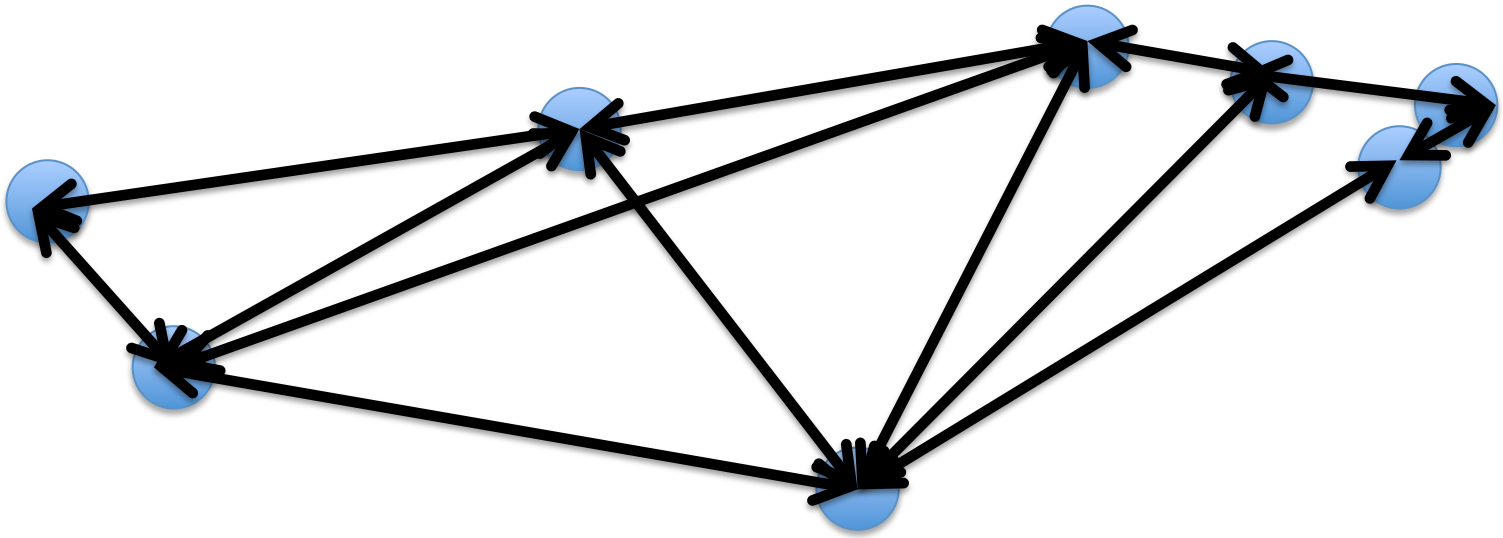
Breadth first search algorithm

```
def BFS(graph, start, end, q = []):
    initPath = [start]
    q.append(initPath)
    while len(q) != 0:
        tmpPath = q.pop(0)
        lastNode = tmpPath[len(tmpPath) - 1]
        print 'Current dequeued path:', printPath(tmpPath)
        if lastNode == end:
            return tmpPath
        for linkNode in graph.childrenOf(lastNode):
            if linkNode not in tmpPath:
                newPath = tmpPath + [linkNode]
                q.append( newPath)
    return None
```

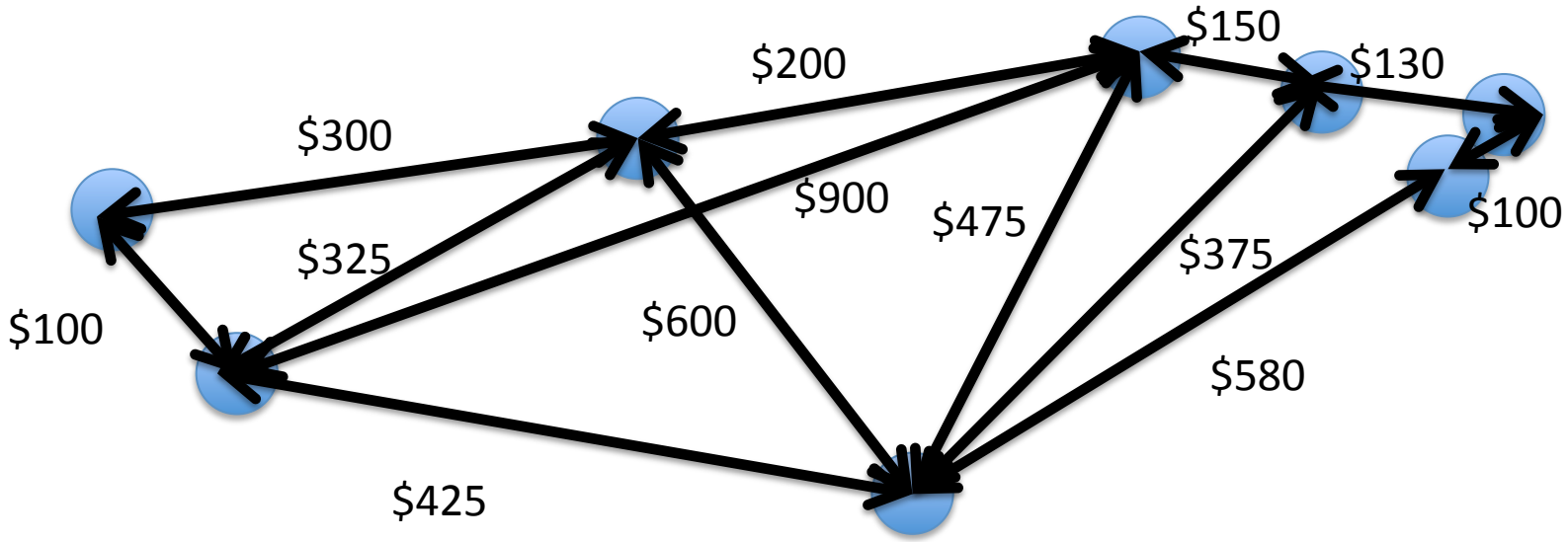
Dealing with weighted graphs

- Our DFS and BFS algorithms find paths with fewest edges
- What if our edges are weighted (e.g. cost of each leg of flight, not just number of legs in flight)?
- Easy to modify DFS, by summing up weights rather than just length of path
- For BFS, first found solution may not be best, and one needs more sophisticated methods

Graph abstraction



Weighted graph abstraction



Why graph optimization?

- Many problems easily expressed as a set of transitions between states of a system
  - Such problems naturally approached as a graph search
  - Example – travel through a physical network, like a road system
  - Example – planning the actions of an autonomous agent, like a robot
  - Example – finding a sequence of actions that convert the state of a physical system to a desired goal

Why graph optimization?

- Depth first and breadth first search find a sequence of transitions that transform a system to a desired goal state
- These methods can find optimal solutions to wide range of problems described as set of objects with defined relationships or set of states of a system with defined transition actions