On the Science Behind Computing (Part I)

Samir Khuller *

1 Graphs and their Representation

We had several discussions about graphs and even informally discussed some simple algorithms that processed graphs to achieve certain objectives. However we only drew pictures of graphs – computer algorithms however need other ways of representing graphs, so that they can be created and manipulated efficiently. The precise representation used will depend on the exact objectives – representations often have different properties and some would enable certain graph operations efficiently, while others would enable other operations efficiently. The two main graph representations we will talk about are the *Adjacency matrix* representation and the *Adjacency list* representation.

An adjacency matrix is essentially a two-dimensional array. Earlier we learnt how to create a one dimensional array in Ruby – two dimensional arrays can be created as well. To create a two dimensional array, we first create a one dimensional array, and define each array entry as a new one-dimensional array.

The following piece of code defines a two dimensional array `a`.

```ruby
def mda(width,height)
a = Array.new(width)
a.length.times{|j| a[j]= Array.new(height) }
return a
end
```

We can refer to elements in the array by writing `a[3][4]`. We are now ready to define the adjacency matrix. We set `a[i][j]` to 1 if there is an edge in the graph connecting nodes `i` and `j` and 0 otherwise. Just by looking up the entry `a[i][j]` we can know if there is an edge in the graph connecting nodes `i` and `j`.

The only problem with this representation is that it uses a lot of storage space – for example for a graph with 10 million nodes (10^7), we need storage space of 10^14 since there is one entry for every pair of nodes. One Kilobit (Kb) is 10^3 bits. One Megabit (Mb) is 10^6 bits, and one Gigabit (Gb) is 10^9 bits.

While this representation makes it easy to look up whether or not a pair of nodes `(i, j)` are adjacent or not, the main problem is that the space used is very large. In addition, most of the entries in the array `a[i]` are actually 0, so we cannot easily find out the neighbors of node `i` – we have to scan the entire array `a[i]` to find the neighbors of `i`. This is unacceptable, since node `i` may only have a small number of neighbors compared to the number of nodes in the graph.

---

*Department of Computer Science, University of Maryland, College Park, MD 20742. E-mail: samir@cs.umd.edu.*
A better representation is to store in a[i] all the neighbors of node i. Since this is an array, it is easy to store a large set of neighboring nodes (upto say our maximum degree $\Delta$).

An array is associated with each node in the graph, and contain all the neighbors of the node. An array is a one dimensional structure.

We can create a graph with $N$ nodes by defining a collection of $N$ arrays $G[i]$.

N=4
G=Array.new
# want G.length to be the number of nodes in the graph
#G[i] is the array corresponding to the neighbors of node i.
N.times{|i| G[i]=Array.new}

How do we write into a specific box?
$G[0][2]$ will index into the 3rd box of $G[0]$. So we can set $G[0][2]=3$ to indicate that node 0 has neighbor node 3.

Suppose we wish to represent the following graph (see Fig. 1) in the adjacency list form.

The following code in ruby will construct the appropriate adjacency list representation. Note that node 0 has two neighbors – namely nodes 1 and 2 and we append to $G[0]$ nodes 1 and 2 etc.

N=7
G=Array.new
N.times{|j| G[j]=Array.new  }
G[0]<<1<<2
G[1]<<0<<2<<6
G[2]<<0<<1<<3<<4
G[3]<<2<<4
G[4]<<2<<3<<5
G[5]<<4
G[6]<<1

Figure 1: Figure showing a small graph.
Now that we have constructed this representation we can do a collection of interesting things.

**List the neighbors of a node**
We now describe a function that will list all the neighbors of a given node. This function is extremely simple.

```ruby
def listnbrs(i)
  G[i].length.times{|j| print G[i][j]}
  print "\n"
end
```

**Count the total number of edges in a graph**
The simple idea is that if we add up the degrees of all the nodes, then we get a quantity that is exactly twice the total number of edges.

```ruby
def notadj(a,b)
  #returns true if a and b are not adjacent
  #search for b in a’s adjacency list
  G[a].length.times{|i|
    if (G[a][i] == b)
      return false
    end
  }
  return true
end
```

```ruby
def notadj(a,b)
  #returns true if a and b are not adjacent
  #search for b in a’s adjacency list
  G[a].length.times{|i|
    if (G[a][i] == b)
      return false
    end
  }
  return true
end
```

```ruby
def listnbrs(i)
  G[i].length.times{|j| print G[i][j]}
  print "\n"
end
```

```ruby
def notadj(a,b)
  #returns true if a and b are not adjacent
  #search for b in a’s adjacency list
  G[a].length.times{|i|
    if (G[a][i] == b)
      return false
    end
  }
  return true
end
```

```ruby
def notadj(a,b)
  #returns true if a and b are not adjacent
  #search for b in a’s adjacency list
  G[a].length.times{|i|
    if (G[a][i] == b)
      return false
    end
  }
  return true
end
```

```ruby
def notadj(a,b)
  #returns true if a and b are not adjacent
  #search for b in a’s adjacency list
  G[a].length.times{|i|
    if (G[a][i] == b)
      return false
    end
  }
  return true
end
```

```ruby
def notadj(a,b)
  #returns true if a and b are not adjacent
  #search for b in a’s adjacency list
  G[a].length.times{|i|
    if (G[a][i] == b)
      return false
    end
  }
  return true
end
```

```ruby
def notadj(a,b)
  #returns true if a and b are not adjacent
  #search for b in a’s adjacency list
  G[a].length.times{|i|
    if (G[a][i] == b)
      return false
    end
  }
  return true
end
```

```ruby
def notadj(a,b)
  #returns true if a and b are not adjacent
  #search for b in a’s adjacency list
  G[a].length.times{|i|
    if (G[a][i] == b)
      return false
    end
  }
  return true
end
```

```ruby
def notadj(a,b)
  #returns true if a and b are not adjacent
  #search for b in a’s adjacency list
  G[a].length.times{|i|
    if (G[a][i] == b)
      return false
    end
  }
  return true
end
```

```ruby
def notadj(a,b)
  #returns true if a and b are not adjacent
  #search for b in a’s adjacency list
  G[a].length.times{|i|
    if (G[a][i] == b)
      return false
    end
  }
  return true
end
```

```ruby
def notadj(a,b)
  #returns true if a and b are not adjacent
  #search for b in a’s adjacency list
  G[a].length.times{|i|
    if (G[a][i] == b)
      return false
    end
  }
  return true
end
```

```ruby
def notadj(a,b)
  #returns true if a and b are not adjacent
  #search for b in a’s adjacency list
  G[a].length.times{|i|
    if (G[a][i] == b)
      return false
    end
  }
  return true
end
```

```ruby
def notadj(a,b)
  #returns true if a and b are not adjacent
  #search for b in a’s adjacency list
  G[a].length.times{|i|
    if (G[a][i] == b)
      return false
    end
  }
  return true
end
```

```ruby
def notadj(a,b)
  #returns true if a and b are not adjacent
  #search for b in a’s adjacency list
  G[a].length.times{|i|
    if (G[a][i] == b)
      return false
    end
  }
  return true
end
```

```ruby
def notadj(a,b)
  #returns true if a and b are not adjacent
  #search for b in a’s adjacency list
  G[a].length.times{|i|
    if (G[a][i] == b)
      return false
    end
  }
  return true
end
```

```ruby
def notadj(a,b)
  #returns true if a and b are not adjacent
  #search for b in a’s adjacency list
  G[a].length.times{|i|
    if (G[a][i] == b)
      return false
    end
  }
  return true
end
```

```ruby
def notadj(a,b)
  #returns true if a and b are not adjacent
  #search for b in a’s adjacency list
  G[a].length.times{|i|
    if (G[a][i] == b)
      return false
    end
  }
  return true
end
```

```ruby
def notadj(a,b)
  #returns true if a and b are not adjacent
  #search for b in a’s adjacency list
  G[a].length.times{|i|
    if (G[a][i] == b)
      return false
    end
  }
  return true
end
```

```ruby
def notadj(a,b)
  #returns true if a and b are not adjacent
  #search for b in a’s adjacency list
  G[a].length.times{|i|
    if (G[a][i] == b)
      return false
    end
  }
  return true
end
```

```ruby
def notadj(a,b)
  #returns true if a and b are not adjacent
  #search for b in a’s adjacency list
  G[a].length.times{|i|
    if (G[a][i] == b)
      return false
    end
  }
  return true
end
```

```ruby
def notadj(a,b)
  #returns true if a and b are not adjacent
  #search for b in a’s adjacency list
  G[a].length.times{|i|
    if (G[a][i] == b)
      return false
    end
  }
  return true
end
```

```ruby
def notadj(a,b)
  #returns true if a and b are not adjacent
  #search for b in a’s adjacency list
  G[a].length.times{|i|
    if (G[a][i] == b)
      return false
    end
  }
  return true
end
```

```ruby
def notadj(a,b)
  #returns true if a and b are not adjacent
  #search for b in a’s adjacency list
  G[a].length.times{|i|
    if (G[a][i] == b)
      return false
    end
  }
  return true
end
```

```ruby
def notadj(a,b)
  #returns true if a and b are not adjacent
  #search for b in a’s adjacency list
  G[a].length.times{|i|
    if (G[a][i] == b)
      return false
    end
  }
  return true
end
```

```ruby
def notadj(a,b)
  #returns true if a and b are not adjacent
  #search for b in a’s adjacency list
  G[a].length.times{|i|
    if (G[a][i] == b)
      return false
    end
  }
  return true
end
```

```ruby
def notadj(a,b)
  #returns true if a and b are not adjacent
  #search for b in a’s adjacency list
  G[a].length.times{|i|
    if (G[a][i] == b)
      return false
    end
  }
  return true
end
```

```ruby
def notadj(a,b)
  #returns true if a and b are not adjacent
  #search for b in a’s adjacency list
  G[a].length.times{|i|
    if (G[a][i] == b)
      return false
    end
  }
  return true
end
```

```ruby
def notadj(a,b)
  #returns true if a and b are not adjacent
  #search for b in a’s adjacency list
  G[a].length.times{|i|
    if (G[a][i] == b)
      return false
    end
  }
  return true
end
```

```ruby
def notadj(a,b)
  #returns true if a and b are not adjacent
  #search for b in a’s adjacency list
  G[a].length.times{|i|
    if (G[a][i] == b)
      return false
    end
  }
  return true
end
```
def recpairs(j):
    # generate all recommendations for node j
    # these are nodes at distance exactly 2
    # x is defined as a neighbor of j in each iteration
    G[j].length.times{|i|
        x=G[j][i]
        # now look at x’s adj list to see which nodes are to be chosen
        # if the node is j, we skip it and if its adj to j we skip it
        # ok to print duplicates
        G[x].length.times{|f|
            if ((not(G[x][f] == j)) and notadj(G[x][f],j))
                print("(#{j},#{G[x][f]})")
            end
        }
    }
end

N.times{|p|
    recpairs(p)
    print("\n")
}