Date
February 3, 2021

classmate
Date
Page

WEEK 8

Graphs - Trains

TRAINS

- Train dataset - information about trains and stations
  → Each train is a route list of stations
  → Each station is a route list of trains passing through
- Compute pairs of stations connected by a direct train
- Represent start and end station of trains in a nested dictionary.
  → trains[t][start] , trains[t][end]

DIRECT ROUTES

- Stations A & B such that a train starts at A and ends at B.
- First, compile the list of stations from 'trains'.
- Create a matrix to record direct routes
- Map station ~~map~~ names to row and column indices.
- Populate the matrix

CODE →

```
Procedure DirectRoutes (trains)
    stations = { }
    foreach t in keys (trains) {
        stations [trains [t] [start]] = True
        stations [trains [t] [end]]   = True
    }
    n = length (keys (stations))
    direct = CreateMatrix (n, n)
    strindex = { }
    i = 0
    foreach s in keys (stations) {
        strindex [s] = i
        i = i + 1
    }
    foreach t in keys (trains) {
        i = strindex [trains [t] [start]]
        j = strindex [train [t] [end]]
        direct [i] [j] = 1
    }
    return (direct)
End DirectRoutes
```

ONE HOP ROUTES

- Stations A and B such that you can reach B from A by changing one train
- Iterate through intermediate stations
  → set onehop [i] [j] = 1, if there is a connection via intermediate station k.
- More useful to let onehop [i] [j] mean "connected with at most one hop"
  → intialise onehop to include direct routes

CODE →

```
Procedure WithinOneHop (direct)
    n = length (keys (direct))
    onehop = CreateMatrix (n, n)
    foreach i in rows (direct) {
        foreach j in columns (direct) {
            onehop [i] [j] = direct [i] [j]
            foreach k in columns (direct) {
                if (direct [i] [k] == 1 and direct [k] [j] == 1) {
                    onehop [i] [j] = 1
                }
            }
        }
    }
    return (onehop)
End WithinOneHop
```

TWO HOP ROUTES

- Stations A and B such that you can reach B from A by changing at most two trains
- Extend a one hop route from i to k by a direct train from k to j
- Iterate through intermediate stations
  → Combine information in direct and onehop

→ Check onehop [i][k] and direct [k][j]

CODE →

```
Procedure WithinTwoHops ( direct , onehop)
    n = length ( keys(direct))
    twohops = CreateMatrix (n,n)
    foreach i in rows (direct) {
        foreach j in columns (direct) {
            twohops [i][j] = onehop [i][j]
            foreach k in columns (direct) {
                if (onehop[i][k] == 1 and direct[k][j] == 1){
                    twohops [i][j] = 1
                }
            }
        }
    }
    return (twohops)
End WithinTwoHops
```

n HOP ROUTES

- Let nhops record connections from station A to station B by changing at most n trains
- Want to extend nhops to allow one more hop
- Iterate through intermediate stations
- Combine information direct and nhops
  → Extend an nhop route from i to k by a direct train from k to j

→ Check nhops [i][k] and direct [k][j]

CODE →

```
Procedure OneMoreHop ( direct , nhops)
    n = length( keys ( direct))
    onemorehop = CreateMatrix (n, n)
    foreach i in rows(direct) {
        foreach j in columns (direct) {
            onemorehop [i][j] = nhops [i][j]
            foreach k in columns (direct) {
                if (nhops [i][k] == 1 and direct [k][j] == 1) {
                    onemorehop [i][j] = 1
                }
            }
        }
    }
    return (onemorehop)
End OneMoreHop
```

i . . . . j

DISCOVERING PATHS

- Path : sequence of edges from A to B
  → A direct edge is a path of length 1
- Procedure OneMoreHop extends paths of length n to paths of length n+1
- N nodes – shortest path from A to B has at most N-1 edges
  → a longer path would visit a node twice
  – unnecessary loop

- Transitive Closure - pairs of nodes connected by a path
  → Repeat OneMoreHop N-1 times starting with direct

SUMMARY

- Graphs are useful to represent connectivity in a network
- A path is a sequence of edges
- Starting with direct edges, we can iteratively find longer and longer paths
- Compute the transitive closure of edge relation
  → stop with paths of length N-1 for an N node graph

Date
February 4, 2021

Edge Labelled Graphs

ADDING INFO. TO DIRECT ROUTE GRA.

- Stations A and B such that a train starts at A & ends at B
- Create a matrix to record direct routes
  → compile the list of stations from trains
  → Map stations to row, column indices
  → Populate the matrix
- Keep track of trains connecting stations
  → Each entry in the matrix is now a dictionary
  → Initially empty dictionary - no direct connection
  → Add a key for each train connecting a pair of stations
- Information about trains is recorded as a label on the edge
  → Edge labelled graph

CODE →

```
Procedure LabelledDirectRoutes (trains)
:
foreach r in rows (direct) {
    foreach c in columns (direct) {
        direct [i][j] = { }
    }
}
foreach t in keys (trains) {
```

```
    i = stnindex[trains[t][start]]
    j = stnindex[trains[t][end]]
    direct[i][j][t] = True
  }
  return(direct)
End LabelledDirectRoutes
```

DISTANCES B/W STATIONS

- For each direct train record the distance it travels
  → Add an extra key, trains[t][distance]
- Compute the shortest distance by direct trains
  → Trains may take different routes b/w same pair of stations
- Graph directdist
  → Edge label : directdist[i][j] is the shortest distance b/w i & j.
- Compute the shortest distance by direct trains
- Edge labelled graph directdist
- When we discover a direct route for the first time, set the distance
- If we find a new direct route between an already connected pair, update the distance to the minimum

CODE →

```
Procedure DirectDistance(trains)
  ⋮
  directdist = CreateMatrix(n,n)
  ⋮
  foreach t in keys(trains){
    i = stn2idx[trains[t][start]]
    j = stn2idx[trains[t][end]]
    if(directdist[i][j] == 0){
      directdist[i][j] = trains[t][distance]
    }
    else{
      directdist[i][j] = min(directdist[i][j], trains[t][distance])
    }
  }
  return(directdist)
End DirectDistance
```

ONE HOP DISTANCE

- We start with the matrix of direct distances
- Initialise one hop distance to direct distance
- Each time we discover a one hop route, update the one hop distance.
- Iterate this to find length of the shortest path between each pair of stations
  → modify the transitive closure calculation to record minimum distance path.

CODE →

```
Procedure OneHopDistance (directdist)
  n = length (keys (directdist))
  onehopdist = CreateMatrix (n,n)
  foreach i in rows (directdist) {
    foreach j in columns (directdist) {
      onehopdist [i][j] = directdist [i][j]
      foreach k in columns (directdist) {
        if (directdist [i][k] > 0 and directdist [k][j] > 0) {
          newdist = directdist [i][k] + directdist [k][j]
          if (onehopdist [i][j] > 0) {
            onehopdist [i][j] = min (newdist, onehopdist [i][j])
          }
          else {
            onehopdist [i][j] = newdist
          }
        }
      }
    }
  }
  return (onehopdist)
End OneHopDistance
```

SUMMARY

- We can represent extra info. in a graph via edge labels
  - Train name, distance
  - Foreach edge in the graph, replace 1 in matrix by edge label
- Iteratively update labels
  - Compute shortest distance path b/w each pair of stations

Date February 6, 2021

WEEK 9

AQ 9.2 → Q4

Recursion

INDUCTIVE DEFINITIONS

- Many computations are naturally defined inductively
  - → Base case: directly return the value
  - → Inductive step: compute value in terms of smaller arguments
- Factorial
  - → $n! = n \times (n-1) \times \ldots \times 2 \times 1$
  - → $0!$ is defined to be 1
  - → factorial (0) = 1
  - → For $n > 0$, **factorial (n) = n * factorial (n-1)**

CODE →

```
Procedure Factorial (n)
  if (n == 0) {
    return (1)
  }
  else {
    return (n* factorial (n-1))
  }
End Factorial
```

- Recursive Procedure
  - → factorial (n) is suspended till factorial (n-1) returns a value.

INDUCTIVE DEFINITIONS ON LISTS

- Inductive functions on lists
  → Base Case : Empty list
  → Inductive step: Compute value in terms first element and rest

- Sum of numbers in a list
  → If l == [], sum is 0

  → Otherwise, add first(l) to sum of rest(l)

CODE:
```
Procedure ListSum (l)
  if ( l == [] ) {
    return (0)
  }
  else {
    return ( first(l) + listsum(rest(l)) )
  }
End ListSum
```

→ Can also add last(l) to sum of init(l)

CODE :
```
Procedure ListSum2 (l)
  if ( l == [] ) {
    return (0)
  }
  else {
    return ( last(l) + listSum2(init(l)))
  }
End ListSum2
```

INSERTION SORT

- Build up a sorted prefix

- Extend the sorted prefix by inserting the next element in the correct position.

CODE:
```
Procedure InsertionSort (l)
  sortedList = []
  foreach z in l {
    sorted_list = sortedListInsert ( sortedList , z)
  }
  return ( sortedList)
End InsertionSort

Procedure SortedListInsert ( l , z)
  newlist = [ ]
  inserted = false
  foreach z in l {
    if ( not (inserted)) {
      if (x < z) {
        newlist = newlist ++ [x]
        inserted = True
      }
    }
  newlist = newlist ++ [z]
  }
  if ( not (inserted)) {
      newlist = newlist ++ [x]
    }
  return (newlist)
End SortedInsertList
```

INSERTION SORT, INDUCTIVELY

- List of length 1 or less is sorted
- For longer lists, insert first(l) into sorted rest(l)

CODE :

```
Procedure InsertionSort (l)
  if ( length(l) <= 1) {
    return (l)
  }
  else {
    return (SortedListInsert (InsertionSort (rest(l)), first(l)))
  }
End InsertionSort

Procedure SortedListInsert ( l , x)
  newlist = [ ]
  inserted = False
  foreach z in l {
    if ( not(inserted)) {
      if (x<z) {
        newlist = newlist ++ [x]
        inserted = True
      }
    }
    newlist = newlist ++ [z]
  }
  if (not(inserted)) {
    newlist = newlist ++ [x]
  }
  return (newlist)
End SortedListInsert
```

SUMMARY

- Many functions are naturally defined in an inductive manner
  - → Base case and inductive step
  - → For numeric functions, base case is typically 0 or 1
  - → For lists, base case is typically length 0 or length 1
- Use recursive procedures to compute such functions
  - → Base case : value is explicitly calculated & returned
  - → Inductive case : value requires procedure to evaluated on a smaller input
  - → Suspend the current computation till the recursive computation terminates
- WARNING : Without properly defined base cases, recursive procedures will not terminate.

A mutual recursion is a form of recursion in which two objects are defined in terms of each other.

Recursion is a process in which function calls itself until termination condition is not true.

It requires a base case and an inductive step to complete the process.

It makes tree traversal easier.

Depth-First Search

REACHABILITY IN GRAPHS

- What are the vertices reachable from node i ?
- Start from i , visit a neighbour j
- Suspend the exploration of i and explore j instead
- Continue till you reach a vertex with no unexplored neighbours
- Backtrack to nearest suspended vertex that still has an unexplored neighbour
- Best defined recursively
  - → Maintain information about visited nodes in a dictionary visited
  - → Recursively update visited each time we explore an unvisited neighbour

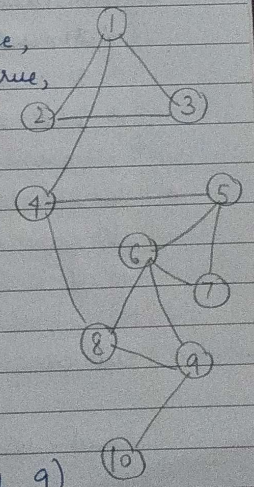
DEPTH FIRST SEARCH

- Maintain info about visited notes in a dictionary visited
- Recursively update visited each time we explore an unvisited neighbour
- To explore vertices reachable from i
  - → Initialise visited = { }
  - → visited = DFS (graph, visited , i)
  - → keys (visited) is set of nodes that can be reached from i
  - → If keys (visited) includes all nodes, the graph is connected.

CODE:

```
Procedure DFS ( graph , visited , i )
  visited [i] = True
  foreach j in columns (graph) {
    if ( graph[i][j] == 1 and not( isKey(visited , j ))) {
        visited = DFS (graph, visited , j)
    }
  }
  return ( visited)
End DFS
```

EXAMPLE: output:
visited = {4: True , 1: True, 2: True,
3: True , 5: True, 6: True , 7: True,
8: True , 9: True, 10: True }



code:
DFS (graph , visited, 4)
DFS (graph , visited, 1)
DFS (graph , visited ,2)
DFS (graph , visited , 3)
DFS (graph , visited , 5)
DFS (graph , visited , 6)
DFS (graph , visited , 7)
DFS (graph , visited , 8) DFS (graph , visited , 9)
DFS (graph, visited , 10)

SUMMARY

- Depth First Search is a systematic procedure to explore a graph
- Recursively visit all unexplored graphs
- Keep track of visited vertices in a dictionary
- Can discover properties of the graph - for instance, is it connected?

DFS : 1) It is a recursive algorith
2) It uses the idea of backtracking
3) It explores all possible searches at each node before backtracking.

* A standard DFS implementation puts each vertex of the graph into one of the two categories : visited & not visited
* A DFS tree don't contain cycles
* In DFS technique if all searches exhaust at one of the node, it goes back to 'just previous node'.

Date February 8, 2021
MONDAY

WEEK 10
Encapsulation

WHAT IS ENCAPSULATION ?

- Procedures that we have seen so far have been unanchored.
  → It seems more natural to attach the procedure to the data elements on which it operates
- Encapsulate the data elements and the procedures into one package
  → which is called an object, hence the popular term object oriented computing / programming
- The procedures encapsulated in the object act as the interfaces through which the external world can interact with the object.
- The object can hide details of the implementation from the external world
  → The changed implementation may involve additional (intermediate) data elements & additional procedures
  → Any changes made to the implementation does not impact the external world, since the procedure interface is not changed
- Allows for separation of concerns - separate the "what ?" from "how ?".

DO WE NEED ANYTHING BEYOND PROCEDURES ?

- Procedures already provide some kind of encapsulation
  - → Interface via the parameters and return value
  - → Hiding of variables used within the procedure
- But we could have the following issues with procedures:
  - → Procedures could have side effects - some of them are desirable
  - → We may need to call a sequence of procedures to achieve something - we expect the object to hold the state between the procedure calls (ATM example)

DATA TYPES & ENCAPSULATION

- Recall datatypes?
  - → Basic datatypes : integer, character, boolean
  - → Subtype of a datatype to restrict the values & operations
  - → Records, Lists, Strings : compound datatypes
  - → Each card (table row) can be represented using a datatype
  - → The collection of cards can also be represented as datatypes
- Operations on a datatype
  - → Well defined operations on the basic datatypes & their subtypes
  - → Some operations on the string datatype
- But what about operations on compound datatypes?
- What if we allowed to attach our own procedures to a datatype?
  - → Allow a datatype to have procedure fields
  - → Just as X.F represents the field F of X, we can use X.F(a,b) to represent a procedure field of X which takes parameters a and b.

EXAMPLE :

(1) Classroom Scores Dataset

- Questions more frequently asked of the scores dataset
  - → What is the avg in a subject - say Maths?
  - → What is the overall avg?
- Suppose that we've to ask these questions many times?
  - → It is wasteful to carry out same computation again & again.
  - → Can we not store the answer of a question, & just return the saved answer when the question is asked again?
- This will work as long as the data is static.

ENCAPSULATION

- We could create an object CT (for class teacher) of datatype ClassAve
  - → ClassAve needs only the list of total marks of the students
- ClassAve can have a field marklist that holds the total marks of all the students in the classroom dataset
- We can now add a procedure average() to ClassAve

to find the average of the list.

- Since we want to store the answer after the first time, we could have another field aValue that will hold the computed value of average. Initially aValue = -1
  - → CT.average() first checks if CT.aValue is -1
  - → If no, it just returns CT.aValue
  - → If it is -1, this means that the average has not been computed yet. So, it computes the average by summing the marks in the list and dividing the sum by the length of the list.

- What about the avg. values of the individual subjects?

- Just like CT, we could have objects PhT, MaT and ChT (for Phy, Maths & Chem. Teacher) that each hold (or have access to) the entire classroom dataset.
  - → But again as we observed with CT, PhT needs to hold only the list of Physics marks of the students, similarly MaT & ChT need only hold the Maths & Chemistry marks lists.

- So let us say that each of these objects are also of the same datatype ClassAvg, but their marks list field holds the list of marks of the respective subject.

- Is this enough? What happens when we call PhT.average()?
  - → PhT.average() checks if PhT.aValue is -1.
  - → If not, it just returns aValue.
  - → Otherwise, it computes the average from the marks in the marks list - which is just the average of the Physics marks!

- So the same datatype classAvg can be used for all the objects, CT, PhT, ChT and MaT.

COMPARE PARAMETERISED PROCEDURE with ENCAPSULATION

- Procedure AvgMarks takes a field as a parameter
  - → We can call AvgMarks with Total or subject name

```
AvgTotal = AvgMarks(Total)
AvgMaths = AvgMarks(Maths)
AvgPhysics = AvgMarks(Physics)
AvgChemistry = AvgMarks(Chem)
```

- Datatype ClassAvg has a procedure average() within it
  - → We call average() for each object of ClassAvg datatype
  - → Advantage: result is stored, next call is much faster
  - → Disadvantage: object needs to be created & initialised

```
AvgTotal = CT.average()
AvgMaths = MaT.average()
AvgPhysics = PhT.average()
AvgChemistry = ChT.average()
```

- Caller is unaware of what is happening inside.

BUT WHAT IF WE THE DATASET IS NOT STATIC?

- If the dataset changes (consider for example that we add a new student to class), then the stored average values will be wrong! How do we deal with this?

- We need to manage the addition of a new student carefully
  → write a procedure addStudent(newMark) in the ClassAve datatype which takes as parameter the marks of the new student (total, or respective subject marks)
  → a new student can be added only via the use of this procedure

- addStudent will need to append newMark at the end of its marksList

- addStudent will also need to reset aValue to -1, so that the average will get computed again.

COMPLETE EXAMPLE

- So what does ClassAve datatype look like at the end of all this?

- Has two data fields and two procedures
  → field marksList contains a list of marks
  → field aValue which is either -1 (not computed) or holds the computed average value.
  → procedure average() returns aValue if it is not -1, else it computes the avg of marksList, stores it in aValue and returns it.
  → procedure addStudent(newMark) appends newMark to the end of marksList and resets aValue to -1.

- Note that the procedures 'average' and 'addStudent' are assumed to have access to the fields marksList and aValue, so we can reference these fields by just using their name (it is as if they are implicitly passed as parameters to the procedure)

- Note also that the procedures are allowed to (and expected to!) make changes to these fields - so the potential side-effects are made explicit

- But what if the external world access marksList or aValue directly?

- ClassAve should be able to restrict access to its fields

- This is typically done by declaring the field as either a private or a public field.

- ClassAve has two private fields and two procedures:
  → private field marksList contains a list of marks
  → private field aValue which is either -1 (not computed) or holds the computed average value.
  → public procedure average() returns aValue if it is not -1, else it computes the avg of marksList & stores it in aValue and returns it.
  → public procedure addStudent(newMark) appends newMark to the end of marksList & resets aValue to -1

- Note that a procedure could also be declared private in which case it is not available to outside world.

(2) Shopping Bills DataSet

- Questions asked of the shopping bill dataset
  - → How many items of a specific category (e.g shirts) were sold?
  - → What is the minimum, maximum and avg. unit price for that category?
- We could define a datatype Category and create an object Shirts of this datatype
- What are its fields?
  - → copy of the entire shopping bill dataset can be stored but this is wasteful
  - → We only need the list of items (i.e rows of the bills) which are of that category
  - → Category could have a private field itemList which carries all the row items from all the bills which are all of the same category
- The datatype could also encapsulate procedures that provide the desired information:
  - → count() returns the no. of items (i.e size of the list)
  - → min() returns the least value of unit price at which the category item was sold
  - → max() returns the largest value of the unit price
  - → average() returns the average across all unit prices for that category
- The Category datatype encapsulates one private field and four procedures
  - → private field itemList carries the list of items, which is hidden
  - → public procedures count(), min(), max() & average() that anyone can call
- What if we want to do this for a particular shop? Or for a specific customer?
  - → The itemList can store only the items sold by that shop
  - → ... or by a specific customer
  - → The procedures will all work without any change!
- Multiple objects of type Category can be created - the list being filtered by category alone, or by a combination of category and customer name and shop name, or anything else
- What if we want to accelerate the execution of these procedures?
  - → Just store their results in private fields - say cValue for count, minValue for min, maxValue for max and aValue for average.
- Can you see how the pattern for encapsulating the Category is quite similar to that for encapsulating the ClassAve?
- If Shirts is an object of this datatype, we may want to find out how many shirts were sold?

→ Note that this may not be the same as count(), since one row in the bill can have multiple items
→ We can write a procedure called number() which finds the sum of the quantity field of the list items

- But number() makes sense only if the category is sold in discrete units (like shirts). What if the category quantity is measured in Kg (for eg. grapes)?

→ For such categories, we can define a procedure quantity() that finds the sum of the (possibly fractional) quantity fields.

- The issue is that just like number() does not make sense for grapes, a measure of quantity in Kg does not make sense for shirts!

USING DERIVED DATATYPES

- While the generic procedures count(), min(), max() and average() worked for all the categories, there could be procedures that work only for some categories and don't work for others.

→ number() works for shirts but not for grapes
→ quantity() works for grapes, but may not work for shirts

- We can write derived datatypes to deal with this.

→ Create derived types NumberCategory & QtyCategory of Category.
→ All the generic procedures count(), min(), max() & average() are available for objects of that derived datatypes also

- We could then add more specific procedures to each derived type that will be available only to objects of that derived type.

→ number() procedure is defined in NumberCategory
→ quantity() procedure is defined in QtyCategory

SUMMARY

- Encapsulation allows procedures to be packaged together with the data elements on which they operate.

→ This is achieved by allowing procedures in addition to fields in a datatype
→ Procedures are the interfaces for others to interact with the objects of this datatype
→ The procedures can have side effects in terms of modifying the field values of the object
→ Fields (or procedures) can be hidden from the outside world by making them as private

- Encapsulation provides an increased level of modularisation when compared to procedures

→ Allows state to be retained b/w calls ... can be used

to speed up the procedures.

→ Side effects are made explicit & more natural

→ We can use derived types to extend the functionality to specific instances when needed

→ But the disadvantage is that objects needs to be created & initialised at the start

- We can find common "object oriented computing" patterns between diff. examples.