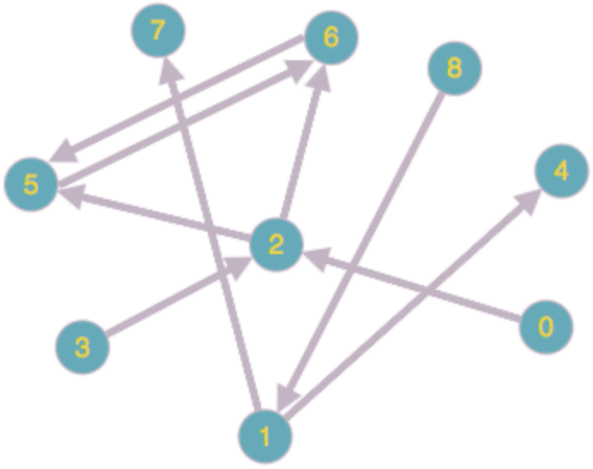
Part 1: Graph Generation



(a) Convert the above graph into an adjacency list

(b) Write a function that can generate an adjacency list based on an input condition that takes two vertices i,j and returns True if the edge i,j is in the graph. This function also takes in the bound of vertices in the graph, Low .. High as integers.

(c) Write a function that can generate a graph using the OS.rand function, which takes as input a percentage value (integer from 0 to 100) and returns the graph where the probability of an edge between any two vertices is the input percentage.

Part 2: Adapt transitive closure algorithms

(a) Read through the section 6.8.1, and describe the first two transitive closure algorithms in your own words. What role does the FoldL play in the declarative algorithm? Then step through the algorithms with the graph from Part 1, (a). Is the return value the exact same in both algorithms?

(b) For the Declarative algorithm, use memory cells to track (1) number of edges added to the adjacency list (you will need to go into the code for Union), (2) number of times SY is returned instead of {Union SY SX}, and (3) largest increase in size of SY when {Union SY SX} is called.

(c) For the Stateful algorithm, use memory cells to track (1) number of edges added and (2) number of times GM is accessed (including updates).

Part 3: Comparisons

(a) Write a function that takes in two adjacency lists, and checks that they are equal (assuming the indices are in order).

(b) Write a procedure that takes a graph as input, and runs both the declarative and stateful algorithms, displaying the accumulated values for both algorithms, and checking to make sure that both algorithms return the same adjacency lists using the function from (a). Include Browse statements before and after each algorithm is ran, so that you can time the algorithms.

(c) Using the procedure from (b), run tests on graphs of different sparsity (from part 1 (c)) where the input probability is varied. Also, calculate the runtime of these algorithms a stopwatch. Describe the results. What did you find?

Part 4: Follow-up Questions

(a) What can you say about the dependency of components within a stateful versus declarative program? Would it be trivial to make the stateful approach concurrent with threading? If so, how could you disentangle the components?

(b) Do the above algorithms work for graphs with disconnected components?

(c) Is there a lazy approach that can save work? In other words, is every computation within the algorithms necessary?

Code from the book

1st Declarative Transitive Closure:

```
fun {DeclTrans G}
  Xs={Map G fun {$ X#_} X end}
in
  {Fold LXs
  fun {$ InG X}
  SX = {Succ X InG} in
    {Map InG
    fun{$ Y#SY}
      Y#if {Member X SY} then
        {Union SY SX} else SY end
    end}
  end G}
```

```
end

fun {Succ X G}
  case G of Y#SY|G2 then
    if X==Y then SY else {Succ X G2} end
  end
end

fun {Union A B}
  case A#B
  of nil#B then B
  [] A#nil then A
  [] (X|A2)#(Y|B2) then
    if X==Y then X|{Union A2 B2}
    elseif X<Y then X|{Union A2 B}
    elseif X>Y then Y|{Union A B2}
    end
  end
end
```

Stateful Transitive Closure:

```
proc {StateTrans GM}
  L={Array.low GM}
  H={Array.high GM}
in
  for K in L..H do
    for I in L..H do
      if GM.I.K then
        for J in L..H do
          if GM.K.J then GM.I.J:=true
          end
        end
      end
    end
  end
end
```

List to Martix:

```
fun {L2M GL}
  M={Map GL fun {$ I#_} I end}
  L={FoldL M Min M.1}
  H={FoldL M Max M.1}
```

```
  GM={NewArray L H unit}
in
  for I#Ns in GL do
    GM.I:={NewArray L H false}
    for J in Ns do GM.I.J:=true end end
  GM
end
```

Matrix to List:

```
fun {M2L GM}
  L={Array.low GM}
  H={Array.high GM}
in
  for I in L..H collect:C do
    {C I#for J in L..H collect:D do
      if GM.I.J then {D J} end
    end}
  end
end
```