Algorithms CS435

October 2021

Below is the template version of breadth-first search. The hook operations are in bold font.

**Algorithm *BFS(G)***

**Input** graph ***G***

**Output** labels the edges of *G*as discovery edges **and cross edges**

***initHesult(G)***

**for all u E** *G. vertices()*

*serLabel(u, UNEXPLORED)*

**postinitVertex(u)**

**for all** *e c aedges()*

*setLabel(e, UNEXPLORED)*

***postlnitEdge(e)***

***for al l v*** c *G.vertices()*

*if* ***isNextComponent(G, v)***   
 ***preComponentVisit(G, v)***

*bfsTraversal(G, v)*

***postCornponentVisit(G, v) return result(G)***

**Algorithm *bfsTraversal(G,*** *s)*   
 ***Q***  *new empty queue*

*setLabel(s, VISITED)*

*Q. enqueue(s)*

***startBFS(G, s)***

**while** *—.QisEinpiy() do*

*v*  *Q.dequeue()*

***preVertexVisit(G, v)***

**for** *all e* **E** *GincidentEdges(v)* **do**   
 ***preEdgeVisit(G, v,***

***if getLabel(e) - UNEXPLORED***   
 ***w***  ***opposite(v, e)***

***edgeVisit(G, v, e, w)***

**Algorithm *isNextComponent(G, v)***   
 **return** *getLabel(v) =* ***UNEXPLORED***

**if** *getLabelM*  *=* ***UNEXPLORED***

***preDiscoveryEdgeVisit(G, v, e, w)*** *setLabel(e, DISCOVERY)*

*seiLabei(w, VISITED)*

*Q. enqueue(w)*

else

***postDiscoveryEdgeVisit(G, v, e, w)***

*setLabel(e, CROSS)*

***crossEdge Visit(G, v, e, w)***

***postEdgeVisit(G, v, e)***   
 ***postVertexVisit(G, v)***   
***finishBFS(G, s)***

**Algorithms CS435**

***List ADT:***

***first(), last(), before(p), after(p), replaceElement(p, o), swapElements(p, q),***

insertBefore(p, o), insertAfter(p, o), insertFirst(o), insertLast(o), remove(p),

size(), isEmpty(), elements() (All of the operations using Rank, but inefficient)

Sequence ADT:

(All of the above List operations), elemAtRank(r), replaceAtRank (r, o),

insertAtRank(r, o), removeAtRank(r), atRank(r), rankOf(p)

BinaryTree ADT:

root(), parent(v), children(v), leftChild(v), rightChild(v), sibling(v),

isInternal(v), isExternal(v), isRoot(v), size(), elements(), positions(),

swapElements(v, w), replaceElement(v, e)

Dictionary ADT (HT and BST based)

findValue(k), insertItem(k, e), removeItem(k), keys(), values(), items()

(General) Graph ADT

numVertices(), numEdges(), vertices(), edges(), aVertex(),

degree(v), adjacentVertices(v), incidentEdges(v), endVertices(e),

opposite(v, e), areAdjacent(v, w), valueAt(v), valueAt(e)

insertVertex(o), removeVertex(v), insertEdge(v, w, o), removeEdge(e)

***Algorithm Design***

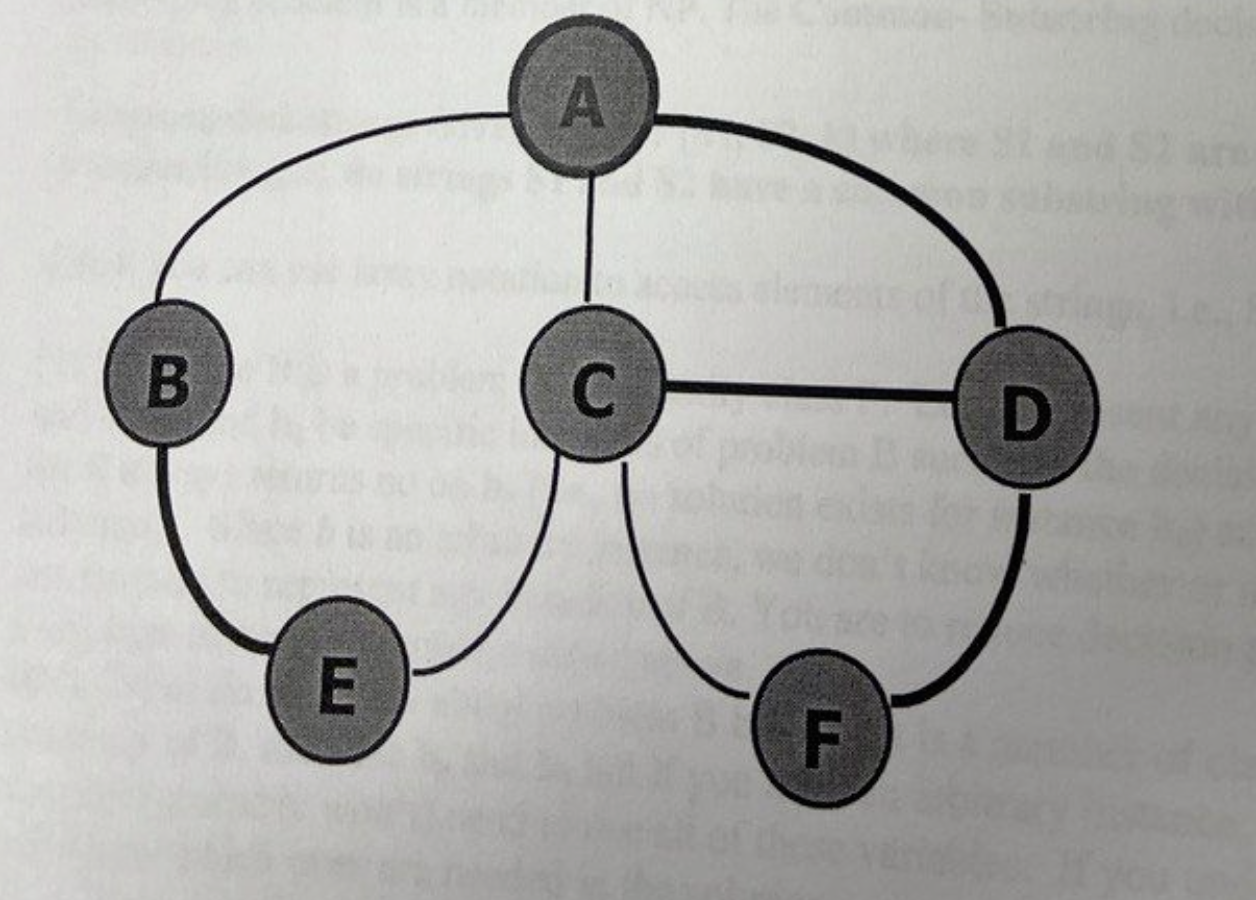
***1.***  ***[25] Given*** a graph G=(V,E) and **a *Sequence* T** of vertices where T⊆V, design **a pseudo code**

algorithm ***findEdges(G,*** T) that finds the edges that connect any pair of the vertices ***in*** Sequence

T. For example, if T=(C,D,F) and G is the graph above, then **findEdges** would return a

sequence {(F,C),(C,D),(D,F)}. Similarly, if **T={B,D,F},** then **findEdges** would return asequence *{(D,F)}* since (D,F) is the only edge connecting any pair of the vertices in T. To receive full   
credit, you must use the BFS *Template* with no unnecessary loops other than those *in* the  
Template *[5-10* points]. **Hint:** note thatT is a **Sequence,** NOTa graph. Use any data structures*if they* improve the efficiency of your solution which could affect how you use the template.

***[10]*** Analyze the complexity of your algorithm by analyzing your pseudo code and **each line of the *above* BFS template. Do line by line analysis on the template algorithm on page 1.**

****

|  |  |
| --- | --- |
| **Algorithm findEdges(G, T)**  **vertexOfT := new Dictionary(HT)**  **for all v in T.elements() do**  **vertexOfT.insertItem(v, YES)**  **return BFS(G)**  **Alothitm initResult(G)**  **result := new Stack**  **Algorithm postDiscoveryEdgeVisit(G, v, e, w)**  **if vertexOfT .findValue(v)=YES /\ vertexOfT .findValue(w)=YES then**  **result.insertLast((v,w))**  **Algorithm result(G)**  **return result** |  |

2. [10] Given a graph G=(V,E) and a Sequence **T** of vertices where T⊆V, design a pseudo code algorithm **isClique(G, T)** that decides whether or not the vertices in **T** form a clique. A clique **T** is asubset of the verticesof **G** such that every vertex in **T** is adjacent to every other vertex in **T**.   
 If the vertices in T form a clique, then return true, otherwise return false. For example, the sequence **T**={C,D,F] is a clique in the above graph because each vertex is adjacent to the other two vertices in Sequence **T**. Similarly, every pair of adjacent vertices like S={B,D} is a clique of size 2. However, if **T**={A,C,F,D}, then **isClique** would return false since **A** and **F** are not adjacent. **Hint**: this can be decided based on the number of edges returned by your solution to   
question 1above together with the number of vertices in **T.**

|  |  |
| --- | --- |
| **Algorithm isCilque(G, T)**  **vertexOfT := new Dictionary(HT)**  **S := T**  **for all v in T.elements() do**  **vertexOfT.insertItem(v, YES)**  **return BFS(G)**  **Alothitm initResult(G)**  **isClique := True**  **Algorithm postDiscoveryEdgeVisit(G, v, e, w)**  **if vertexOfT .findValue(w)=YES /\ isClique is true then**  **cnt = 0**  **for all e in G.incidentEdges(w) do**  **o := G.oppsite(w, e)**  **if vertexOfT .findValue(o)=YES then**  **cnt ++**  **if cnt != S.size() – 1 then**  **isClique = false**  **Algorithm result(G)**  **return isClique** |  |

3. [35] Design a pseudo-code algorithm, **totalWeightPerComponent(G,T),** that takes a weighted graph **G=(V,E)** and a Sequence **T** of edges from G, i.e., T⊆E and returns a Sequence containing the total edge weight of each **connected component** of the subgraph formed by **GT=(V,T),** i.e., the subgraph composed of all the vertices in G but only the edges in T. Your output should be a Sequence containing the total edge weight in each component, i.e., the sum of the weight of the edges ineach component (this will of course include only the edges in T and there will likely be more components than in the graph **G** since **T** may not include all the edges in **E**). For example, suppose a graph has two components (after excluding the edges not in T); if the first component has 8 vertices and 20 edges with total edge-weight of 55 (the sum of the weights of the 20 edges) and if the second has 9 vertices and 28 edges with total edge-weight of 47, then your algorithm would return a Sequence containing {55,47}. [5-10] bonus points if there are no unnecessary loops other than those in the Template. **Hint:** use labels like we did in the homework and examples in the notes to skip vertices and/or edges

|  |  |
| --- | --- |
| **Algorithm totalWeightPerComponent(G, T)**  **edgeOfT := new Dictionary(HT)**  **for all v in T.elements() do**  **edgeOfT.insertItem(e, YES)**  **return BFS(G)**  **Alothitm initResult(G)**  **perWeightList := new Dictionary(HT)**  **cmp = 1**  **perWeightList.insert(cmp, 0)**  **Algorithm preComponentVisit(G, v)**  **cmp += 1**  **Algorithm postDiscoveryEdgeVisit(G, v, e, w)**  **if edgeOfT.findValue(e)=YES then**  **t := perWeightList.findValue(cmp) + e.getWeight()**  **perWeightList.insertItem(cmp, t)**  **Algorithm result(G)**  **return perWeightList** |  |

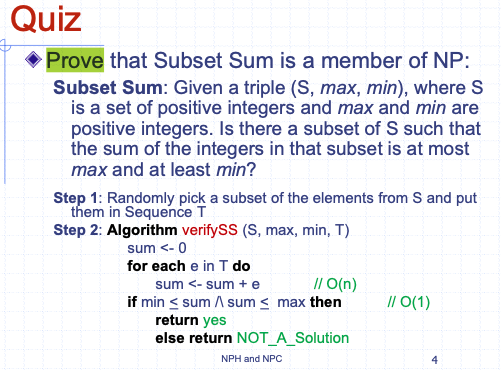
**NP and NP-Complete**

**Decision Problems and Reductions**

**4.**  **(a) [10]** A substring of a string S is a consecutive sequence of characters that occurs in S. For example, the substrings of a string "abcc" are the empty string "", and "a", "b", "c", **"ab", "bc",** "cc", "abc", "bcc", and *"abcc".* Given two strings, a common substring would be a substring thatoccurs in both strings. For example, given S1 ="abedefg" and S2= "xyzcdefabc”, then "def”, *"cde", and* "abc" are common substrings of S1 and S2 with length 3. Prove that the **Common-Substring** problem is a member of NP. The **Common- Substring** decision problem can be stated as follows:

**Common-Substring: Given a triple (S1, S2, k) where S1 and S2 are strings and k is a positive integer, do strings S1 and S2 have a common substring with length k?**

***Hint:*** you can use array notation to access elements of the strings, i.e., **S1[i].**

****

|  |  |
| --- | --- |
| Algorithm commonSubString(S1, S2, k)  dictS1 := new Dictionary(HT)  dictS2 := new Dictionary(HT)  dictS1 := makeDict(S1, k)  dictS2 := makeDict(S2, k)  result := new Stack  iter = dictS1.items()  while iter .hashNext() then  (key, value) := iter.next()  if dictS2.findValue(key) then  result.insertLast(value)  return result  Algorithm makeDict(S, k)  i := 0  dictS := new Dictionary(HT)  while i < i.size()-k do  dictS.insertItem(minS[i:i+k], 1)  i := i + 1  return dictS |  |
| Randomly pick string of elements with length k from S1 or S2 and them in Sequence T  Algorithm verifyCSS(S1, S2, k, T)  for each e in T.elements() do  if !checkContains(S1, e) \/ !checkContains(S2, e) then  return NO\_A\_Solution  return YES | O(n^2)This line is poly time |

**The Common-Substring problem is a member of NP because it can be verified in polynomial time.**

**5.** [10] Suppose B is a problem in complexity class P. Let ***b*** represent any arbitrary instance of B

and let **bo** and **b1** be specific instances of problem B such that the decision/verification algorithm   
for B always returns no on **bo** (i.e., no solution exists for instance **bo**) and yes on **b1** (has a solution). Since *b* is an arbitrary instance, we don't know whether or not ***b***has a solution so it can be used to represent any instance of B. You are to reduce decision problem B to the **Common-Substring** problem stated above.

**Hint:** What do weknow about problem B because it is a member of class P? If you need specific enstances of B, then use **bo** and **b1** but if you need an arbitrary instance, then use ***b;***note that you may not (probably won't) need to use all of these variables. If you understand reductions, you will know which ones are needed in the solution.

**Problem B can be reduced to the Common-Substring problem, utilizing the fact that B is in the complexity class P.**

|  |  |
| --- | --- |
| ReduceBtoCommonSubstring(b):  S1 = EncodeInstance(b) // Convert instance b of problem B into a string S1  S2 = "abc" // Arbitrary string used for the reduction  k = 3 // Length of the common substring to search for  result = CommonSubstring(S1, S2, k)// Invoke the Common-Substring function  R := new Stack  R.insertLast(2)  if result == True then  return (R, 2, 2)  else  return (R, 1, 1)  if result == True:  return "Yes"  else:  return "No" |  |

**Algorithms CS435**

6. [10] Given a graph G, the **Graph Coloring problem** attempts to assign colors to each vertex of G such that no two adjacent vertices are the same color; the optimization problem tries to do this with the fewest number of colors. Convert/restate the **Graph Coloring problem** (finding the minimum number of colors needed) as a decision problem that can decide whether the instance has a solution.

**Assuming there is no cycles in graph the minimum no of colors will be 2**

**If each edge on graph has n(n-1)/2 then we need n colors**

|  |  |
| --- | --- |
| Algorithm **graphColoring**(G)  return BFS(G)  Algorithm **initResult**(G)  colorDict = new Stack  colorDict.insert(Blue)  colorDict.insert(Red)  colorCNT := 0  Algorithm **preComponentVisit**(G, v)  cntComp ++  Algorithm visitCrossEdge()  colorDict.insert(RandomNewColor)  Algorithm **result**(G)  return colorDict.size()  Algorithm getIndex()  colorCNT ++  Return colorCNT % colorDict.size()  Algorithm **preDiscEdgeVisit**(G, v, e,w)  w.color = colorDict[getIndex()] | Algorithm **graphColoring**(G)  return BFS(G)  Algorithm **initResult**(G)  colorDict = new Stack  colorDict.insert(Blue)  colorDict.insert(Red)  colorCNT := 0  Algorithm **preComponentVisit**(G, v)  cntComp ++  Algorithm visitCrossEdge()  colorDict.insert(RandomNewColor)  Algorithm **result**(G)  return colorDict.size()  Algorithm getIndex()  colorCNT ++  Return colorCNT % colorDict.size()  Algorithm **preDiscEdgeVisit**(G, v, e,w)  w.color = colorDict[getIndex()] |

7. An independent set is a **subset** T of the vertices of G=(V,E), i.e., T⊆V such that none of the vertices in T are adjacent to each other. For example, in the graph above, the vertices A, E, and F form an independent set of size 3 since none of these vertices are adjacent. Similarly, vertices B and D form an independent set of size 2 since B and D are not adjacent.

The **Independent Set** decision problem can be stated as follows:

**Independent-Set: Given an integer K and a graph G, does G have an independent set of size K?**

[10]: Prove that the **Independent Set problem** is a member of **NP.**

**Answer**

**The Independent Set problem is a member of NP because given a potential solution, it can be verified in polynomial time, and if a solution exists, it can be found in polynomial time using a nondeterministic algorithm.**

**8.** [5] The **Clique** optimization problem searches for the largest clique in a graph **G. The Clique** decision problem can bestated as follows:

**Clique: Given a pair (G, K) where G=(V,E) is a graph and *K* is a positive integer, does there exist a clique S in G, such that size of S equals K?**

Can we conclude anything if there is a polynomial time reduction of the **Clique** problem to the **Independent Set** problem (assume that you correctly answered the previous question regarding the **Independent Set** problem)? Recall, from the chart of reductions in the lecture notes, that Circuit-Sat has been reduced to SAT and SAT has been reduced to 3-CNF-Sat and 3-CNF-Sat has been reduced to the **Clique** problem. If there is a conclusion, then state and justifyit otherwise explain why nothing can be concluded.

**Notation: A —>p** B means instances of problem **A** can be reduced to instances of problem B by function p in polynomial time.

9. Answer **true** or **false** to each of the following questions (a) to (j). if true, briefly justify your answer (using at most 2-3 sentences in the space provided below). If false, give a counter example, such as, "A could be MST and B could be halting problem" or "A could be in NPC and B in P", etc. **Zero points without a justification**. If true, explain based on the definitions of P, NP, NPH, and NPC.

For parts (a) to (j), assume A and B are **specific** decision problems and that the subscript *f* denotes the polynomial-time function for reducing A to B. **(Please put your explanation on the same *page* or the back so it's easy to find; this will save me a lot of time and will result in the grading being\_ finished sooner)**

(a)[2] *if* A —>fB and A∈P, then B∈P

False, because A can be NP and B can be P, then you cannot NP reduce to P

(b) [2] if A —>f B and B -> gA, then A, B∈P

False, because A can be NP and B can be P, then you cannot NP reduce to P

(c)[2] if A ->fB and A∈NPC, then B∈NPC

True, because A and B always has solutions, and they are same

(d) [2] if A ->f B and B∈NPC, then A∈NP

False , Because A can have solution in P , while B can be NPH

(e) [2] if A ->fB and B∈P, then A∈NP

False, If B is p and A is NP or A can be NPH then it’s false , as we can find solution for A and verify it but we cannot verify solution for B.

(f) [2] if A->f B and A, B∈NP, then B->gA

False, because A can be MST of P, but B can Subset Problem of NP

(g) [2] if A-**>f** B and A∈NPH and B∈P, then P =NP

False, because A cannot be reduced to P

NPH cannot be reduced to P

(h) [2] if A->**f** B and B∈NP, then A∈NP

False, A can be NP and B can be P

[2] if A->**f** B and B∈NPH, then A∈NP

False because B cannot be P, whereas A can be P

(j) [2] if A->**f** B and A is the Halting Problem, then B∈NPH

False, because A is decideable problem where as B can be NPC and maybe get solution.

10. [5] **Extra credit**: Explain how and why Sudoku is a 9-Coloring problem if the 9 by 9 grid is represented as a Graph with 81 vertices. If you are not familiar with Sudoku or didn't hear me describe it in class, then skip this question.

By solving the 9-Coloring problem for the Sudoku graph, we effectively solve the Sudoku puzzle by assigning digits (colors) to each cell (vertex) such that the Sudoku rules are satisfied.