1.

a)

i.

Need to both show problem is NP (1), then reduce HP to it (2).

(1) Show HP(s, t) is NP:

Using a guess & check:

Guess path (I.e. list of edges from s, t). Verify this is Hamiltonian path: I.e. check path contains all nodes of graph, check starts at s, ends at t. This verification is poly.

(2) Show poly reduction from HP to HP(s, t):

Run HP(s, t) on the same graph, using s & t as any arbitrary single node (so s & t are the same).

ii.

Need to both show problem is NP (1), then reduce HP to it (2).

(1) Show HPPE is NP:

Using a guess & check:

Guess path (I.e. list of edges).

Verify this is Hamiltonian path: I.e. check path contains all nodes of graph, check starts and ends in same place.

Verify for each pair, check exactly one of the two is contained in the path.

This verification is poly.

(2) Show poly reduction from HP to HPPE:

Construct G’ as follows: Remove any loops (doesn’t impact HP). Then create a “triangle” of nodes x, y, z connected to each other. Call edge xy e1, edge xz e2. Add (e1, e2) to the set S, fulfilling the non-empty requirement.  
Now connect x to all nodes in G.  
If G had a HP originally, G’ still has a HPPE by starting at z, then y, then x. We’ve used e1 but not e2 as required. Then from x it can go to the HP in G like normal.  
If G didn’t have a HP, G’ doesn’t have HPPE either, because adding these nodes didn’t create any new paths between nodes in G (x needs to be adjacent to y or z in the HP, or you will miss y or z and not have a HP).

b)

i. (tutorial q)

Run machine for L1, run machine for L2 (one after the other, this is still P time). Accept if both machines accept, else reject.

ii.

Try all splittings (iterate along): break into two halves, check first accepted by machine for L, check second half rejected by machine for L.

iii.

False if co-NP /= NP (likely).

As assume L’ in NP. Then we can provide a machine which can accept co-NP in NP time:

Give y, which we want to check if y in co-NP.

With arbitrary member of L, say x, concatenate: xy. Run machine for L’ on this. Accepts iff y not in L.

2.

a)

i.

Any language in NL can be logspace reduced to this problem.

ii.

See lecture notes

iii.

Need to both show problem is NL (1), then reduce RCH to it (2).

(1) Show MODRCH is in NL:

Not sure if this works...:

We know co-NL = NL. So we know ~RCH is also in NL. We can just compose two machines M1 for RCH(x,y) and M2 for ~RCH(x,z) to form ND I/O TM called M. Now M only accepts if both M1 and M2 accept. There must be one such branch iff x reaches y and x doesn’t reach z.

(2) Show poly reduction from RCH to MODRCH:

Set x & y as in RCH, same nodes. Set z as x, which will trivially be not reachable from x (mmm not sure about this, edge case depending on exact design of RCH).

b)

i.

See lecture notes

ii.

3.

a)

See lecture notes

b)

i.

Use binary tree OR’ing together all inputs. Binary trees give log time.

ii.

First layer: AND together all pairs of inputs. Next layers: OR these together in binary tree. 1 + binary tree gives log time.

c)

Similarly to the section in lecture notes, we square adjacency matrix log n times. This gives us all RCH answers at once.

The difference here is we have optimization instead of decision. So instead of computing a matrix element of the square matrix like so:

A\_{ik} = \lor\_{i} (a\_{ij} \land a\_{jk})

We do (replace /\ with sum, replace \/ with max):

A\_{ik} = max ( a\_{i1} + a\_{1k}, … , a\_{iN} + a\_{Nk} )

4.

a)

i.

Take any language L in NP, we want to show it must be in DP.

First consider the empty set language (no words) decided by a machine that always rejects. This is in NP. Therefore the entire alphabet language (all words) is in co-NP, decided by a machine that always accepts.

Now using the definition of DP, take L1 as L in NP. Take L2 as the entire alphabet language. The intersect language is just L again, so it must be in DP. Thus L in DP.

ii.

See lecture notes.

iii.

Oracles are weirdly defined such that you can check if a word is not in a language, even if a machine for that were to never halt.

So given any language in DP, we can decide it with a P algo with access to a NP oracle, since the P algo can just check if its in L1 (using the oracle), then check if its in L2 (using the oracle, guaranteed result). If it's in the first, but not the second, accept.

iv.

$$P^{DP} \subseteq P^{NP}$$

$$P^{DP} \subseteq P^{P^{NP}} = P^{NP} $$ (P^P adds no power)

$$P^{NP} \subseteq P^{DP}$$

Follows directly from $$NP \subseteq DP$$

$$P^{DP} = P^{NP}$$

b)

i.

First, consider the problem: is there a simple path of size k in graph G?

This is in NP. To see this, use a guess and check. Guess a set of vertices of size k + 1 (i.e. k edges), and verify this is a simple path: check each sequential vertex in this list is connected (in the right direction), check no repeated vertices in list. This check is in poly time, so the problem is in NP.

Also, see that since the above in is NP, the co problem: is there **no** simple path of size k in graph G, is in co-NP.

Now for MAXPATH. Use the above algorithm to check if there is a simple path of of size k (which is in NP), and use the co algorithm to check if there is no simple path for all size greater than k (i.e. k+1, k+2 ... n) up to the number of nodes (note poly number of repeats & this is in co-NP). Since if there's no path of any sizes greater than k, the path of size k must be the longest, so we can use the outcome from this to decide MAXPATH. This is in NP ∩ co-NP = DP, as required.

ii.

Again use the fact that the problem: is there a simple path of size k in graph G? is in NP.

For k counting down from the number of nodes, apply the above algorithm, as the NP oracle, and stop when we get a positive result. So the first positive result tells us there is a simple path of size k in the graph, which is the longest length.

Secondly, we need to check if the path of this length is unique. Keeping a counter, for every edge in the graph, we test omit that edge, then consult the NP oracle if there is still a simple path of the max length. If we get a negative result (no path anymore), this edge must be in **ALL** (if there are multiple) longest simple paths (since if it were only in one path, there would still be a simple path of this longest length). In this case we increment the counter. Repeat this for all edges (not forgetting to always put edges back).

Lastly after these tests, if the counter is equal to the max length, all the shared edges are in one solution, so it is unique. If the counter is less than the max length, there are edges which are not shared by all solutions, so there is no unique solution.