Worth: 5%

1. **[10 marks]**

   In the lecture we have shown that 3-Sat is NPC. Now state the decision problem 2-Sat and show that 2-Sat is, in fact, in $\mathcal{P}$.

   Solution. Each clause in a 2-Cnf formula is of the form $(x \vee y)$, which is equivalent to $(\neg x \implies y)$, and also to $(\neg y \implies x)$. Using this transformation, we construct a graph $G = (V, E)$ from a 2-Cnf formula φ as follows: we define a vertex for each variable and its negation, and we define an edge between two nodes x and y if and only if there is a clause in φ that is equivalent to $(x \implies y)$, namely, a clause of the form $(\neg x \vee y)$.

   Observe that if G contains an *edge* from x to y, then it also contains an *edge* from $\neg y$ to $\neg x$, simply because $(x \implies y)$ is equivalent to $(\neg y \implies \neg x)$. Consequently, if G contains a *path* from x to y, then it also contains a *path* from $\neg y$ to $\neg x$.

   Also observe that if φ is satisfiable and there is a path from x to y in G, by transitivity of implication this is equivalent to $x \implies y$.

   Claim: A 2-Cnf formula φ is unsatisfiable if and only if there exists a variable x such that there exists a path from x to $\neg x$ and a path from $\neg x$ to x in G.

   Proof. Assume φ is satisfiable. By the above observation, we have $x \implies \neg x$ and $\neg x \implies x$. Since $x \implies y$ is true iff either x is false or y is true, it follows that $x \implies \neg x$ is true iff x is false, and $\neg x \implies x$ is true iff $\neg x$ is false iff x is true. Since, x cannot both be true and false, we have reached a contradiction. And hence, φ is not satisfiable.

   Conversely, suppose there are no such paths in G. We construct a satisfying assignment as follows: pick a literal x in G that has not been assigned a truth value yet and that does not have a path from x to $\neg x$. Assign the truth value True to x. Then assign True to all vertices reachable from x, and assign False to their negations. Repeat this until all vertices have been assigned a truth value. Check that thiis procedure is well defined: if there is a path from x to y and a path from x to $\neg y$, then by above observations there would be a path from x to $\neg y$ and a path from $\neg y$ to $\neg x$, thereby leading to a path from x to $\neg x$, which yields a contradiction. □

   Because of the claim, we obtain the following efficient algorithm to decide satisfiability of a 2-Sat formula.

```
1: procedure 2-Sat(φ)
2:     construct the graph G = (V, E) from φ as mentioned above
3:     for x ∈ V do
4:         Run BFS and compute the shortest distance from x to all vertices reachable from x and store
   them
5:     for x ∈ V do
6:         if there is a path from x to ¬x and a path from ¬x to x then
7:             return False
8:     return True
```

Complexity: $\mathcal{O}(|V|(|V| + |E|)) = \mathcal{O}(|V|^2 + |V||E|)$.

□

2. **[15 marks]**

Consider the following DisjointHamiltonianPaths decision problem ("DHP" for short).

- *Input:* Graph $G = (V, E)$ — G may be directed or undirected.
- *Output:* Does G contain at least two edge-disjoint Hamiltonian paths?

(Two paths are said to be *edge-disjoint* if there is no edge that belongs to both paths.)

Write a *detailed* proof that DisjointHamiltonianPaths is *NP*-complete. State what you are doing at each step of your solution: it will be graded on its structure as much as on its content.

Solution. • First, we show that DHP $\in$ *NP* by describing a polytime verifier for DHP.

VerifyDHP($G = (V, E), C$):
where $C = ([u_1, \ldots, u_n], [v_1, \ldots, v_n])$ is a pair of permutations of V
Check that G contains all the edges $(u_1, u_2), \ldots, (u_{n-1}, u_n), (v_1, v_2), \ldots, (v_{n-1}, v_n)$.
Check that no edge of G belongs to both paths $u_1, \ldots, u_n$ and $v_1, \ldots, v_n$.
Return True if both conditions hold; return False otherwise.

Clearly, VerifyDHP runs in polytime for every input and it outputs True for some certificate C iff G contains two edge-disjoint Hamiltonian paths.

- Next, we show that DHP is *NP*-hard by proving UHP $\leq_p$ DHP, where UHP is the *undirected* Hamiltonian Path decision problem.

Given undirected graph G, output directed graph G' that contains the same vertices as G and edges in both directions for each undirected edge in G. Formally, if $G = (V, E)$, $G' = (V, E')$ where $E' = \big\{(u, v), (v, u) : \{u, v\} \in E\big\}$.

Clearly, G' can be computed in polytime from G.
Also, if G contains some Hamiltonian path $v_1, \ldots, v_n$, then G' contains the two Hamiltonian paths $v_1, \ldots, v_n$ and $v_n, \ldots, v_1$, with no directed edge belonging to both paths.
Finally, if G' contains two edge-disjoint Hamiltonian paths $v_1, \ldots, v_n$ and $u_1, \ldots, u_n$, then $v_1, \ldots, v_n$ is a Hamiltonian path in G.
Since UHP is *NP*-hard, so is DHP.

□

3. **[15 marks]**

Give a precise definition for the Disjoint Hamiltonian Paths *Search Problem*. Then, write a *detailed* argument that this problem is polynomial-time self-reducible. Once again, your solution will be graded on its structure as well as its content, so make sure to state what you are doing at each step.

Solution. The Disjoint Hamiltonian Paths Search Problem ("DHPS" for short) is defined as follows:

- *Input:* Graph $G = (V, E)$ — G may be directed or undirected.
- *Output:* A pair of edge-disjoint Hamiltonian paths in G, if they exist — the special value nil otherwise.

Suppose that DHP-D is an algorithm that correctly solves the DHP decision problem with runtime $t(n, m)$ (where $n = |V|$ and $m = |E|$).

Consider the following algorithm to solve the DHPS problem.

```
DHP-S(G = (V, E)):
    if not DHP-D(G):  return NIL
    for e ∈ E:
        if DHP-D(G − e):  # where G − e = (V, E − {e})
            G := G − e
    return E
```

Then DHP-S runs in worst-case time $O(m \cdot t(n,m) + m(n+m))$, which is polynomial if $t(n,m)$ is.

Moreover, DHP-D(G) is a loop invariant: it is obviously true before the loop starts and remains true throughout the execution, because of the explicit if-statements. This means that when the loop ends, E contains every edge in a pair of edge-disjoint Hamiltonian paths (by the loop invariant) and no other edge (because every other edge is removed by the loop).

Note that the algorithm does not produce the paths themselves. Separating the edges into those two paths is a much more complicated problem!

□

4. **[20 marks]**

For each of the following decision problems D, state whether $D \in P$, $D \in NP$ or $D \in coNP$ — make the strongest claim that you can. Then, justify your answer by writing down an explicit algorithm for D and explaining why it satisfies the properties required for your answer. *Do **not** attempt to justify that other complexity classes are incorrect; focus on providing evidence that your choice is correct.*

(a) **[10 marks]** SomeShortPaths ("SSP" for short)

Input: Undirected graph $G = (V,E)$, vertices $s,t \in V$, non-negative integer $k \leq |V|$.
Output: Does G contain *some* simple path from s to t with no more than k edges on the path?

Solution. SSP $\in P$. On input $G = (V,E)$, $s,t \in V$, $k \leq |V|$:

```
run BFS in G starting from s
let ℓ be the number of edges on the path from s to t found by BFS
return ℓ ≤ k
```

Because BFS is known to find a path with the minimum number of edges, the algorithm returns True iff G contains a simple path from s to t with at most k edges. Moreover, BFS runs in polytime so the entire algorithm is polytime.

□

(b) **[10 marks]** AllShortPaths ("ASP" for short)

Input: Undirected graph $G = (V,E)$, vertices $s,t \in V$, non-negative integer $k \leq |V|$.
Output: Does *every* simple path in G from s to t contain no more than k edges?

Solution. ASP $\in coNP$. On input $G = (V,E)$, $s,t \in V$, $k \leq |V|$, c:

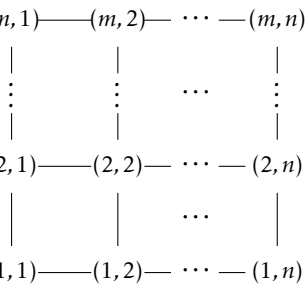
```
# where c is a sequence of edges from G
check that c represents a simple path in G from s to t
return len(c) > k
```

Clearly, the algorithm runs in polytime. Moreover, it outputs True for some value of certificate c iff there is some simple path in G from s to t with more than k edges, iff G,s,t,k is a no-instance.
(Note: It's acceptable to invert the output of the verifier.)

□

5. **[20 marks]**

Your friends want to break into the lucrative coffee shop market by opening a new chain called *The Coffee Pot*. They have a map of the street corners in a neighbourhood of Toronto (shown on the right), and estimates $p_{i,j}$ of the profits they can make if they open a shop on corner (i,j), for each corner. However, municipal regulations forbid them from opening shops on corners $(i-1,j)$, $(i+1,j)$, $(i,j-1)$, and $(i,j+1)$ (for those corners that exist) if they open a shop on corner (i,j). As you can guess, they would like to select street corners where to open shops in order to maximize their profits!



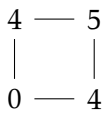
(a) **[5 marks]** Consider the following greedy algorithm to try and select street corners.

```
C := {(i, j) : 1 ≤ i ≤ m, 1 ≤ j ≤ n}   # C is the set of every available corner
S := ∅   # S is the current selection of corners
while C ≠ ∅:
    pick (i, j) ∈ C with the maximum value of p_{i,j}
    # Add (i, j) to the selection and remove it (as well as all corners adjacent to it) from C.
    S := S ∪ {(i, j)}
    C := C − {(i, j), (i − 1, j), (i + 1, j), (i, j − 1), (i, j + 1)}
return S
```

Give a precise counter-example to show that this greedy algorithm does not always find an optimal solution. State clearly the solution found by the greedy algorithm, and show that it is not optimal by giving another selection with larger profit.

(b) **[15 marks]** Prove that the greedy algorithm from part (a) has an approximation ratio of 4. (Hint: Let S be the selection returned by the greedy algorithm and let T be any other valid selection of street corners. Show that for all $(i,j) \in T$, either $(i,j) \in S$ or there is an adjacent $(i',j') \in S$ with $p_{i',j'} \geq p_{i,j}$. What does this means for all $(i,j) \in S$ and their adjacent corners?)

Solution. (a) For the input below (where we've indicated the profit of each corner), the algorithm returns the corners with profits 5 and 0, for a total of 5. However, picking both corners with profits 4 would give a total profit of 8 instead.



(b) Note that the input to the problem can be represented as an undirected graph G, and valid selections of corners are the same as independent sets in G.

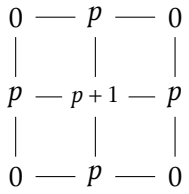
Let $A(G)$ be the smallest total profit of any independent set returned by the greedy algorithm, and $M(G)$ be the *maximum* profit of *all* independent sets of G. We prove that for all inputs G, $A(G) \geq M(G)/4$ — and that for at least one input G_0, $A(G_0) = M(G_0)/4$.

Let S be any independent set returned by the algorithm, and T be any independent set with maximum profit in G. For all $v \in T$, if $v \notin S$, then there is some corner $v' \in S$ such that $(v',v) \in E$ and $p(v') \geq p(v)$ — otherwise, v could be added to S. When v was removed from G by the algorithm, it was because of some adjacent corner v' being added to S, which means that at that point, v' had the largest profit among all remaining corners, including v.

Since no corner in S has more than 4 neighbours, for all $v \in S$, there are at most 4 corners $v_1, v_2, v_3, v_4 \in T$ such that $p(v) \geq p(v_1)$, $p(v) \geq p(v_2)$, $p(v) \geq p(v_3)$, $p(v) \geq p(v_4)$. In other words, for

all $v \in S$, $4p(v) \geq p(v_1) + p(v_2) + p(v_3) + p(v_4)$, in the worst case, to "cover" all corners in T. Hence, $4p(S) \geq p(T)$, *i.e.*, $A(G) \geq M(G)/4$, as desired.

To show that $A(G)$ can be equal to $M(G)/4$, consider the input below. The algorithm will select the corners with profits $(p+1)+0+0+0+0 = p+1$ while the maximum profit is obtained by picking the corners $p+p+p+p = 4p$. This is not quite the desired factor of 4, though it can be made arbitrarily close to it by picking p large enough. To get a factor of 4 exactly, simply set the profit of the middle "corner" to p. Then, the selection returned by the algorithm is no longer *guaranteed* to be sub-optimal, but it is *possible* that the algorithm will select the middle "corner" first, and end up with a solution whose value is exactly 1/4 of the optimum.



□

6. **[20 marks]**

You are using a multi-processor system to process a set of jobs coming in to the system each day. For each job $i = 1, 2, \ldots, n$, you know v_i, the time it takes one processor to complete the job.

Each job can be assigned to one and only one of m processors — no parallel processing here! The processors are labelled $\{A_1, A_2, \ldots, A_m\}$. Your job as a computer scientist is to assign jobs to processors so that each processor processes a set of jobs with a reasonably large total running time. Thus given an assignment of each job to one processor, we can define the *spread* of this assignment to be the minimum over $j = 1, 2, \ldots, m$ of the total running time of the jobs on processor A_j.

Example: Suppose there are 6 jobs with running times $3, 4, 12, 2, 4, 6$, and there are $m = 3$ processors. Then, in this instance, one could achieve a spread of 9 by assigning jobs $\{1, 2, 4\}$ to the first processor (A_1), job 3 to the second processor (A_2), and jobs 5 and 6 to the third processor (A_3).

The ultimate goal is find an assignment of jobs to processors that **maximizes** the spread. Unfortunately, this optimization problem is *NP*-Hard (you do not need to prove this). So instead, we will try to approximate it.

(a) **[15 marks]**

Give a polynomial-time algorithm that approximates the maximum spread to within a factor of 2. That is, if the maximum spread is s, then your algorithm should produce a selection of processors for each job that has spread at least $s/2$. In your algorithm you may assume that no single job has a running time that is significantly above the average; specifically, if $V = \sum_{i=1}^{n} v_i$ denotes the total running time of all jobs, then you may assume that no single job has a running time exceeding $\frac{V}{2m}$.

Show that your algorithm achieves the required approximation ratio.

(b) **[5 marks]**

Give an example of an instance on which the algorithm you designed in part (6a) does not find an optimal solution (that is, one of maximum spread). Say what the optimal solution is in your instance, and what your algorithm finds.

Solution. (a)

ALLOCATE$(m, v[1,\ldots,n])$:
  $A[1] = \cdots = A[m] = []$ # $A[j]$ is the list of jobs assigned to processor j
  $a[1] = \cdots = a[m] = 0$ # $a[j]$ is the total time of jobs in $A[j]$
  for $i := 1, 2, \ldots, n$:
    # add job i to the processor with the current smallest total time
    find j such that $a[j]$ is minimum
    $A[j] := A[j] \cup \{v[j]\}$
    $a[j] := a[j] + v[j]$
  return $A[1], \ldots, A[m]$

For an arbitrary input $(m, v[1,\ldots,n])$, let OPT be the maximum spread possible and let GS be the spread of the solution returned by the algorithm (GS is short for "Greedy Spread"). We show that $GS \geq OPT/2$.

First, note that $OPT \leq V/m$. Otherwise, every processor would have a total strictly greater than V/m which would make the sum of the processors' runtime strictly greater than V, a contradiction.

Next, let $A[i], \ldots, A[m]$ be the solution returned by the algorithm, with total processor times $a[1], \ldots, a[m]$. Let $a[k]$ be the minimum processor time (so $a[k] \leq a[j]$ for all j and $GS = a[k]$).

For each processor j, let b_j be the running time of the *last* job added to $A[j]$. The fact that b_j is added to $A[j]$ instead of another processor means that $a[j] - b_j$ is no larger than any other processor total when b_j is added. In particular, $a[j] - b_j \leq a[k]$. Since $b_j \leq V/2m$, this means $a[j] \leq a[k] + V/2m$ for every j.

But this means $V = \sum_{j=1}^{m} a[j] \leq \sum_{j=1}^{m} (a[k] + V/2m) = m(a[k] + V/2m)$. Solving for $a[k]$ yields $a[k] \geq V/2m$.

Hence, $GS = a[k] \geq V/2m \geq OPT/2$, as desired.

(b) Consider the input $[3, 3, 2, 2, 2]$ with $m = 2$ processors.

The algorithm assigns jobs $[v_1, v_3, v_5] = [3, 2, 2]$ to A_1 (with total time 7) and jobs $[v_2, v_4] = [3, 2]$ to A_2 (with total time 5). So the spread of the algorithm's output is 5.

However, it is possible to achieve a spread of 6 by assigning jobs as follows: $A_1 = [v_1, v_2] = [3, 3]$, $A_2 = [v_3, v_4, v_5] = [2, 2, 2]$.

□