PRIFYSGOL ABERYSTWYTH UNIVERSITY

SEMESTER ONE 2025

COMPUTER SCIENCE

CS31920 : ADVANCED ALGORITHMS

Time allowed: **2 HOURS**

Answer all **THREE** questions

All questions carry equal marks.

YOU MAY BRING THE FOLLOWING INTO THE EXAMINATION:
UP TO 3 SHEETS OF NOTES ON A4 PAPER (WHICH MAY BE DOUBLE-SIDED). THERE ARE NO ADDITIONAL RESTRICTIONS ON CONTENT OR FORMATTING OF THESE NOTES.
NO ELECTRONIC DEVICES.

CALCULATORS ARE NOT PERMITTED.

STUDENTS MAY SUBMIT ANSWERS TO THIS PAPER IN EITHER WELSH OR ENGLISH

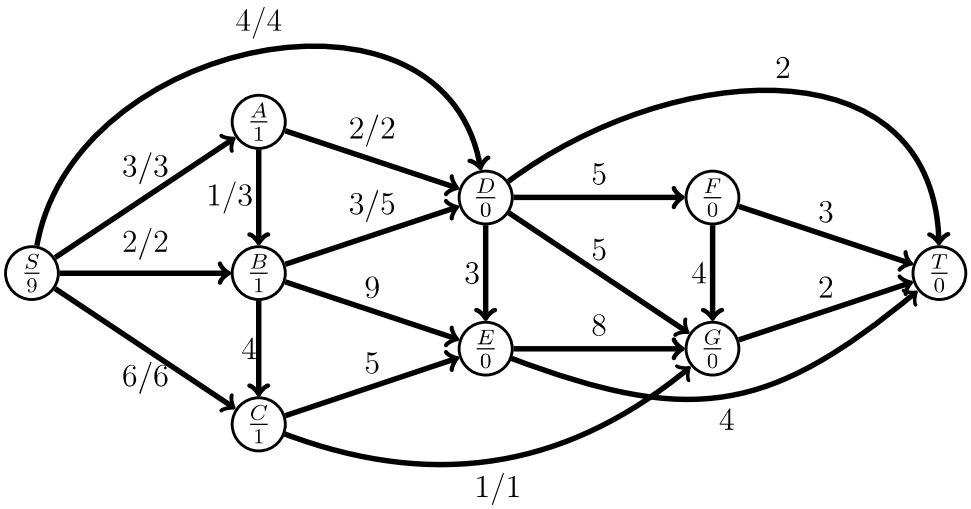
Once seated, students may complete the front cover of the answer book(s) and the attendance slip ONLY.

DO NOT TURN THE EXAMINATION QUESTION PAPER OVER UNTIL INSTRUCTED TO DO SO.

1. **Maximum flow**

a) **Goldberg/Tarjan algorithms for computing maximum flow**

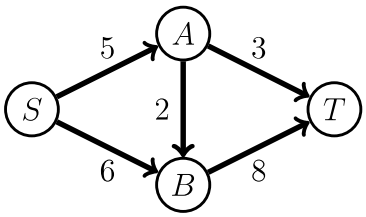
The graph below is a flow network with source S, sink T, a valid node labelling, and a pre-flow. Nodes have names $V = \{A, B, C, D, E, F, G, S, T\}$ and each node has a label. If, for example, the node F has label 3 than we would indicate this as $\frac{F}{3}$ in the picture below. Edges are arrows and annotated either with one or two numbers. If an edge is annotated with one number (for example, $\xrightarrow{5}$) then the number indicates the edge capacity (edge capacity 5 in this example). If an edge is annotated with two numbers (for example, $\xrightarrow{2/4}$) then the numbers indicate the pre-flow and edge capacity (pre-flow 2 over an edge with capacity 4 in this example). Consider the flow network and pre-flow below. (1) For each node except source and sink provide the excess value. (2) Provide a list of all applicable relabel and push operations. For each operation include the change in value(s) that it implies. Don't forget to indicate the name of the node for relabel operations and the name of the edge (in the form $B \to C$ for an edge from B to C) for push operations.



[11 marks]

b) **Linear program modelling a flow problem**

Consider the flow network below. Define an equivalent linear program. Explain the different components and their purpose.



[11 marks]

c) **Algorithms for the maximum flow problem**

Name three different algorithms for the maximum flow problem, each following a different approach. For each algorithm, give a very rough description (in no more than three sentences) of the main algorithmic ideas. For two of them, say when it would be a good idea to use this algorithm. [11 marks]

2. Difficult problems and how to solve them

a) A variant of the maximum flow problem

We consider a variant of the maximum flow problem that we call maximum flow problem with demands. Input is a flow network where additionally each edge does not only have a capacity c but also a demand d (with $0 \leq d \leq c$ for all demands). A flow in this network is only a valid solution for the maximum flow problem with demands, if the flow on each edge is at least as large as the demand (but still not bigger than the capacity). The definition of value of the flow is the same as in the maximum flow problem. Output is a maximum flow if one exists or 'no solution' if there is no flow that can satisfy the demand on all edges. What can you say about the difficulty of the problem? How would you propose to solve it? [11 marks]

b) Variants of the travelling salesperson problem (TSP)

We consider three variants of the TSP, all defined on a complete, directed graph with n nodes (meaning that for any two nodes $v_1 \neq v_2$, the edges (v_1, v_2) and (v_2, v_1) both exist). The variants differ only in the way the edge weights are defined. For each variant, what can you say about the difficulty of computing a shortest tour that starts and ends at the same node and visits each other node exactly once? What can you say about the difficulty of computing such a tour that is at most twice as long as a shortest tour?

1. All edge weights are non-negative.
2. All edge weights are non-negative and obey the triangle inequality (meaning that for any three different nodes v_1, v_2, v_3, the inequality $w_{v_1,v_2} \leq w_{v_1,v_3} + w_{v_3,v_2}$ holds).
3. All edge weights are 1.

[11 marks]

c) Using a SAT solver

Assume someone gives you solver for the satisfiability problem that is very fast to find an assignment of truth values to variables that satisfies as many clauses simultaneously as possible, given that the instance contains up to 500 variables. How would that be useful when trying to solve different problems that are in NP, including NP-hard problems? [11 marks]

3. Approximations

a) Approximating the metric TSP

You will remember that we considered a 2-approximation for the metric TSP: you compute a minimum spanning tree, use its edges to reach all nodes (possibly repeating nodes), transforming this into a tour by taking shortcuts (i. e., avoiding previously visited nodes and going to the next unvisited node directly).

A friend claims that local search (using 2-OPT as neighbourhood) is a more useful algorithm for the TSP because it can deliver solutions that are less than twice as long as an optimal solution, different from the 2-approximation above. Also, your friend argues, local search can easily be applied to the general TSP and the Euclidean TSP, both not possible with the 2-approximation algorithm that you remember.

Discuss all three aspects of this claim (better solutions; applicable to the general TSP; applicable to the Euclidean TSP). [11 marks]

b) Approximating the knapsack problem

Remember the knapsack problem, where an input consists of n items with weights $w_1, w_2, \ldots, w_n \in \mathbb{N}$ and values $v_1, v_2, \ldots, v_n \in \mathbb{N}$ and a weight limit $W \in \mathbb{N}$, and the output is a selection of items $S \subseteq \{1, 2, \ldots, n\}$ that adheres to the weight limit, $\sum_{i \in S} w_i \leq W$, and maximises the overall value, $\sum_{i \in S} v_i$.

A friend suggests the following approximation algorithm. We create a new input with the same number of items, with equal weights and the same weight limit. The values of the new items are $\tilde{v}_i = \lfloor v_i / j \rfloor$, where $\lfloor \cdot \rfloor$ denotes rounding down to the nearest non-negative integer, and $j = \max\{v_1, v_2, \ldots, v_n\}/(11n)$ (where v_i is the value of the item in the original input). For this new input we apply a dynamic programming algorithm that computes an optimal solution, using the approach that computes for each $i \in \{1, 2, \ldots, n\}$ and each $v \in \left\{0, 1, \ldots, \sum_{i=1}^{n} \tilde{v}_i\right\}$ the smallest weight $w(i, v)$ to achieve exactly value v with a selection of only the first i items. Your friend can prove that this algorithm is an 1.1-approximation for the knapsack problem.

Does this algorithm work at all: will it always produce at least a solution that respects the weight limit? What is the runtime of this algorithm? [11 marks]

c) **Approximating MAXkSAT**

Consider the following problem that we will call MAXkSAT: an input consists of m clauses over n Boolean variables, each clause containing k different variables. Output is an assignment that satisfies as many clauses simultaneously as possible.

For the clause c_j, let $I_i^+ = \{i \mid x_i \text{ appears in } c_j\}$ be the set of indices of variables that appear in clause c_j in positive form, and let $I_j^+ = \{i \mid \overline{x_i} \text{ appears in } c_j\}$ be the set of indices of variables that appear in clause c_j in negative form.

Consider the following algorithm for MAXkSAT and analyse its runtime. Provide the runtime (using asymptotic notation, ideally $\Theta(\cdot)$) as a function of n, m, and k.

for all $i \in \{1, 2, \ldots, n\}$ **do**
    Set $a_i = 1/2$.
for all $i \in \{1, 2, \ldots, n\}$ **do**
    Set $a_i = 0$.
    for all $j \in \{1, 2, \ldots, m\}$ **do**
        Set $e_j = 1 - \left(\prod_{h \in I_j^+} (1 - a_h) \right) \times \prod_{h \in I_j^-} a_h$.
    Set $E_0 = \sum_{j=1}^{m} e_j$.
    Set $a_i = 1$.
    for all $j \in \{1, 2, \ldots, m\}$ **do**
        Set $e_j = 1 - \left(\prod_{h \in I_j^+} (1 - a_h) \right) \times \prod_{h \in I_j^-} a_h$.
    Set $E_1 = \sum_{j=1}^{m} e_j$.
    if $E_0 > E_1$ **then**
        Set $a_i = 0$.
Output $a_1, a_2, \ldots, a_n$

The algorithm is motivated by the following observations. If $a_i \in \{0, 1\}$ is the value of x_i then $c_j = 1 - \left(\prod_{i \in I_j^+} (1 - a_i) \right) \times \prod_{i \in I_j^-} a_i$: the clause c_j is satisfied (has the value 1), unless all variables that appear in positive form are set to 0 and all variables that appear in negative form are set to 1. If we set all $a_i = 1/2$ the expression $e_j = 1 - \left(\prod_{i \in I_j^+} (1 - a_i) \right) \times \prod_{i \in I_j^-} a_i$ expresses the probability that the clause c_j is satisfied if all variables independently are set to either 0 and 1 with

equal probability. Since all clauses contain exactly k different variables, we have $1 - \left(\prod_{i \in I_j^+} (1 - a_i) \right) \times \prod_{i \in I_j^-} a_i = 1 - 1/2^k$ for all j in the beginning when $a_i = 1/2$ for all i. This implies $\sum_{j=1}^{m} e_j = m \times \left(1 - 1/2^k\right)$ initially. Setting a_i to either 0 or 1, depending on which value makes $\sum_{j=1}^{m} e_j = m \times \left(1 - 1/2^k\right)$ larger, ensures that at the end the assignment $a_1, a_2, \ldots, a_n$ satisfies at least $m \times \left(1 - 1/2^k\right)$ clauses.

What is the approximation ratio this algorithm guarantees? Express it in the form $1 + x$ for some appropriate x. How does the approximation ratio change with k? For what values of k can you find a better algorithm? [11 marks]

END OF PAPER