CS4234: Optimisation Algorithms Tutorial 1

M(W)VC, (I)LP, Max-Clique, Graph-Coloring, PS1 Debrief

V1.6: Steven Halim *August 22, 2022*

MODAL ANSWER IS FOR OUR CLASS USAGE ONLY; NOT TO BE DISTRIBUTED IN PUBLIC

Preliminaries

As this is the first tutorial session, let's use the first few minutes for the tutor (Audrey (2 classes: T1+T3) and Steven (1 class: T2) for this S1 AY2022/23) to know slightly more about you.

Also to discuss if Group 02 should be canceled and its 5 members scattered to Group 01 or 03....; eventually Group 02 is retained and a few students from Group 01+03 are moved to Group 02 instead to make all three tutorial groups balanced.

Discussion Points

Q1: Can we solve Min-Vertex-Cover in polynomial time if all vertices in the input graph have degrees **not more than 2**? What if we want to solve Min-Weight-Vertex-Cover variant on that graph type?

Short Answer: A vertex with degree 0 implies an isolated vertex (guaranteed to be outside VC so this case can be safely ignored on either unweighted or weighted version), a vertex with degree 1 implies that it is a leaf vertex connected to another vertex, a vertex with degree 2 implies it is part of either a linked list or a cycle (remarks: year after year, a few students failed to recognize the possibility that cycle graph C_n — see `https://en.wikipedia.org/wiki/Cycle_graph` — is also a valid input for the given constraints).

For unweighted version, we can find each CC (which is a line/linked list or cycle per CC) using $O(N)$ DFS/BFS, then report floor(CC_size/2) to cover a CC that is a line/linked list or ceil(CC_size/2) to cover a CC that is a cycle. Then repeat for all CCs.

However, the strategy above will not work on weighted version and we need a slightly modified $O(N)$ DP (on Tree) instead (the slight modification is 'trivial': instead of 1/0 for taking/not-taking a vertex, we pay $w(v)$/0 for taking or not taking vertex v with weight $w(v)$, see CP4 Book 2 page 449. Again, isolated vertex is never taken and a linked list/line is a tree also. For a cycle: we need to do something to break the cycle by removing an arbitrary edge (u, v), try putting u into vertex cover, solve the resulting line optimally, try putting v into vertex cover, solve the resulting line optimally, and report the best answer out of these two possibilities).

Q2: In lecture, we have seen a Dynamic Programming (DP) solution to solve the Min-Vertex-Cover on a (Binary) Tree. How about the following Greedy Algorithm (solution for CLRS Ex 35.1-4): First, take any leaf in the tree, then add its parent to the (minimal) vertex cover, then delete the leaf and parent and all associated edges. Repeat this process until there is no vertex remain in the tree. Is this a correct Greedy Algorithm? Hint: Check `https://visualgo.net/en/mvc`, 'MVC on Tree', the 'Greedy MVC on Tree' option. Can it be used on the Min-Weight-Vertex-Cover variant?

Short Answer: Yes (for MVC), this greedy algorithm has a proof of correctness (for MVC).

Optimal substructure: We can argue that the behavior of this $O(N)$ Greedy Algorithm precisely does a Vertex Cover (cover and delete the edges selected, then we have a smaller subtree, repeat until done).

Exchange argument for the greedy-choice property: "We can transform an optimal MVC that contains some leaf vertices into MVC (of same size) that does not contain any leaf".

Again, it will not work on the weighted version (MWVC), so we just use the $O(N)$ DP (on Tree) again as shown earlier in Q1 instead.

Q3: A carpenter makes tables and chairs. Each table can be sold for a profit of 25 SGD and each chair for a profit of 11 SGD. The carpenter can afford to spend up to 40 hours per week working and takes 5 hours to make a table and 3 hours to make a chair. The owner requires that the carpenter makes at least 3 times as many chairs as tables (so that he can package it as dining table set). Tables take up 4 times as much storage space as chairs and there is room for at most 10 tables each week.

Formulate this problem as a Linear Programming problem (write in standard form) and solve it (graphically, with Excel, with lp_solve, or with a (Simplex? or Brute Force?) program :O).

Now if the solution is a floating point numbers, please round them so that we have an Integer solution. Are you sure your Integer solution is the optimal one?

Short Answer: If you work the LP correctly, you will have Tables (T): 2.85, Chairs (C): 8.57, profit: 165.71. But obviously we can't produce 2.85 Tables and 8.57 Chairs so the LP below is lacking one more constraint: The integer constraint on T and C (and afterwards, we will have Integer (Linear) Program (ILP) instead).

$$
\begin{array}{rcl}
\max(25*T+11*C) & & \textit{such that:} \\
5*T+3*C & \leq 40 & \textit{working time constraint} \\
3*T-C & \leq 0 & \textit{dining table set constraint} \\
4*T+C & \leq 40 & \textit{storage capacity constraint} \\
T & \geq 0 & \textit{non-negative constraint} \\
C & \geq 0 & \textit{non-negative constraint}
\end{array}
$$

If both T and C are rounded down, we have T: 2, C: 8, profit: 138, surely feasible.
But if we round down T: 2 and round up C: 9, we have a better profit 149, still feasible.
However the optimal answer is T: 2 and C: 10, profit: 160 (try Excel Simplex tool ***with additional Integer constraint*** *or even just brute force small T and C for this toy problem), this optimal answer can't be found by just rounding (to either direction) as the integrality gap is somewhat higher in this problem than in MWVC problem discussed in Lecture 02, highlighting the (NP)-hardness of truly solving ILP...*

Q4: The year is 2022 (After IOI 2020+2021 (and also IOI 2022) are over). Steven returned to his VisuAlgo project and want to add a few new features involving various NP-hard optimization problems (assume that nobody has proven whether $P = NP$ (or not) by then). You are one of his Final Year Project (FYP) student who is assigned to do Max-Clique visualization (which has not appeared in `https://visualgo.net/` yet...). If you forget, the decision version of Clique is defined as follows: "Given a graph $G = (V, E)$ and an integer k, is there a subset $C \subseteq V$ of size k such that C is a clique in G? The Max-Clique optimization variant seeks for the subset C with the largest possible k in G. Here are Steven's requirements:

1. VisuAlgo user *must* be able to draw his/her own graph input, as per VisuAlgo standard.

2. Steven dislikes graphs that are drawn with edge crossings, so he has put on a check in his "Draw Graph" feature so that only graphs without edge crossing are allowed in VisuAlgo.
3. VisuAlgo must be very responsive, i.e., upon clicking "Show Max-Clique" button, a Java?Script (the year is 2022, remember) routine has to quickly compute the answer in not more than 1 second. Longer than that, Steven says that the visitors will press Alt+F4 instead.

What is the *best* algorithm that will you implement as Java?Script routine of your MAX-CLIQUE visualization in VisuAlgo? What is the time complexity? Is your algorithm seeks for an approximate solution or an optimal solution? What are the rough limit of n and m, the number of vertices and edges, that you will set in your visualization?

Answer: The key to solve this problem is to realize that requirement no 2 is essentially restricting the input as a planar graph. In Section 4.1.2 of the extra reading material 'Understanding Unsolvable Problem' of Lecture 1 written by ex student Jonathan Irvin Gunawan, we are told that there will not be a MAX-CLIQUE *of more than K_4 in planar graph as presence of K_5 will imply that the graph is not planar (either via Graph Coloring 4 color theorem or if you are into graph theory, search 'Kuratowski's Theorem', e.g.,* `https://en.wikipedia.org/wiki/Planar_graph`*). Therefore we can just do an $O(n^4)$ brute force search. Try all quartet of 4 vertices in n and see if all ${}_4C_2 = 6$ edges between those 4 vertices are present in G. If we cannot find clique of size 4, we try triples of 3 vertices. Otherwise, we highlight any edge for K_2. If G has no edge at all, we say that the* MAX-CLIQUE *is any single vertex. (And if G has no vertex at all, the answer is 0).*

But we can improve further. We try $O(m^2)$ brute forcing pair of edges in G instead and test whether the 4 (or 3, if one end point overlaps) vertices involved in the pair of edges form a clique of size 4 (or 3, if one end point overlaps). In planar graph, $m = O(n)$, to be precise $m = 3n - 6$ (for $n > 2$) — continue your Google searches on this special graph, so this algorithm is just $O(n^2)$.

We can thus easily allow users to draw a large graph in VisuAlgo (e.g., a few hundred vertices) and still able to run in under one second... The limiting factor is now the user him/herself. Not many normal human will want to draw manual test cases that large, so maybe a button that says 'generate random planar graph of n vertices' is needed...

Q5: Follow up of Q4 above. All Steven's requirements are identical, just that you are another FYP student who is assigned to do GRAPH-COLORING visualization (which also has not appeared in `https://visualgo.net/` yet...). The decision version of GRAPH-COLORING is defined as follows: "Given a graph $G = (V, E)$ and an integer k, can we assign a color (an integer in $[1..k]$) to each vertex such that no two adjacent vertices (that share an edge) are of the same color? The GRAPH-COLORING optimization variant seeks for coloring with the smallest possible k.

Answer: For planar graph, the 4 color theorem says that we can color the graph with at most 4 color. A simple backtracking algorithm that tries all color per vertex has a worst case time complexity $O(4^n)$ but early pruning for any infeasible solution may help avoid this worst case behavior. We probably can do faster with DP with subset (not thought carefully yet) but as of now, there is no known specific planar graph property that can be used to solve GRAPH-COLORING *in polynomial time.*

So the rough limit is probably not in order of hundreds vertices, but low tens of vertices.

PS1 Debrief: To be summarized briefly: What are the underlying problems+(alternative) solutions of PS1 Task A/B/C/D/E? What are the required algorithms for each task?

Short Answers have been presented at the back of Lecture 02, review the recording.

CS4234: Optimisation Algorithms Tutorial 2

MSC, Steiner-Tree, M(W)FES, 2-CNF-SAT, PS2

V1.7: A/Prof Steven Halim *September 04, 2023*

Discussion Points

Q1: We discussed Integer Linear Programming (ILP) in Lecture 02. Now express the proven NP-hard optimization problem: Min-Set-Cover problem that we discussed in Lecture 03a as an ILP!

Answer: The key discussion point is to help students write their own ILP for certain relevant problems. Min or Max? What are the decision variables? What are the objective functions? Are the decision variables have to be integer? (put that down immediately). Then ask: What are the (easiest) way to express the non trivial constraints? Are all constraints (and objective function) linear?

$$\min \sum_{i \in S} x_i \quad \text{where} \quad \text{// } x_i = 1/0 = \text{use/not-use this subset } i\text{, min total used}$$

$$x_i \in \{0, 1\} \quad \text{// this is the standard integer/Boolean constraint}$$

$$\forall j \in X, \sum_{i \in S \text{ where } j \in S_i} x_i \geq 1 \quad \text{// for every element in } X\text{, at least one selected set must cover it}$$

Q2: Now express what we know as a P problem: Min-Spanning-Tree problem as an ILP! Will you solve MST problem that way instead of using what you already know from earlier modules?

Answer: See below for the ILP formulation. The no-cycle in any subset of vertices constraint is short to express but exponential in size in practice (that complex constraints are really needed to prevent cycles)... So, I don't think we will go that route. MST problem already has efficient $O(E \log V)$ algorithm(s) in Prim's or Kruskal's (and there are a few others) that are simpler than the ILP route.

$$\min \sum_{ij \in E} x_{ij} \times w(i,j) \quad \text{where} \quad \text{// } x_{ij} = 1/0 = \text{use/not-use edge } (i, j) \in E$$

$$\forall ij \in E, x_{ij} \in \{0, 1\} \quad \text{// standard}$$

$$\sum_{ij \in E} x_{ij} = n - 1 \quad \text{// choose only } n\text{-1 edges (convert this to } \leq)$$

$$\forall S \subset V, S \neq \emptyset, \sum_{ij \in E \text{ where } i \in S \text{ and } j \in S} x_{ij} \leq |\ S\ | - 1 \quad \text{// any non } \emptyset \text{ subset of } k \text{ vertices have } \leq k\text{-1 edges}$$

Q3: In Lecture03b, we discussed the EUCLIDEAN-STEINER-TREE problem. Now let's spend some time discussing (and maybe proving) some of these properties that were only discussed briefly in Lecture 03b. The TA will draw a few (e.g., $n = 7$) 'random' points on Euclidean plane (the whiteboard) and we will try to apply these known properties to help us derive an optimal (or at least (visually) good enough) Euclidean Steiner Tree:

- Each Steiner point in an optimal solution has degree 3.
- The three lines entering a Steiner point form 120 degree angles, in an optimal solution.
- An optimal solution has at most $n - 2$ Steiner points.

Answer:
There is a key Lemma for subsequent statements (which we will not prove but shown instead): No edges in Steiner tree can meet at an angle of less than 120 degrees. We can always split these two edges involving three vertices (that meet at angle less than 120 degrees), put a new Steiner point in the middle (chosen carefully with some 'Toricelli point' rule, see `https://en.wikipedia.org/wiki/Fermat_point` *), and reconnect them. This can be demonstrated live on the whiteboard.*

Each Steiner point in an optimal solution has degree 3 → Because the degree of a Steiner point S_j, i.e., $d(S_j)$ cannot be 1 (what is the point of using a Steiner point in this case?) and cannot be 2 (that Steiner point S_j can be removed and we have a simpler straight-line edge), so $d(S_j) \geq 3$. A circle has 360 degrees. The no-more than 120 degree angles Lemma above force each Steiner point S_j to have degree $d(S_j) \leq 3$. Combining these two facts together, $d(S_j) = 3$.

The three lines entering a Steiner point form 120 degree angles, in an optimal solution → Obvious, from 360 degrees divided by 3 sectors = 120 degrees each.

An optimal solution has at most $n - 2$ Steiner points →. Let say our Steiner tree has n required points + k additional Steiner points. A tree with $n + k$ vertices has $n + k - 1$ edges. As each Steiner point has exactly degree 3 (see above) and each required points has at least *degree 1 (it has to be connected somehow), the number of edges must be* at least *$(n * 1 + k * 3)/2$ (divide by 2 to avoid overcount). Combining the two statements, we have $n + k - 1 \geq (n + 3k)/2$ or $2n + 2k - 2 \geq n + 3k$ or $n - 2 \geq k$ or $k \leq n - 2$.*

Q4: Prof Halim needs to continually increase the number of (NP-)hard problems that has been exposed to CS4234 students so far (to have a more interesting Midterm Test and Final Assessment). So let's study this yet other problem MIN-WEIGHT-FEEDBACK-EDGE-SET (MWFES) and MIN-FEEDBACK-EDGE-SET (MFES) for the unweighted version.

You are given a *directed weighted* graph $G = (V, E)$ with (positive) weights $w : E \to \mathbb{R}$ and $|E| \geq 1$. Your goal is to delete the some edges to produce an acyclic graph with the maximum remaining edge weights. That is:

> Find a minimum weight set of edges F such that $G = (V, E \setminus F)$ is acyclic.

Part 0. Draw an (small) example graph and explain this problem to your tutorial group.

Short Answer: Just draw a random small instance by yourself.

Part 1. If G is an *undirected weighted* graph, give an efficient algorithm to solve this problem *optimally*.

Short Answer: First, find a maximum *spanning tree T (spanning forest if the input graph is disconnected).*

(Simple: Reverse-sorted Kruskal's or Prim's with Max PQ). Then let F be all the edges not *in the tree T. Why does this work? (there is a simple proof).*

Part 2. In the case of directed graphs, the MFES (or MWFES) problem is NP-hard (for now, just assume that it is really NP-hard and later in (the optional) Part 4, we will show you the details). Now, consider the following (heuristic) algorithm: Given a directed graph $G = (V, E)$ and weights 1 (or w) for MFES (or MWFES) version, respectively:

1. Let the vertices be $V = v_1, v_2, \ldots, v_n$.
2. Build graph $G_f = (V, E_f)$ containing only forward edges. That is, $E_f = (v_i, v_j)$ where $i < j$.
3. Build graph $G_b = (V, E_b)$ containing only backward edges. That is, $E_b = (v_j, v_i)$ where $i < j$.
4. Return the graph G_f or G_b containing more edges (or higher total weighted edges for the weighted MWFES version) as the acyclic graph (and the other non-selected edges as the removed edges F).

Show that:

- Both G_f and G_b are valid solutions to the MFES/MWFES problem.
- Show that the algorithm is *not* a good approximation algorithm for MFES/MWFES.

Short Answer: To show that they are both valid solutions, you need to show that the resulting graph is acyclic, which follows from the construction.

To see that the algorithm is not a good approximation algorithm for MFES, construct a graph so that both G_f and G_b have $|E|/2$ edges, but the original graph G is acyclic (i.e., $F = \emptyset$ in the optimal solution — do not delete anything). For example: G with $V = 2$ vertices and $E = 2$ edges: $0 \to 1$ and $3 \to 2$. The algorithm will delete 1 of these 2 edges while the optimal answer is 0 (delete nothing) and 1/0 is undefined.

Part 3. Now consider the *complementary* (the dual) problem:

Find a maximum weight subgraph $G' \subset G$ that is acyclic.

Notice that *weight* here refers to the sum of the edge weights. Now show that the (heuristic) algorithm above is actually a 2-approximation algorithm.

Short Answer: The key here is to observe that $OPT \leq \sum_{e \in E} w(e)$
($OPT == \sum_{e \in E} w(e)$ if G itself is already acyclic, otherwise $OPT < \sum_{e \in E} w(e)$),

We see that $weight(G_f) + weight(G_b) = \sum_{e \in E} w(e)$, so either G_f or G_b contains at least half the total weight and we will return one of them (the larger one).

Thus, the returned answer is at least OPT/2.

Now, which problem in PS2 S1 AY2023/24 is exactly this one?
It is problem C, /exitsinexcess.

Part 4 (optional to save time, just read the modal answer). Assume G is a *directed* graph. Show that now it is NP-hard to solve by reduction from the VERTEX-COVER problem. That is, given an instance of VERTEX-COVER problem, show how to reduce it into an instance of FEEDBACK-EDGE-SET problem.

Answer:
Solution ideas. Given an instance $G = (V, E)$ of vertex cover (i.e., an undirected graph), we build a new directed graph in two steps. First, for each (undirected) edges $e = (u, v)$ in E, we add two directed edges $e_1 = (u, v)$ and $e_2 = (v, u)$ to E_1. Now we have the graph $G' = (V, E_1)$ that is directed. Next, for every node u in V, we create two nodes u_{in} and u_{out} and add them to V'. Now we build edges set E_2 as follows: for every edge (u, v) in E_1 add edge (u_{out}, v_{in}) to E_2. Finally, for every node u, add edge (u_{in}, u_{out}) to E_2. Notice that in the resulting graph $G'' = (V', E_2)$, node u_{in} has only one outgoing edge (to u_{out})) and node u_{out} has only one incoming edge from u_{in}. Give all edges weight 1. Draw a picture.

If a solution removes the edge (u_{in}, u_{out}), then add edge u to the vertex cover. To show that the reduction yields the desired result, we need the following:

- Show that this constructs a vertex cover (i.e., if there is an uncovered edge, it would have formed a cycle in the graph $G = (V', E_2)$).
- Argue that the cost of the resulting vertex cover is $\leq$ the number of edges removed in $G = (V', E_2)$.
- Given the VERTEX-COVER, construct an edge set F of the exact same size that solves the acyclic graph problem by adding (u_{in}, u_{out}) to F for every vertex u in the vertex cover (and vice versa).

We have shown how to reduce an NP-hard VERTEX-COVER problem to FEEDBACK-EDGE-SET problem, establishing the NP-hardness of the FEEDBACK-EDGE-SET problem too (and later the NP-hardness of the MIN-(WEIGHT-)FEEDBACK-EDGE-SET too.

Q5: The `3-CNF-SAT` problem is one of the classic baseline problem to show the NP-hardness of a few other classic problems, e.g. `3-CNF-SAT` $\leq_p$ `Clique` $\leq_p$ `Vertex-Cover` (we briefly shown `Clique` $\leq_p$ `Vertex-Cover` in Lecture01). One of the PS2 'new' question (it appeared in 2021) involves a special case `2-CNF-SAT` of this `3-CNF-SAT`.

`2-CNF-SAT` problem is defined as follows: Given a conjunction of disjunctions ("and of ors") where each disjunction ("the or operation") has three (2) arguments that may be variables or the negation of variables, find a truth (T/F) assignment to these variables that makes the formula true. For example, $\phi = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$ is satisfiable.

Part 1. Find a truth assignment to x_1, x_2 and x_3 so that the given ϕ is satisfiable.

Short Answer: $x_1 = x_3 = T$, $x_2 = F$.

Part 2. Which problem in PS2 is actually `2-CNF-SAT` problem?

Short Answer: It is problem D, /buriedtreasure2. Sample test case two is can be reduced into $\phi_2 = (x_1 \vee x_2) \wedge (x_2 \vee \neg x_1) \wedge (\neg x_1 \vee \neg x_2)$ *that is satisfiable when* $x_1 = F$ *(a trap) and* $x_2 = T$ *(a treasure).*

If you do not know this polynomial solution for `2-CNF-SAT`, you may have to try $O(2^m \times n)$ *complete search solution (for each location, test if it is either a Treasure or a Trap (2 choices each, so* 2^m *overall), and see if all* n *treasure maps are reasonable). This is hopelessly TLE as both* m *and* n *can be up to* 10^5.

Part 3. What is the high-level idea to solve this special case of an NP-complete decision problem?

Short Answer: Each clause in a 2-CNF formula $(a \vee b)$ can be written as two directed implications: $\neg a \rightarrow b$ and $\neg b \rightarrow a$. We can build an implication graph from the given formula ϕ. Then, ϕ is satisfiable if and only if there is no variable that belongs to the same Strongly Connected Component (SCC) as its negation (there is a longer construction algorithm for this case). Checking SCC can be done using Kosaraju's or Tarjan's SCC finding algorithm, that runs in $O(V + E) = O(m + n)$ in this case.

The short explanation on why this works is that if we have a variable, say x_i and its negation $\neg x_i$ in the same SCC of the implication graph, e.g., $(x_i \vee x_i) \wedge (\neg x_i \vee \neg x_i)$, then whether we set x_i to True or to False, we will always end up with a contradiction. PS: `https://visualgo.net/en/dfsbfs` *has a 2-SAT Checker (but it is currently buggy - as of 23 Aug 2023 :(..., hopefully Prof Halim or his FYP/USR student can quickly isolate the bug and kill it).*

CS4234: Optimisation Algorithms Tutorial 3

TSP, Max-Independent-Set, PS3

V1.7: A/Prof Steven Halim *September 11, 2023*

MODAL ANSWER IS FOR OUR CLASS USAGE ONLY; NOT TO BE DISTRIBUTED IN PUBLIC

Discussion Points

Q1: In Lecture 04, Prof Halim did not show any approximation algorithm for the G-NR-TSP variant of TSP (that is, the general, non-metric version and without repeated vertex). He says that 'it is (NP-)hard even to approximate'. Why?

For details: See Section 35.2.2 of CLRS (page 1115-1116 in the 3rd edition or page 1113-114 in the 4th edition) if you have either edition of the book.

In short: If $P \neq NP$, then for any constant $\rho \geq 1$, there is no *polynomial-time approximation algorithm with approximation ratio ρ for the* G-NR-TSP *variant of TSP. The proof by contradiction cleverly uses the 'hypothetical approximation algorithm A' that runs in polynomial-time for* G-NR-TSP*. Let assumes such an algorithm A exists. Then, we can use A to polynomially solve* Hamiltonian-Cycle *problem on graph $G = (V, E)$ by placing edge weights strategically: 1 for edges in E and $\rho \times |V| + 1$ otherwise. With such construction, we can decide if G has a hamiltonian cycle (total cost is $|V|$) or not (uses at least one of the costly edge with cost $\rho \times |V| + 1$; even if the other $|V| - 1$ other edges cost 1 unit, the total cost is at least $\rho \times |V|$)... If approximation algorithm A exists, A can be used to decide if G has a hamiltonian cycle (A will give a tour of no more than $\rho \times |V|$) or not (A will give a tour of cost $> \rho \times |V|$). Contradiction, this cannot happen if $P \neq NP$. So the conclusion is algorithm A cannot exists if $P \neq NP$.*

Notice that on Metric version (e.g., the one in `https://visualgo.net/en/tsp`*), we cannot manipulate edge weights like this.*

Q2: In class, we formulated the TSP (4 variants) in terms of a set of points V and a distance function d that gives the distance between any two points in V (thus, a complete graph with V^2 edges that surely has $(V-1)!$ Hamiltonian Cycles). What if the input to the problem is a graph $G = (V, E)$ with E weighted edges and $0 \leq |E| \leq V * (V-1)/2$?

Define a version of TSP for graphs, and explain whether or not it remains approximate-able using the techniques discussed in class. (Consider these: What if there is no Hamiltonian Cycle in the input to begin with? What if the input graph is actually disconnected?)

Answer: In the general or graph versions of TSP (including the graph version of General-Steiner-Tree *last week), some edges may not exist if we are not given the complete graph. We can cope with some cases if we first calculate the metric completion (thus we get a complete graph again, in metric space), then use a TSP algorithm, e.g., the 2-approximation or the 1.5-approximation (Christofides's) algorithm, and finally reconstruct a tour of the same (or lesser) cost in the original graph (but the tour may contain repeats).*

In general, if you have a graph, it is often (but not always) useful to think about the metric completion, because then you can imagine vertices connected by a distance metric that satisfies the triangle inequality.

But beware:
1. If the input graph is disconnected, we cannot solve the TSP even with metric completion...
2. (Even if the input graph is connected and we have used the metric completion): If we are asked to deal with the M-NR variant, we may have a problem because if the edges used are 'virtual edges' and when we reconstruct the paths in the original graph, we have to repeat (at least one) vertex.

This is one of the reason why TSP input is almost always a complete graph.

Q3: Think about the *Euclidean* TSP where each point has (x, y)-coordinates to identify it (for simplicity, let's assume that all x-coordinates of the n points are different). Imagine that Prof Halim only wants "cycles" that proceed in one direction, e.g., left-to-right. For example, a legal output is a cycle $(v_1, v_2, v_3, v_4, v_1)$ where for each $i < n$, we know that $v_i.x < v_{i+1}.x$. (Only in the last step of the cycle, going back from v_n to v_1, we are allowed to go to the left.) Is there an efficient algorithm to solve this *non-standard* version of TSP problem? Next, give an example where the cycle that is found in this case is very bad compared to the optimal TSP cycle.

Short Answer: This is not a variant of TSP called Bitonic TSP (if you are interested, you can read CP4 Book 2 page 443-444 for this Bitonic TSP variant), but actually far simpler. The solution: $\{v_1, v_2, ..., v_n\}$ can be found by sorting the points according to their x coordinates using any $O(n \log n)$ sorting algorithm.

However, as there will only be one such valid cycle, we can give a sawtooth like test case that will make this algorithm runs very badly compared if we don't have the left-to-right restriction, i.e., the original TSP.

Q4: Prof Halim needs to continually increase the number of (NP-)hard problems that has been exposed to CS4234 students so far (to have more interesting Midterm Test and Final Assessment). So let's study this yet other problem MAX-INDEPENDENT-SET (MIS). It is very similar to MVC and defined as follows: Given a graph $G = (V, E)$, pick the maximum-size set $I \subset V$ so that no two vertices in I share an edge.

You are told that MIS is also an NP-hard optimization problem (proof omitted, but you can reduce NP-hard VC to IS easily), but here you are given a network that is arranged as a grid, as in Figure 1:

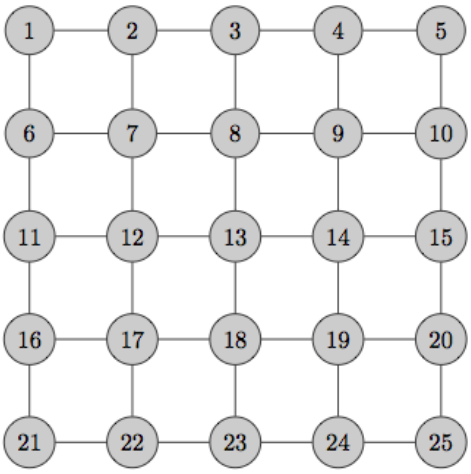

Figure 1: A Grid Graph of size 5×5.

Part 1. The grid has n vertices, and each of the vertices (except for those on the edges) has four neighbors. The goal of this question is to develop an algorithm for finding a maximum-sized MAX-INDEPENDENT-SET (MIS) for this graph (or an approximation). What is the MIS (and its size) of Figure 1 above?

Short Answer: Sub-optimal: 12, e.g., {2, 6, ..., 4, 8, 12, 16, ..., 10, 14, 18, 22, ..., 20, 24}.
Optimal: 13, e.g., {1, ..., 3, 7, 11, ..., 5, 9, 13, 17, 21, ..., 15, 19, 23, ..., 25}.
Notice that a maximal *independent set (the Sub-optimal answer above) is one where no new vertices can be added, i.e., all vertices are neighbors of some vertex in the current maximal independent set. Note that the term* maximal *is easily confused with the term* maximum *(the Optimal answer above).*

For an extra challenge: Tutor can remove a few random vertices from the grid, so that the grid has hole(s).

Part 2. Consider the following greedy algorithm for graph $G = (V, E)$:

- Set $I = \emptyset$;
- Repeat until V is empty:
  - Choose any arbitrary vertex $u \in V$,
  - Add u to I,
  - Delete u all the neighbors of $u \in V$.

Now argue that this algorithm is a 2.5-approximation of optimal on grid graph like in Figure 1 above. (But first, show that it is a correct algorithm that produces an independent set!)

Short Answer: First, it surely produces an Independent Set as after u is added into I, all neighbors of $u \in V$ are deleted, so we will never have two vertices u and v inside I where there is an edge (u, v) in the (grid) graph.

Notice that in any graph $G = (V, E)$, the OPT of this MIS problem is always not more than n, i.e., $OPT \leq n$ (or $n \geq OPT$). All you need to do next is argue that this procedure adds at least $n/5$ vertices to I, i.e., that the loop continues at least $n/5$ times no matter what order of vertices that you add to I as there are only at most 5 vertices removed at every step (vertex added to I and at most $\Delta = 4$ of its neighboring vertices).
So $|I| \geq n/5$.
So $|I| \geq OPT/5$ (recall that MIS is a maximizing problem).

Longer Proof: We upper bound the number of vertices in $V \setminus I$ (the deleted vertices) as follows. A vertex u is in $V \setminus I$ because it is removed as a neighbor of some vertex $v \in I$ when the Greedy algorithm adds v into I. Count +1 deleted vertex u due to v. A vertex $v \in I$ can be counted at most Δ times since it has at most Δ neighbors. Hence we have that $|V \setminus I| \leq \Delta \times |I|$.

Since every vertex is either in I or $V \setminus I$ we have $|I| + |V \setminus I| = n$.
Therefore $|I| + \Delta \times |I| \geq n$.
Therefore $(1 + \Delta) \times |I| \geq n$.
Finally $|I| \geq n/(1 + \Delta)$.
$\Delta = 4$ in a grid graph, hence $|I| \geq n/(1 + 4) = n/5$, so it is a 5-Approximation Algorithm (as above).

What we have really shows in that if the graph has degree Δ, then any maximal independent set is a $(\Delta + 1)$-approximation. In practice, many real-world systems rely on Maximal Independent Sets (much easier to find) instead of really Maximum Independent Sets (NP-hard).

Note: A stronger bound can be found for the given grid graph by noticing that $OPT \leq \lceil n/2 \rceil$ on such a

grid graph of size $n \times n$ with no hole. To simplify the analysis a bit, we can assume that n is even, so $OPT \leq n/2$ and $n \geq 2 * OPT$.
So $|I| \geq n/5$ (still the same argument as above).
So $|I| \geq 2 * OPT/5$ (a 2.5-approximation).

Note 2: There was discussions during tutorial of previous AYs that this 2.5-approximation bound is not really tight and can be lowered further because we don't always remove 5 vertices at every step (we only do this at the early steps involving non corner/side vertices), but the potentially tighter bound is left as optional exercise for now...

Note 3: All these approximation algorithms are actually 'useless', as MIS on grid graph can actually be solved in polynomial time once you know the correct algorithm (in the next few weeks).

Part 3. (Optional to save time, just read the modal answer): Write down MIS as an ILP!

$$\begin{aligned} \max \sum_{i=1}^{n} x_i \quad & \text{where} \quad && // \ x_i = 1/0 = \text{use/not-use this vertex } i\text{, maximize size} \\ \forall i \in V, x_i \in \quad & \{0,1\} && // \text{ this is the standard integer/Boolean constraint} \\ \forall i,j \in V, x_i + x_j \leq \quad & 1 && // \text{ this is the Independent Set constraint} \end{aligned}$$

We add vertex i into to the Independent Set if $x_1 = 1$. Observe that this yields an Independent Set (i.e., no two neighbors are both in the Independent Set), and that it is optimal as we maximizes the cardinality.

Q5: One of the PS3 problems involves another new NP-hard problem that has not been discussed earlier (in lecture and/or previous tutorial). Which PS3 problem is it? What is the underlying NP-complete decision problem? What is the general idea to tackle this problem?

Short Answer: It is problem A (/sumsets). This is an NP-complete problem Subset-Sum*, i.e., is there a subset of integers in a set of (distinct) integers S of size N that sums to a target integer k. /sumsets is a 'special-case' of this NP-hard optimization problem as it asks for the largest d (one of the integers in S) such that a subset of* exactly 3 *integers $(a + b + c)$ from S sums to d. Subset-Sum have two known special cases: 2-SUM, and 3-SUM (this one). See help, e.g.,* `https://www.geeksforgeeks.org/find-a-triplet-that-sum-to-a-given-value/` *for inspiration.*

CS4234: Optimisation Algorithms Tutorial 4

Max-Flow, Min-Cut, Ford-Fulkerson Analysis

V1.7: A/Prof Steven Halim *September 18, 2023*

MODAL ANSWER IS FOR OUR CLASS USAGE ONLY; NOT TO BE DISTRIBUTED IN PUBLIC

Preliminaries

Before starting T04, we will run a comparison between the two similar Ford-Fulkerson methods that use 'shortest-augmenting-path-first' strategy: Edmonds-Karp vs Dinic's algorithm using `https://visualgo.net/en/maxflow` on a small random flow graph.

Short Answer: This is done on the spot to give students one more round of discussion about Edmonds-Karp vs Dinic's. Both uses BFS (unweighted SSSP) to identify shortest (in terms of numbers of edges used) augmenting paths first.

Edmonds-Karp uses $O(m)$ BFS on the entire residual graph per iteration. There are at most $O(mn)$ iterations (not proven), so Edmonds-Karp runs in $O(m^2 \times n)$.

Dinic's use $O(m)$ BFS to identify level graphs (subgraphs/smaller graphs) starting from the shortest level graph first. There are at most $O(n)$ level graphs (easily shown), and even if we exhaust all flows in a level graph in a naive $O(nm)$ time, Dinic's runs in $O(n^2 \times m)$ time overall.

And since $m > n$ in 99% of typical flow graphs, Dinic's is considered a better algorithm than Edmonds-Karp and should be the default max flow algorithm used in this class.

FYI: There are other (more complicated) Max Flow algorithms beyond the scope of this module, see `https://en.wikipedia.org/wiki/Maximum_flow_problem#Algorithms` ...

PS: Read `https://www.wisdom.weizmann.ac.il/~oded/even-dini.html` as supplementary reading, for fun :).

Discussion Points

Q1: Write Max-Flow as a Linear Program. Again, are you going to solve Max-Flow that way?

Short Answer: One possible LP that is derived really from the basic definition of Max-Flow is shown below. It will produce integer answers if the capacities are all integers (we do not have to also constraint the flow of each edge to be integers). We will likely not go through this route and use the more established graph-based Max-Flow algorithms. PS: Colleagues/students from other departments may disagree, i.e., `https://nusmods.com/courses/MA3252/linear-and-network-optimisation` or `https://nusmods.com/courses/IE4210/operations-research-ii`.

$$\max \sum_{j:(s,j)\in E} f_{sj} \quad \text{where} \quad \text{// maximize flow that goes out from source vertex } s$$
$$\forall i,j \in E, f_{ij} \geq \quad 0 \quad \text{// } f_{ij} \text{ is the (integer) flow in edge } (i,j)$$
$$\forall i,j \in E, f_{ij} \leq \quad c_{ij} \quad \text{// capacity constraints}$$
$$\forall j \in V \backslash \{s,t\}, \sum_{i:(i,j)\in E} f_{ij} = \sum_{k:(j,k)\in E} f_{jk} \quad \text{// flow conservation constraints}$$

Q2: Show how to use (standard) Ford-Fulkerson algorithm to find a maximum sized matching on *Bipartite* Graph, or formally known as the Max-Cardinality-Bipartite-Matching (MCBM) that will be properly discussed on Lecture 6 (please read ahead). Prove that the result is a matching, and that it is the maximum-sized matching. Analyze the running time of your algorithm. Note that we have seen the topic of Graph Matching briefly in Lecture 1 (Deterministic Vertex Cover-2), Lecture 4 (part of Christofides's algorithm), and will properly come back to this topic on Lecture 6+7.

Answer: Let the vertices in Bipartite Graph be partitioned into set A and set B. Add a source vertex s and connect s to all vertices in set A with capacity 1. Add a sink vertex t and connect all vertices in set B to t, also with capacity 1. Give all the input bipartite graph edges from A to B capacity 1. Direct all edges from s to A to B to t (the directed edges are important, don't use bidirectional edges to avoid silly subtle bugs). Run any Max Flow algorithm. Every edge from A to B with flow 1 is in the matching, all the others are not.

Why does this work? First, since each vertex in A has one incoming edge with capacity 1, each vertex in A can only have one outgoing edge with capacity 1 (flow conservation). The same is true for each vertex in B, and hence we get a matching. For short Proof by Contradiction on maximality, assume there is a larger matching M. Then we can use that to construct a larger flow f, which is a contradiction that we have found a max flow.

*The (rough) running time if we use standard Ford-Fulkerson is *just* $O(m^2)$ as $U = 1$ in this unit-capacity flow graph. Faster runtime of $O(\sqrt{n} \times m)$ is possible if we use Dinic's algorithm (essentially Hopcroft-Karp algorithm, to be revisited in Graph Matching lectures).*

This reduction from MCBM *to* Max-Flow *can only be done on Bipartite Graph. It does not work on general graph. It is also shown in* `https://visualgo.net/en/maxflow`*, select 'Modeling'.*

Q3a: Assume that you have an (un)directed graph $G = (V, E)$ with a source vertex s and a target vertex t (the graph is *unweighted*). Give an algorithm that finds the maximum number of *edge-disjoint paths* from s to t. Two paths P_1 and P_2 are called edge-disjoint if they do not share any edges—but they may share a vertex. Is this a Max Flow problem? What is the running time of your algorithm?

Short Answer: Just run any Max Flow algorithm on that input graph verbatim (unweighted graph can be seen as a flow graph with unit or weight 1 capacity edges, and even if the input graph is weighted, we can edit all edge weights to unit or weight 1 capacity edges too; also directed graphs can be made into undirected graphs too by listing the bidirectional edges twice). The resulting max flow is simply the number of edge-disjoint paths in G. Again, quick analysis: $O(m^2)$.

Q3b: What if you want to find the maximum number of *vertex-disjoint paths* from s to t instead? Two paths are called vertex-disjoint if they do not share any vertex. Is this (still) a Max Flow problem?

Answer: For this, we need to assign vertex capacities = 1 (most likely a new technique for many students) on all vertices too to constraint it to be used only once. We can split that vertex v into v_{in} and v_{out}. All edges that goes to v will be connected to v_{in}. All edges that goes out from v will be connected from v_{out}. Then, we place the vertex weight as weight of directed edge $v_{in} \rightarrow v_{out}$. At this point, we have the standard flow network again and we can use Q3a answer. Note that we only add n more vertices and n more edges in the transformed graph so the quick time complexity analysis is about $O((m+n)^2)$.

Q4: Please read `https://onlinejudge.org/external/117/11757.pdf` (that person *returned to that club two years ago, but by now he had left to another interesting league*) and try to reduce this problem into a max flow problem, solve it using $O(n^2 \times m)$ Dinic's algorithm (assuming that you have such implementation ready), and analyze its time complexity.

Answer: It is a creative problem about min cut :O... We build the flow graph with a bit of simple geometry involving circle.

The source s is the top of the field. The sink t is the bottom of the field. We connect s/t to nearby robot defenders that are within distance d away from the top/bottom of the field, respectively, with capacity 1 (to say that if we have to go through them, we need to handle 1 tackle).

Then we try all pairs of N^2 robot defenders and connect them with edge with capacity 1 if their Euclidean distance is less than or equal to 2d (we could put edge with capacity 2 here as we are technically fighting 2 robot defenders if the winger cross midway of the 2 robot defenders, but most likely the min cut answer is bounded by the next vertex capacity constraints). We then put vertex capacity 1 for each vertices (as each robot defender can only tackle once – this constraint may be missed by some students). Let's direct all edges from s (top of the field) to t (bottom of the field) to avoid double count.

Finally, we find the min cut of this flow graph using any max flow algorithm.

Analysis: $n = 2 * N + 2$ (remember that we use vertex splitting) and $m = N * N + N + N + N$. Plugging in the maximum numbers, $n = 202$ and $m = 10\,300$. So in the worst case, $O(n^2 \times m)$ Dinic's algorithm (the fastest Ford-Fulkerson algorithm mentioned in class so far) will need: $420\,281\,200$ operations?? See Q5b.

Q5a: Can we write the runtime of Ford-Fulkerson algorithm as $O(m \times F)$ where F is the eventual max flow value of the input graph? This is an output-sensitive analysis, whereby the runtime speed of the algorithm depends on the size of the output.

Answer: Yes, it is tighter but not known until runtime and some theoretician doesn't like this... It also gives precursor to Q5b.

Q5b: Now revisit problem in Q4 above and consider this 'Competitive Programmer' analysis that is not frequently found in Computer Science textbooks about max flow algorithms. Assuming that we are using the theoretically faster $O(n^2 \times m)$ Dinic's algorithm, can we analyze its time complexity as:
$O(min(\sum_{u:(s,u)\in E} c_{su}, \sum_{u:(u,t)\in E} c_{ut}, n^2) \times m)$.
Explain what are we trying to do here?

Answer: This can be a tighter analysis for some input flow graph: The max flow is upper bounded by the minimum of sum of edge capacities that goes out from source s AND sum of edge capacities that goes into sink t. The number of required iterations may not be 'that large'...

Many flow graphs have (much) smaller such bound than the textbook $O(n^2 \times m)$ time complexity of the fastest algorithm discussed in class so far: Dinic's algorithm.

For Q4, we have $O(min(n, n, n^2) \times m)$ as there are at most n robot defenders. This is up to $100 \times 10\,300 \approx 1M$ operations per test case in the worst case, which is 'small enough'.

Post Tutorial

Students can discuss PS3 ideas with the tutor if you still struggle as this is the 'last' tutorial before PS3 is due (remember: no tutorial on Monday of recess week...).

PS: Max flow is an interesting algorithm that will take time to master (properly). You are advised to use the recess week to revise the first half of CS4234.

CS4234: Optimisation Algorithms Tutorial 5

Max-Flow Part 2 + Graph-Matching Part 1

V1.7: A/Prof Steven Halim *October 02, 2023*

MODAL ANSWER IS FOR OUR CLASS USAGE ONLY; NOT TO BE DISTRIBUTED IN PUBLIC

Discussion Points

Q1: Please read `https://onlinejudge.org/external/128/12873.pdf` and try to reduce this problem into a max flow problem, solve it using $O(n^2 \times m)$ Dinic's algorithm (assuming that you have such implementation ready), and analyze its time complexity. Follow-up question: Can we solve this problem as a pure MCBM problem and solve it with Augmenting Path algorithm++ (the one with randomized greedy pre-processing step)?

Answer: It is like Bipartite Matching that was discussed earlier in T04, but with some capacities... Also known as: An assignment problem (discussed in details in page 428-429 in Competitive Programming 4 Book 2 and also have been modeled in `https://visualgo.net/en/maxflow`, Modeling, Bipartite Matching, select left side 1 option);

It appeared in LA 6851 - Bangkok14; Our top NUS ICPC team solved this problem rather fast en-route to become the runner up (unfortunately didn't win) of that contest.

Analysis: $n = P + S + 2$ and $m = P * S + P + S$. Plugging in the maximum numbers, $n = 522$ and $m = 10\,520$. So in the worst case, $O(n^2 \times m)$ Dinic's algorithm (the fastest Ford-Fulkerson algorithm mentioned in class so far) will need: $2\,866\,531\,680$ operations?? (that is 28 more than the typical limit of 10^8 operations in 1 second in Kattis...)

Not really, again using the tighter analysis from T04, we have $O(min(P, S * C) \times (P \times S + P + S))$ or $500 \times (500 \times 20 + 500 + 20) = 5\,260\,000$. Now we are quite confident that our computer can solve this problem 'within 1 second'...

Note that trying to solve this problem as a pure MCBM (each vertex can only be matched once), we need to 'blow up the vertices' according to its capacity. This is not a scalable transformation for large capacities.

Q2: In Lecture 6, we have learned that there are some Bipartite Matching problem that admits Greedy solution. As an exercise, try solving `https://nus.kattis.com/problems/froshweek2` using a Greedy algorithm. Can we hope to pass the time limit if we use any MCBM (or Max Flow) algorithm?

Answer: This problem is clearly an MCBM problem. The left set are the tasks, the right set are the quiet times. There is an edge between task i and quiet time j if the length of task i is $\leq$ the length of quiet time j. However, this problem admits a greedy strategy: After sorting tasks and quiet times in non-decreasing order, we can greedily match task i (shortest task) with the first quiet time j that has length $\geq$ length of task i (see Figure 1 for a hidden e-Lecture slide that TAs cannot show on their VisuAlgo account). A simple greedy exchange argument proof can be supplied to show that any optimal bipartite matching can be transformed into this one. The time complexity is thus bounded by the need to sort the length of the tasks and the quiet times, so $O(n \log n + m \log m)$.

Note that since $n, m \leq 200\,000$, it is near impossible to use a proper MCBM (or Max Flow) algorithm even if we use MCBM algorithm with the $O(n^{2.5})$ time complexity.

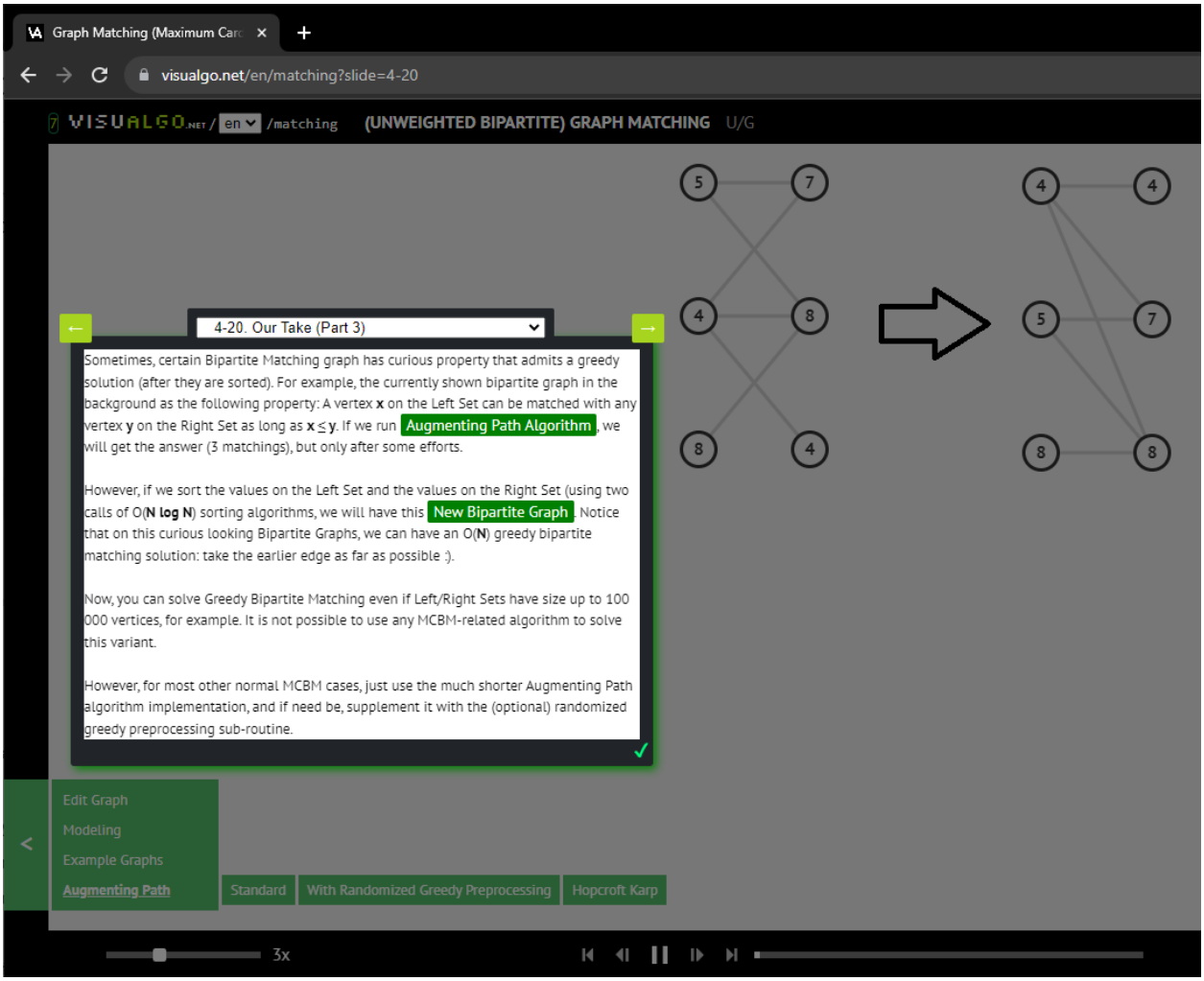

Figure 1: Greedy MCBM Example.

Q3: Back in Lecture 1, we have learned about the Min-Vertex-Cover (MVC) problem. In T03, we also have learned about the Max-Independent-Set (MIS) problem. In Lecture 6, we have seen the special cases of solving MVC if it is asked on *Bipartite Graph*, but only the easier part of Konig's theorem, i.e., "In any bipartite graph, the number of edges in a maximum matching equals the number of vertices in a minimum vertex cover.".

In problem `https://nus.kattis.com/problems/bilateral`, you are asked to find MVC and *one possible solution* too... In short, given a Bipartite Graph $G = (V_L, V_R), E$ of approximately 2000 vertices and up to 10 000 edges, show how to find the MVC (the optimal solution, not just the cardinality of the optimal solution) on G by reducing those problems into MCBM (you can then use a MCBM-specific algorithm or further reduce them into Max Flow problems and use a Max Flow algorithm).

Follow up question: How to find MIS instead of MVC? Then, what if the MVC/MIS problems asked are the weighted variants?

Answer: This /bilateral problem is again a Bipartite Graph problem. Left set: Employees in the Stockholm office [1000..1999]. Right set: Employees in the London office [2000.2999]. Team (of two), essentially an edge, can only cross from one in the Stockholm office and another employee in the London office.

Now, | MVC | in Bipartite Graph equals to | MCBM | (notice conversion from $\geq$ in general graph to == in Bipartite graph) due to Denes Konig's theorem and we can use any Max Flow (preferably Dinic's, it has tighter time complexity for Bipartite Matching graph) or the faster specific MCBM algorithms (preferably the one with randomized greedy pre-processing step).

Jeno Egervary gave similar proof on weighted variant of MVC but this time, we will need to use Max Flow. As | MCBM | = max flow = min cut of the correctly modeled flow graph on that bipartite graph (capacities of s to left set equal to the weights of vertices on left set; capacity of right set to t equal to the weights of vertices on right set; capacities of edges in the middle are ∞ and will not be part of the min-cut), we can use any Max Flow algorithm (preferably Dinic's) to find the max flow = min cut = | MCBM | value.

So far there are three known ways to show the MVC certificate (version 2 and 3 are currently shown in VisuAlgo MVC page):

1. The edges that are part of the min cut will only be the s to left set or right set to t. Exclude s and t and you will get the certificate of the MVC.

2. The { t-component $\cap$ left-set } $\cup$ { s-component $\cap$ right-set }.

3. The Konig's constructive proof: U = unmatched vertices on the left set, Z = vertices along the augmenting path that starts from vertices in U, and certificate K of the MVC = $\{L \setminus Z\} \cup \{R \cap Z\}$.

We only talk about MVC just now. How about MIS? As MVC and MIS are complementary, the | MIS | in (not necessarily Bipartite) Graph equals to | V |-| MCBM |. For weighted version: (sum of all vertex weights-MVC).

Please review this at `https://visualgo.net/en/mvc`, load/draw any bipartite graph and click MVC on Bipartite Graph, Konig's Theorem.

Q4: Discuss the solution for `https://nus.kattis.com/problems/taxicab`. Is this (a special case of) an NP-hard problem? Which PS4 task is this one?

*Answer: This problem is actually an NP-hard `Min-Path-Cover` problem (what is the minimum number of paths needed to cover the graph $G = (V, E)$?). However, it is posed on a Directed **Acyclic** Graph (DAG) structure and there is a special case polynomial algorithm for this MPC on DAG problem by reducing this problem into MCBM with a 'special construction'. This is a rare graph matching modeling problem and we do not expect students to recognize this modeling if one has not been exposed to this technique before.*

Basically, we want to cover n vertices with minimal number of paths. Initially, we start with n length-1 paths (that contains each vertex by itself). Then we construct MCBM modeling as shown in Figure 2. Every time there is a matching, we can use the same path/edge (same taxi) to cover both vertices, i.e. we need one less path. If we maximize the MCBM, we have the least amount of paths used. See CP4 Section 8.6.8 (page 454) for more details. PS4 B - airports is (the harder form of) this problem.

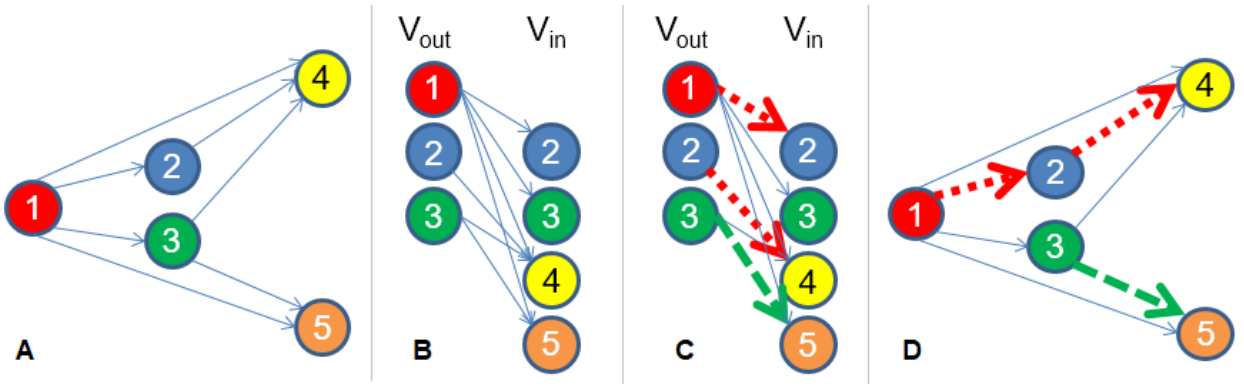

Figure 2: `Min-Path-Cover` on DAG example: 5 taxis, draw an edge between u and v if a taxi can serve both u and v on time. The answer for this test case is 2 paths: $0 \rightarrow 1 \rightarrow 2$ and $3 \rightarrow 5$.

Q5: Discuss the general idea of PS4 A.

PS4 A - antennaplacement can be viewed as a Min-Set-Cover problem. The Universe U contains all the points of interests (the ''s). The subsets are the antennas (a short edge between a point of interest and its 4 N/E/S/W neighbors). Picking one antenna can cover at least 1 point of interest, but we prefer that it covers 2 points of interests at the same time. If we attack the problem this way, we may end up with a really NP-hard problem with exponential complete search solution (TLE).*

However, if we realize that this problem is set on a grid *graph, we may be able to identify potential 'hidden' bipartite graph of this problem.*

Pause to let students think...

One of the common transformation of grid graph into bipartite graph is to split the cells based on chess-board (white-black) pattern. We put all points of interests on white cells on the left set and put all points of interests on black cells on the right. Now, if we model it this way, an antenna (a short edge) now only connects a cell on the left with another neighboring cell on the right (because the color of any cell in a chess board to its 4 N/E/S/W neighbor always change). Now what problem that we want to solve? We want to pick as few antennas (edges) as possible to cover all vertices... okay, but we have not seen this type of problem before. Can we transform this problem into something else that you have actually seen before in the first half of CS4234?

TAs are supposed to stop here...

Hint: MIS...

OPTIONAL: In 06.MatchingLecture1.pdf and 07.MatchingLecture2.pdf (later), we use *Berge's theorem* for Augmenting Path(++) algorithm, Kuhn-Munkres (Hungarian) algorithm, and also Edmonds' Matching algorithm.

From `https://en.wikipedia.org/wiki/Berge%27s_theorem`, the wording of Berge's theorem is as follows: "In graph theory, Berge's theorem states that a matching M in a graph G is maximum (contains the largest possible number of edges) if and only if there is no augmenting path (a path that starts and ends on free (unmatched) vertices, and alternates between edges in and not in the matching) with M".

In 06.MatchingLecture1.pdf, Prof Halim has shown the proof. Students who are already comfortable

with the presentation of this proof can leave the tutorial. TA can re-present the proof one more time for the remainder.

We will discuss `https://en.wikipedia.org/wiki/Berge%27s_theorem#Proof`.

The proof has two parts, the forwards direction and the backwards direction (if and only if).

The forwards proof is easier: M in G is maximum $\rightarrow$ there is no augmenting path in G w.r.t M.
`https://www.youtube.com/watch?v=93EVpMqOv5s`
Proof by contradiction: Suppose M in G is maximum but G ***has*** *an augmenting path w.r.t matching M. Now, this augmenting path can be flipped into another matching M' that takes the other edges along the augmenting path. Thus, $|M'| = |M| + 1$, contradicting the statement that M is maximum matching.*

The backwards proof is a bit harder: M in G is maximum $\leftarrow$ there is no augmenting path in G w.r.t M.
`https://www.youtube.com/watch?v=KJns-h3jXNc`
Proof by contradiction: Suppose there is no augmenting path in G w.r.t M but M in G is ***not*** *maximum, i.e., we have M' that is larger than M. We take a symmetric difference (new term for many of us probably) of M' and M to produce a new graph G' that has the same vertices as G, but only edges that are involved in either M' or M. Let's observe this new graph G'. Notice that G' will only consist of vertices with degree 0 (isolated vertices, will not help with this proof), degree 1 (endpoint of an augmenting path), or degree 2 (in the middle of augmenting path, a vertex that connects an edge in M and another edge in M'). Graph with degree not more than 2 can only consist of paths or cycles (recall MVC special case on such graphs in Tutorial01?).*

We won't have odd-length cycle as the edges in G' only comes from M and M'.
We can have even-length cycle but it doesn't help with this proof (equal size in both M and M').
We can have even-length path but it also doesn't help with this proof (also equal size in both M and M').
Lastly, we can have odd-length path where the path starts and ends with edges from the 'larger' M' and edges in M are slightly inside... Now what is this? This is an augmenting path w.r.t. M, so again we arrive at a contradiction.

Conclusion: Berge's lemma is correct and it is an integral part of Augmenting Path(++) algorithm, Kuhn-Munkres (Hungarian) algorithm, and also Edmonds' Matching algorithm.