Missouri University of Science & Technology | Department of Computer Science
Spring 2024 | **CS 2500: Algorithms (Sec: 102)**

Exam 2

Name: Jacob Lee **Email:** jk1m3w

Please read the following instructions

1. This is a **open-book, take-home exam**. However, students are not allowed to interact or discuss with each other in solving these problems.
2. Write your solutions (**in detail**) only within the provided space on numbered pages. You are not allowed to take any further space than what is provided to you.
3. Use a pen, rather than a pencil to make sure your solutions are legible. Your grades will depend on how well the instructor can understand your solution.
4. Your solutions are **due by Friday, Apr 19, 2024, by 11:59 PM.** No further extensions will be granted under any circumstance.
5. Lastly, please do not forget to write your full name at the top of this page (within the header).

DO NOT WRITE ANYTHING BELOW THIS LINE ON THIS PAGE

Problem	**Max. Possible Points**	**Awarded Points**
1	5	
2	5	
3	5	
Total	15	

Problem 1 Graph Algorithms *5 pts.*

Clearly indicate the following spanning trees in the weighted graph pictured below, assuming that node-1 is the start vertex. Some of them have more than one correct answer.

Note: You do not have to demonstrate the algorithm. Just depict the final result.

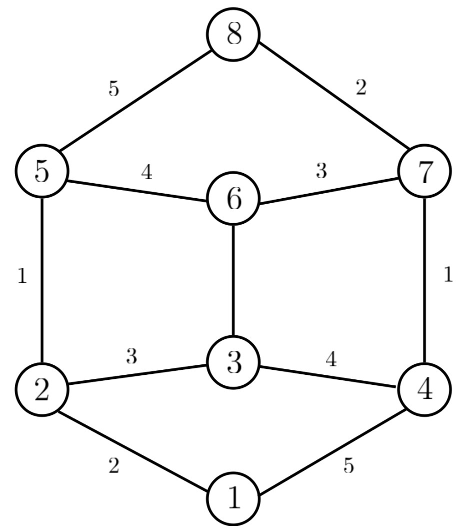
(a) A breadth-first spanning tree, with start node as 1

(b) A depth-first spanning tree rooted, with start node as 1

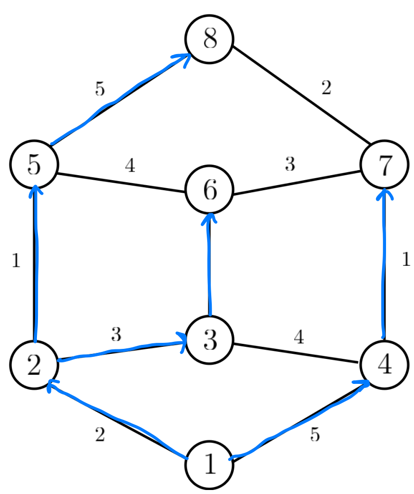
(c) A shortest-path spanning tree, with start node as 1

(d) A minimum spanning tree

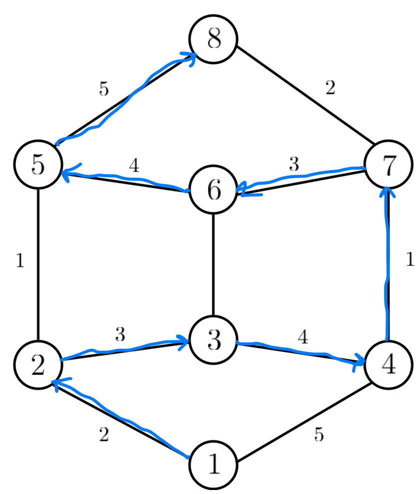
(e) A maximum spanning tree



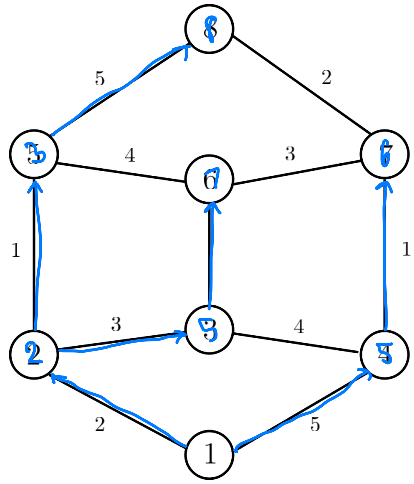
Solution 1



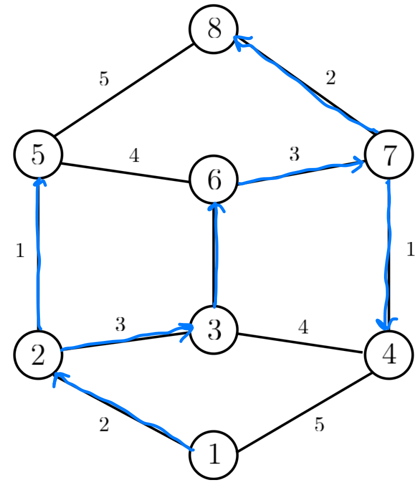
(a) Breadth-first spanning tree, rooted in 1.



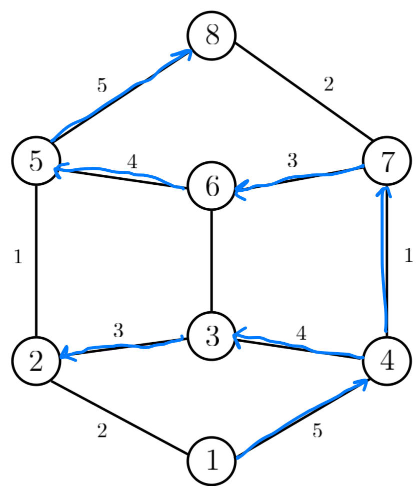
(a) Depth-first spanning tree, rooted in 1.



(a) A shortest-path spanning tree, rooted in 1.



(a) Minimum spanning tree



(a) Maximum spanning tree

Problem 2 Topological Ordering and Cycles *5 pts.*

Most software projects consists of multiple modules that are interdependent on each other. For example, Module A may depend on Module B, Module B may depend on Module C, and Module C may depend on Module A. This forms a cycle in the dependency graph, which can cause issues during the build process. Given the dependency graph on the various software modules, develop a single algorithm called `TopoSortCycles(`G`)` on a dependency (directed) graph G that does both of the following:

(a) Return the topological ordering of modules using `DFS_Topo(`G, s`)` subroutine (given below) to build the software. (2 pts)

DFS_TOPO$(G, s, CurrentLabel)$

for each vertex $v \in G.V$
    $v.explored = 0$
$s.explored = 1$
for each vertex $v \in G.V$
    if $v.explored = 1$
        DFS_TOPO(G, v)
$s.order = CurrentLabel$ // Assign position of s in the topological ordering
$CurrentLabel = CurrentLabel - 1$
return $CurrentLabel$

(b) Detect any cycle in the dependency graph, in which case, it should throw an error and print the cycle. (3 pts)

Solution 2

```
def DFS_Topological (v):
    v.explored = True
    v.visiting = True
    for w in G.adj[v]:
        if not w.explored:
            DFS_Topological (w)
        elif w.visiting:
            print (f"cycle detected including vertex {w}")
            has_cycle = True
    v.visiting = False
    v.finished = True
    topo_order.insert (0,v)
```

```
def TopoSortCycles(G):
    topo_order = []
    has_cycle = False
    for v in G.V:
        if not v.explored:
            DFS_Topological(v)
    if has_cylce:
        print("cycle detected, no topological order possible.")
    else:
        return topo_order
```

Problem 3 Fractional Knapsack *5 pts.*

Consider a fractional Knapsack problem with n divisible items, where v_i and w_i are the value and weight of the i^{th} item respectively. Assume the size of the Knapsack is W.

(a) Design an optimal greedy algorithm, i.e. model the fractional Knapsack as a multi-stage decision problem (clearly define the state, decision, outcome and reward variables, and the greedy decision philosophy based on a myopic score used to optimize at each stage.) (3 pts)

(b) Prove the correctness of your greedy algorithm for the fractional Knapsack problem. (2 pts)

Solution 3(a)

Multi-Stage Decision Problem Model: At the K^{th} stage, we have

- State s_K: the current capacity of the Knapsack at stage K
- Decision x_K: the fraction of the item K to include in the Knapsack
- Outcome y_K: the new state after making the decision x_K
- Reward u_K: the value added to the Knapsack by including x_K of item K

Mypoic (Greedy) Decision: Choose item with the highest value to weight ratio then repeat until Knapsack's capacity is full or there are no more items left.

Pseudocode:

```
GreedyFracKnapsack(items):
    items.sort(key= item: item.value / item.weight, reverse=True)
    total value = 0
    for item in items:
        if capacity == 0:
            break
        amount = min(item.weight, capacity)
        total_value += amount * (item.value / item.weight)
        capacity -= amount
    return total_value
```

Solution 3(b)

Loop Invariant:

(Hint: Your loop invariant should ensure that the output in the final iteration, i.e. decision stage, is the optimal set of items with maximum total value that fits in the Knapsack.)

In the reminder of this proof, the loop invariant holds true in every iteration (decision stage):

Initialization: Before the loop begins, the maximum value is 0 which is the optimal solution for a Knapsack of 0 capacity

Maintenance: Assume the loop invariant holds true at the end of $(k-1)^{th}$ iteration. Then, during the k^{th} iteration, let the residual capacity of the Knapsack be $c = C - \sum_{i \in \texttt{Knapsack}} w_i$.

next item picked by `GreedyFracKnapsack`(items)

Assuming that the Knapsack's value is optimal for a given capacity before adding a new item, after adding a part of or a whole of the next item, it remains optimal

Termination: At the end of the loop, the Knapsack's capacity is exhausted, and the value in the Knapsack is the maximum possible for the given capacity.