Graphs and Algorithms Notes

Hashan Punchihewa

https://hashanp.xyz

Contents

1 Graphs and Algorithms **1**

1.1 Graphs 1

1.2 Graph Algorithms 6

1.3 Algorithmic Analysis 13

1.4 Complexity 22

© Hashan Punchihewa 2017-2018. All rights reserved.

Chapter 1

Graphs and Algorithms

1.1 Graphs

Definition 1.1.1

- *A graph G consists of a set of nodes, denoted as $\mathrm{nodes}(G)$ and a set of arcs denoted as $\mathrm{arcs}(G)$ such that for each $a \in \mathrm{arcs}(G)$, there is a corresponding unordered pair of nodes. We say that the arc a is between these two nodes.*
- *Parallel arcs are arcs that are between the same nodes.*
- *A loop is where the nodes in the corresponding pair for an arc are the same.*
- *A simple graph has no parallel arcs or loops.*
- *If a node n appears in the corresponding pair of nodes for an arc a, a is incident to n.*
- *If n and n' are the corresponding pair of nodes for the same arc a, then n is adjacent to n' and vice versa.*
- *The degree of a node is the number of arcs incident to a node, where loops are counted twice.*

Proposition 1.1.1

1. *The sum of all degrees in a graph is twice the number of arcs, and therefore even.*
2. *The number of nodes with an odd degree is even.*

Definition 1.1.2

For a graphs G and G':

- *G is a subgraph of G' if $\mathrm{nodes}(G) \subseteq \mathrm{nodes}(G')$ and $\mathrm{arcs}(G) \subseteq \mathrm{arcs}(G')$.*
- *If G is a subgraph of G' and $\mathrm{nodes}(G) = \mathrm{nodes}(G')$, then G spans G'.*

- *A subset* $X \subseteq \text{nodes}(G)$ *induces a subgraph* $G[X] \subseteq G$*, with nodes* X *and all arcs in* G *which contain only nodes in* X*.*
- *A subgraph* $G' = G[X]$ *is an induced (or full) subgraph of* G*, for some* $X \subseteq \text{nodes}(G)$*.*

Definition 1.1.3

- *An adjacency matrix for a graph containing* n *nodes is an* n *by* n *matrix, where the element at* (i, j)*th position contains the number of arcs between the* i*th and* j*th node. Note that loops are counted twice.*
- *An adjacency list is a representation of a graph consisting of an array of linked lists. There is a linked list for each node that contains every other node that the node is incident to. Each arc appears twice, except for loops.*

Note that adjacency matrices are a better representation for graphs with lots of arcs, since an adjacency matrix has n^2 entries, where n is the number of nodes. But an adjacency list is a better representation for graphs, where the number of arcs is much less than n^2, since it has at most size $n + 2m$, where m is the number of arcs.

Definition 1.1.4

- *A graph isomorphism between two graphs* G *and* G' *consists of a bijection* $f : \text{nodes}(G) \to \text{nodes}(G')$ *and a bijection* $g : \text{arcs}(G) \to \text{arcs}(G')$*, such that for an arc* a *between* n *and* n'*, then* $g(a)$ *is between* $f(n)$ *and* $f(n')$*.*
- *This equivalent to saying that the adjacency matrices are the same, except the ordering of rows and columns may be different.*
- *An isomorphism between the same graph is called an automorphism.*
- *Every graph has at least one automorphism, the identity automorphism.*
- *If there exists an isomorphism between two graphs, then the graphs are called isomorphic.*

Definition 1.1.5

- *An algorithm is said to consume* $O(g(n))$ *of some resource with respect to a measure of input size* n*, if the amount of that resource consumed is bounded above, by a constant factor of* $g(n)$*, after a certain input size.*
- *More formally, if the amount of the resource consumed is* $f(n)$*, then* $f(n)$ *is* $O(g(n))$*, or even* $f(n) = O(g(n))$ *if:*

$$\exists m \in \mathbb{N} \; \exists c \in \mathbb{R}^+ \; \forall n \geq m \; f(n) \leq cg(n)$$

- *It is said* $f(n) = \Theta(g(n))$ *if* $f(n) = O(g(n))$ *and* $g(n) = O(f(n))$*.*
- *If the resource is time, then* $O(1)$ *is called constant time. Similarly,* $O(n)$ *is called linear time,* $O(n^2)$ *is called quadratic time, etc.*

- *In addition, $O(n^k)$ for some $k > 0$ is called polynomial time, and $O(k^n)$ for some $k > 0$ is called exponential time.*
- *$O(\log n)$ is called logarithmic time and $O(n!)$ is called factorial time.*

The notation introduced above is called big-O notation, and its principal advantage is that it abstracts machine and implementation dependent constant factors, and concentrates on the rate at which resource consumption increases, as input size increases. The trivial algorithm for checking for graph isomorphisms, by considering every rearrangement, takes $O(n!)$, i.e. factorial time complexity.

Proposition 1.1.2 *Detecting whether two graphs are isomorphic can be solved in at least quasi-polynomial time.*

Definition 1.1.6

- *A graph is planar if it can be drawn so that no arcs cross.*
- *The complete graph of n nodes, denoted to as K_n, is a simple graph where every distinct pair of nodes has an arc between them.*
- *A graph is bipartite if it can be partitioned into two sets, such that there are no arcs between nodes in the same set.*
- *The complete bipartite graph of n_1, n_2 nodes, denoted as K_{n_1,n_2} consists of n_1 nodes in one set and n_2 nodes in the other set, with every possible arc, without violating the condition of being bipartite.*
- *Two graphs are homeomorphic if they can be obtained from the same graph, by a series of operations replacing nodes $x - y$, with nodes $x - z - y$, i.e. introducing a new node between any two incident nodes.*
- *A dual graph for a map is a graph where every country is a node, and borders are represented as arcs.*
- *A graph is k-colourable if the nodes of the graph can be coloured with k colours.*

Proposition 1.1.3

- *Any non-planar graph, must have a subgraph homeomorphic to K_5 or $K_{3,3}$.*
- *Any planar graph, splits a graph into regions called faces, and it must be the case that $F = A - N + 2$, where F is the number of faces, A is the number of arcs, and N the number of nodes. This is called Euler's formula.*
- *Detecting where a graph with n nodes and m arcs can be done in $O(n + m)$.*
- *Dual graphs of maps are always simple and planar.*
- *Every simple planar graph is the dual graph for some map.*
- *The dual graph of a map being k-colourable is equivalent to a map being colourable with k colours.*

- *(Four Colour Theorem) Every simple planar graph is 4-colourable and every map can be coloured with 4 colours.*

Definition 1.1.7

- *A path is a sequence of adjacent arcs.*
- *Two nodes are connected if there exists a path between them. Note a node is trivially connected to itself.*
- *Two nodes being connected forms an equivalence relation, since it is reflexive, symmetric and transitive. The equivalence classes of this relation are called the connected components of a graph.*
- *A graph is connected if every pair of nodes is connected.*
- *A cycle is a path which starts and finishes at the same node, but does not use the same arc twice.*
- *A cyclic graph contains at least one cycle.*
- *An acyclic graph contains no cycles.*
- *An Euler path is a path which uses each arc exactly once.*
- *An Euler circuit (Or cycle) is a cycle which uses each arc exactly once.*

Proposition 1.1.4

- *A connected graph has an Euler path if and only if there are either 0 or 2 odd-degree nodes.*
- *A connected graph has an Euler circuit if and only if there are only even-degree nodes.*

Proof: Consider an odd-degree node. There are an odd number of arcs leading to it. When a node is used as an intermediate node, the arc leading to it, must be followed by an arc leading from the node. So any node used exclusively as an intermediate node, cannot be an odd-degree node. So an odd-degree must appear as the start node and/or the end node. If the start and the end node are the same, i.e. an Euler circuit, however, then that node will have even-degree, so there'll be 0 odd-degree nodes. If they are different then there will be 2 odd-degree nodes.

A proof in the other direction is beyond the scope of the course.

Definition 1.1.8

- *A Hamiltonian path is a path that visits every node exactly once.*
- *A Hamiltonian circuit is a cycle that visits every node exactly once.*

Proposition 1.1.5

- *For a Hamiltonian path to exist, a graph must be connected.*

- *For a Hamiltonian circuit to exist, each node must have a degree greater than or equal to 2.*

Problem 1 ***(Hamiltonian Circuit Problem)***

Given a graph G determine if there is a Hamiltonian circuit.

Proposition 1.1.6 *The Hamiltonian Circuit Problem is NP-Complete.*

A naïve algorithm to solve the problem would take $O(n!)$ time by checking every possible path.

Definition 1.1.9

- *A rooted graph is a graph, with a special node called the root.*
- *A tree is an acyclic, connected, rooted graph.*
- *A nonrooted tree is an acyclic, connected graph.*
- *The depth of a node x is the distance between the root and x.*
- *The depth of a tree is the maximum depth of any node.*
- *If x is a node, which isn't the root, then it has a parent which is the unique adjacent node on the path to the root.*
- *If x is the parent of y, y is the child of x.*
- *A spanning tree T of a graph G, is a non-rooted tree that span G.*

Proposition 1.1.7 *A tree with n nodes will have $n - 1$ arcs.*

Proof: Every node, but the root will have a parent. Each non-root node, will have exactly one arc to its parent. An arc, will always be the one node that connects a child to its parent for some child node. So for every node, except the root, there is one arc.

Definition 1.1.10

- *A directed graph is the same as an ordinary graph, except that for each arc $a \in \text{arcs}(G)$ there is a corresponding ordered pair of nodes, not unordered.*
- *The in-degree of a node, is the number of arcs entering a node.*
- *The out-degree of a node, is the number of arcs leaving a node.*

Proposition 1.1.8

For a directed graph, the number of arcs, the number of in-degrees and the number of out-degrees are equal.

1.2 Graph Algorithms

There are two standard algorithms for search graphs: Breadth-First Search (BFS) and Depth-First Search (DFS). DFS goes down the graph as far as it can, before backtracking. It is a recursive algorithm, with the following pseudocode:

```
DEPTH-FIRST-SEARCH(x)
1  visited[x] = TRUE
2  for z = adj[x]
3      if not visited[z]:
4          DEPTH-FIRST-SEARCH(z)
```

Since DEPTH-FIRST-SEARCH is applied to each node once, and each application runs through arcs incident to any given node once, the running time $O(n+m)$, where n is the number of nodes and m the number of arcs. In the worst case, $m = n^2$, so the running time would be $O(n^2)$. The space complexity depends is $O(h)$ where h is depth from the starting node, where $h = n$ is the worst case, so space complexity would be $O(n)$. This is solely due to recursion.

```
BREADTH-FIRST-SEARCH(x)
1  visited[x] = TRUE
2  Q = QUEUE()
3  ENQUEUE(Q, x)
4  while not EMPTY(Q)
5      y = DEQUEUE(Q)
6      for z = adj[y]
7          if not visited[z]:
8              visited[x] = TRUE
9              ENQUEUE(Q, z)
```

BFS, encounters each node once, and each arc incident to any given node is looked at once, so, the time complexity is the same, $O(n+m)$. The worst-case space complexity is $O(n)$, since you may end up needing to hold all the nodes in your queue.

Both of these algorithms can be used to check is a graph is connected, by checking if all nodes have been visited. Both algorithms can be used to find a spanning tree. Depth-first search can be used to detect cycles in a graph. This is as you can detect a cycle by finding a node, that has been visited, that is not the parent of the current node. Breadth-first search can also be used, but you need to keep track of parents. Breadth-first search finds the shortest path to any reachable node, whereas depth-first search may give a longer path than necessary. In addition, a connected graph with n nodes has a cycle, if it has n or more arcs.

Proposition 1.2.1 *If a is an arc in a graph G, with a spanning tree T generated by depth-first search, where the endpoints of a are x and y, then in T, either x is an ancestor of y or y is an ancestor of x, regardless of if a is in T.*

Definition 1.2.1

- *A weighted graph G is a simple graph, with a weight function $W : \mathrm{arcs}(G) \to \mathbb{R}^+$, where $\mathbb{R}^+$ denotes the non-negative reals.*
- *A minimum spanning tree is the spanning tree where there is no other spanning tree where the the sum of the weights of the constituent arcs is smaller.*

There are two principal algorithms to find a minimum spanning tree: Prim's algorithm and Kruskal's algorithm.

Prim's algorithm in essence starts with a single node, which forms the initial tree, and adds the arc, with the shortest weight in the fringe. It does this until all nodes are in the tree. The fringe refers to the arcs, not in the tree, which leads to a node, which is not in the tree.

```
PRIM'S-ALGORITHM(G)
 1  tree.start = TRUE
 2  for x ∈ adj[start]
 3      fringe[x] = TRUE
 4      parent[x] = start
 5      weight[x] = W(start, x)
 6  while not EMPTY(fringe)
 7      f = x, such that fringe[x] and weight[x] is minimum.
 8      fringe[f] = FALSE
 9      tree[f] = TRUE
10      for y ∈ adj[f]
11          if not tree[y]
12              if not fringe[y]
13                  if W(f, y) < weight[y]
14                      weight[y] = W(f, y)
15                      parent[y] = f
16              else
17                  fringe[y] = TRUE
18                  weight[y] = W(f, y)
19                  parent[y] = f
```

Testing whether the fringe is empty, takes constant time, finding the fringe node x such that $weight[x]$, is minimum also take constant time. The same is true for going through the adjacent nodes. There are $O(n)$ executions of the while loop. Therefore, the algorithm takes time $O(n^2)$.

Proposition 1.2.2 *Prim's algorithm correctly constructs the minimum spanning tree for a graph G with n nodes.*

Proof: Let T_n be the minimum spanning tree at stage n, where T_0 consists solely of the $start$ node, and T_{k+1} is obtained by adding arc a_{k+1} to T_k.

First it must be shown that T_k is a subgraph of a minimum spanning tree T' of G by induction. The base case, is where there is one node $start$ and no arcs, in T_0. Clearly $T_0 \subseteq T'$ for any minimum spanning tree T'. The inductive case is where $T_k \subseteq T'$. Observe, that if $a_{k+1} \in \text{arcs}(T')$ then $T_{k+1} \subseteq T'$. If it wasn't the case that $a_{k+1} \in \text{arcs}(T')$, then there must already be a path in the tree from the endpoints x and y. This path must cross between the current tree and the fringe. Call the arc that does this, a. A new spanning tree can be formed by removing the existing a and adding a_{k+1}. Since the algorithm chose a_{k+1} not a, this means $W(a_{k+1}) \leq W(a)$, so this new spanning tree must be a minimum spanning tree.

Since T_{n-1} has $n-1$ arcs, and is a subset of a spanning tree T', this means that T_{n-1} is a minimum spanning tree, since all spanning trees have $n-1$ arcs.

Prim's algorithm can also be implemented using a data structure called a priority queue, where each item is given a key, usually a natural number indicating its priority. It has the following interface: PQCREATE, EMPTY, INSERT, GET-MINIMUM, DELETE-MINIMUM and DECREASE-KEY. Priority queues can be implemented using binary heaps, where GET-MINIMUM and EMPTY are constant time operations and the others are logarithmic time.

```
PRIM'S-ALGORITHM(G)
 1  Q = PQCREATE()
 2  for x ∈ nodes(G)
 3      key[x] = ∞
 4      parent[x] = NULL
 5      INSERT(Q, x)
 6  DECREASE-KEY(Q, start, 0)
 7  while not EMPTY(Q)
 8      f = GET-MINIMUM(Q)
 9      DELETE-MINIMUM(Q)
10      tree[f] = TRUE
11      for y ∈ adj[f]
12          if not tree[y]
13              if W(f, y) < key[y]
14                  DECREASE-KEY(Q, y, W(f, y))
15                  parent[y] = f
```

The number of times the inner for-loop executes is proportional to the number of arcs. Since there are at most n nodes in the priority queue, DECREASE-KEY takes time $O(\log n)$. So, all in all, an implementation of Prim's with priority queues takes time $O(m \log n)$, where n is the number of nodes, and m the number of arcs. Therefore, Prim's with priority queues is better when the graph is sparse, and classic Prim's is better when the graph is dense.

An alternative algorithm to find the minimum spanning tree is Kruskal's where, rather than taking the smallest arc in the fringe, the smallest arc that doesn't introduce a cycle is added. This means when constructing the minimum spanning tree there will be several connected

components, that needed to be kept track of. Intuitively, at the beginning each node can be considered as its own connected component, and at each iteration, two connected components are made into a single connected component.

To do this a data structure called union find, that implements dynamic equivalence classes is used. It has the operations UFCREATE, FIND and UNION. Internally a union find data structure stores an array. Each connected component is stored as a tree, where the nth item in the array, stores the index of the parent of the nth node. If the nth item is n, then this means it is the root. Union involves taking the tree of least weight and appending it to the tree of greater weight. Note, that this is not a binary tree. To store sizes, a separate array is used. By appending the smaller tree to the larger tree, this means the tree is at most of depth $\lfloor \log n \rfloor$ Each node at the beginning is given its own equivalence class, with labels 1 to n. Then a find operation takes $O(\log n)$ time, to find the root of the tree, and union takes $O(1)$ time.

KRUSKAL'S-ALGORITHM(G)

```
1   Q = PQCreate(n)
2   for a ∈ arcs(G)
3       Insert(Q, a, W(a))
4   T = UFCreate(n)
5   F = ∅
6   while not Empty(Q)
7       (x, y) = Get-Minimum(Q)
8       x' = Find(T, x)
9       y' = Find(T, y)
10      if x' ≠ y'
11          Insert(F, (x, y))
12          Union(T, x', y')
```

The number of times the while-loop runs is proportional to the number of arcs in G, m, and since the complexity of FIND is $O(\log n)$, then the complexity of the overall algorithm is $\Theta(m \log n)$. In fact, the complexity of Kruskal's algorithm can be improved by path compression, so that it becomes $O((n + m) \log^* n)$, where $\log^* n$ is an extremely slow-growing function. Here when finding the root for a node x, set its parent to be the root, if it isn't already. This speeds up FIND operations in the future. Fibonacci heaps can also be used to implement priority queue where all operations are constant time, except for DELETE-MINIMUM, which is logarithmic time. The overall complexity of Prim's using Fibonacci heap priority queues is $O(m + n \log n)$.

Here is the Warshall algorithm for finding if two nodes are connected, where b is an 2 dimensional array of booleans where $b[i, j]$ is true if there is an arc between nodes i and j.

WARSHALL'S-ALGORITHM(b)

```
1  for k = 1 to n
2      for i = 1 to n
3          for j = 1 to n
4              b_ij = b_ij ∨ (b_ik ∧ b_kj)
5  return b
```

This can be extended into Floyd's algorithm, or the Floyd-Warshall algorithm:

FLOYD-WARSHALL-ALGORITHM(b)

```
1  for k = 1 to n
2      for i = 1 to n
3          for j = 1 to n
4              b_ij = min(b_ij, b_ik + b_kj)
5  return b
```

The Floyd-Warshall algorithm takes cubic time, $O(n^3)$. There exists a quadratic time, $O(n^2)$ algorithm to find the shortest distance between two specific nodes, called Dijkstra's algorithm, which is based on Prim's algorithm. In this algorithm, each node in the fringe is marked with the shortest distance found from the start node to this node, found so far. The algorithm continues until the destination node is found. An implementation based on classic Prim's goes as follows. The Floyd-Warshall algorithm is an example of dynamic programming.

DIJKSTRA'S-ALGORITHM($start$, end)

```
 1  tree[start] = TRUE
 2  for x ∈ adj[start]
 3      fringe[x] = TRUE
 4      parent[x] = start
 5      distance[x] = W(start, x)
 6  while not tree[finish] and not EMPTY(fringe)
 7      f = x, such that fringe[x] and distance[x] is minimum.
 8      fringe[x] = FALSE
 9      tree[x] = TRUE
10      for y ∈ adj[f]
11          if not tree[y]
12              if fringe[y]
13                  if distance[f] + W[f, y] < distance[y]
14                      distance[y] = distance[f] + W[f, y]
15                      parent[y] = f
16              else
17                  fringe[y] = TRUE
18                  distance[y] = distance[f] + W[f, y]
19                  parent[y] = f
20  return distance[finish]
```

Proposition 1.2.3 *Dijkstra's algorithm finds the shortest path between two nodes.*

Proof: The algorithm terminates because the size of the tree increases, at each iteration of the while-loop, the size of the tree increases. To show the correctness of the algorithm, we use three invariants:

1. If x is a tree node (other than $start$) , then $parent[x]$ is a tree node.
2. If x is a tree node (other than $start$) then $distance[x]$ is the shortest path, and $parent[x]$ is the predecessor along this path.
3. If f is a fringe node, then $distance[f]$ is the length of the shortest path, using only tree nodes, and $parent[x]$ is its parent along this path.

When the algorithm terminates, $finish$ is a tree node, therefore $distance[finish]$ is the shortest path by the second invariant. We just need to show all three invariants hold at the beginning and are maintained by the algorithm.

Initialisation: The first invariant holds, as the only tree node is $start$, which is exempt from the condition. The same reasoning applies to the second invariant. The only fringe nodes are those adjacent to $start$. It should be observed, that the only tree nodes is $start$ so the shortest distance, can only be from the arc between $start$ and that adjacent node. So the third invariant holds.

Maintenance: The first invariant holds, as it is only added to the tree, if $fringe[x]$ is TRUE. It is the case that $fringe[x]$ is initially false for all x, and is only set TRUE, when $parent[x]$ is set, to a tree node. The second invariant holds, as if it didn't, this would mean there is a shorter path. Suppose there was a shorter path, and y be the first node not to belong to the tree. the length of this path, must be at least as long than $distance[y]$, which must be at least as long as $distance[f]$. This is due to the 3rd invariant. So there cannot be a shorter path. The 3rd invariant is maintained, as when a node y is added to the fringe, it hasn't been in the fringe before, which means it isn't adjacent to any tree node. So the shortest path must involve f, and it is already known by 2nd invariant that $distance[f]$ is the shortest distance to f. Similarly, when another f node is added to the tree, the only case where there is a shorter path for a fringe node y, is if f is adjacent to y, which the algorithm will discover, and change $distance[y]$ to be equal to the length of the shortest path.

The algorithm can also be written to use priority queues. Dijkstra's algorithm with priority queues using binary heaps, take $O(m \log n)$ time, and with priority queues using fibonacci heaps takes $O(m + n \log n)$ time.

DIJKSTRA'S-ALGORITHM(G)

```
 1  Q = PQCreate()
 2  for x ∈ nodes(G)
 3      key[x] = ∞
 4      parent[x] = NULL
 5      Insert(Q, x)
 6  key[start] = 0
 7  while not tree[finish] and not Empty(Q)
 8      f = Get-Minimum(Q)
 9      Delete-Minimum(Q)
10      tree[f] = TRUE
11      for y ∈ adj[f]
12          if not tree[y]
13              if key[f] + W[f, y] < key[y]:
14              Decrease-Key(Q, y, key[f] + W[f, y])
15              parent[y] = f
```

Problem 2 ***(Travelling Salesman Problem)***

Given a complete weighted graph, find the Hamiltonian circuit of minimum weight.

This problem, like the Hamiltonian circuit problem is NP-complete.

BELLMAN-HELD-KARP'S ALGORITHM(G, W)

```
 1  Choose start ∈ nodes(G)
 2  for x ∈ nodes(G) \ {start}
 3      C[∅, x] = W[start, x]
 4  for S ⊆ nodes(G) \ {start} with S ≠ ∅
 5      for x ∈ nodes(G) \ (S ∪ {start})
 6          C[S, x] = ∞
 7          for y ∈ S
 8              C[S, x] = min(C[S \ {y}, y] + W[y, x], C[S, x])
 9  opt = ∞
10  for x ∈ nodes(G) \ start
11      opt = min(opt, C[nodes(G) \ {start, x}, x] + W[x, start])
12  return opt
```

There are $O(2^n)$ non-empty subsets and $O(n^2)$ work performed by each subset, so the algorithm takes time $O(n^2 2^n)$. The algorithm can be very easily modified to solve the Hamiltonian Circuit problem. One algorithm to approximate the Travelling Salesman Problem uses the nearest neighbour heuristic, by choosing the shortest available arc each time. However, this can lead to extremely sub-optimal solutions.

1.3 Algorithmic Analysis

Problem 3 ***(Searching Problem)***

Take a list L and a search item x, and return an index i such that $L[i] = x$, or -1 if there is no i such that $L[i] = x$.

LINEAR-SEARCH(L, x)

```
1  for i = 0 to L.length − 1
2      if L[i] = x
3          return i
4  return -1
```

The linear search algorithm takes n comparisons in the worst case and 1 comparison in the best case.

Proposition 1.3.1 *Linear search is optimal in the worst case in the number of comparisons.*

Proof: Given a search item x, and a list L which doesn't contain x, suppose there was a way to conclude that L doesn't contain x without inspecting a certain index i. Then passing a modified version of the list which does contain x at index i would lead to the algorithm giving the wrong output.

Problem 4 ***(Ordered Searching Problem)***

Take an ordered list L and a search item x, and return an index i such that $L[i] = x$, or -1 if there is no i such that $L[i] = x$.

A modified version of the linear search algorithm is:

MODIFIED-LINEAR-SEARCH(L, x)

```
1  for i = 0 to L.length − 1
2      if L[i] = x
3          return i
4      else if L[i] > x
5          break
6  return -1
```

This still performs n comparisons in the worst case. A better algorithm is binary search.

BINARY-SEARCH(L, x)

```
1   l = 0
2   u = L.length − 1
3   while l ≤ u
4       i = ⌊(l + u)/2⌋
5       if L[i] < x
6           l = l + 1
7       else if L[i] == x
8           return i
9       else
10          u = u − 1
11  return -1
```

The recurrence relation for the number of comparisons in binary search is:

$$W(1) = 1$$
$$W(n) = 1 + W(\lfloor n/2 \rfloor)$$

Solving this recurrence gives:

$$W(n) = 1 + \lfloor \log n \rfloor$$

Proposition 1.3.2

$$\lceil \log(n+1) \rceil = 1 + \lfloor \log n \rfloor$$

Proposition 1.3.3 *A binary tree of depth d can have at most $2^{d+1} - 1$.*

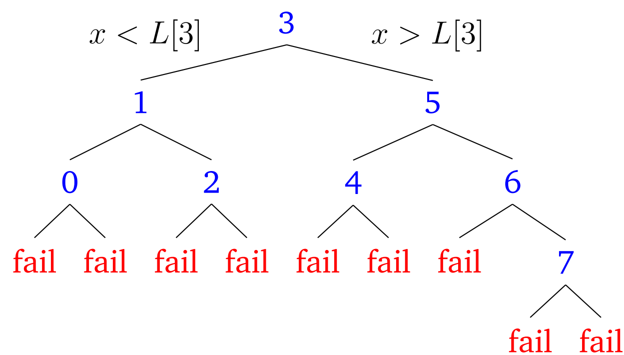
Proof: Base Case: The proposition holds for a tree of depth 0.

Inductive Case: Suppose the proposition holds for a tree of depth k. A tree of depth $k+1$, has a root node, and two subtrees, which can have at most $2^{k+1} - 1$ nodes. So the overall tree can have at most $1 + 2(2^{k+1} - 1)$ nodes, which is the same as $1 + 2^{k+2} - 2$, i.e. $2^{k+2} - 1$ or $2^{(k+1)+1} - 1$ So the proposition holds.

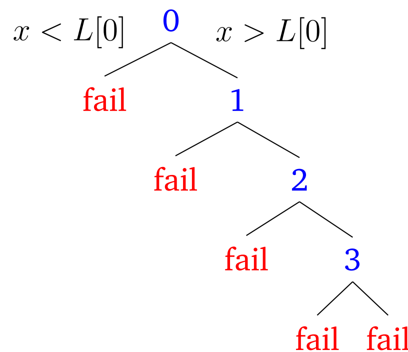
Proposition 1.3.4 *Binary search is optimal in the number of comparisons in the worst case.*

Proof: Any series of comparisons can be represented as a binary decision tree. The tree must have n nodes. A binary tree with depth d can have at most $2^{d+1} - 1$ nodes. Therefore, the smallest tree which can store n nodes, must have depth $1 + \lfloor \log n \rfloor$. It can be shown by rearranging that $d + 1 \geq \log(n+1)$, and since $d + 1$ is an integer, $d + 1 \geq \lceil \log(n+1) \rceil$, which is the same as $d + 1 \geq 1 + \lfloor \log n \rfloor$. So $d + 1 \geq 1 + \lfloor \log n \rfloor$. A comparison must be made for each 'level' of the tree. Observe that the number of 'levels' is one more than the tree's depth i.e. $d + 1$. So at least $1 + \lfloor \log n \rfloor$ must be made.

Here is a decision tree for binary search where $n = 8$:



Whereas modified linear search for $n = 4$ would like this:

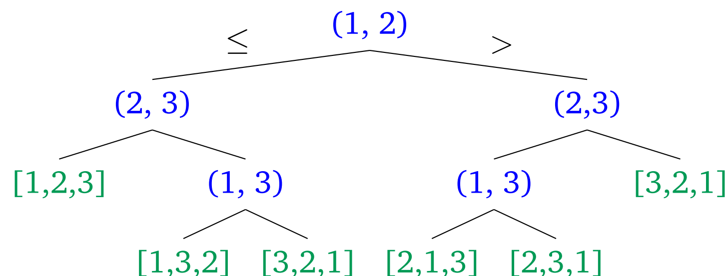


Proposition 1.3.5 *A tree of depth d has at most 2^d leaves.*

Proof: Base Case: Consider a tree of depth 0. This has 1 leaf. So the proposition holds.

Inductive Case: Assume the proposition holds for a trees of depth k. Then a tree of depth $k + 1$, has a root and 2 trees of depth at most k, so has at most $2(2^k)$ leaves, i.e. 2^{k+1}, so the proposition holds.

Here is a decision tree for a list of length 3:



Proposition 1.3.6 *An algorithm for the sorting problem must perform at least $\lceil \log n! \rceil$ comparisons.*

Proof: There are $n!$ permutations of a list of length n that must be differentiated against. This means that there are $n!$ leaves. A tree of depth d has 2^d leaves. So $2^d \geq n!$. So

$d \geq \log(n!)$. Since d is an integer, $d \geq \lceil \log(n!) \rceil$. Since a comparison must be made at every level of the tree, at least $\lceil \log(n!) \rceil$ comparisons must be made.

Definition 1.3.1 *The total path length of a tree is the sum of the depths of all the leaf nodes.*

If a decision tree has total path length b and $n!$ leaves, then the average number of comparisons is $b/n!$. It is the case that the total path length is lowest when leaves are at roughly equal depth.

Definition 1.3.2 *A tree of depth d is balanced if every leaf is at depth d or $d - 1$.*

Proposition 1.3.7 *If a tree is unbalanced, one can find a balanced tree with the same number of leaves, without increasing total path length.*

Proposition 1.3.8 *Any algorithm for sorting a list of length n must perform at least $\lfloor \log(n!) \rfloor$ comparisons in the average case.*

Proof: This is as in a balanced tree of depth d leaves are at depth d or $d-1$. This is as if the decision tree isn't balanced, then a balanced one can be found with a shorter total path length, hence a better average case.

Proposition 1.3.9 *The standard matrix multiplication algorithm performs $\Theta(n^3)$ multiplications, whereas Strassen's performs $\Theta(n^{\log 7})$.*

INSERTION-SORT(A)

1 **for** $i = 1$ **to** $A.length - 1$
2 $\quad key = A[i]$
3 $\quad j = i - 1$
4 $\quad$ **while** $j > 0$ and $A[j] > key$
5 $\quad\quad A[j+1] = A[j]$
6 $\quad\quad j = j - 1$
7 $\quad A[j+1] = key$

Proposition 1.3.10 *In the average and worst case, insertion sort is bounded by $\Theta(n^2)$.*

Proof: Notice in insertion sort, the total number of comparisons of the while loop on line 4, depends on the number of inversions in the sequence. In the worst case, there are $\frac{n(n-1)}{2}$ inversions, and so in the average case, there are $\frac{n(n-1)}{4}$ inversions.

```
MERGE(A, l, u)
 1  middle = ⌊(l + u)/2⌋ − l
 2  len = u − l + 1
 3  counterA = 0
 4  counterB = middle
 5  B = A[l..u]
 6  for i = l to len
 7      if counterA = middle ∨ (counterB ≠ len ∧ A[counterB + l] < A[counterA + l])
 8          A[i] = B[counterB]
 9      else
10          A[i] = B[counterA]
```

```
MERGE-SORT(A, l, u)
1  if l ≥ u − 1
2      return
3  middle = ⌊(l + u)/2⌋
4  MERGE-SORT(A, l, middle)
5  MERGE-SORT(A, middle + 1, u)
6  MERGE(A, l, u)
```

The recurrence relation for merge sort is:

$$\begin{aligned} W(1) &= 0 \\ W(n) &= n - 1 + W(\lceil n/2 \rceil) + W(\lfloor n/2 \rfloor) \end{aligned}$$

Solving this recurrence by assuming $n = 2^k$, gives us:

$$W(n) = n \log n - n + 1$$

Note that the recurrence, like many recurrences can be solved by unrolling, and assuming the input size is a convenient power.

Thus merge sort takes time $\Theta(n \log n)$. This is of the same order as $\log(n!)$, and can be shown by integrating:

$$\begin{aligned} \int_1^n \ln x \; dx &= [x \ln x - x]_1^n \\ &= n \ln n - n + 1 \\ &= \Theta(n \log n) \end{aligned}$$

```
SPLIT(A, l, u)
 1  d = A.[l]
 2  i = l + 1
 3  j = u
 4  while i ≤ j
 5      if L[i] ≤ d
 6          i = i + 1
 7      else
 8          SWAP(i, j)
 9          j = j − 1
10  SWAP(l, j)
```

```
QUICKSORT(A, l, u)
1  if l ≥ u − 1
2      return
3  middle = SPLIT(A, l, u)
4  QUICKSORT(A, l, pivot)
5  QUICKSORT(A, pivot + 1, u)
```

The worst case for quicksort is where the pivot is at either end. This leads to the following recurrence:

$$W(1) = 0$$
$$W(n) = n - 1 + W(n-1)$$

The average case recurrence for quicksort is:

$$A(n) = n - 1 + \frac{1}{n}\sum_{s=1}^{n}(A(s-1) + A(n-s))$$

This is the same as:

$$A(1) = 0$$
$$A(n) = n - 1 + \frac{2}{n}\sum_{s=2}^{n-1} A(s)$$

Proposition 1.3.11 *It can be shown that the average number of comparisons for quicksort is* $\Theta(n \log n)$.

Proposition 1.3.12

- *Sum of a geometric series:*

$$\sum_{i=0}^{k} ar^i = \frac{a(r^{k+1} - 1)}{r - 1}$$

- *Therefore, if $r \neq 1$, r is non-negative and r doesn't depend on n, then:*

$$\sum_{i=0}^{k} ar^i = \Theta(t(n))$$

Where $t(n)$ is the largest term in the geometric progression.

Proposition 1.3.13 ***(Master Theorem)***

A recurrence in the form $T(n) = aT(n/b)+f(n)$, has a solution where $E = \log_b a = \log a/\log b$ is the critical exponent:

- *If $f(n) = \Theta(n^E)$, then $T(n) = \Theta(f(n)\log n)$*
- *If $f(n) = O(n^{E-\epsilon})$, for some $\epsilon > 0$, then $T(n) = \Theta(n^E)$.*
- *If $n^{E+\epsilon} = O(f(n))$ for some $\epsilon > 0$, then $T(n) = \Theta(f(n))$*

Definition 1.3.3

- *Heaps are left-complete balanced binary trees, i.e. no node is messing left to a node that is present.*
- *A tree is a minimising partial order tree if the key at any node is less than or equal to that of any children.*
- *A tree is a maximising partial order tree is the key at any node is greater than or equal to that of any children.*
- *A min heap is a heap which is a minimising partial order tree, and a max heap is a heap which is a maximising partial order tree.*

Heaps can be implemented in arrays, where the root is stored at index 0, and the left child of any node at index n is stored at index $2n$ and the right child of any node is stored at index $2n + 1$. This means the parent of any node is stored at index $\lfloor n/2 \rfloor$. The insertion algorithm for a min-heap is as follows:

INSERT(A, x)

1 $A[size] = x$
2 $size = size + 1$
3 PERCOLATE-UP$(A, size)$

PERCOLATE-UP(A, x)

```
1  if x ≥ 1
2      parent = ⌊x/2⌋
3      if A[x] < A[parent]
4          SWAP(A, x, parent)
5          PERCOLATE-UP(A, parent)
```

This is $\Theta(\log n)$. To implement DECREASE-KEY, each element is given an identifier from 0 to $n-1$, and a supplementary node stores the position of the elements by their identifier. Then PERCOLATE-UP can be used to maintain the heap property. This would be $\Theta(\log n)$. Heapsort is an $\Theta(n \log n)$. in-place sorting algorithm. This implementation assumes a max heap:

HEAPSORT(A)

```
1  while size ≠ 0
2      SWAP(A, 0, size)
3      size = size − 1
4      FIX-MAX-HEAP(A)
```

FIX-MAX-HEAP$(A, root)$

```
 1  left = 2 ∗ root
 2  right = 2 ∗ root + 1
 3  if left ≤ size
 4      if left = size ∨ A[left] > A[right]
 5          larger = left
 6      else
 7          larger = right
 8      if A[root] < A[larger]
 9          SWAP(A, root, larger)
10          FIX-MAP-HEAP(A, larger)
```

Here is an algorithm to build a max-heap from an array:

BUILD-MAX-HEAP$(A, root)$

```
1  BUILD-MAX-HEAP(A, 2 ∗ root)
2  BUILD-MAX-HEAP(A, 2 ∗ root + 1)
3  FIX-MAP-HEAP(A, root)
```

FIX-MAX-HEAP performs $\Theta(\log n)$ comparisons and BUILD-MAX-HEAP performs $\Theta(n)$ comparisons.

Problem 5 ***(The word break problem)***

Can a string of characters be broken down into words contained in the dictionary, for instance, "windown" can be broken up into "win down" and "wind own".

A recursive solution, with exponential time complexity:

```
WORD-BREAK(s)
1  if s.length = 0
2      return TRUE
3  for i = 0 to s.length − 1
4      if IN-DICTIONARY(s[0..i]) ∧ WORD-BREAK(s[(i + 1)..])
5          return TRUE
6  return FALSE
```

A solution that uses memoisation, a type of dynamic programming:

```
WORD-BREAK2(s)
 1  if s.length = 0
 2      return TRUE
 3  for i = 1 to s.length − 1
 4      if memo[s] ≠ UNDEFINED
 5          return memo[s]
 6      if IN-DICTIONARY(s[0..i]) ∧ WORD-BREAK2(s[(i + 1)..])
 7          memo[s] = TRUE
 8          return TRUE
 9  memo[s] = FALSE
10  return FALSE
```

This solution has time complexity $\Theta(n^2)$. Here is a solution that uses bottom-up dynamic programming, which also has time complexity $\Theta(n^2)$:

```
WORD-BREAK3(s)
 1  if s.length = 0
 2      return TRUE
 3  wb[0] = TRUE
 4  for i = 1 to s.length
 5      wb[i] = FALSE
 6      for j = 0 to (i − 1)
 7          if wb[j] ∧ IN-DICTIONARY(s[j..(i + 1)])
 8              wb[i] = TRUE
 9              break
10  return wb[n]
```

Dynamic programming breaks down a problem into smaller subproblems that are computed only once. The results of these subproblems are cached, to later compute the bigger problems. The principal advantage of top-down memoisation is if not all subproblems need computing. But a bottom-up non-recursive approach avoids the overheads of recursion.

1.4 Complexity

Proposition 1.4.1 ***(Cook-Karp Thesis)***

A problem is tractable if it can be computed within polynomially many steps in the worst case.

Proposition 1.4.2 *Deciding whether a graph has an Euler path is tractable.*

Proof: This is as it is only necessary to check whether the graph has 0 or 2 odd-degree nodes.

Models of computations are dependent on two main things: what constitutes input size; and what constitutes a computation step?

Proposition 1.4.3 ***(Polynomial Invariance Thesis)***

If a problem can be solved in polynomial time in some reasonable model of computation, then it can be solved in polynomial time, in any other reasonable model of computation.

Examples of unreasonable models of computation are superpolynomial parallelism and unary numbers.

Definition 1.4.1

- *A decision problem $D(x)$ is a function with a binary output i.e. 'yes'/'no' or 'true'/'false'.*
- *A decision problem is decidable in polynomial time if there is an algorithm A that runs in polynomial time to compute it, i.e. in $p(n)$, where n is the input size.*
- *The complexity class P is the set of decision problems that are decidable in polynomial time.*
- *The complexity class NP is the set of decision problems whose solutions are verifiable in polynomial time given some certificate, i.e. there exists another decision problem $E(x, y)$ in P where $D(x) \leftrightarrow \exists y E(x, y)$, and if $E(x, y)$, then $|y| \leq p(|x|)$, for some polynomial p, where y is the certificate.*

Arithmetical operations take polynomial time, where the input size is $\log n$, in terms of a number n.

Proposition 1.4.4 *The Hamiltonian path problem is in* NP.

Proof: This is since the certificate is a Hamiltonian path, whose size must be bounded by a polynomial of the input size.

Proposition 1.4.5 *A polynomial time function will produce output that is bounded in size, by a polynomial function of its input.*

Proof: This is as the algorithm only has polynomial time to build its input.

Proposition 1.4.6 *If f and g are computable in polynomial time, $g \circ f$ is computable in polynomial time.*

Proof: Suppose f is bounded by $p(x)$ and g by $q(x)$. The output of f is bounded by some polynomial $p'(x)$, so the total running time is $p(x) + q(p'(x))$, which is a polynomial in x.

Proposition 1.4.7 *If a decision problem is in* P*, then it is in* NP*.*

Proof: To do this the certificate y is always set to the empty string ϵ, so $D(x) \leftrightarrow E(x, \epsilon)$. So, $D(x) \leftrightarrow \exists y E(x, y)$ always holds and so does $|y| \leq p(|x|)$ holds, if $p(x) = 0$.

Although it is clear $\mathsf{P} \subseteq \mathsf{NP}$, it is not known where $\mathsf{P} \neq \mathsf{NP}$ (i.e. $\mathsf{P} \subsetneq \mathsf{NP}$) or $\mathsf{P} = \mathsf{NP}$.

Definition 1.4.2

- *A decision problem D reduces to a decision problem D', denoted as $D \leq D'$, if there is a reduction function f computable in polynomial time, such that $D(x) \leftrightarrow D'(f(x))$.*
- *A decision problem is NP-hard if for all $D' \in \mathsf{NP}$, we have $D' \leq D$.*
- *A decision problem is NP-complete if:*
  - $D \in \mathsf{NP}$
  - *D is NP-hard*

Observe that NP-hard problems could be harder than NP*. To prove a problem is NP-complete, the usual strategy is to show that it is in* NP *and that some NP-complete problem can be reduced to it, which establishes that it is NP-hard.*

Proposition 1.4.8 *If D' is decidable in polynomial time, and $D \leq D'$, then D is decidable in polynomial time.*

Proof: If $D \leq D'$, then there is a function computable in polynomial time by an algorithm A', $f(x)$, such that $D'(f(x)) \leftrightarrow D(x)$, so an algorithm A to compute $D(x)$ in polynomial time, is to first compute $f(x)$, and then to compute A' on the output of $f(x)$. Since composition of two functions computable in polynomial time, results in a function computable in polynomial time, A is a function computable in polynomial time.

Proposition 1.4.9 *If D' is in* NP*, and $D \leq D'$, then $D \in \mathsf{NP}$.*

Proof: If D' is in NP*, then there exists $E'(x, y)$ computable in polynomial time, such that $D'(x) \leftrightarrow \exists y E'(x, y)$ and $E'(x, y) \to |y| < p(|x|)$ for some polynomial p.To show the same for D, define $E(x, y) \leftrightarrow E'(f(x), y)$. Observe, the E is computable in polynomial time, since it is in essence the composition of two polynomial time functions. In addition, since $D \leq D'$, if $D(x)$, this means $D'(f(x))$, i.e. $\exists y E'(f(x), y)$, which means $\exists y E(x, y)$. And if $E(x, y)$, this implies $E'(f(x), y)$, so $|y| < p(|f(x)|)$ for some polynomial p. Since f is bound by a polynomial, $q(x)$, $|y| < p(q(|x|))$, i.e. $|y| < p'(|x|)$, where $p' = q \circ p$, which is a polynomial.*

Proposition 1.4.10 *The reduction order, $\leq$ is reflexive and transitive.*

Proof: The reduction order is reflexive, as the identity function can be used to reduce one decision problem into itself. The reduction order is transitive, as if $D \leq D'$ for a reduction function f, and $D' \leq D''$ for some reduction function f', then $D \leq D''$ for reduction function $f'' = f' \circ f$, which is computable in polynomial time as it is the composition of two functions computable in polynomial time.

Problem 6 ***(SAT)***

Given a formula ϕ in propositional logic in conjunctive normal form, is ϕ satisfiable?

Proposition 1.4.11 ***(Cook-Levin Theorem)***

SAT is NP-complete.

Problem 7 ***(Hamiltonian path problem)***

The Hamiltonian path decision problem is a generalised version of the Hamiltonian circuit problem. It takes a graph G, outputs whether the graph has a Hamiltonian path.

Observe that this problem can be reduced into the Hamiltonian circuit problem, by adding an additional node, and connecting it to each existing node.

Proposition 1.4.12 *SAT reduces to the Hamiltonian path problem, and since we have already shown that the Hamiltonian path problem is in* NP*, this means it is NP-complete.*

Proposition 1.4.13 *If* $\mathsf{P} \neq \mathsf{NP}$*, then for any NP-hard problem D, $D \notin \mathsf{P}$.*

Proof: Assuming $P \neq NP$, then suppose $D \in \mathsf{P}$. Since for any $D' \in \mathsf{NP}$, $D' \leq D$, this means $D' \in \mathsf{P}$. So $\mathsf{NP} \subseteq \mathsf{P}$. Since it is known $\mathsf{P} \subseteq \mathsf{NP}$, then $\mathsf{P} = \mathsf{NP}$, which contradicts our assumption.

Therefore, if we show a problem is NP-complete, it is intractable, assuming $\mathsf{P} \neq \mathsf{NP}$.

Problem 8 ***(Metric Travelling Salesman Problem)***

The metric travelling salesman problem is restricted to graphs satisfying the triangle inequality where:

$$W(x, z) \leq W(x, y) + W(y, z)$$

Problem 9 ***(Decision Variant of the (Metric) Travelling Salesman Problem)***

This is to find if a solution exists to the (metric) travelling salesman problem, where the solution doesn't exceed a bound B.

Proposition 1.4.14 *The metric travelling salesman problem is NP-hard, in particular, the Hamiltonian path problem reduces to the decision problem variant of the metric travelling salesman problem.*

Proof: If an arc (x, y) is an arc of G, then let (x, y) be an arc of G', with $W(x, y) = 1$. Otherwise let it be an arc of G' with $W(x, y) = 2$. Clearly, the triangle inequality will always hold. Whether or not a tour that includes every node, is equivalent to if there is a solution to the metric travelling salesman problem where the bound is $n + 1$, where n is the number of nodes.

Proposition 1.4.15 *The travelling salesman problem is NP-hard, in particular, the Hamiltonian path problem reduces to the decision problem variant of the travelling salesman problem.*

Proof: This is trivial since the Hamiltonian path problem reduces to the metric travelling salesman problem, which trivially reduces to the travelling salesman problem.

Proposition 1.4.16 *The (metric) travelling salesman problem is NP-complete.*

Proof: It has already been shown that this problem is NP-hard, so it only needs to be shown that it is in NP. This can be seen since the certificate is a path that starts and ends at the same node and visits each city, keeping within the necessary bound. This can be checked in polynomial time.

Problem 10 ***(Vehicle Routing Problem with Capacities)***

Given a weighted graph satisfying the triangle inequality, and a distinguished node $start$ (the depot) and k vehicles with capacity C, and a set of packages of sizes $s_1, \ldots, s_n$ to be delivered to nodes $x_1, \ldots, x_n$ and a budget B, can all packages be delivered within budget B, subject to the total size of packages on each vehicle being less than capacity C. All vehicles must return to the original depot, with each vehicle performing at most one trip.

Proposition 1.4.17 *The vehicle routing problem with capacities is NP-hard.*

Proof: It can be shown that the metric travelling salesman problem (MTSP) reduces into the vehicle routing problem with capacities (VRPC), i.e. MTSP $\leq$ VRPC. The reduction function f:

- Keeps the graph (G, W) the same and keep the bound B the same.
- Makes one node the depot.
- Assigns one package with size 1 to each of the remaining nodes.
- Create one vehicle with capacity $n - 1$.

The reduction function f is computable in polynomial time, and a 'yes' instance to the VRPC under those conditions above, means that it is possible to visit all nodes at least once and return to the start under bound B. In other words:

$$\mathsf{VRPC}(f((G, W), B)) \leftrightarrow \mathsf{MTSP}((G, W), B)$$

Suppose VRPC$(f((G, W), B))$, then there is a path that is a travelling salesman tour, except nodes may be visited more than once. In this case, these nodes can be skipped, and lead to a route just as small if not smaller, due to the triangle inequality. Hence, MTSP$((G, W), B)$.

Alternatively, MTSP$((G, W), B)$, means it is possible to begin at $start$ and visit all nodes, returning back to $start$ within B. This same tour will satisfy VRPC$(f((G, W), B))$, since all nodes need to be visited once, there is only one vehicle and the packages do not exceed vehicle capacity.