T F A greedy algorithm for a problem can never give an optimal solution on all inputs.

Solution: False. Prim's algorithm for minimum spanning trees is a counter-example: it greedily picks edges to cross cuts, but it gives an optimal solution on all inputs.

T F Suppose we have computed a minimum spanning tree of a graph and its weight. If we make a new graph by doubling the weight of every edge in the original graph, we still need $\Omega(E)$ time to compute the cost of the MST of the new graph.

Solution: False. Consider sorting the edges by weight. Doubling the weights does not change the sorted order. But this means that Kruskal's and Prim's algorithm will do the same thing, so the MST is unchanged. Therefore the weight of the new tree is simply twice the weight of the old tree and can be computed in constant time if the original MST and weight are known.

T F 2-3-4 trees are a special case of B-trees.

Solution: True. They are the case when $t = 2$.

T F You have n distinct elements stored in an augmented red-black tree with an extra field per node containing the number of elements in the subtree rooted at that node (including itself). The rank of an element is the number of elements with value less than or equal to it (including itself). The best algorithm for finding the rank of a given element runs in linear time.

Solution: We can augment a red-black tree to support order-statistic queries in $O(\log n)$ time. (For each node, keep track of the number of nodes in it's subtree. This information is easily maintained on rotation, insert, and delete, and is good enough to search for an order-statistic.)

T F Let $G = (V, E)$ be a connected, undirected graph with edge-weight function $w : E$ to reals. Let $u \in V$ be an arbitrary vertex, and let $(u, v) \in E$ be the least-weight edge incident on u; that is, $w(u, v) = \min\{w(u, v') : (u, v') \in E\}$. (u, v) belongs to some minimum spanning tree of G.

Solution: True. Consider any MST. Assume it doesn't contain (u, v), or else we're done. Since it is a spanning tree, there must be a path from u to v. Let (u, x) be the first edge on this path. Delete (u, x) from the tree and add (u, v). Since (u, v) is a least weight-edge from u, we did not increase the cost, so as long as we still have a spanning tree, we still have a minimum spanning tree. Do we still have a spanning tree? We broke the tree in two by deleting an edge. So did we reconnect it, or add an edge in a useless place? Well, there is only one path between a given two nodes in a tree, so u and v were separated when we deleted e. Thus our added edge does in fact give us back a spanning tree.

Problem -2. Minimum and Maximum Spanning Trees

(a) It can be shown that in any minimum spanning tree (of a connected, weighted graph), if we remove an edge (u, v), then the two remaining trees are each MSTs on their respective sets of nodes, and the edge (u, v) is a least-weight edge crossing between those two sets.

These facts inspire the 6.046 Staff to suggest the following algorithm for finding an MST on a graph $G = (V, E)$: split the nodes arbitrarily into two (nearly) equal-sized sets, and recursively find MSTs on those sets. Then connect the two trees with a least-cost edge (which is found by iterating over E).

Would you want to use this algorithm? Why or why not?

Solution: This algorithm is actually *not correct*, as can be seen by the counterexample below. The facts we recalled are essentially irrelevant; it is their *converse* that we would need to prove correctness. Specifically, it *is* true that every MST is the combination of two sub-MSTs (by a light edge), but it is *not* true that every combination of two sub-MSTs (by a light edge) is an MST. In other words, it is not safe to divide the vertices arbitrarily.

A concrete counterexample is the following: vertices A, B, C, D are connected in a cycle, where $w(A, B) = 1$, $w(B, C) = 10$, $w(C, D) = 1$, and $w(D, A) = 10$. A minimum spanning tree consists of the first three edges and has weight 12. However, the algorithm might divide the vertices into sets $\{B, C\}$ and $\{A, D\}$. The MST of each subgraph has weight 10, and a light edge crossing between the two sets has weight 1, for a total weight of 21. This is not an MST.

(b) Consider an undirected, connected, weighted graph G. Suppose you want to find the *maximum* spanning tree. Give an algorithm to find the maximum spanning tree.

Solution: You can make a copy of G called G' in which the weight of every edge is replaced with its negation (i.e. $w(u, v)$ becomes $-w(u, v)$) and compute the minimum spanning tree $MST(G')$ of the resulting graph using Kruskal's algorithm (or any other MST algorithm). The resulting tree corresponds to a maximum weight spanning tree for the original graph G. Note that if it were not a maximum weight spanning tree, then the actual maximum weight spanning tree of G corresponds to a minimum spanning tree with less weight than $MST(G')$, which is a contradiction.

Problem -3. Finding Shortest Paths

We would like solve, as efficiently as possible, the single source shortest paths problem in each of the following graphs. (Recall that in this problem, we must find a shortest path from a designated source vertex to every vertex in the graph.) For each graph, state which algorithm (from among those we have studied in class and section) would be best to use, and give its running time. If you don't know the name of the algorithm, give a *brief* description of it. If the running time depends on which data structures are used in the implementation, choose an arbitrary data structure.

(a) A weighted directed acyclic graph.

Solution: First Topological Sort and then dynamic programming (Refer to DAG-SHORTEST-PATHS on Page 536 in CLR) - $O(V + E)$

(b) A weighted directed graph where all edge weights are non-negative; the graph contains a directed cycle.

Solution: Dijkstra' Algorithm with Fibonacci Heaps - $O(E \lg V)$

(c) A weighted directed graph in which some, but not all, of the edges have negative weights; the graph contains a directed cycle.

Solution: Bellman-Ford's Algorithm - $O(VE)$

Problem -4. Roots of a graph

A **root** of a directed graph $G = (V, E)$ is a node r such that every other node $v \in V$ is reachable from r.

(a) [3 points] Give an example of a graph which does not have a root.

Solution:
Refer Figure 1

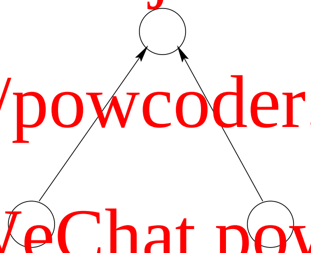


Figure 1: Problem 4-(a)

(b) [10 points] Prove the following claim.

Consider the forest constructed by running depth-first-search (DFS) on the graph G. Let T be the last tree constructed by DFS in this forest and let r be the root of this tree (i.e, DFS constructed T starting from the node r.) If the graph G has a root, then r is a root of G.

Solution: None of the nodes in the trees other than T of the forest constructed by DFS reach r (or r would not be the root of a new tree). If any node in T is a root of the graph, so is r. Thus proved.

(c) [7 points] Using the result proved in **(b)**, give an $O(|V|+|E|)$-algorithm which when given a graph G, finds a root of the graph if one exists, and outputs NIL otherwise.

Solution: From above, we know that if G has a root, then the root of the last tree output by DFS must be a root. Thus, we first run DFS and then check whether the last root r reaches out to all nodes. This can be done by doing DFS again starting with r and checking if the forest has only one root. Clearly, this algorithm takes $O(|V|+|E|)$ time.

```
FIND-ONE-ROOT$(G)$
    for each vertex $u \in V[G]$
        do color$[u] \leftarrow$ WHITE
    for each vertex $u \in V[G]$
        do if color$[u] =$ WHITE
            then last-root $\leftarrow u$          $\triangleright$ store current root
                 DFS-VISIT$(u)$
    $r \leftarrow$ last-root
    TEST-ROOT$(r)$
```

We test whether r is a root by recoloring all nodes white, calling DFS-VISIT(r) and then verifying that every node in G is black. This will show that every node is reachable from r.

(d) [5 points] Suppose you are a given a root r of a graph G. Give an $O(|V|+|E|)$-algorithm to find all the roots of the graph.
(Hint: u is a root of G iff r is reachable from u).

Solution: We now find all other roots as follows. A node u is a root iff u can reach r. This is because if u is a root then it reaches r and conversely, if u reaches r then it reaches every other node as well. We can find all nodes that reach r by reversing the edges of G and then starting a DFS from r in the reversed graph.

```
FIND-ALL-ROOTS$(G)$
    REVERSE-GRAPH$(G)$
    for each vertex $u \in V[G]$
        do color$[u] \leftarrow$ WHITE
    DFS-VISIT$(r)$
    for each vertex $u \in V[G]$
        do if color$[u] =$ BLACK
            then OUTPUT$(u)$          $\triangleright$ $u$ is a root
```

It is easy to see that the procedure REVERSE-GRAPH(G) takes $O(V+E)$ time. Therefore, the algorithm to find all roots takes $O(V+E)$ as well.

Problem -5. Test-Taking Strategies

Consider (if you haven't already!) a quiz with n questions. For each $i = 1, \ldots, n$, question i has integral point value $v_i > 0$ and requires $m_i > 0$ minutes to solve. Suppose further that no partial credit is awarded (unlike this quiz).

Your goal is to come up with an algorithm which, given $v_1, v_2, \ldots, v_n, m_1, m_2, \ldots, m_n$, and V, computes the minimum number of minutes required to earn at least V points on the quiz. For example, you might use this algorithm to determine how quickly you can get an A on the quiz.

(a) Let $M(i, v)$ denote the minimum number of minutes needed to earn v points when you are restricted to selecting from questions 1 through i. Give a recurrence expression for $M(i, v)$.

We shall do the base cases for you: for all i, and $v \leq 0$, $M(i, v) = 0$; for $v > 0$, $M(0, v) = \infty$.

Solution: Because there is no partial credit, we can only choose to either do, or not do, problem i. If we do the problem, it costs m_i minutes, and we should choose an optimal way to earn the remaining $v - v_i$ points from among the other problems. If we don't do the problem, we must choose an optimal way to earn v points from the remaining problems. The faster choice is optimal. This yields the recurrence:

$$M(i, v) = \min \{m_i + M(i - 1, v - v_i), M(i - 1, v)\}$$

(b) Give pseudocode for an $O(nV)$-time dynamic programming algorithm to compute the minimum number of minutes required to earn V points on the quiz.

Solution:

```
FASTEST($\{v_i\}, \{m_i\}, V$)
  ▷ fill in $n \times V$ array for $M$; base case first
  for $v \leftarrow 1$ to $V$
    $M[0, v] \leftarrow \infty$
  ▷ now fill rest of table
  for $i \leftarrow 1$ to $n$
    for $v \leftarrow 1$ to $V$
      ▷ compute first term of recurrence
      if $v - v_i \leq 0$
        $a \leftarrow m_i$
      else
        $a \leftarrow m_i + M[i-1, v - v_i]$
      ▷ fill table entry $i, v$
      $M[i, v] \leftarrow \min\{a, M[i-1, v]\}$
  return $M[n, V]$
```

Filling in each cell takes $O(1)$ time, for a total running time of $O(nV)$. The entry $M[n, V]$ is, by definition, the minimum number of minutes needed to earn V points, when all n problems are available to be answered. This is the quantity we desire.

(c) Explain how to extend your solution from the previous part to output a list S of the questions to solve, such that $V \leq \sum_{i \in S} v_i$ and $\sum_{i \in S} m_i$ is minimized.

Solution: In each cell of the table containing value $M(i, v)$, we keep an additional bit corresponding to whether the minimum was $m_i + M(i-1, v - v_1)$ or $M(i-1, v)$, i.e. whether it is fastest to do problem i or not. After the table has been filled out, we start from the cell for $M(n, V)$ and trace backwards in the following way: if the bit for $M(i, v)$ is set, we include i in S and go to the cell for $M(i-1, v - v_i)$; otherwise we exclude i from S and go to the cell for $M(i-1, v)$. The process terminates when we reach the cell for $M(i, v')$ for some $v' \leq 0$.

It is also possible to do this without keeping the extra bit (just by looking at both possibilities and tracing back to the smallest one).

(d) Suppose partial credit is given, so that the number of points you receive on a question is proportional to the number of minutes you spend working on it. That is, you earn v_i/m_i points per minute on question i (up to a total of v_i points), and you can work for fractions of minutes. Give an $O(n \log n)$-time algorithm to determine which questions to solve (and how much time to devote to them) in order to receive V points the fastest.

Solution: A greedy approach works here (as it does in the fractional knapsack problem, although the problems are not identical). Specifically, we sort the problems in descending value of v_i/m_i (i.e., decreasing "rate," or "bang for the buck"). We then iterate over the list, choosing to do each corresponding problem until V points are accumulated (so only the last problem chosen might be left partially completed). The running time is dominated by the sort, which is $O(n \log n)$. Correctness follows by the following argument: suppose an optimal choice of problems only did part of problem j and some of problem k, where $v_j/m_j > v_k/m_k$. Then for some of the points gained on problem k, there is a faster way to gain them from problem j (because j hasn't been completed). This contradicts the optimality of the choice that we assumed. Therefore it is safe to greedily choose to do as much of a remaining problem having highest "rate" as needed.

1. Kruskal's algorithm for solving the Minimum Spanning Tree Problem is

    1. **an optimal and efficient algorithm**
    2. an optimal and inefficient algorithm
    3. an approximate and efficient algorithm
    4. an approximate and inefficient algorithm
    5. None of these

2. Which of the following are true of a tree?

    1. any graph that is connected and has bridges
    2. any graph that is connected and has no bridges
    3. **any graph that is connected and has no circuits**
    4. any graph that has no circuits
    5. None of the above

3. If a tree has 98 edges, then the number of its vertices is:

    1. 97
    2. **99**
    3. 100
    4. 98
    5. None of these

4. Prim's algorithm applied to in Figure 1, starting at C yields the following sequence of edges:

    1. CD, CB, CA
    2. CD, CA, CB
    3. CA, CD, CB
    4. **CA, CB, CD**
    5. None of the above

5. How many spanning trees does the graph below has?:

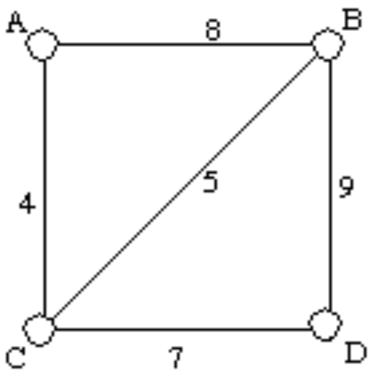


Figure 1

1. 6
2. **8**
3. 10
4. 4
5. None of these

6. For finding minimum spanning trees, Kruskal's algorithm is:

    1. efficient
    2. exact
    3. greedy
    4. a & b
    5. a & c
    6. b & c
    7. **all of the above**
    8. none of the above

7. Every graph with N vertices and N - 1 edges is a tree.
    1. True
    2. **False**

8. Use Kruskal's algorithm to find a minimum spanning tree of the given graph below. Draw the resulting spanning tree and list the edges in the order they are picked by Kruskal's algorithm.

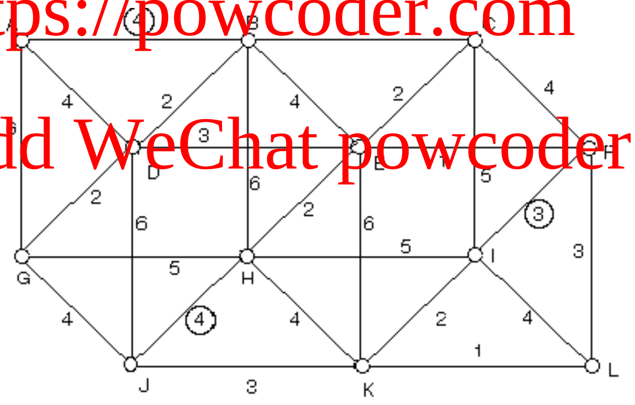


Figure 2

Group 1 (edges with weight 1)

KL, BC, EF all three in any (random) order

Group 2 (edges with weight 2; leaving those that will close a circuit out)

EC, IK, EH, BD, DG in any order, but after all three edges Group 1 are picked.

Group 3 (edges with weight 3; leaving those that will close a circuit out)

JK, and (IF or FL, only one of these) in any order

Group 4 (edges with weight 4; leaving those that will close a circuit out)

either AB or AD (only one of them)

The number of edges picked is 11, which is one less than the number of vertices.

9. Use Prim's algorithm, starting at H, to find a minimum spanning tree of the graph given above (Figure 2). Draw the resulting spanning tree and list the edges in the order they are picked by Prim's algorithm.

Prim's algorithm picks the edges in the following order:

HE, EF, EC, BC, BD, DG, FI, IK, KL, KJ, AB

or

HE, EF, EC, BC, BD, DG, FL, KL, IK, KJ, AB

1. For each graph below, answer the following:

   (a) List the edges and vertices of the graph.
   $G_1 : V = \{r, s, t, u, v, w, x, y, z\}, E = \{sv, su, sw, wx, vy, vr, yz, rt\}$
   $G_2 : V = \{r, s, t, u, v, w, x, y, z, a, b\}$,
   $E = \{su, sv, sr, uw, vz, vr, rb, wz, wx, xy, xt, at, ab\}$

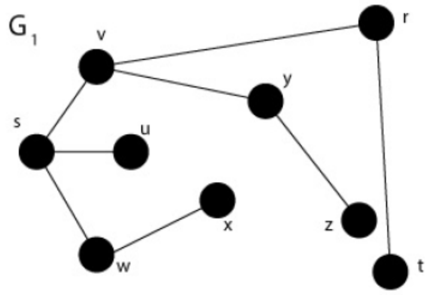
   (b) How many edges and vertices are there?
   G_1 : 9 vertices, 8 edges
   G_2 : 11 vertices, 13 edges

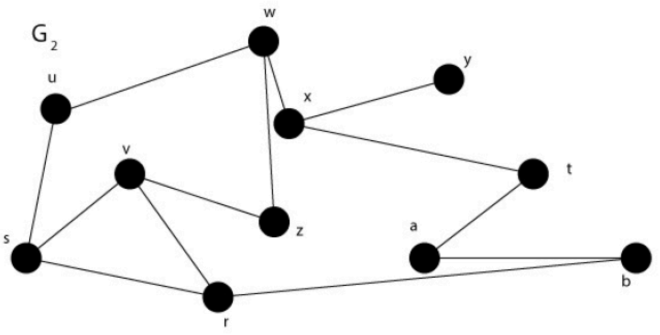
   (c) List the neighbours of the vertex v
   $G_1 : r, s, y$
   $G_2 : r, s, z$

   (d) How many edges are incident with s
   G_1 : 3 The edges are: sv, su, sw
   G_2 : 3 The edges are; sv, su, sr

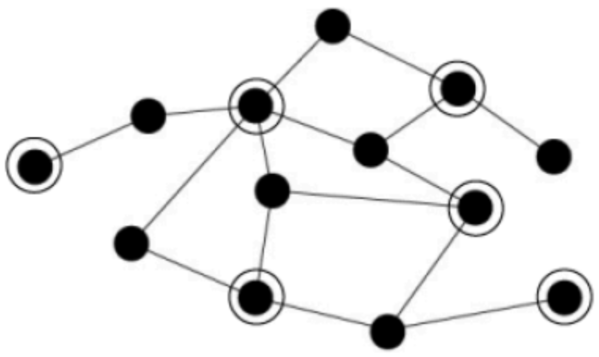
   (e) Find a walk between s and t. Is your walk a path? Why or why not?
   Answers will vary for G_2 from student to student one possible answer is provided.
   There is a unique walk in G_1.
   $G_1 : s, v, r, t$. This walk is a path as no vertices repeat.
   $G_2 : s, v, r, s, u, w, z, v, r, b, a, t$ This walk is not a path as multiple vertices (s, v, r) repeat.

   (f) Is the graph a tree? if not, find a cycle.
   G_1 : This graph is a tree.
   G_2 : This is not a tree. One cycle is s, v, r, s

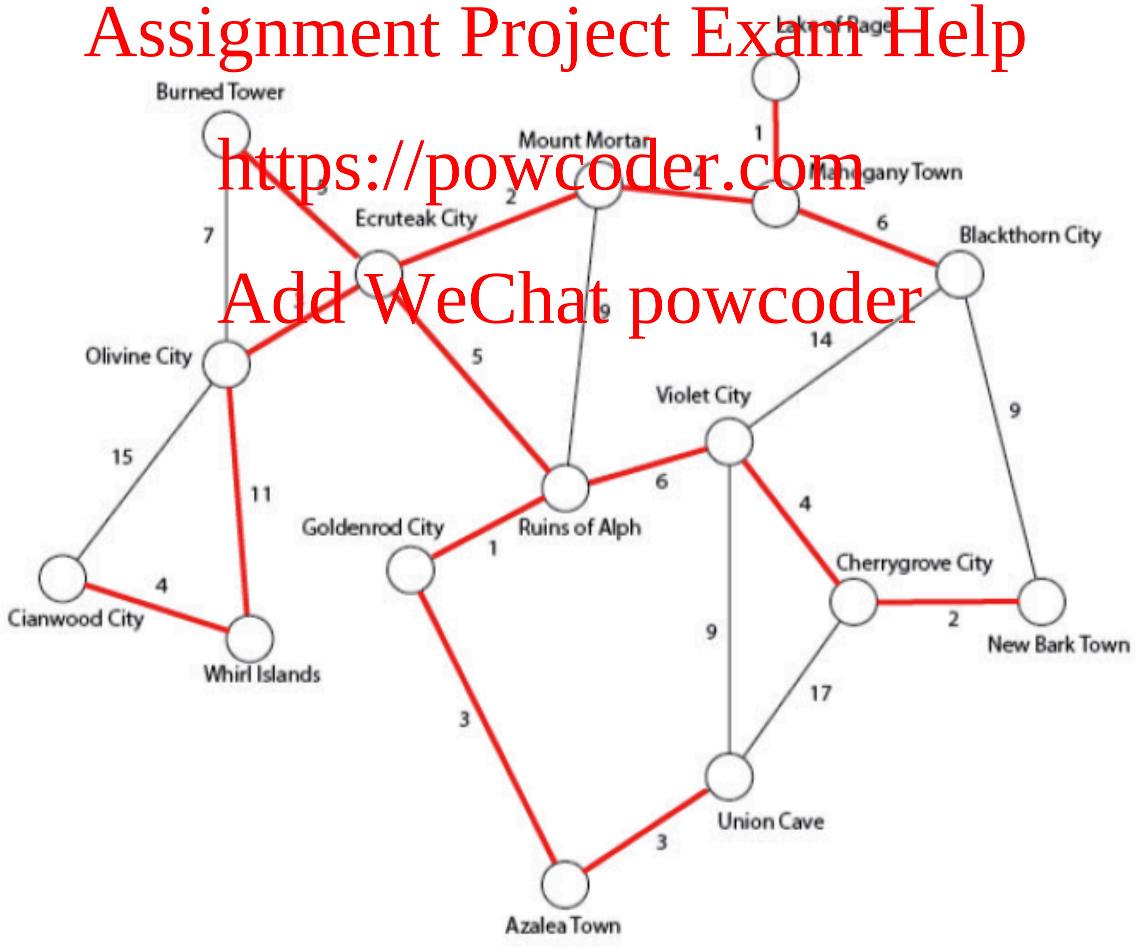




Determine whether the following graph is bipartite. If it is, find the groups A and B.
This graph is bipartite. Group A is the circled vertices. Group B is the uncircled vertices.



Using Prim's Algorithm, find a minimum spanning tree for the following railroad network. Use New Bark Town as your starting vertex.
The minimum spanning tree found using Prim's Algorithm is highlighted in red



(4 points) Consider an undirected graph G with unique positive weights. Suppose it has a minimum spanning tree T. If we square all the edge weights and compute the MST again, do we still get the same tree structure? Explain briefly.

Solution: Yes we get the same tree. The minimum spanning tree only depends on the ordering among the edges. This is because the only thing we do with edges is compare them.

(5 points) A new startup *FastRoute* wants to route information along a path in a communication network, represented as a graph. Each vertex represents a router and each edge a wire between routers. The wires are weighted by the maximum bandwidth they can support. *FastRoute* comes to you and asks you to develop an algorithm to find the path with maximum bandwidth from any source s to any destination t. As you would expect, the *bandwidth* of a path is the minimum of the bandwidths of the edges on that path; the minimum edge is the *bottleneck*.

Explain how to modify Dijkstra's algorithm to do this. In particular, how would you change the priority queue and the following relax step?

```
fun relax (Q, (u,v,w)) = PQ.insert (d(u) + w, v) Q
```

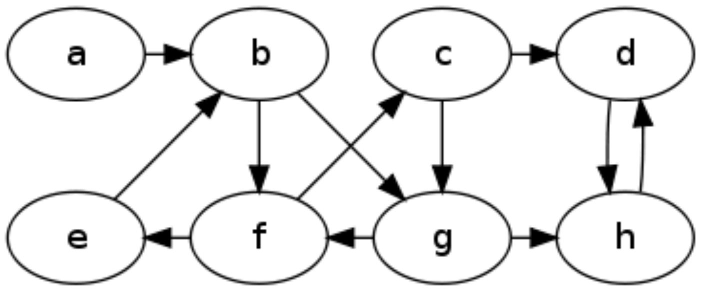
Justify your answer.

Solution: We'll use a max priority queue instead of a min priority queue used in Dijkstra's. We will also modify the relax step to insert into the priority queue $\min(d(u), w)$ because the quality of a path is the minimum of the edge weights. These changes don't affect the correctness of Dijkstra's, so we could explore the vertices like in Dijkstra's.

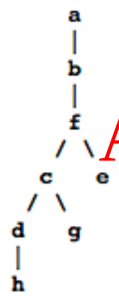
(5 points) Given a graph with integer edge weights between 1 and 5 (inclusive), you want to find the shortest *weighted* path between a pair of vertices. How would you reduce this problem to the shortest *unweighted* path problem, which can be solved using BFS?

Solution: Replace each edge with weight i with a simple path of i edges each with weight 1. Then solve with BFS.

Problem : Show the DFS tree that results from running DFS on the following graph and classify the edges as tree edges, back edges, forward edges, or cross edges. Start at vertex a and examine edges in alphabetical order of destination vertex.



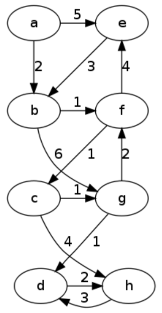
ANS



- Tree edges: see above
- Forward edges: (b,g)
- Back edges: (e,b), (g,f) (h,d)
- Cross edges: (g,h)

Problem : List the strongly connected components of the graph in the previous problem in order of topological sort.
{a}, {b,c,e,f,g}, {d,h}

Problem : Illustrate the execution of Dijkstra's algorithm on the following graph, using a as the start vertex. Show the priorites before each dequeue operation and show the shortest paths tree at the end of the algorithm.

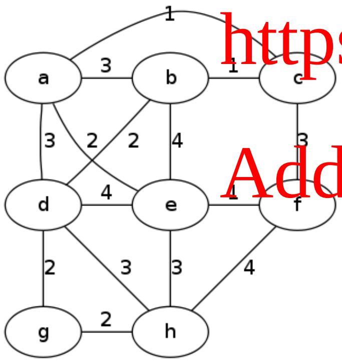


ANS

		Priority after dequeueing							
Vertex	**Priority**	**a**	**b**	**f**	**c**	**e**	**g**	**d**	**h**
a	**0**								
b	∞	**2**							
c	∞			**4**					
d	∞						**6**		
e	∞	**5**							
f	∞		**3**						
g	∞		**8**		**5**				
h	∞				**8**				

```
a-e
|
b-f
 /
c-g
 x
d h
```

Problem : Illustrate the execution of Prim's algorithm on the following graph, using a as the start vertex. Show the priorites before each dequeue operation and show the final MST.

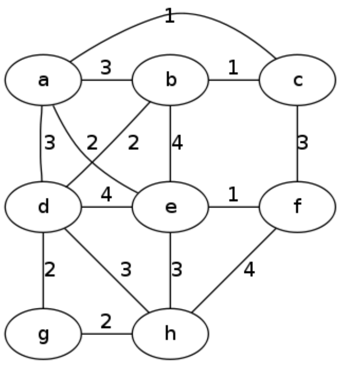


ANS

		Priority after dequeueing							
Vertex	**Priority**	**a**	**c**	**b**	**d**	**e**	**f**	**g**	**h**
a	**0**								
b	∞	**3**	**1**						
c	∞	**1**							
d	∞	**3**		**2**					
e	∞	**2**							
f	∞		**3**			**1**			
g	∞				**2**				
h	∞				**3**			**2**	

```
  _
 / \
a b-c
 x
d e-f
|
g-h
```

Problem : Illustrate the execution of Kruskal's algorithm on the graph from the previous problem. Show the disjoint sets after each edge is added to the tree and show the final MST.



{a,c} {b,c} {e,f} {a,e} {b,d} {d,g} {g,h} {a,b} {a,d} {c,f} {d,h} {e,h} {b,e} {d,e} {f,h}

- **{a} {b} {c} {d} {e} {f} {g} {h}**
- **Add {a,c}: {a,c} {b} {d} {e} {f} {g} {h}**
- **Add {b,c}: {a,b,c} {d} {e} {f} {g} {h}**
- **Add {e,f}: {a,b,c} {d} {e,f} {g} {h}**
- **Add {a,e}: {a,b,c,e,f} {d} {g} {h}**
- **Add {b,d}: {a,b,c,d,e,f} {g} {h}**
- **Add {d,g}: {a,b,c,d,e,f,g} {h}**
- **Add {g,h}: {a,b,c,d,e,f,g,h}**
- **Skip the rest**

```
  _
 / \
a b-c
 x
d e-f
|
g-h
```

Graph Theory Problems and Solutions

Tom Davis
tomrdavis@earthlink.net
http://www.geometer.org/mathcircles
November 11, 2005

1 Problems

1. Prove that the sum of the degrees of the vertices of any finite graph is even.

2. Show that every simple graph has two vertices of the same degree.

3. Show that if n people attend a party and some shake hands with others (but not with themselves), then at the end, there are at least two people who have shaken hands with the same number of people.

4. Prove that a complete graph with n vertices contains $n(n-1)/2$ edges.

5. Prove that a finite graph is bipartite if and only if it contains no cycles of odd length.

6. Show that if every component of a graph is bipartite, then the graph is bipartite.

7. Prove that if u is a vertex of odd degree in a graph, then there exists a path from u to another vertex v of the graph where v also has odd degree.

8. If the distance $d(u, v)$ between two vertices u and v that can be connected by a path in a graph is defined to be the length of the shortest path connecting them, then prove that the distance function satisfies the triangle inequality: $d(u, v) + d(v, w) \geq d(u, w)$.

9. Show that any graph where the degree of every vertex is even has an Eulerian cycle. Show that if there are exactly two vertices a and b of odd degree, there is an Eulerian path from a to b. Show that if there are more than two vertices of odd degree, it is impossible to construct an Eulerian path.

10. Show that in a directed graph where every vertex has the same number of incoming as outgoing paths there exists an Eulerian path for the graph.

11. Consider the sequence 01110100 as being arranged in a circular pattern. Notice that every one of the eight possible binary triples: 000, 001, 011, . . . , 111 appear exactly once in the circular list. Can you construct a similar list of length 16 where all the four binary digit patterns appear exactly once each? Of length 32 where all five binary digit patterns appear exactly once?

12. An n-cube is a cube in n dimensions. A cube in one dimension is a line segment; in two dimensions, it's a square, in three, a normal cube, and in general, to go to the next dimension, a copy of the cube is made and all corresponding vertices are connected. If we consider

the cube to be composed of the vertices and edges only, show that every n-cube has a Hamiltonian circuit.

13. Show that a tree with n vertices has exactly $n-1$ edges.

14. If u and v are two vertices of a tree, show that there is a unique path connecting them.

15. Show that any tree with at least two vertices is bipartite.

16. Solve Instant Insanity.

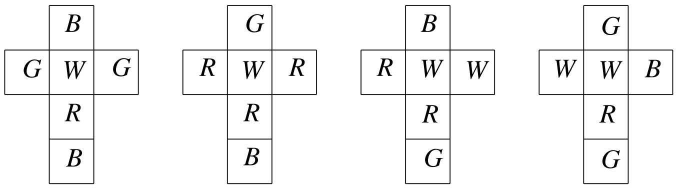


Figure 1: Instant Insanity Blocks

Figure 1 shows four unwrapped cubes that form the instant insanity puzzle. The letters "R", "W", "B" and "G" stand for the colors "red", "white", "blue" and "green". The object of the puzzle is to stack the blocks in a pile of 4 in such a way that each of the colors appears exactly once on each of the four sides of the stack.

17. (The Schroder-Bernstein Theorem) Show that if set A can be mapped $1-1$ onto a subset of B and B can be mapped $1-1$ onto a subset of A, then sets A and B have the same cardinality. (Two sets have the same cardinality if there exists a $1-1$ and onto mapping between them.)

2 Solutions

1. Prove that the sum of the degrees of the vertices of any finite graph is even.

   Proof: Each edge ends at two vertices. If we begin with just the vertices and no edges, every vertex has degree zero, so the sum of those degrees is zero, an even number. Now add edges one at a time, each of which connects one vertex to another, or connects a vertex to itself (if you allow that). Either the degree of two vertices is increased by one (for a total of two) or one vertex's degree is increased by two. In either case, the sum of the degrees is increased by two, so the sum remains even.

2. Show that every simple finite graph has two vertices of the same degree.

   Proof: This can be shown using the pigeon hole principle. Assume that the graph has n vertices. Each of those vertices is connected to either 0, 1, 2, $\ldots$, $n-1$ other vertices. If any of the vertices is connected to $n-1$ vertices, then it is connected to all the others, so there cannot be a vertex connected to 0 others. Thus it is impossible to have a graph with n vertices where one is vertex has degree 0 and another has degree $n-1$. Thus the vertices can have at most $n-1$ different degrees, but since there are n vertices, at least two must have the same degree.

3. Show that if n people attend a party and some shake hands with others (but not with themselves), then at the end, there are at least two people who have shaken hands with the same number of people.

   Proof: See problem 2. Each person is a vertex, and a handshake with another person is an edge to that person.

4. Prove that a complete graph with n vertices contains $n(n-1)/2$ edges.

   Proof: This is easy to prove by induction. If $n = 1$, zero edges are required, and $1(1-0)/2 = 0$.

   Assume that a complete graph with k vertices has $k(k-1)/2$. When we add the $(k+1)^{\text{st}}$ vertex, we need to connect it to the k original vertices, requiring k additional edges. We will then have $k(k-1)/2 + k = (k+1)((k+1)-1)/2$ vertices, and we are done.

5. Prove that a finite graph is bipartite if and only if it contains no cycles of odd length.

   Proof: To show that a graph is bipartite, we need to show that we can divide its vertices into two subsets A and B such that every edge in the graph connects a vertex in set A to a vertex in set B.

   Proceed as follows: Choose any vertex from the graph and put it in set A. Follow every edge from that vertex and put all vertices at the other end in set B. Erase all the vertices you used. Now for every vertex in B, follow all edges from each and put the vertices on the other end in A, erasing all the vertices you used. Alternate back and forth in this manner until you cannot proceed. This process cannot encounter a vertex that is already in one set that needs to be moved to the other, since if it did, that would represent an odd number of steps from it to itself, so there would be a cycle of odd length.

   If the graph is not connected, there may still be vertices that have not been assigned. Repeat the process in the previous paragraph until all vertices are assigned either to set A or to set B.

There is no reason that the graph has to be finite for this argument to work, but the proof does have to be modified slightly, and probably requires the axiom of choice to proove: divide the graph into connected components and select a vertex from each component and put it in set A. Then use the same process as above. The “select a vertex from each component” requires the axiom of choice.

6. Show that if every component of a graph is bipartite, then the graph is bipartite.

   Proof: If the components are divided into sets A_1 and B_1, A_2 and B_2, et cetera, then let $A = \cup_i A_i$ and $B = \cup_i B_i$.

7. Prove that if u is a vertex of odd degree in a graph, then there exists a path from u to another vertex v of the graph where v also has odd degree.

   Proof: We will build a path that does not reuse any edges. As we build the path, imagine erasing the edge we used to leave so that we will not use it again. Begin at vertex u and select an arbitrary path away from it. This will be the first component of the path. If, at any point, the path reaches a vertex of odd degree, we will be done, but each time we arrive at a vertex of even degree, we are guaranteed that there is another vertex out, and, having left, we effectively erase two edges from that meet at the vertex. Since the vertex originally was of even degree, coming in and going out reduces its degree by two, so it remains even. In this way, there is *always* a way to continue when we arrive at a vertex of even degree. Since there are only a finite number of edges, the tour must end eventually, and the only way it can end is if we arrive at a vertex of odd degree.

8. If the distance $d(u, v)$ between two vertices u and v that can be connected by a path in a graph is defined to be the length of the shortest path connecting them, then prove that the distance function satisfies the triangle inequality: $d(u, v) + d(v, w) \geq d(u, w)$.

   Proof: If you simply connect the paths from u to v to the path connecting v to w you will have a valid path of length $d(u, v) + d(v, w)$. Since we are looking for the path of minimal length, if there is a shorter path it will be shorter than this one, so the triangle inequality will be satisfied.

9. Show that any graph where the degree of every vertex is even has an Eulerian cycle. Show that if there are exactly two vertices a and b of odd degree, there is an Eulerian path from a to b. Show that if there are more than two vertices of odd degree, it is impossible to construct an Eulerian path.

   Proof: One way to prove this is by induction on the number of vertices. We will first solve the problem in the case that there are two vertices of odd degree. (If all vertices have even degree, temporarily remove some edge in the graph between vertices a and b and then a and b will have odd degree. Find the path from a to b which we will show how to do below, and then follow the removed edge from b back to a to make a cycle.)

   Suppose the odd-degree vertices are a and b. Begin at a and follow edges from one vertex to the next, crossing off edges so that you won’t use them again until you arrive at vertex b and you have used all the vertices into b. Why is it certain that you will eventually arrive at b? Well, suppose that you don’t. How could this happen? After you leave a, if you arrive at a vertex that is not b, there were, before you arrived, an even number of unused edges leading into it. That means that when you arrive, there is guaranteed to be an unused path away from that vertex, so you can continue your route. After entering and leaving a vertex, you reduce the number of edges by 2, so the vertex remains one with an even number (possibly zero) of unused paths. So if you have not yet arrived at vertex b, you can never get stuck at any other vertex, since there's always a way out. Since the graph is finite, you cannot continue forever, so eventually you will have to arrive at vertex b. (And it has to be possible to get to vertex b since the graph is connected.)

Note that a similar argument can be used to show that you can wait until you have used all the edges connecting to b. If b has more than one edge, leave each time you arrive until you get stuck at b.

Now you have a path something like this: $(a, a_1, a_2, \ldots, a_n, b)$ leading from a to b. If all the edges are used in this path, you are done. If not, imagine that you have erased all the edges that you used. What remains will be a number of components of the graph (perhaps only one) where all the members of each component have even degree. Since b will not be in any of the components, all of them must have fewer vertices than the original graph.

10. Show that in a directed graph where every vertex has the same number of incoming as outgoing paths there exists an Eulerian path for the graph.

    Proof: The proof above works basically without change. Note that each time a vertex is visited, one incoming and one outgoing node is used, so the equality of incoming and outgoing edges is preserved.

11. Consider the sequence 01110100 as being arranged in a circular pattern. Notice that every one of the eight possible binary triples: 000, 001, 011, . . . , 111 appear exactly once in the circular list. Can you construct a similar list of length 16 where all the four binary digit patterns appear exactly once each? Of length 32 where all five binary digit patterns appear exactly once?

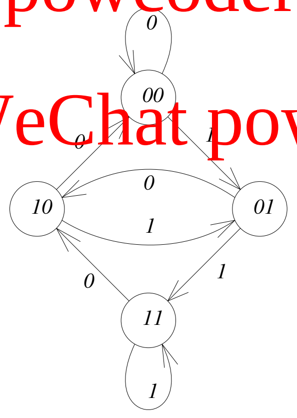


Figure 2: Circular Binary Patterns

Proof: Consider the figure 11. It is a graph with four vertices, each labeled with one of the possible pairs of binary digits. Imagine that each represents the last two digits in the pattern so far. The arrows leading away from a vertex are labeled either 0 or 1: the two possible values for the next digit that can be added to the pattern, and the ends of the arrows indicate the new final two digits. To achieve every possible three-digit combination, we need to traverse the graph with an Eulerian cycle. Because of the result in the previous problem, there are always two arrows out and two arrows into each vertex, and because of the result in the previous problem, there must be an Eulerian circuit.

The situation is the same for any number of digits except that the graph will become more and more complex. For the 4-digit version, there will be 8 vertices and 16 edges, and so on. But in every case, there will be two edges entering and two leaving each vertex, so an Eulerian circuit is possible.

12. An n-cube is a cube in n dimensions. A cube in one dimension is a line segment; in two dimensions, it's a square, in three, a normal cube, and in general, to go to the next dimension, a copy of the cube is made and all corresponding vertices are connected. If we consider the cube to be composed of the vertices and edges only, show that every n-cube has a Hamiltonian circuit.

    Proof: The proof is easy, and can be done by induction. If $n = 1$, we simply need to visit each vertex of a two-vertex graph with an edge connecting them.

    Assume that it's true for $n = k$. To build a $(k + 1)$-cube, we take two copies of the k-cube and connect the corresponding edges. Take the Hamiltonian circuit on one cube and reverse it on the other. Then choose an edge on one that is part of the circuit and the corresponding edge on the other and delete them from the circuit. Finally, add to the path connections from the corresponding endpoints on the cubes which will produce a circuit on the $(k + 1)$-cube.

13. Show that a tree with n vertices has exactly $n - 1$ edges.

    Proof: We can prove this by induction. If $n = 1$, the graph cannot have any edges or there would be a loop, with the vertex connecting to itself, so there must be $n - 1 = 0$ edges.

    Suppose that every tree with k vertices has precisely $k - 1$ edges. If the tree T contains $k + 1$ vertices, we will show that it contains a vertex with a single edge connected to it. If not, start at any vertex, and start following edges marking each vertex as we pass it. If we ever come to a marked vertex, there is a loop in the edges which is impossible. But since each vertex is assumed to have more than one vertex coming out, there is never a reason that we have to stop at one, so we much eventually encounter a marked vertex, which is a contradiction.

    Take the vertex with a single edge connecting to it, and delete it and the edge from the tree T. The new graph T' will have k vertices. It must be connected, since the only thing we lopped off was a vertex that was not connected to anything else, and all other vertices must be connected. If there were no loops before, removing an edge certainly cannot produce a loop, so T' is a tree. By the induction hypothesis, T' has $k - 1$ edges. But to convert T' to T we need to add one edge and one vertex, so T also satisfies the formula.

14. If u and v are two vertices of a tree, show that there is a unique path connecting them.

    Since it's a tree, it is connected, and therefore there has to be at least one path connecting u and v. Suppose there are two different paths P and Q connecting u to v. Reverse Q to make a path Q' leading from v to u, and the path made by concatenating P and Q' leads from u back to itself. Now this path PQ' is not necessarily be a loop, since it may use some of the same edges in both directions, but we did assume that there are some differences between P and Q.

    We can, from PQ', generate a simple loop. Begin at u and continue back one node at a time until the paths P and Q' differ. (Of course they may differ right away.) At this point, the paths bifurcate. Continue along both paths of the bifurcation until they join again for the first time, and this must occur eventually, since we know they are joined at the end. The two fragments of P and Q' form a (possibly) smaller circuit in the tree, which is impossible.

15. Show that any tree with at least two vertices is bipartite.

    Proof: This is a trivial consequence of problem 5. A tree has *no* cycles, so it certainly does not contain any cycles of odd length.

16. Solve Instant Insanity.

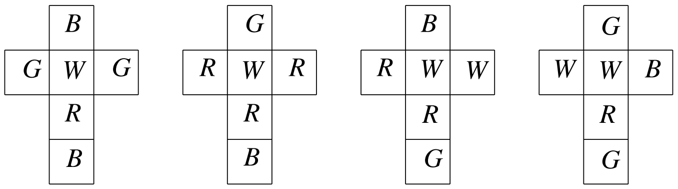


Figure 3: Instant Insanity Blocks

Figure 3 shows four unwrapped cubes that form the instant insanity puzzle. The letters “R”, “W”, “B” and “G” stand for the colors “red”, “white”, “blue” and “green”. The object of the puzzle is to stack the blocks in a pile of 4 in such a way that each of the colors appears exactly once on each of the four sides of the stack.

Solution: Any cube can have any pair of opposite faces appear on opposite sides of the stack of four. By turning the cube upside-down, the clockwise-counterclockwise orientation of those faces can be flipped, if necessary.

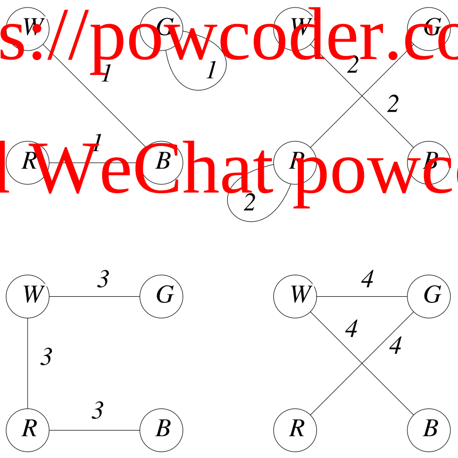


Figure 4: Individual Cube Graphs

For each cube, we’ll draw a graph connecting the pairs of colors on opposite faces, so each cube’s graph will consist of four vertices, corresponding to the four possible colors, and three edges that correspond to the pairs of colors that are opposite each other on each different cube. All of the four graphs are illustrated in figure 4.

Next, combine all sets of graph edges using the same set of four vertices, but retain the labels on the edges so that we obtain the graph in figure 5, which includes multiple connections between some pairs of vertices.

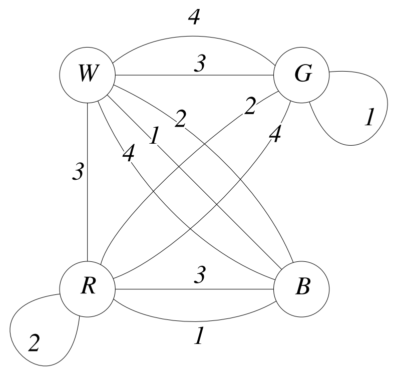


Figure 5: Combined Cube Graphs

To solve instant insanity, we need to find two subgraphs, each of which uses all four nodes, and each of which uses one each of the four numbered vertices. Each node must have exactly two of the edges coming from it. A little experimentation indicates that there is only one way to do this, and it is illustrated in figure 6.

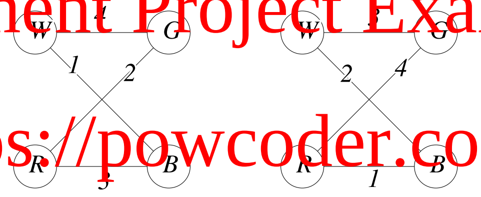


Figure 6: Instant Insanity Solution

If we align the front and back faces of the cube as mandated by the graph on the left of figure 6 and the left and right faces as mandated by the graph on the right of that figure then stack them on the top and bottom faces, we will obtain the unique solution.

17. (The Schröder-Bernstein Theorem) Show that if set A can be mapped $1-1$ onto a subset of B and B can be mapped $1-1$ onto a subset of A, then sets A and B have the same cardinality. (Two sets have the same cardinality if there exists a $1-1$ and onto mapping between them.)

    Proof: What we'd like to prove is that if the function $f : A \rightarrow B$ is an injection from A into B and if $g : B \rightarrow A$ is an injection from set B into A, then we can find a bijection $h : A \leftrightarrow B$ mapping A $1-1$ and onto B.

    Graph theory provides a nice way to visualize the proof. Imagine drawing a horizontal line and placing a vertex (dot) for every element of set A above the line and a vertex for every element of B below the line[1]. Next, for every element $a \in A$, draw a blue arrow from a to $f(a)$. All of these blue arrows will go down. Similarly, for every element $b \in B$, draw a red arrow from b to $g(b)$. All the red arrows will go up.

[1] For those who know about sets of *large* cardinality, it is clear that this may be impossible, since the sets will have more elements than there are points of a plane, but for visualization purposes, this is not a bad way to think about what's going on.

Since f and g are injections, it's clear that every $a \in A$ will have a blue arrow going down from it to some element in B, and similarly, every $b \in B$ will have a red arrow coming up from it. But since f and g are merely injections, it is not necessarily the case that every $a \in A$ will have a red arrow coming up into it, and similarly, there may be elements $b \in B$ that have no blue arrows going down into them.

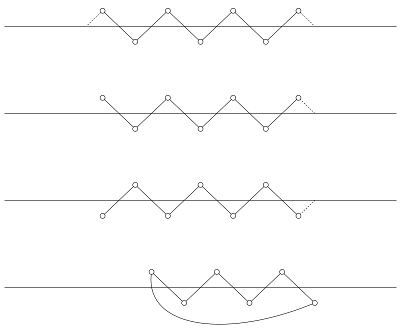


Figure 7: The Schröder-Bernstein Theorem

If we look at the situations that can occur, there are really only four possibilities, all of which are illustrated in Figure 7. The four horizontal lines in the figure are meant to represent different parts of the one horizonal line with the vertices representing elements of A above and elements of B below.

Imagine starting at any vertex in set A. It will surely have a line connecting it down to a vertex in B. Then that one will have a line going up to A, then down to B, then up to A, and on and on. Now, either it continues forever, or it loops back to the initial element in A. It cannot come back to any other element in the chain, or there would be two arrows coming to that point, violating the injectiveness of g. If it's a closed loop, it looks like the bottom example in Figure 7.

If it goes on forever without looping, then look at the vertex in A that we started with. If nothing points to it, we obtain a situation like that in the second line in the figure. If something points to it, keep looking back either until it continues forever (as in the top illustration), or until it halts either with an element of A or B.

If we start with any element in B, we can just do exactly the same thing and again, we'll obtain one of the four situations above: a finite loop, a semi-infinite chain beginning with an element of A, a semi-infinite chain beginning with an element of B, or a doubly-infinite chain. Note that *every* element of A and B lies in a chain of one of these four types.

From the figure, it is easy to construct a bijective function $h : A \leftrightarrow B$ as follows. If the element $a \in A$ is in a chain like the first, second, or fourth lines in Figure 7, then $h(a) = f(a)$. If a's chain is like that in the third line, then $h(a) = g^{-1}(a)$.