CSE 5311 Homework #2 Solutions

Fall 2002

1. Problem 26.2-2, also solve using preflow-push

Figure 1 gives the results using Edmonds-Karp. Augmenting paths are indicated by dashed lines.

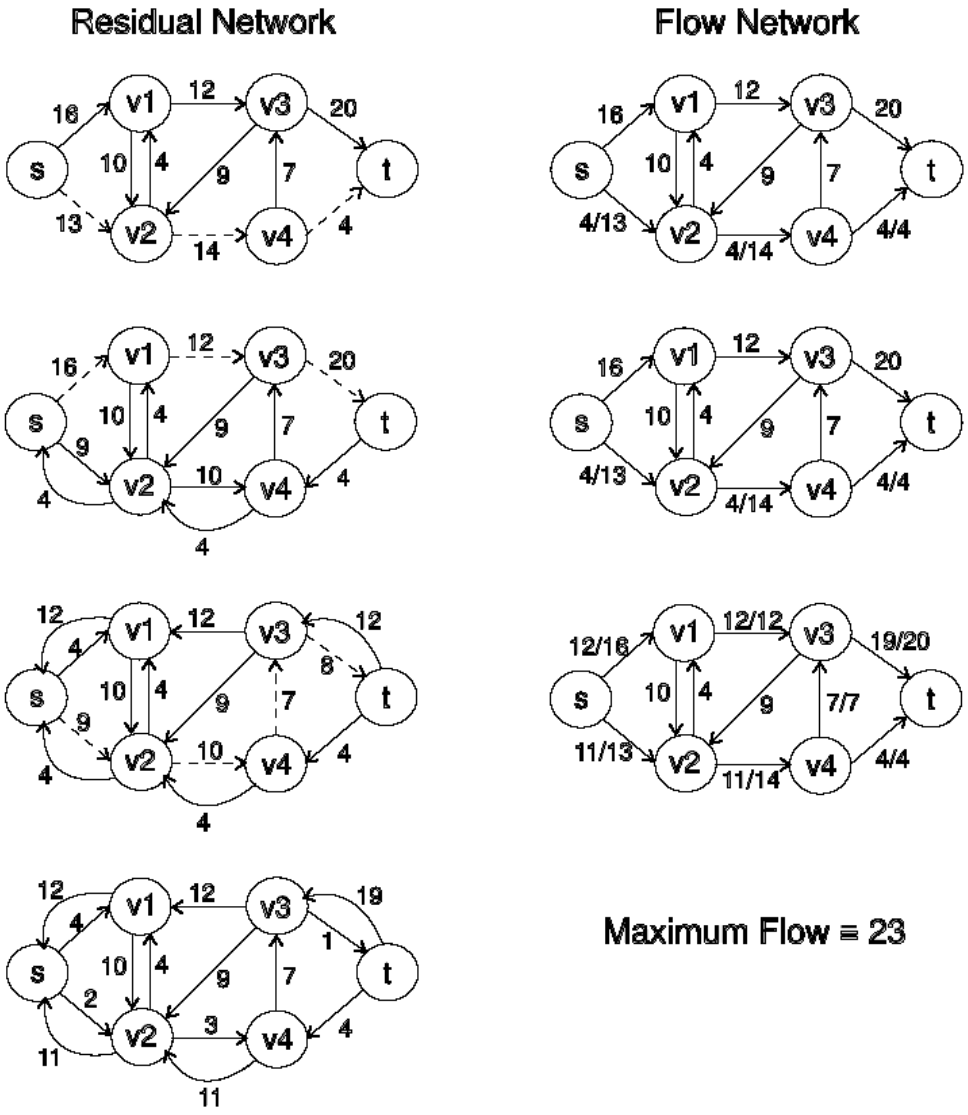


Figure 1: Edmonds-Karp Solution

```
6 10
0 1 16
0 2 13
1 3 12
1 2 10
2 1 4
2 4 14
3 2 9
3 5 20
4 3 7
4 5 4
```

Table 1: Input File for preflowPushFIFO.c

For the preflow-push results you may use preflowPushFIFO.c located in the web site (http://reptar.uta.edu/NOTES5311/preflowPushFIFO.c). Table 1 gives an input file which encodes the figure from the book into the proper format for that program. The source is assigned a node number of 0 and the sink a node number of 5. The other node numbers correspond to the numbers in the diagram. The output is as follows:

```
vertex height excess
   0      6    -29
   1      0     16
   2      0     13
   3      0      0
   4      0      0
   5      0      0
 tail head capacity    flow
   0    1       16      16
   0    2       13      13
   1    2       10       0
   1    3       12       0
   2    1        4       0
   2    4       14       0
   3    2        9       0
   3    5       20       0
   4    3        7       0
   4    5        4       0
debug: lifting 1 from 0 to 1
debug: pushing 10 units from 1 to 2
debug: pushing 6 units from 1 to 3
debug: lifting 2 from 0 to 1
debug: pushing 14 units from 2 to 4
debug: lifting 2 from 1 to 2
debug: pushing 9 units from 2 to 1
debug: lifting 3 from 0 to 1
debug: pushing 6 units from 3 to 5
debug: lifting 4 from 0 to 1
debug: pushing 4 units from 4 to 5
debug: lifting 4 from 1 to 2
debug: pushing 7 units from 4 to 3
```

```
debug: lifting 4 from 2 to 3
debug: pushing 3 units from 4 to 2
debug: lifting 1 from 1 to 2
debug: pushing 6 units from 1 to 3
debug: lifting 1 from 2 to 3
debug: pushing 3 units from 1 to 2
debug: pushing 13 units from 3 to 5
debug: lifting 2 from 2 to 4
debug: pushing 6 units from 2 to 1
debug: lifting 1 from 3 to 5
debug: pushing 6 units from 1 to 2
debug: pushing 3 units from 2 to 4
debug: lifting 2 from 4 to 6
debug: pushing 3 units from 2 to 1
debug: lifting 4 from 3 to 7
debug: pushing 3 units from 4 to 2
debug: lifting 1 from 5 to 7
debug: pushing 3 units from 1 to 0
debug: lifting 2 from 6 to 7
debug: pushing 3 units from 2 to 0
total flow is 23
flows along edges:
0->1 has 13
0->2 has 10
1->2 has 1
1->3 has 12
2->4 has 11
3->5 has 19
4->3 has 7
4->5 has 4
```

2. Lattice for Stable Marriage Problem

Figure 2 shows the male-oriented lattice.

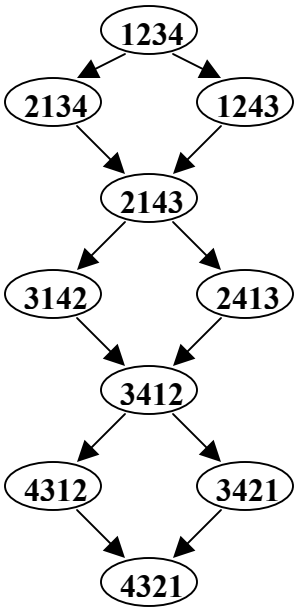


Figure 2: Stable-Marriage Lattice

3. Problem 22-1

a. Undirected Graph with breadth-first search (BFS):

- Because of the properties of BFS–the way nodes are visited in BFS, there can be no back and forward edges. The edges that constitute as back and forward edges in depth-first search (DFS), have already been visited through the parent and are tree edges in a BFS of an undirected graph.
- Again, because of the way nodes are visited in a BFS, for each tree edge (u, v), $d[u]$ has to be equal to $d[u] + 1$.
- Two kinds of situations may arise where there can be cross edges. These are depicted in figure 3. In one case $d[v] = d[u]$ and in the other case $d[v] = d[u]+1$.

b. Directed Graph with BFS:

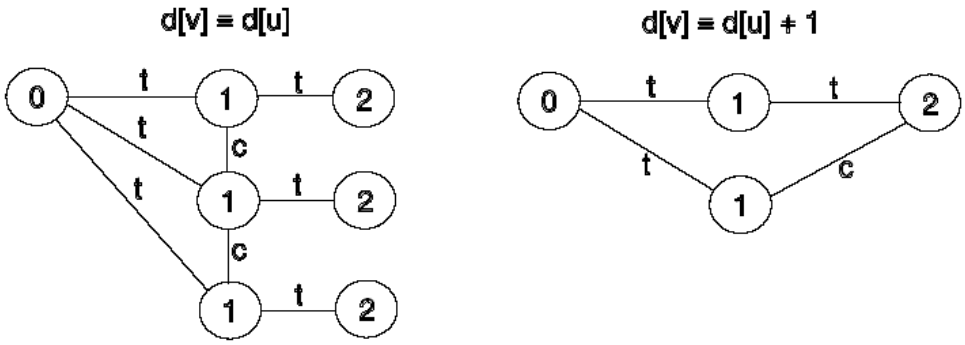


Figure 3: Undirected Graph Using BFS

- Because the graph is directed, it can now have back edges. However, there are still no forward edges, because of the way nodes are visited in a BFS. The forward edges of a DFS are tree edges in a BFS already visited by the parent node.

- **Same as a) 2 above.**
- **Same as a) 2 above. So, d[v] ≤ d[u] +1.**

- Two examples of a back edge in a BFS of a directed graph are given in figure 4. In one case we have $d[v] < d[u]$ and in the other case we have $0 \leq d[v] < d[u]$.

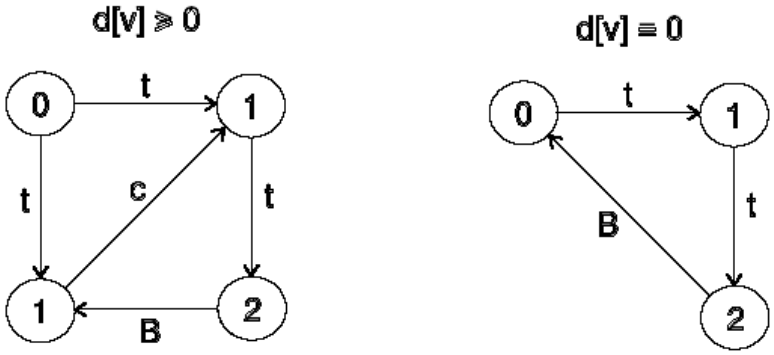


Figure 4: Directed Graph Using BFS

4. Max Flow Using Preflow-Push

The following output was obtained by using preflow-push located in the web site (http://reptar.uta.edu/NOTES5311/preflowPushFIFO.c). The source is assigned a node number of 0 and the sink a node number of 3. The nodes labeled A and B are given labels of 1 and 2, respectively.

```
debug:  after initialization
  i height excess
  0      4      0
  1      0     10
  2      0     50
  3      0      0
  i   j capacity ...flow
  0   1       10      10
  0   2       50      50
  1   3      100       0
  2   3       20       0
debug:  (1,0) changed minHeight to 4
debug:  (1,3) changed minHeight to 0
debug:  lifting 1 from 0 to 1
debug:  (2,0) changed minHeight to 4
debug:  (2,3) changed minHeight to 0
debug:  lifting 2 from 0 to 1
```

```
debug:
  i height excess
  0      4      0
  1      1     10
  2      1     50
  3      0      0
  i   j capacity ...flow
  0   1       10      10
  0   2       50      50
  1   3      100       0
  2   3       20       0
debug:  pushing 10 units from 1 to 3
debug:  pushing 20 units from 2 to 3
debug:
  i height excess
  0      4      0
  1      1      0
  2      1     30
  3      0     30
  i   j capacity ...flow
  0   1       10      10
  0   2       50      50
  1   3      100      10
  2   3       20      20
debug:  (2,0) changed minHeight to 4
debug:  lifting 2 from 1 to 5
debug:
  i height excess
  0      4      0
  1      1      0
  2      5     30
  3      0     30
  i   j capacity ...flow
  0   1       10      10
  0   2       50      50
  1   3      100      10
  2   3       20      20
debug:  pushing 30 units from 2 to 0
debug:
  i height excess
  0      4     30
```

```
  1       1       0
  2       5       0
  3       0      30
  i    j capacity ...flow
  0    1        10       10
  0    2        50       20
  1    3       100       10
  2    3        20       20
debug:
  i height excess
  0      4      30
  1      1       0
  2      5       0
  3      0      30
  i    j capacity ...flow
  0    1        10       10
  0    2        50       20
  1    3       100       10
  2    3        20       20
final result:
  i height excess
  0      4      30
  1      1       0
  2      5       0
  3      0      30
  i    j capacity ...flow
  0    1        10       10
  0    2        50       20
  1    3       100       10
  2    3        20       20
```

5. KMP Fail Links

The Knuth-Morris-Pratt fails links (both methods) for the search pattern *abracadabra* are given in the table below:

	a	b	r	a	c	a	d	a	b	r	a
Method 1	0	1	1	1	2	1	2	1	2	3	4
Method 2	0	1	1	0	2	0	2	0	1	1	0

S[3]: case 1
S[4]: case 1
S[5]: case 1
S[6]: case 1
S[7]: case 1
S[8]: case 1
S[9]: case 2, kprime=2
S[10]: case 2, kprime=3
S[11]: case 3

1	**a**	**11**
2	**b**	**0**
3	**r**	**0**
4	**a**	**1**
5	**c**	**0**
6	**a**	**1**
7	**d**	**0**
8	**a**	**4**
9	**b**	**0**
10	**r**	**0**
11	**a**	**1**

6. Complexity of Recursive Matrix Multiplication

Suppose that matrix multiplication is implemented in a recursive decomposition fashion like Strassen's methods. However, instead of using his equations we use the everyday ones, i.e., $C_{ij} = A_{i1} * B_{1j} + A_{i2} * B_{2j}$. What is the asymptotic complexity, based on the number of scalar multiplies and additions/subtractions?

For $C = AB$, we divide each of A, B, C into four $\frac{n}{2} \times \frac{n}{2}$ matrices. Then,

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & g \\ f & h \end{pmatrix}$$

Then according to: $C_{ij} = A_{i1} * B_{1j} + A_{i2} * B_{2j}$ we have:

r = ae + bf
s = ag + bh
t = ce + df
u = cg + dh

Each of these four equations specifies two multiplications of $\frac{n}{2} \times \frac{n}{2}$ and addition of $\frac{n}{2} \times \frac{n}{2}$ products. Then the number of multiplies is 8 and the number of additions is 4.

1. Let $M(k)$ be the # of scale multiplies for $n = 2$. Then,

$$\begin{array}{rcl} M(0) & = & 1 \\ M(1) & = & 8 \\ \hline M(k) & = & 8M(k-1) \\ \hline M(n) & = & n^3 \end{array}$$

2. Let $P(k)$ be the # of additions for $n = 2$. Then,

$$\begin{array}{rcl} P(0) & = & 0 \\ P(1) & = & 4 \\ \hline P(k) & = & 8P(k-1) + n^2 \\ \hline P(n) & = & 8P(n/2) + n^2 \\ P(n) & = & \Theta(n^3) \end{array}$$

So, the asymptotic complexity is $\Theta(n^3)$.

7. Rectangle-Fit Problem is NP-Complete

This problem can be reduced from the bin-packing problem. The rectangle-fit problem is obviously in NP because it can easily be verified in polynomial time whether a given set of rectangles fits within another given rectangle with dimensions x and y.

To prove that it is NP-hard, we reduce it from the bin-packing problem in the following manner. The bin-packing problem says that given a set of objects, each weith size ≥ 1, then k is the minimum number of unit-sized bins needed to pack all the objects. There is a direct relationship between the rectangle-fit problem and the bin-packing problem **as can be seen from the diagram given in Figure 5.**

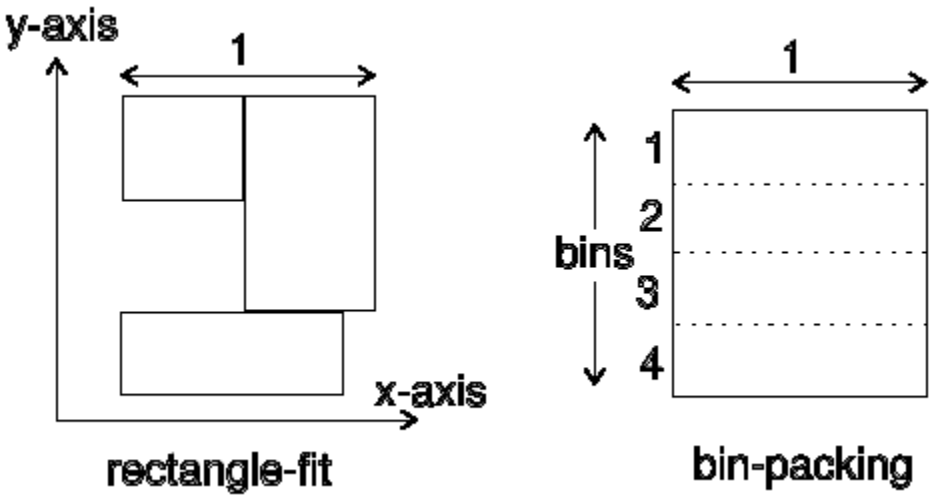


Figure 5: Rectangle Fit and Bin Packing

The reduction is obvious from Figure 1. We set dimension x of the minimum sized rectangle needed to fit all the other rectangles equal to unit size. Dimension y is given by the number (k) of bins required to cover the total width of the set of rectangles.

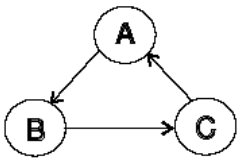
Thus, the rectangle-fit problem will have a covering rectangle of dimensions x and y if and only if the bin-packing problem packs all the object into k bins of size x such that $kw = y$, where w is the width of each bin. Thus, the rectangle-fit problem is NP-complete.

8. Graph Transitivity Problem is NP-Complete

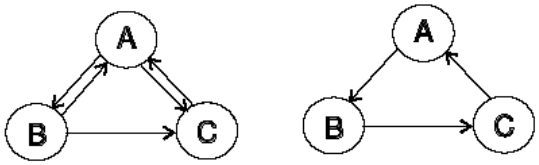
To prove that the problem is NP-complete we prove that it is in NP and that it is NP-hard by reducing from another NP-complete problem.

The problem is obviously in NP because it can be easily verified in polynomial time whether all the directed paths re preserved after removing the given k edges.

To prove that the problem is NP-hard, it is reduced from the Directed Hamiltonian Circuit problem. The Directed Hamiltonian Circuit problem says that given a directed graph G, the graph has a simple directed cycle containing each vertex in G.



As can be seen from the preceding diagram of a directed Hamiltonian circuit, the number of edges in such a cycle is equal to the number of vertices in the graph. Hence the reduction is as follows. The given problem can have k our of $|E|$ edges removed while still preserving all directed paths if and only if the corresponding Directed Hamiltonian Circuit has a simple cycle of size $|V|$, where $k = |E| - |V|$. So, for the example given below, $k = 2$ is the optimal (maximum) number of edges that can be removed while still preserving all directed paths from the original graph. Thus, this problem is NP-complete.



9. Problem 34-1

a) INDEPENDENT-SET = $\{< G, K >|$ graph G has an independent set of size $k\}$.

To prove this problem is NP-complete we first show that the independent-set problem is in NP. Suppose we are given a graph $G = (V, E)$ and an integer k. The certificate we choose is te independent set $V' \subseteq V$ itself. The verification algorithm affirms that $|V'| = k$ and then for each edge $(u, v) \in E$ it check that at most one of u and v is in V'. This verification can be performed straight-forwardly in polynomial time.

We prove that the independent-set problem is NP-hard by showing that CLIQUE $\propto$ INDEPENDENT-SET. This reduction is based on the notion of the "complement"

of a graph. Given an undirected graph $G = (V, E)$, we define the complement of G as $\overline{G} = (V, \overline{E})$, where $\overline{E} = \{(u, v) \mid (u, v) \notin E\}$.

The reduction algorithm takes as input an instance $< G, K >$ of the clique problem. It computes the complement $\overline{G}$, which is easily obtained in polynomial time. The output of the reduction algorithm is the instance $< \overline{G}, K >$ of the independent-set problem with the same k vertices in the independent set as in the clique. The

proof is very similar to the vertex-cover problem proof on p. 1006. Using the same examples as given in the book (figure 34.15 on p. 1007) we have G yielding the clique V' = {u, v, x, y} and $\overline{G}$ yielding the independent set V' = {u, v, x, y}.

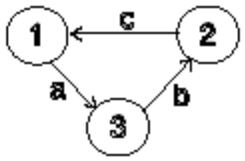
b) Not done.

c) It can be observed that when each vertex in G has degree 2, the graph is just a simple cycle. In this case, an independent set of maximum size can be obtained by starting at any vertex and picking each alternate vertex on the cycle until the size of the independent set is $\lfloor |V|/2 \rfloor$. The running time of this algorithm is obviously $O(|V|)$ (or $O(|E|)$ since $|V| = |E|$ in this case).

d) Using figure 26.7 on p.665, it can be seen that when graph G is bipartite, then the independent-set (maximum size) is the side with the larger number of vertices (the set of vertices in L). The running time will be $O(|V|)$.

10. Problem 35.2-2

For each combination of 3 vertices ($\binom{n}{3}$), calculate $x = c - a - b$ where $c, a, , b$ are the cost of the edges.



If $x < 0$, ignore. Otherwise keep track of $(x)_{max}$. Add $(x)_{max}$ to the cost of each edge. The resulting traveling-salesperson problem (TSP) satisfies the triangle inequality i.e., **c(u, w) ≤ c(u, v) + c(v, w). Using the example from p.1013 (reproduced here in fig 6) we find that $(x)_{max}$ = 5-1-2=2. Adding 2 to each edge yields the second graph in fig 6 which satisfies the triangle inequality.Since the same number is added to each**

edge, the two instances will have the same set of optimal tours. This transformation takes polynomial time because $\binom{n}{3} = O(n^3)$.

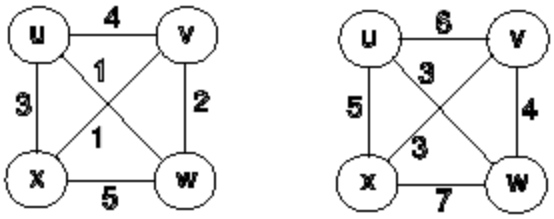


Figure 6: Triangular Inequalities

Now we explain why such a polynomial-time transformation does not contradict theorem 35.3 (p. 1031), assuming that P ≠ NP. The TSP with triangle inequality has a polynomial-time approximation algorithm with a ratio bound $\rho \geq 1$ i.e., $\text{cost}(\Delta) \leq \rho \cdot \text{opt}(\Delta)$, where $\text{opt}(\Delta)$ denotes the cost of an optimal tour for the TSP with triangle inequality. So for the original TSP without triangle inequality (denoted by $\not\Delta$) we can write $\text{cost}(\not\Delta) \leq \rho \cdot \text{opt}(\not\Delta)$ or $\text{cost}(\Delta) - X|V| \leq \rho(\text{opt}(\Delta) - X|V|)$, where X is the constant added to each edge to convert to triangle inequality and hence $X|V|$ is the total increase in cost for the triangle inequality TSP. The above equation can be rewritten as $\text{cost}(\Delta) \leq \rho \cdot \text{opt}(\Delta) - \underbrace{(\rho - 1)X|V|}_{\text{positive for } \rho > 1}$, which may or may not be true. Hence the **transformation does not contradict theorem 35.3.**

11. Problem 35.2-3

Figure 7 shows how the closest-point heuristic works on the graph used in the example on p. 1029.

From the manner in which nodes are added to the cycle in this heuristic, it can be observed (comparing with figure 35.2 on p. 1029) that the final cycle has an inherent MST within it and that its length will be less than the length of the full walk which traverses every edge of the MST twice. Hence, from analysis in theorem 35.2 (p. 1030) we have c(closest-point heuristic) $\leq c(w)$ and hence c(closest-point heuristic) $\leq 2c(H^*)$ where $c(H^*)$ is the cost of the optimal tour. Hence the closest-point heuristic has a ration bound of 2.

12. Problem 35.2-4

We are given that the vertices of the TSP are points in the plane and the cost $c(u, v)$ is the euclidean distance between point u and v. Fig 8 gives an example where the solid lines indicate a TSP optimal tour. Using triangle inequality, $z \leq x + y$ and $c \leq a + b$. Hence replacing $x + b$ and $a + y$ with c and z gives us a shorter tour which contradicts the assumption that the original tour was an optimal tour. Hence an optimal tour will never cross itself.

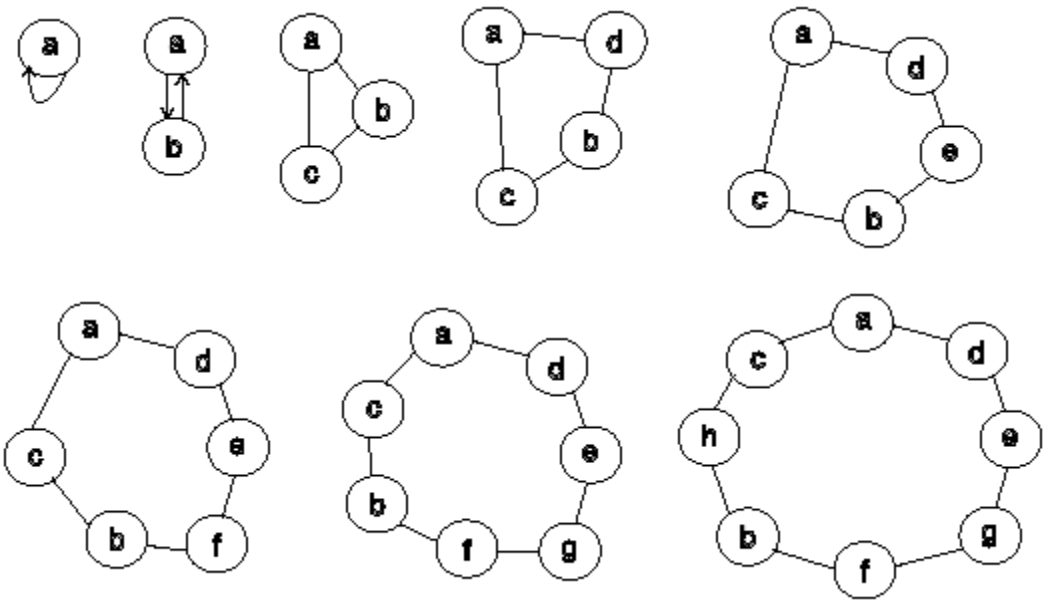


Figure 7: Closest-Point Heuristic

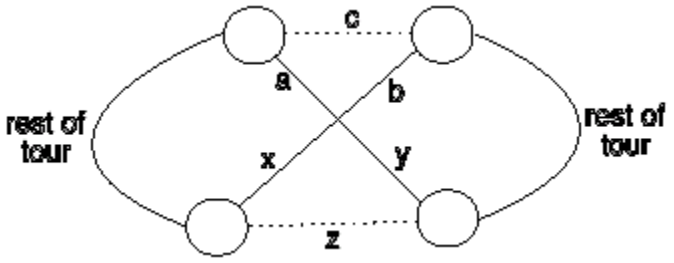


Figure 8: Traveling Salesperson Problem

13. Problem 35.3-1

Here X= {a,d,e,h,i,l,n,o,r,s,t,u} (see p. 1033-1034) and each of the given words is a

subset of X. The cover produced by GREEDY-SET-COVER when ties are broken in favor of the word appears first in the dictionary is: thread, lost, drain, shun.

14. Problem 35.3-2

The decision version of the set-covering problem is:

$$\text{SET-COVER} = \{< X, F, k >| \text{ set } X \text{ has a set cover of size } k\}$$

where X and F are as defined on p. 1033.

To show that this problem is NP-complete, we first show that SET-COVER $\in NP$. Suppose we are given a set X, a family F of subsets of X, and an integer k. The certificate we choose is the set cover $C' \subseteq F$ itself. The verification algorithm affirms that $|C'| = k$ (i.e., the number of subsets in $C' = k$) and then it checks whether each element of X is in at least one of the subset of C'. This verification can be performed in polynomial time **(with a proof very similar to the vertex-cover problem proof on p. 1006).**

We now prove that the set-covering problem is NP-hard by showing that VERTEX-COVER $\propto$ SET-COVER. The reduction is a direct mapping from VERTEX-COVER to SET-COVER.

$X = E$ i.e., the elements of X in SET-COVER are the edges $(u, v) \in E$ in VERTEX-COVER.

$|F| = |V|$ i.e. the number of subsets in the family F of subsets in SET-COVER is equal to the number of vertices in VERTEX-COVER, where each subset contains the edges from one vertex in VERTEX-COVER.

Thus, it is obvious that a vertex cover of size k in VERTEX-COVER is equivalent to a set cover of size k in SET-COVER where the subsets in the set cover correspond to the set of edges of the vertices in the vertex cover. In other words, the graph $G = (V, E)$ has a vertex cover of size k if and only if the set X has a set cover of size k. The proof starting from either side is obvious.