Algorithm Design and Analysis

Homework #6

Due: 1pm, Monday, January 9, 2012

=== Homework submission instructions ===

- Submit the answers for writing problems (including your programming report) through the CEIBA system (electronic copy) or to the TA in R432 (hard copy). Please write down your name and school ID in the header of your documents. You also need to submit your programming assignment (problem 1) to the Judgegirl System(`http://katrina.csie.ntu.edu.tw/judgegirl/`).

- Each student may only choose to submit the homework in one way; either all as hard copies or all through CEIBA except the programming assignment. If you submit your homework partially in one way and partially in the other way, you might only get the score of the part submitted as hard copies or the part submitted through CEIBA (the part that the TA chooses).

- If you choose to submit the answers of the writing problems through CEIBA, please combine the answers of all writing problems into only one file in the doc/docx or pdf format, with the file name in the format of "hw6_[student ID].{pdf,docx,doc}" (e.g. "hw6_b99902010.pdf"); otherwise, you might only get the score of one of the files (the one that the TA chooses).

- For each problem, please list your references (they can be the names of the classmates you discussed the problem with, the URL of the information you found on the Internet, or the names of the books you read). The TA can deduct up to 100% of the score assigned to the problems where you don't list your references.

Problem 1. (40%) **The n-puzzle problem.** The n-puzzle is a sliding puzzle which consists of a frame of numbered square tiles in random order with one tile missing. If the size is 3x3, then the puzzle is called the 9-puzzle and if 4x4, the puzzle is called the 16-puzzle. The goal of the puzzle is to place the tiles in order by making sliding moves (horizontally or vertically, but not diagonally) that use the empty space.

The following example shows a sequence of legal moves from an initial board configuration to the desired board configuration for a 9-puzzle.

```
  1 3   1   3   1 2 3   1 2 3   1 2 3
4 2 5   4 2 5   4   5   4 5     4 5 6
7 8 6   7 8 6   7 8 6   7 8 6   7 8
```

Finding *one* solution is relatively easy. However, it has been shown that finding the "shortest" solution (the solution with the minimum number of moves) is **NP-hard**. In this programming assignment, we ask you to write a program to solve this NP-hard problem.

We will now describe a classic solution to the problem, which illustrates a general artificial intelligence methodology known as the **A* algorithm**. In the A* algorithm, we always pick the one board configuration which seems most likely to be the one that leads to the shortest solution to evaluate. For the n-puzzle, it works as follows. First, insert the starting board configuration into a priority queue. Then, delete the board configuration with the minimum priority from the queue. It will turn into several new board configuration after moving a tile into the empty space; we put all possible new board configurations after one move back to the priority queue. Repeat this procedure until the board configuration dequeued represents desired board configuration.

The key question is how we determine the likelihood of one board configuration being the one that leads to the shortest solution. A heuristic based on the **Manhattan distance** between the current board configuration and the desired board configuration can be used for this purpose, and can be calculated for a particular board configuration c as the following:

$$h(c) = \sum_{i=1}^{n-1} (|\bar{x}_i - x_i| + |\bar{y}_i - y_i|) \tag{1}$$

where $\bar{x}_i$ and $\bar{y}_i$ are the x and y coordinates of the i-th tile in the desired board configuration, and x_i and y_i are the x and y coordinates of the i-th tile in the current board configuration. Note that $h(c)$ is a lower bound of the number of moves from the current board configuration c to the desired board configuration. Then the "priority" of a board

configuration when inserting into the priority queue is given by:

$$p(c) = h(c) + \epsilon(c) \tag{2}$$

where $\epsilon(c)$ is the actual number of moves (or, cost) used by the program to reach board configuration c from the starting board configuration.

One thing to note is that not all problems are solvable, i.e., some initial board configurations cannot be altered to reach the desired board configurations by any number of moves. However, we will make sure that all the input data sets which we use to test your program are solvable.

Your program should follow the input and output formats specified below.

Input

The first line has $\sqrt{n}$. The board is $\sqrt{n} \times \sqrt{n}$ in size. $3^2 \leq n \leq 10^2$.

The next $\sqrt{n}$ lines show the initial board configuration. The number are separated by a single space character ' '.

The tiles are numbered from 1 to $n - 1$. The empty tile is represented by a single '@' character.

Output

The number m in one line. m = the number of moves in the shortest solution.

Sample Input and Output

(input)

```
3
@ 1 3
4 2 5
7 8 6
```

(output)

```
4
```

Please write a program to solve the n-puzzle problem (25%). Please also submit a report (10%) which includes a technical specification of your program, which helps the TA to understand your program. In addition, please answer the following questions in your report.

- Suppose that we replace the Manhattan distance heuristic with a different one, which is no longer a lower bound of the number of moves from the current board configuration to the desired board configuration. Can we still guarantee that the A* algorithm will generate a shortest solution? Please explain. (5%)

- (bonus problem) Can you modify your algorithm to make sure that in the case that the input given to your program is not solvable, your program will be able to output -1 (indicating that the problem is not solvable) in *polynomial time*? Please explain. (bonus 10%)

Sol: To be announced later.

Problem 2. Please spend some time to read the NP-Completeness course material we covered in the class so far (section 34.1 and 34.2 in the textbook). Then solve the following to problems:

a. (15%) Show that the class P, viewed as a set of languages, is closed under union, intersection, concatenation, complement, and Kleene star. That is, if $L_1, L_2 \in P$, then $L_1 \cup L_2 \in P$, $L_1 \cap L_2 \in P$, $L_1 L_2 \in P$, $\overline{L_1} \in P$, and $L_1^* \in P$ (Problem 34.1-6 in the textbook).

Sol:

(1) $L_1 \cup L_2 \in P$

Since L_1 and L_2 are both in P, there exists machines M_1 and M_2 which decides L_1 and L_2, respectively. Design a machine M_3 which takes w as input as the following:

$M_3(w)$:
1 Run M_1 with input w. If M_1 accepts w, then *accept*.
2 Run M_2 with input w. If M_2 accepts w, then *accept*.

It is clear that M_3 accepts w if and only if M_1 or M_2 accepts w and M_3 runs in polynomial time.

(2) $L_1 \cap L_2 \in P$

Since L_1 and L_2 are both in P, there exists machines M_1 and M_2 which decides L_1 and L_2, respectively. Design a machine M_3 which takes w as input as the following:

$M_3(w)$:

1 Run M_1 with input w. If M_1 accepts w, then run M_2 on w, else *reject*.

2 If M_2 also accepts w, then *accept*, else *reject*.

It is clear that M_3 accepts w if and only if both M_1 and M_2 accepts w and M_3 runs in polynomial time.

(3) $L_1L_2 \in P$

Since L_1 and L_2 are both in P, there exists machines M_1 and M_2 which decides L_1 and L_2, respectively. Design a machine M_3, which takes $w = a_1a_2 \cdots a_n$ with each $a_i \in \Sigma$ as input, as the following:

$M_3(w)$:

1 For $i = 0, 1, 2, ..., n$

2 Run M_1 with input $w_1 = a_1a_2...a_i$ and
run M_1 with input $w_2 = a_{i+1}a_{i+2}...a_n$.
If both M_1 and M_2 accepts w, then *accept*.

3 If none of the iterations in Stage 2 accept, then *reject*.

It is clear that M_3 accepts w if and only if $w = w_1w_2$, where M_1 accepts w_1 and M_2 accepts w_2. Also, M_3 runs in polynomial time.

(4) $\overline{L_1} \in P$

Since L_1 is in P, there exists machines M_1 that decides L_1. Design a machine M_2 which takes w as input as the following:

$M_2(w)$:

1 Run M_1 with input w. If M_1 accepts w, then *reject*, else *accept*.

It is clear that M_2 accepts w if and only if M_1 rejects w, and M_2 runs in polynomial time.

(5) $L_1^* \in P$

Since L_1 is in P, there exists machines M_1 that decides L_1. By using dynamic programming, a machine M_2 that decides L_1^* in polynomial time, which takes $w = w_1 w_2 ... w_n$ as input, can be constructed as follows:

$M_2(w)$:

1 **if** $w = \epsilon$, then *accept*
2 **for** $i = 1$ to n
3 **for** $j = 1$ to n
4 $A[i, j] = 0$
5 **for** $i = 1$ to n
6 set $A[i, j] = 1$ if $w_i \in L_1$
7 **for** $l = 2$ to n
8 **for** $i = 1$ to $n - l + 1$
9 $j = i + l - 1$
10 set $A[i, j] = 1$ if $w_i ... w_j \in L_1$
11 **for** $k = i$ to $j - 1$
12 set $A[i, j] = 1$ if $A[i, k] = 1$ and $A[k, j] = 1$
13 **if** $A[1, n] = 1$ then *accept*
14 **else** *reject*

Since each step takes polynomial time and there are $O(n^3)$ steps in it, it takes polynomial time for M_2 to decide L_1^*.

b. (9%) Prove that $P \subseteq$ *co-NP* (Problem 34.2-9 in the textbook). *co-NP* is the set of

languages L such that $\overline{L} \in NP$.

Sol:

Let M_1 be the machine which decides L, where $L \in P$. For any $x \notin L$, design a machine M_2 which accepts x in polynomial time as follows:

$M_2(x)$:

1 Run M_1 with input x. If M_1 rejects x, then *accept*.

Therefore, $\overline{L} \in NP$, and thus $L \in co\text{-}NP$.

Problem 3. (24%) Solve problem 34.3 on p.1103 in the textbook. The description for you to understand problem c and d is illustrated in Figure 1.

Sol:

a. First, we randomly choose a vertex in $r \in V$ and color it with 0, then we can do a BFS starting from r to color on graphs so that every child of a node $v \in V$ is colored differently than the color of v itself. By checking whether every adjacent pair of nodes has exactly two different colors, we can decide on whether the graph is 2-colorable or not.

b. Given an undirected graph G and an integer k, the decision version of the graph-coloring problem is to determine whether there exists a function $c : V \to \{1, 2, ..., k\}$ such that $c(u) \neq c(v)$ for every edge $(u, v) \in E$.

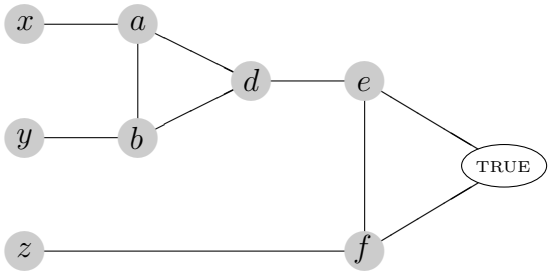
   Suppose the decision problem is solvable in polynomial time. Run the decision algorithm for at most $|V|$ times to ask whether G is r-colorable, starting from $r = 1$ to $|V|$. Once the algorithm returns TRUE at the i-th iteration, then the minimum number of colors needed to color G must be i.

   On the other direction, suppose that the graph-coloring problem is solvable in polynomial time. Run the algorithm to find the minimum number r such that G is r-colorable. Then given any $k \in \mathbb{N}$, we can determine whether G is k-colorable by examining whether k is larger or equal than r.

c. Let $k = 3$ and run our algorithm for solving the decision problem in part b. Then given any undirected graph $G = (V, E)$, G is 3-colorable if and only if G is k-colorable. Since 3-COLOR is NP-complete, the decision problem must be NP-complete as well.

d. Since each literal and its negation must be colored distinctly by colors other than c(RED), they must be colored by c(TRUE) and c(FALSE), one for each. We can also see that there exists a 3-coloring of the graph containing just the literal edges by coloring it in this way.

e. Here we label the nodes in the widget as follows:



If each of x, y and z is colored c(TRUE) or c(False), to prove that at least one of x, y and z is colored c(TRUE), it suffices to show that $c(x), c(y)$ and $c(z)$ cannot all be c(FALSE). Suppose on the contrary, we can see from the widget that

(1) if $c(a) = c(\text{TRUE})$, then $c(b) = c(\text{FALSE}) \Rightarrow c(d) = c(\text{TRUE}) \Rightarrow c(e) = c(\text{RED})$;

(2) if $c(a) = c(\text{RED})$, then $c(b) = c(\text{TRUE}) \Rightarrow c(d) = c(\text{FALSE}) \Rightarrow c(e) = c(\text{RED})$.

In either of the above two cases, there is no way to color the node f appropriately. Hence the widget is not 3-colorable, and vice versa.

f. Combining the results in part d and part e to construct the graph for each clause in the given 3-CNF, we can see that the 3-CNF is satisfiable if and only if the corresponding graph is 3-colorable, thus we get a polynomial-time reduction from 3-CNF-SAT to 3-COLOR. Since 3-CNF-SAT is NP-complete, 3-COLOR is also NP-complete.

Problem 4. (12%) A graph $G_1 = (V_1, E_1)$ is said to be isomorphic to a graph $G_2 = (V_2, E_2)$ if there exists a one-to-one correspondence $f : V_1 \to V_2$ such that for any two

vertices u and v of G_1, $(u, v) \in E_1$ if and only if $(f(u), f(v)) \in E_2$. That is, u and v are adjacent in G_1 if and only if $f(u)$ and $f(v)$ are adjacent in G_2. The *subgraph-isomorphism problem* takes two undirected graphs G_1 and G_2, and it asks whether G_1 is isomorphic to a subgraph of G_2. Show that the subgraph-isomorphism problem is NP-complete. Hint: This problem is related to the *clique problem* (Section 34.5.1 on p.1086-1089 in the textbook).

Sol: We can reduce this problem to the clique problem. Given an undirected graphs G_2, let $G_1 = K_i$, which is the complete graph of i vertices, where $i \leq |G_2|$. Then G_1 is the subgraph of G_2 if and only if G_2 has a clique of size i. By the above problem reduction, it is clear that if the clique problem can be solved in polynomial time, the the subgraph isomorphism problem can be solved in polynomial time. Since the clique problem is NP-complete, the subgraph isomorphism problem is also NP-complete.

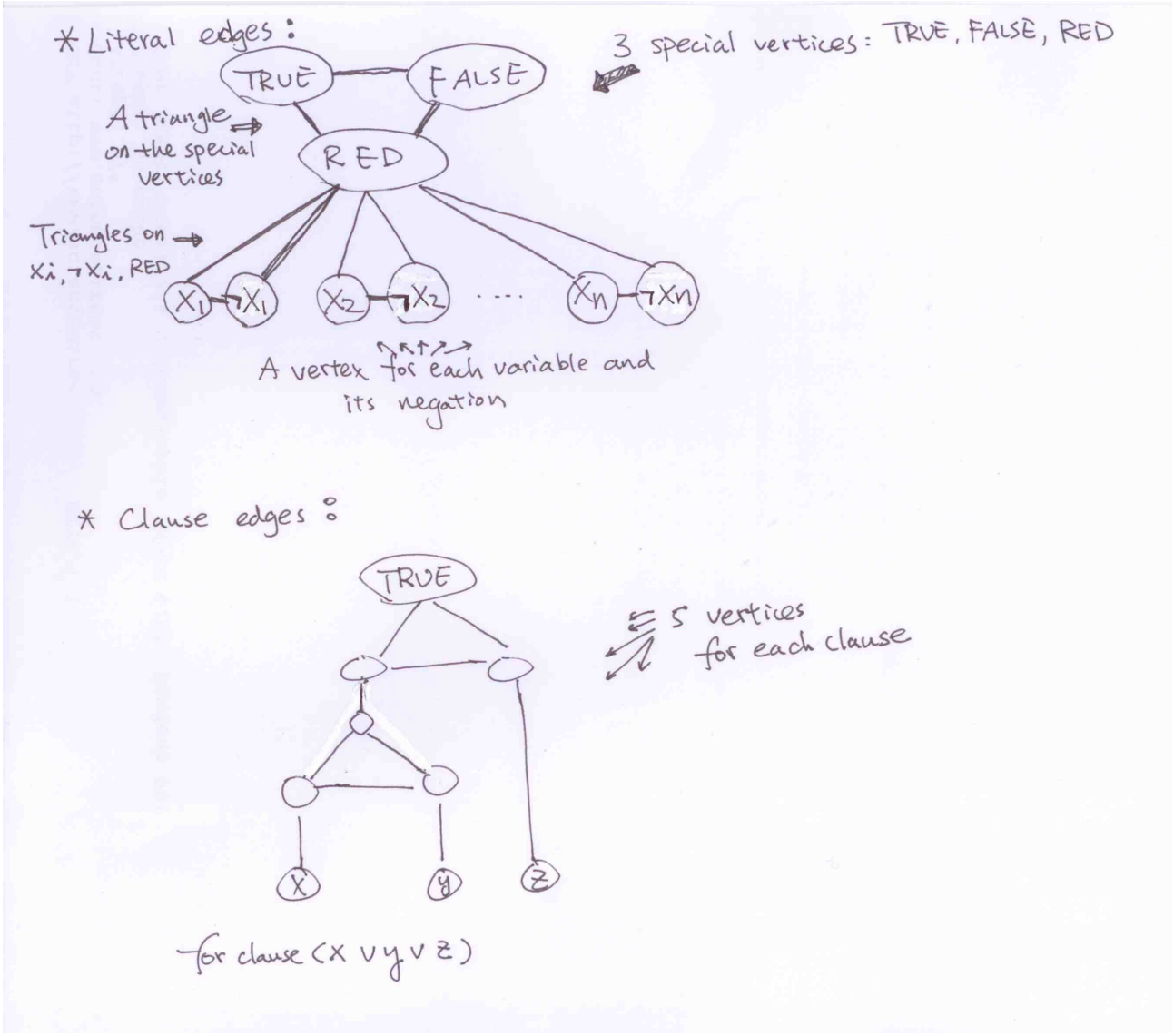


Figure 1: Clarification of some descriptions in Problem 3