Solutions to FIT5047 Tutorial on Problem Solving as Search: Graphsearch

Exercise 1: Search Algorithms

(a) Why can you stop BFS when you generate/find the goal, i.e., you don't need to wait for the goal to reach the top of the list in OPEN?

SOLUTION:
Because all the nodes generated later will have the same or greater cost than the cost of the nodes just generated.

(b) How is DFS (or DLS) different from Backtrack?

SOLUTION:
Backtrack generates one child at a time, and deletes it when it backtracks. DFS generates all the children of a node, and deletes all of them when a solution has not been found down that path.

(c) Show that if $h(n)$ overestimates $h^*(n)$, Algorithm A may return a non-optimal solution. You can use an example.

SOLUTION:
Say we have two nodes A and B. Assume

- $g^*(A) = g(A) = g(B) = g^*(B) = g$,
- $h^*(A) = 8$ and $h^*(B) = 10$, but
- $h(A) = 12 > h^*(A)$ and $h(B) = 11 > h^*(B)$.

So, $f(A) = g(A) + h(A) = g + 12$ and $f(B) = g(B) + h(B) = g + 11$. Therefore, we expand B, reaching the goal with $f^*(B) = g + 10$.

Since $f^*(B) = g + 10 < f(A) = g(A) + h(A) = g + 12$, A is not expanded, and the optimal solution of $g^*(A) + h^*(A) = g + 8$ is not found.

(d) Why does monotonicity of h imply that when you expand a node you have found the optimal path to it?

SOLUTION (slide 85):
Monotonicity is a stricter requirement than admissibility. If a heuristic is monotonic, no pointer redirection is required for the nodes in CLOSED.

Consider a start node s with three children n_1, n_2 and n_3.

Say $f(n_1) = g(n_1) + h(n_1) \leq f(n_i)$ for $i = 2, 3$, so n_1 is expanded first, yielding n_4 and n_5.

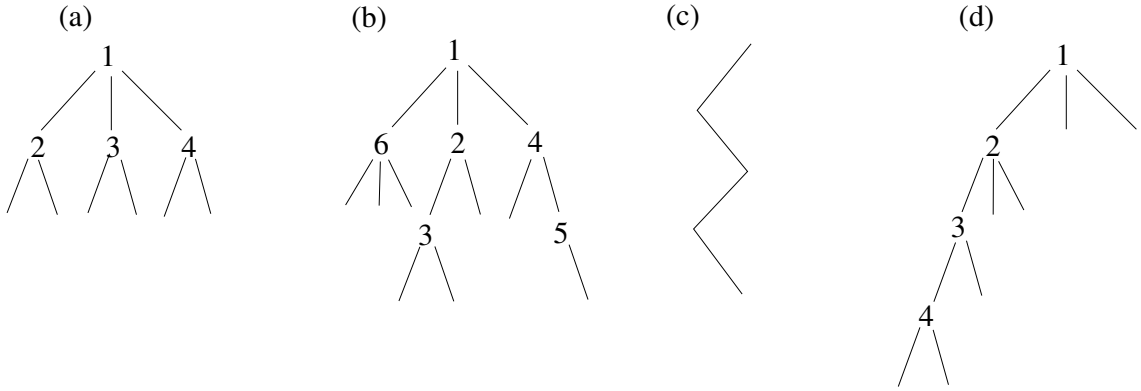
Now, n_4 is expanded, which means that
$f(n_4) = g(n_4) + h(n_4) = g(n_1) + c(n_1, n_4) + h(n_4) \leq f(n_i)$ for $i = 2, 3, 5$.

Now say that n_2 is expanded, and it has an arc to n_4. If h is monotonic, the path going to n_4 through n_2 is worse than the previously found path to n_4 (through n_1) because:
$f(n_4) = g(n_1) + c(n_1, n_4) + h(n_4) \leq f(n_2) = g(n_2) + h(n_2) \leq g(n_2) + c(n_2, n_4) + h(n_4)$

Hence, $g(n_1) + c(n_1, n_4) \leq g(n_2) + c(n_2, n_4)$ [going through n_1 < going through n_2].

Exercise 2: Search Trees

Each of the following search trees has a distinctive structure and order of expansion which can be produced by a particular search procedure. For each tree write the name of the search procedure which can generate it. The possible names are: backtrack, breadth first search, depth first search and A (with non-zero g and h). Where applicable, the nodes are labeled with the order in which they are expanded.



SOLUTION:

(a) Breadth First Search, but it could also be A or A*.

(b) A or A*.

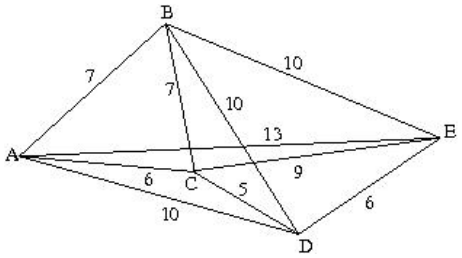
(c) Backtrack, but it could also be Depth First Search (if each expansion has one option only), or A (or A*) if the expanded node has the best f value and there is only one option.

(d) Depth First Search, but it could also be A (or A*) if the expanded node has the best f value.

Exercise 3: Algorithm A/A*

Consider the Traveling Salesman Problem:

A salesman must visit each of n cities. There is a road between each pair of cities. Starting at city #1, find the route of minimal distance that visits each of the cities only once and returns to city #1.



(a) Propose two (non-zero) h functions for this problem. Is either of these h functions a lower bound of h^*? Apply algorithm A with these h functions to the 5 city problem below.

SOLUTION:

1. The Graphsearch algorithm expands a node by generating all its successors that are **not** ancestors of this node. Once all the successors are established, they are put in a list called *OPEN*, which may contain other nodes that have not been expanded yet. The algorithm then selects the next node to be expanded from **all** the nodes in *OPEN*. This selection may either be arbitrarily performed, or according to heuristic merit. Arbitrary selections are: *expand the node that is lowest in the tree*, which leads to a depth-first-search strategy, or *expand the node which is highest in the tree*, which leads to a breadth-first-search strategy. In any event, keep in mind that **all the nodes in OPEN are taken into consideration**, not just the successors of the last node expanded[1].
2. Algorithm A is a type of Graphsearch algorithm which uses the following function to determine which node to expand next:
   $f(n) = g(n) + h(n)$
   Where:
   - *g(n)* - is the **cumulative** cost of arriving from the start node to node n, i.e., the cost of the entire path from start to n.
   - *h(n)* - is an estimate of the cost of arriving from node n to the goal node. This estimate does **not** include the cost of arriving to node n.
     If we can infer that this estimate is always less than the cost of the best path from node n to the goal (which we don't know at the moment), then A becomes A*.

   Thus *f(n)* is a composite measure of the effort or cost incurred in reaching node n plus the cost we are expecting to incur in order to arrive from node n to the goal. In other words, it is a measure of the cost of getting from the start to the goal **through node n**. You should keep in mind, that algorithm A **has to consider the cost of arriving at node n**. If it disregards this cost, then it is merely considering the promise of a particular node (e.g., node n), without taking into consideration how much it costs to arrive at this node. This would yield to a strategy similar to depth-first-search, and by definition, it would no longer be algorithm A.

HEURISTIC FUNCTION

There are a number of heuristic functions that can be applied to this problem, we shall now present a few, and thereafter we shall present the search-tree reached for the last of these functions.

1. h_1 = Number of cities left to be visited.
   This function represents a lower bound to *h**. However it does not provide information with respect to the relative promise of each path, since at each level the remaining number of cities to be visited is the same.
2. h_2 = Distance to the closest city.
   This function is also less than *h**, since it takes into consideration only the closest city (which, clearly, has to be visited). It disregards the constraint that all cities should be visited.
3. $$h_3 = \sum_{i \in C'} \min_{x \in \text{neighbour}(i)} (dist(x \to i)) \quad (1)$$
   where C' = {cities left to be visited}.

[1] This description does not include the pointer-redirection operations performed in step 7 of algorithm Graph-search in the class handout.

This function takes into consideration the minimum distance to and from each city. It disregards the constraint that we have to leave the city where we are now and the fact that we cannot depart from most previously visited cities, except the last one. It is also a lower bound of h^*.

4.
$$h_4 = \sum_{i \in C'} \min_{x \in C''}(dist(x \to i)) \tag{2}$$

where C' = cities not yet visited, and

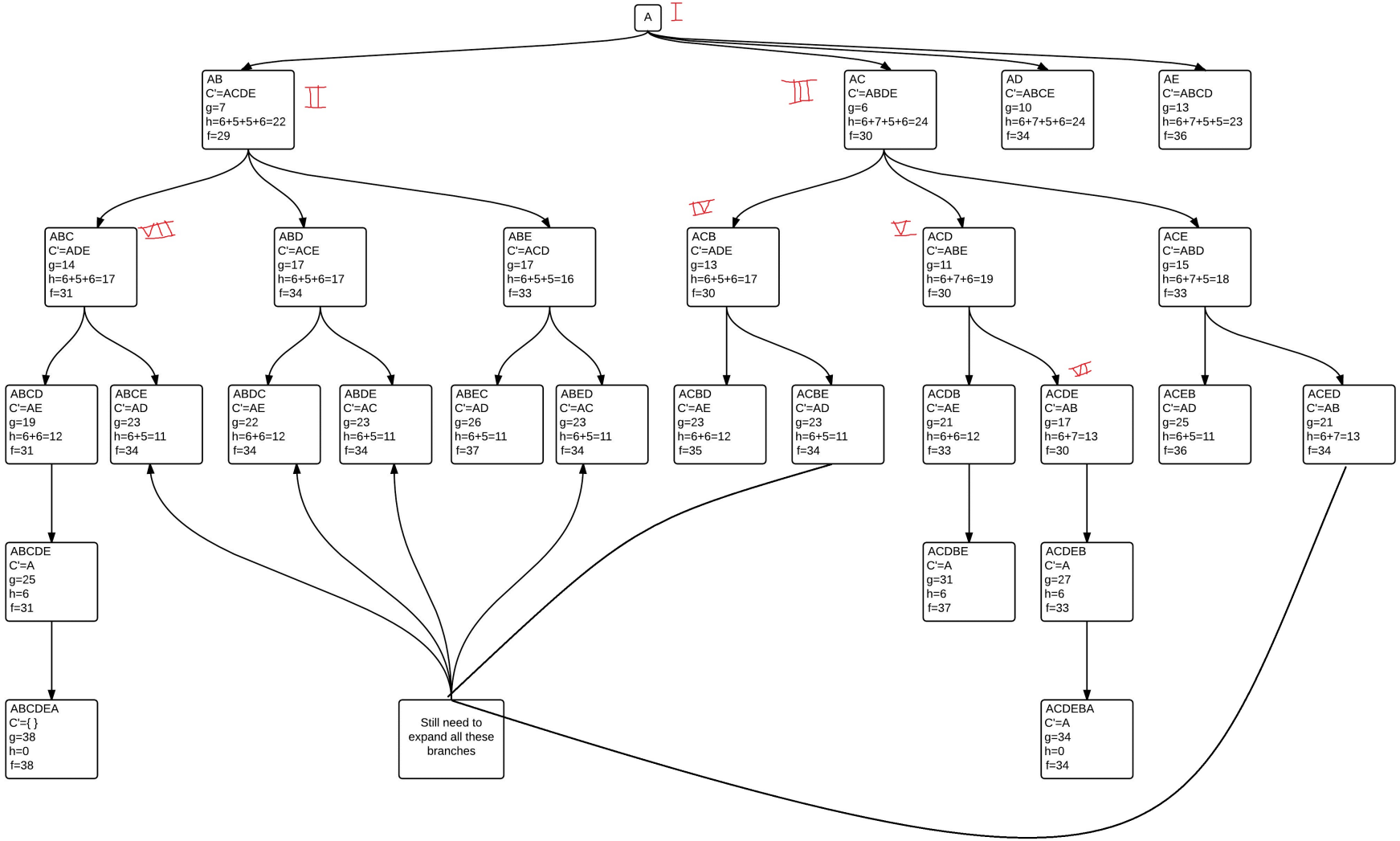
- For $i \neq A$: C'' = {C' - A} $\bigcup$ {last-city-visited} (for each city i not yet visited, except A, which is visited at the end, we consider the distance from {all the not-yet-visited neighbours of i plus the last visited city} to i)
- For $i = A$ and $|C'| > 1$: C'' = {C' - A}
- For $i = A$ and $|C'| = 1$: C'' = {last city visited}

The difference between h_3 and h_4 is that h_3 considers the minimum distance to a particular city **from all the cities**, while h_4 takes into consideration only the cities that remain to be visited + the last city visited (except for A, which must be the last city to be visited). Thus, the estimate provided by h_4 is more accurate than the one provided by h_3, but still is a lower bound of h^*.
For example, in the given problem, if we already visited ABC, in this order, and D,E and A remain to be visited.

$$\begin{aligned} h_3 &= \min(AD, BD, CD, ED) + \min(AE, BE, CE, DE) + \\ &\quad \min(BA, CA, DA, EA) \\ &= 5 + 6 + 6 = 17 \end{aligned}$$

$$h_4 = \min(CD, ED) + \min(CE, DE) + \min(DA, EA) = 5 + 6 + 10 = 21 \tag{3}$$

(b) The following diagram contains the search tree produced by algorithm A when using h_3.


A
I
AB
C'=ACDE
g=7
h=6+5+5+6=22
f=29
II
AC
C'=ABDE
g=6
h=6+7+5+6=24
f=30
III
AD
C'=ABCE
g=10
h=6+7+5+6=24
f=34
AE
C'=ABCD
g=13
h=6+7+5+5=23
f=36
ABC
C'=ADE
g=14
h=6+5+6=17
f=31
VII
ABD
C'=ACE
g=17
h=6+5+6=17
f=34
ABE
C'=ACD
g=17
h=6+5+5=16
f=33
ACB
C'=ADE
g=13
h=6+5+6=17
f=30
IV
ACD
C'=ABE
g=11
h=6+7+6=19
f=30
V
ACE
C'=ABD
g=15
h=6+7+5=18
f=33
ABCD
C'=AE
g=19
h=6+6=12
f=31
ABCE
C'=AD
g=23
h=6+5=11
f=34
ABDC
C'=AE
g=22
h=6+6=12
f=34
ABDE
C'=AC
g=23
h=6+5=11
f=34
ABEC
C'=AD
g=26
h=6+5=11
f=37
ABED
C'=AC
g=23
h=6+5=11
f=34
ACBD
C'=AE
g=23
h=6+6=12
f=35
ACBE
C'=AD
g=23
h=6+5=11
f=34
ACDB
C'=AE
g=21
h=6+6=12
f=33
ACDE
C'=AB
g=17
h=6+7=13
f=30
VI
ACEB
C'=AD
g=25
h=6+5=11
f=36
ACED
C'=AB
g=21
h=6+7=13
f=34
ABCDE
C'=A
g=25
h=6
f=31
ACDBE
C'=A
g=31
h=6
f=37
ACDEB
C'=A
g=27
h=6
f=33
ABCDEA
C'={}
g=38
h=0
f=38
Still need to expand all these branches
ACDEBA
C'=A
g=34
h=0
f=34

The following diagram contains the search tree produced by algorithm A when using h_4.

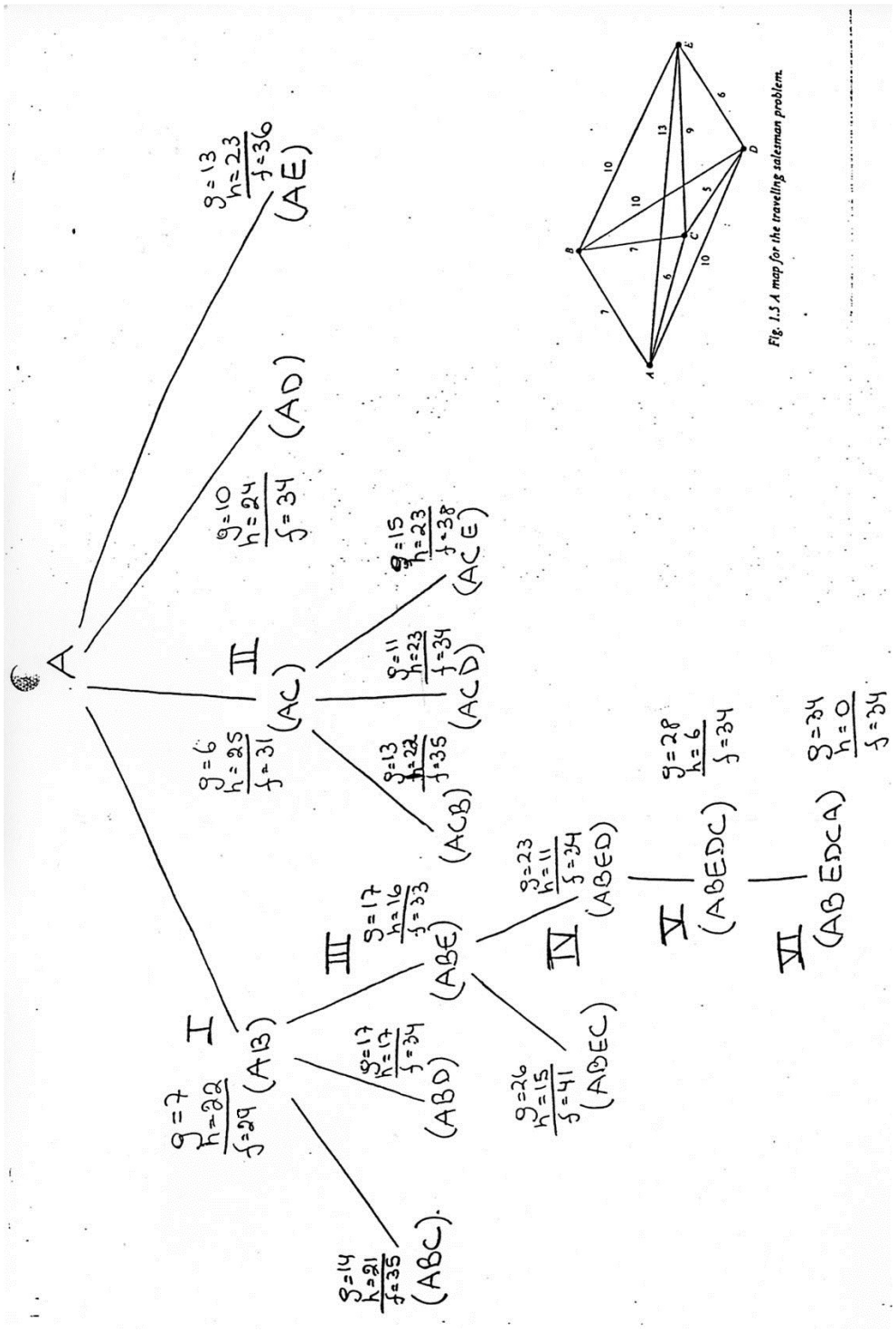


Fig. 1.5 A map for the traveling salesman problem.

(c) How would you use Hill Climbing to solve this problem? How about simulated annealing? Give an example with different temperatures, and illustrate a few steps.

Hill Climbing SOLUTION:

Start with some configuration, say one obtained by a Greedy algorithm, and then apply changes. You have one operator:

Swap(i,j) – Swap city i with city j.

However, if we consider all the permutations, we have combinatorial explosion. If we only consider neighbouring cities, we might not get the optimum.

Starting with the Greedy solution: A-C-D-E-B-A (34):

Swap C and D $\Rightarrow$ A-D-C-E-B-A (41)

Swap D and E $\Rightarrow$ A-C-E-D-B-A (38)

Swap E and B $\Rightarrow$ A-C-D-B-E-A (44)

Actually, the Greedy solution is optimal.

==========

Simulated Annealing SOLUTION (slide 101):

Note that this is a **minimization** problem, while the algorithm in the slide is for maximization. So, I have used ΔE = Value(new-state) − Value(current-state).

Starting with the Greedy solution: A-C-D-E-B-A (34) – keep this solution as Best-so-far.

Swap C and D $\Rightarrow$ A-D-C-E-B-A (41) – with probability Pr = $e^{-\frac{\Delta E}{T}}$ keep this solution, where

$\Delta E = 41 - 34 = 7$ and $T = 100 \Rightarrow$ Pr = $e^{-\frac{7}{100}} = 0.9324$

What would be the probability if $T = 1000$?

$e^{-\frac{7}{1000}} = 0.993$

Now, swap C and E $\Rightarrow$ A-D-E-C-B-A (10+6+9+7+7=39) – since this solution is better than 41, keep it (but Best-so-far is still the best).

Exercise 4: Algorithm A*

Consider a modified version of the 8-puzzle, where moving the blank tile to the center square has a cost of 4 (the rest of the moves cost 1). The initial configuration is:

2	8	3
1	6	4
7		5

Where the goal 8-puzzle is the following:

1	2	3
8		4
7	6	5

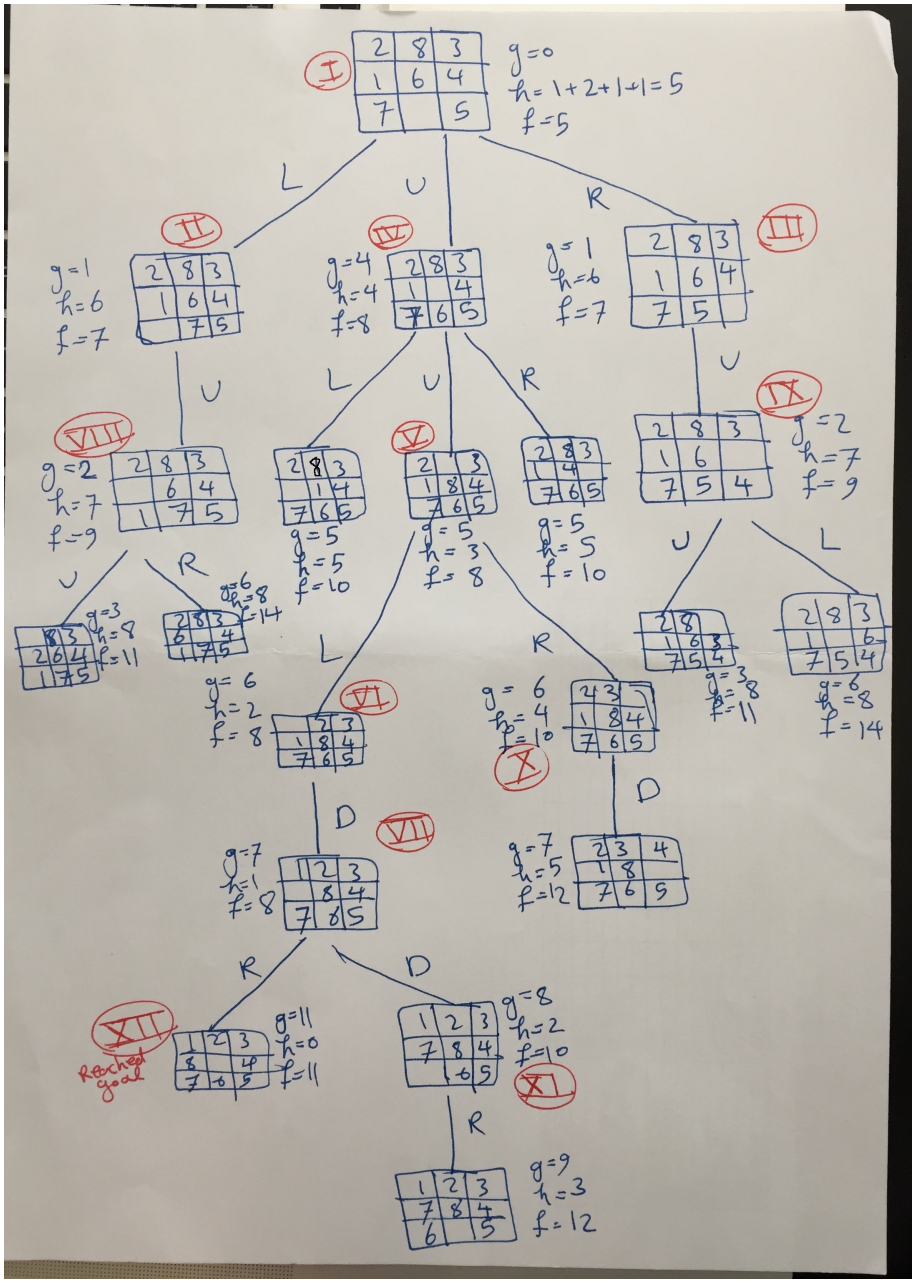
(a) Apply algorithm A* to find the shortest path between the initial and the final configuration. Use Manhattan Distance as your heuristic function.

In the Search Tree produced by the algorithm, indicate clearly the order of expansion of each node (i.e., when is a node **expanded**, not generated), and the values of *g, h* and *f*. Label the moves clearly.

(b) Describe what happens upon reaching the goal state. If you think that A* requires any modifications to improve its efficiency for this problem, describe and justify the modifications you envisage.

SOLUTION:

(a) The solution tree is shown below.



(b) Upon reaching the goal state, subtract 3 from the final g. This will reduce the number of states whose f is less than the f of the goal node, thereby reducing the number of nodes inspected. This modification is justified, since all the paths to the goal state will have to move the blank tile into the center, thereby adding a cost of 4.

The resultant solution would expand the goal node in the eighth expansion (VIII), instead of the twelfth (XII).