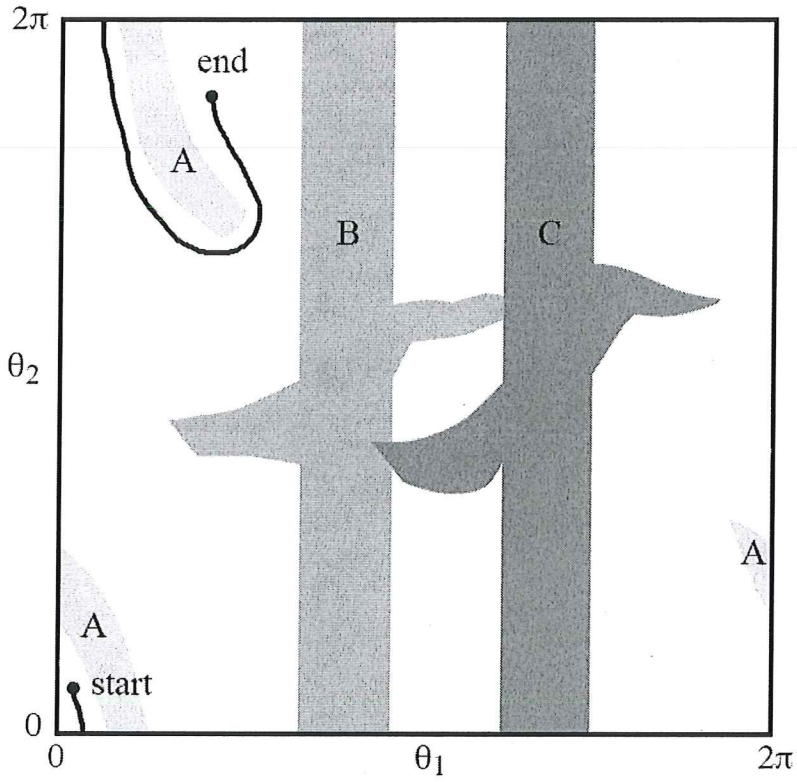
HW5 Chi Zhang

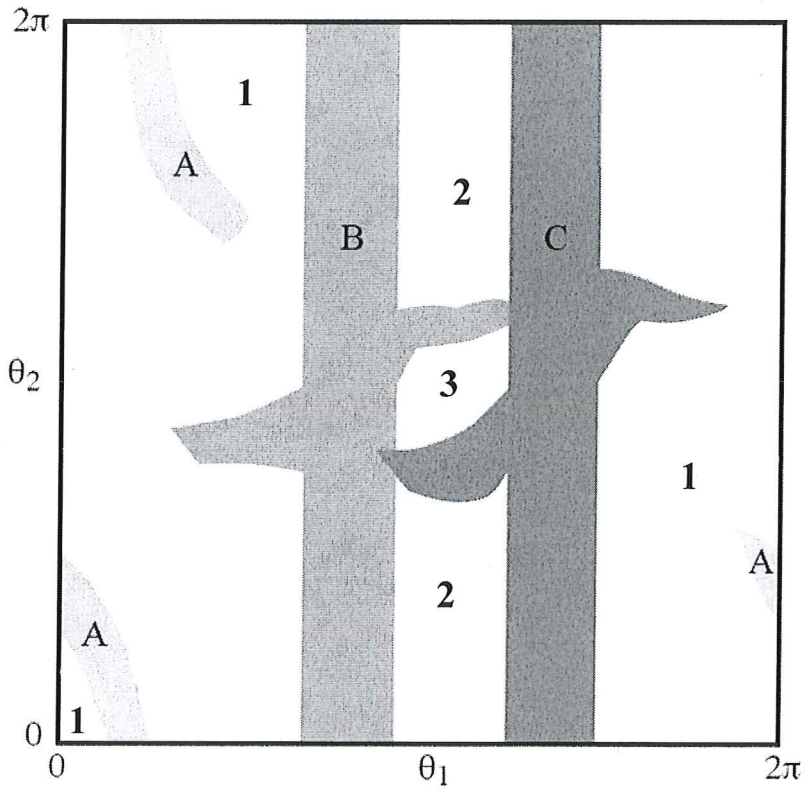
60/60

1.



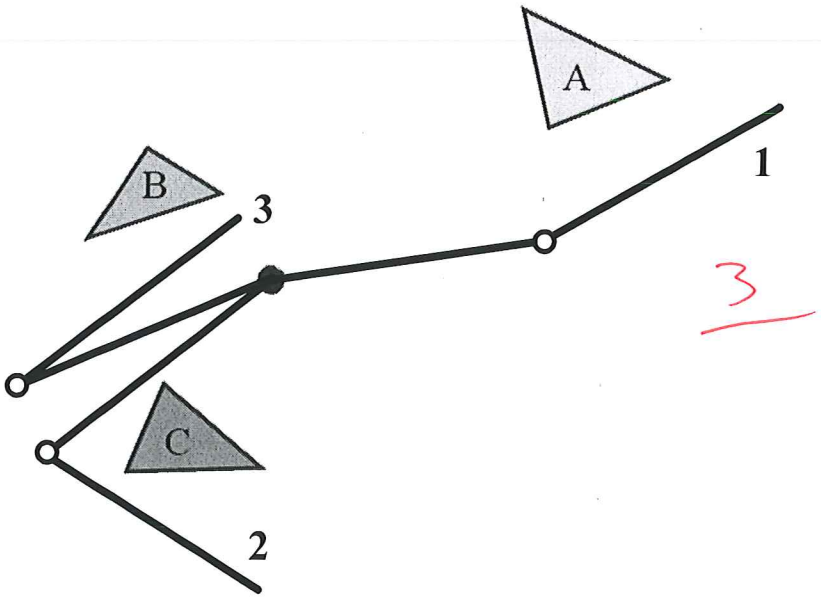
5

2.



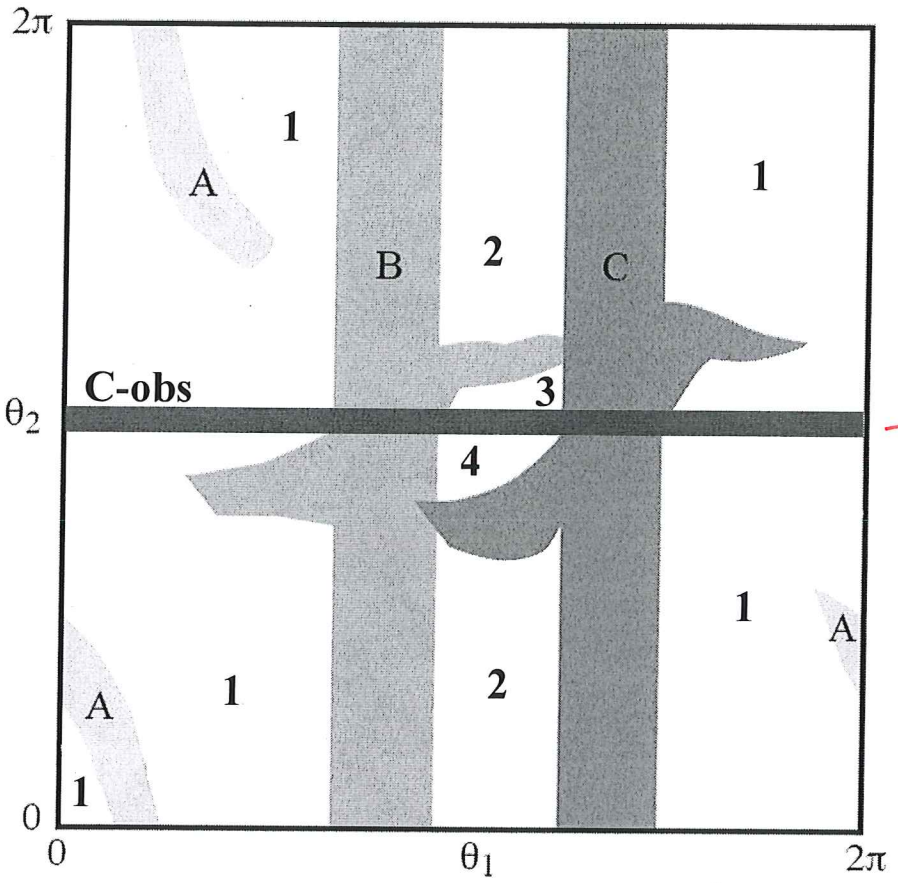
3

8



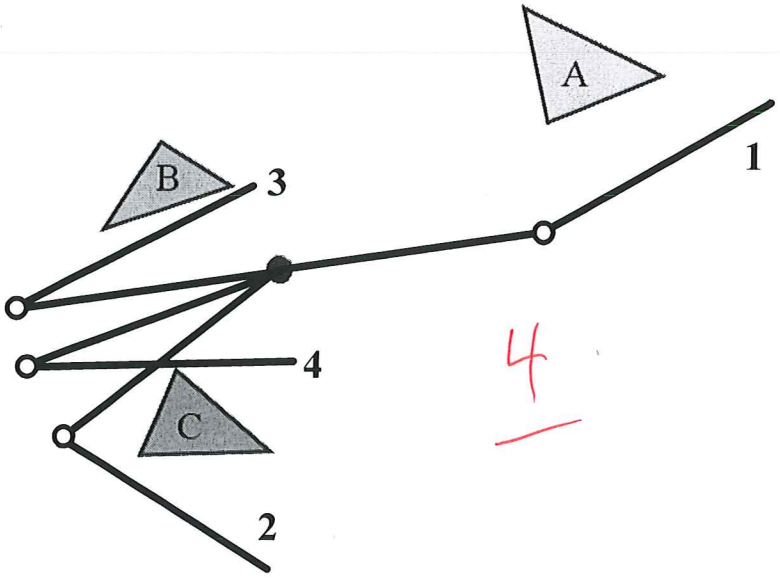
3

3.



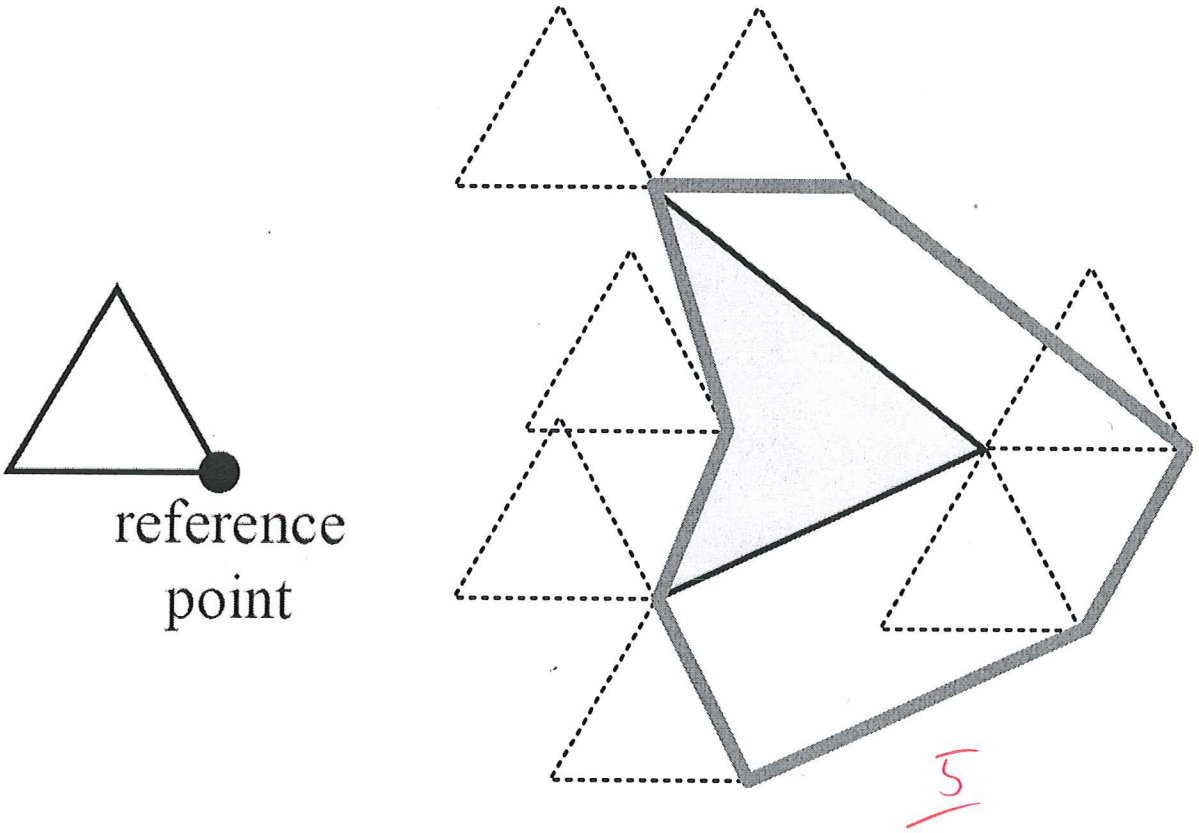
4

7



4

4.



5

(a)

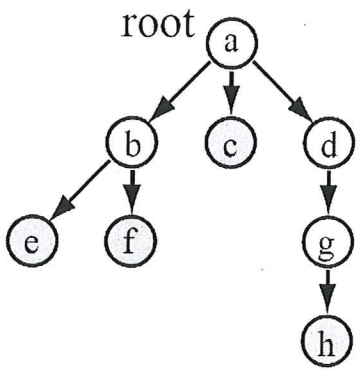
7.

Breadth-First-Search:

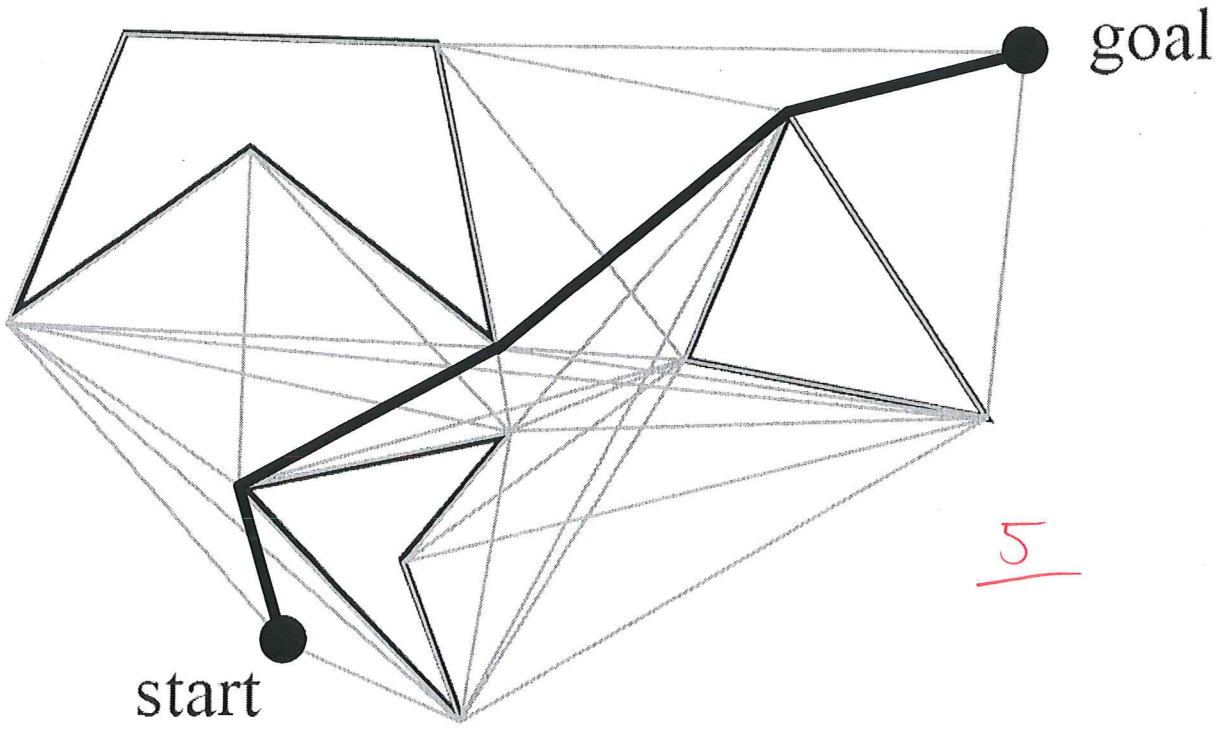
a – b – c – d – e – f – g – h. 3

Depth-First-Search:

a – b – e – f – c – d – g – h. 3



8.



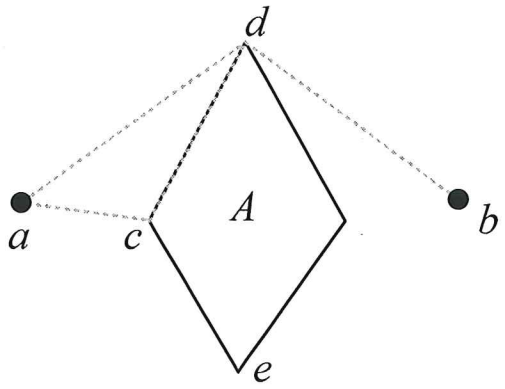
5

(11)

9.

As shown below, "hit the obstacle tangentially" can also be described as "glance the obstacle instead of running into it".

As shown in the figure below, assume that in the optimal path, the path has to go bypass obstacle A from node *a* to *b*. If the optimal path contains edge *a-c*, which is an edge running into the obstacle, then it has to go along edge *c-d* or c-e to go round the obstacle A. In either case, let's say *a-c-d*, it is longer than the tangential edge *a-d*, as in Δacd, $ad < ac + cd$. Therefore, for any edge that runs into the obstacle, another edge that goes along the obstacle is required for the robot to round it, and these 2 edges will be replaced by a shorter edge glancing the obstacle.



5

10. The code and the graph is shown below:

```
function[runtime,success_flag] = P10()
%Generate a random undirected graph and find the shortest path with A* search
%The function returns the runtime in ms of A* search and whether there
%exists a path.
clear;
clc;
N=10;
E=20;
% The random graph contains N nodes and E edges

position = cell(N,1);
for i=1:N;
    position{i}=[100*rand(1),100*rand(1)];
end
%Generate N nodes' x and y coordinate with random numbers in the 100*100 grid

edge = cell(E,1);
i=1;
available_table = triu(ones(N,N),0) - eye(N);
%For the edges, first define an N*N 0-1 matrix representing the possibility
%of adding edges between the two nodes. As this is an undirected graph,
%edge i-j and j-i are the same, so an upper triangle matrix is used here.
%Also, there is no edge i-i, so the diagnal is 0. Therefore, the
%initialized available matrix would be an upper triangle matrix with all
%elements equals to 1, each of which representing a possible edge between
%node i and j.

while (i<=E && sum(sum(available_table))~=0)
    rand_edge = randi(sum(sum(available_table)));
    count = 0;
    break_flag = 0;
    for k = 1:N-1
        for l = k+1:N
```

10

15

```
            if(available_table(k,l) == 1)
                count = count + 1;
            end
            if(count == rand_edge)
                edge{i}(1) = k;
                edge{i}(2) = l;
                available_table(k,l) = 0;
                break_flag = 1;
                break;
            end
        end
        if(break_flag)
            break;
        end
    end
    i = i + 1;
end
%While the number of edges is not E and the available table is not empty,
%generate a random number with the max of the sum of all "1"s in the
%available matrix, which means from the first element representing
%edge 1-2, count all available "1"s till the random number generated, then
%add this edge to the graph, and change the availability of this edge to "0".

%Plot the random graph generated above
for i=1:E
    x = [position{edge{i}(1)}(1),position{edge{i}(2)}(1)];
    y = [position{edge{i}(1)}(2),position{edge{i}(2)}(2)];
    plot(x,y,'g','linewidth',1.5);
    hold on;
end

for i=1:N
    plot(position{i}(1),position{i}(2),'.','MarkerSize',18);
    hold on;
end
axis equal;
axis([0 100 0 100]);


%Use a matrix to memorize the length of edges. If there is no edge between
%the two nodes then set the length infinity. 1000 is large enough here.
edge_length_table = 1000*(ones(N,N)-eye(N));
for i=1:E
    d = sqrt((position{edge{i}(2)}(1)-position{edge{i}(1)}(1))^2 +
(position{edge{i}(2)}(2)-position{edge{i}(1)}(2))^2);
    edge_length_table(edge{i}(1),edge{i}(2)) = d;
    edge_length_table(edge{i}(2),edge{i}(1)) = d;
end

%Initialization for A* Search, initialize the variables needed. The OPEN
%set memorizes both the number of nodes to be explored and the estimated
%total cost.
past_cost = 10000*ones(1,N);
past_cost(1) = 0;
parent = [];
est_total_cost = [];
d0 = sqrt((position{1}(1)-position{N}(1))^2+(position{1}(2)-position{N}(2))^2);
OPEN = [1,d0];
CLOSED = [];
path = [];
success_flag = 0;
```

```
%Use an N*1 cell to memorize all the neighbours of all nodes, judging
%easily by the length of edges in the edge length matrix.
nbr = cell(N,1);
for i=1:N
    for k=1:N
    if(edge_length_table(i,k)~=0 && edge_length_table(i,k)~=1000)
        nbr{i} = [nbr{i},k];
    end
    end
end

%A* Search, use the psudo code in the book, use tic-toc to calculate
%runtime
tic;
while ~isempty(OPEN)
    current = OPEN(1,1);
    OPEN(1,:) = [];
    CLOSED = [current,CLOSED];
    if current==N
        success_flag = 1;
        break;
    end
    for i=1:length(nbr{current})
        if ~ismember(nbr{current}(i),CLOSED)
            tentative_past_cost = past_cost(current) +
edge_length_table(current,nbr{current}(i));
            if tentative_past_cost < past_cost(nbr{current}(i))
                past_cost(nbr{current}(i)) = tentative_past_cost;
                parent(nbr{current}(i)) = current;
                d = sqrt((position{nbr{current}(i)}(1)-
position{N}(1))^2+(position{nbr{current}(i)}(2)-position{N}(2))^2);
                est_total_cost(nbr{current}(i)) = past_cost(nbr{current}(i)) + d;
                OPEN = [OPEN;nbr{current}(i),est_total_cost(nbr{current}(i))];
            end
        end
    end
    OPEN = sortrows(OPEN,2);
end
runtime = 1000*toc;

%If A* search runs successfully, then return the path from 1 to N and draw
%it on the graph: Simply by picking node N, and then the parents of previous
%nodes till node 1.
if success_flag
    i = 1;
    trace = N;
    while(trace~=0)
        path(i) = trace;
        trace = parent(trace);
        i = i + 1;
    end

    for i=1:(length(path)-1)
        x = [position{path(i)}(1),position{path(i+1)}(1)];
        y = [position{path(i)}(2),position{path(i+1)}(2)];
        plot(x,y,':k','linewidth',2.0);
        hold on;
    end
end

%Plot node 1 and node N in a different color
plot(position{1}(1),position{1}(2),'.k','MarkerSize',22);
```

11. Revise code for problem 10 so that it runs 2 different A* search on the same graph and returns the runtime, distance and success-or-not sign for the 2 searches, then use the following code to calculate the max, min and average distance and runtime. The results are shown below the code.

```
clear;
clc;
M = 100;
search_time1 = 0;
success_time1 = 0;
search_time2 = 0;
success_time2 = 0;
avg_distance1 = 0;
avg_distance2 = 0;
max_dist1 = 0;
min_dist1 = 200;
max_dist2 = 0;
min_dist2 = 200;
max_time1 = 0;
min_time1 = 1000;
max_time2 = 0;
min_time2 = 1000;

for i=1:M
    [runtime1,success_flag1,distance1,runtime2,success_flag2,distance2]=A_star();
    search_time1 = search_time1 + runtime1;
    success_time1 = success_time1 + success_flag1;
    avg_distance1 = avg_distance1 + distance1;
    if max_dist1 <= distance1
        max_dist1 = distance1;
    end
    if (min_dist1 >= distance1 && distance1~=0)
        min_dist1 = distance1;
    end
    if max_time1 <= runtime1
        max_time1 = runtime1;
    end
    if min_time1 >= runtime1
        min_time1 = runtime1;
    end

    search_time2 = search_time2 + runtime2;
    success_time2 = success_time2 + success_flag2;
    avg_distance2 = avg_distance2 + distance2;
    if max_dist2 <= distance2
        max_dist2 = distance2;
    end
    if (min_dist2 >= distance2 && distance2~=0)
        min_dist2 = distance2;
    end
    if max_time2 <= runtime2
        max_time2 = runtime2;
    end
    if min_time2 >= runtime2
        min_time2 = runtime2;
    end
end

search_time1 = search_time1/M;
search_time2 = search_time2/M;
avg_distance1 = avg_distance1/success_time1;
avg_distance2 = avg_distance2/success_time2;
```

For convenience, we call the original A* search A*(1), and the one with 10 times large heuristic distance to go A*(2).

SEARCH	AVG. DISTANCE	MAX DISTANCE	MIN DISTANCE	AVG. RUNTIME	MAX RUNTIME	MIN RUNTIME
A*(1)	61.86	102.60	11.76	0.976	5.18	0.352
A*(2)	67.13	198.16	11.76	0.952	2.90	0.288

* Both of the success time is 100 in 100 tests, the time is in ms.

From the result we could see that on average A*(2) runs faster than A*(1), and when A*(1) can find a path, A*(2) can always find one. However, sometimes the path found by A*(2) is longer, it is because when the heuristic distance is over estimated, sometimes the path found by A*(2) will not be optimal. The figure below shows such an example, in which the full line is the path found by A*(1), and the dash line is the path found by A*(2), we can see clearly that path (2) is longer than path (1).

This result meats with the statement on the textbook, the suboptimal A* search on pp. 266.