Fundamental of AI Coursework
15-Puzzle-Problem

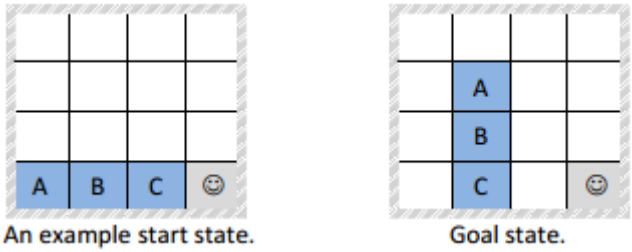
Lei Zhang
ID: 27678385
Email：lz3g15@soton.ac.uk

Abstract

This paper provides four methods for solving 15 puzzle problem, I programming using Matlab. The aim is to compare the differences between those four searching methods. The four searching methods I applied are DFS (deep first search), BFS (breadth first search), IDS (iterative depth search) and greedy search. The paper will gives the performances of all four methods in solving the problems.

Introduction

Brief introduction about the game, the game is an Non-competitive game, it played on a 4*4 board, as shown below, there is 3 characters and 1 block with a smile face inside, the face is a blank block, it can move to neighbors blocks and switch the position with it, and the player keep moving the face block to meet the goal state from the start state. In order to comment the performance comprehensively, I chose several different start states, with different Manhattan distance, at first I thought this should be a great and reasonable way to measure the difficulty, however it turns out sometime is not.



An example start state. Goal state.

Figure-1 Game Example

Four searching methods

DFS-Deep First Search

The first search I use is Deep First Search, this method will make the face block moves randomly in the board, then compare the current state with the goal state after each moves, if the state is the goal state, stop and return, otherwise keep moving until find the right state. This method is easy to apply but it will takes much cost to find the way, like moves millions steps to meet the goal. After I finish this part code, I find a problem, the direction of the face moves is settled before it start to search, the behavior of the block is the same as the DFS, just keep moving until find the right state, however the principle is a little bit different, so I improved it in IDS search. another problem is about the differences of tree search and graph search, tree search allowed to move back to visited block, which may make the block do nothing but move between two same blocks forever, so in order to prevent this situation, I use graph search for all four searching method.

BFS-Breadth First Search

Breadth first search is a better method of search, and the first goal it finds will be the optimal solution. BFS will find all available states to go, and moves to one of them first, then check is it the goal state, it true return and stop, if it false, jump to the other state just opened and check it out, if all states are not the goal state, moves to next depth and do it again until find the true state. however due to my poor programming skills, i didn't forbid the move the same state, in this situation, BFS will find the same state from two different states, and both store it and expand it, this will waste times and node for searching goal state.

IDS-Iterative Depth Search

Iterative Depth Search is almost the DFS, but one thing different, it limits the depth, so the DFS will not expand the node over the limits; it will check the current state, if it is the goal state, stop and return, otherwise, trackback to the previous state and expands another one. IDS can make the search more efficient, and prevent the block from stuck into circulation, and most important, it meet the goal fast. However, just like DFS, IDS may not give the best way towards the goal, in another words, it's not the shortest path.

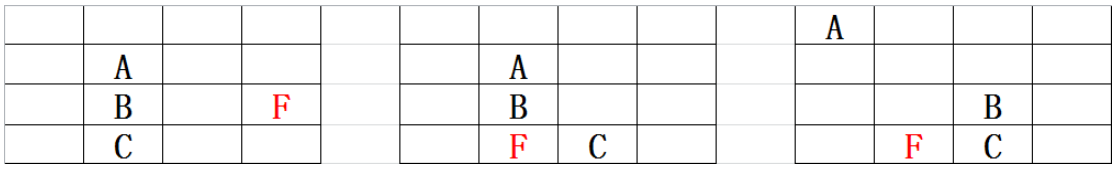
Greedy search

The three searching methods mentioned above all blind search, and now I applied another searching method, Greedy search. Unlike the three searching methods above, Greedy search is a kind of heuristic search. The difference between heuristic search and blind search is heuristic search pick directions. According to a function called greedy function, the face block will pick the node which is more close to the goal, and expand it. The greedy function is the Manhattan distance between the current state and goal state, if the distance is short, means this state is more probability to be right state towards the goal state.

Result and Analysis

This section is to present the result and analysis them, I will compare the performance of these four searching methods using 6 different initial state with different difficulties. The difficulties are defined by the Manhattan distance. The performance will be represented by the number of steps and how much states they searched. The aim is to find out the difference between those four methods.

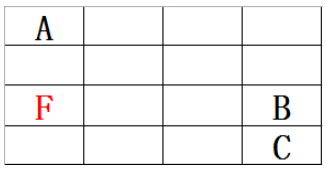
According to the Manhattan distance, I set 5 different initial states, their difficulty are 1,3,6,10 and 18. F represents the moving block. Every initial state I run 3 times to show the difference.



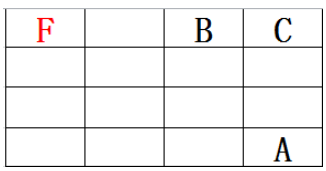
Difficulty-1

Difficulty-3

Difficulty-6



Difficulty-10



Difficulty-18

We can see from figure-2&3, even in low difficulty position, DFS need to search lots of state to find the right state and in the same initial state, the solution can be very different. That is because all moves in DFS are random; it's easy to stick in circulation or moves towards different directions. So whatever the difficulty is, it always searches many states.

However I find that if the character all distributed in the corner or the distance between them is far away like difficulty 10, DFS can get the answer in thousands steps however, IDS and BFS will spend tons of times to search states, the number of state can over million. That is because they need to expand several search paths, which takes a lot of time. This really surprises me. DFS can provide a path, however in most case, it won't be optimal, and however I couldn't ignore the performance when it deals the high level difficulties.

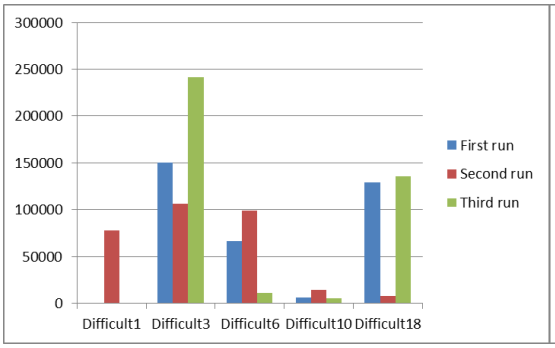


Figure-2 DFS Step Performance

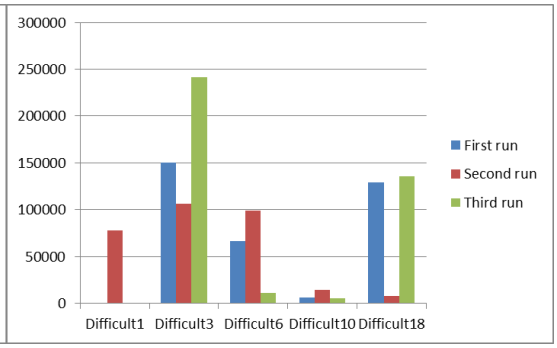


Figure-3 DFS Visited Nodes Performance

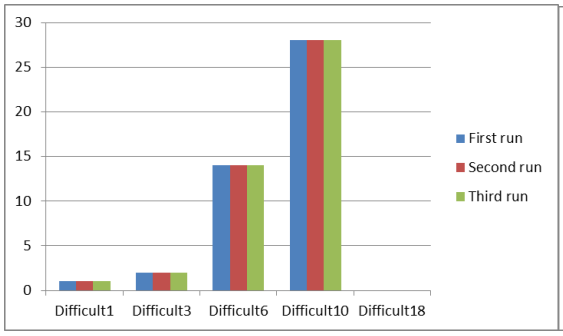


Figure-4 BFS Step Performance

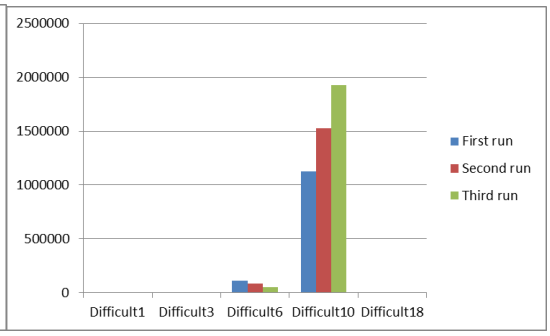


Figure-5 BFS Visited Nodes Performance

From Figure-4&5, I find that BFS will always get the optimal solution, however when dealing with high level puzzles, it takes really long times in searching. For the reason that it need to check all states in the same depth first, so it will hard for BFS to search the big steps puzzle. As I tested, in difficulty 10, BFS search over 1 million states but still can't find the solution. However if the steps is small, BFS can find the optimal solution.

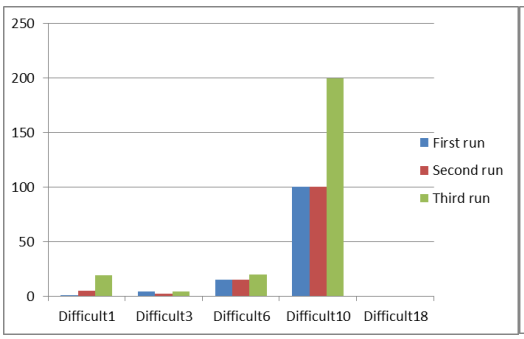


Figure-6 IDS Step Performance

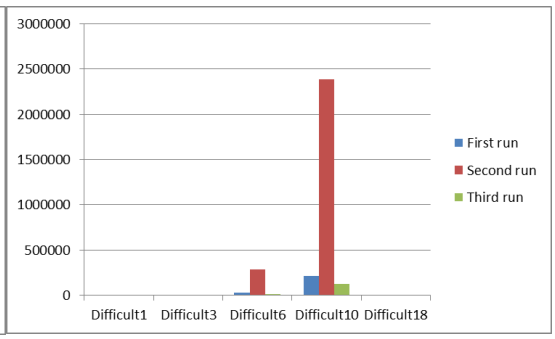
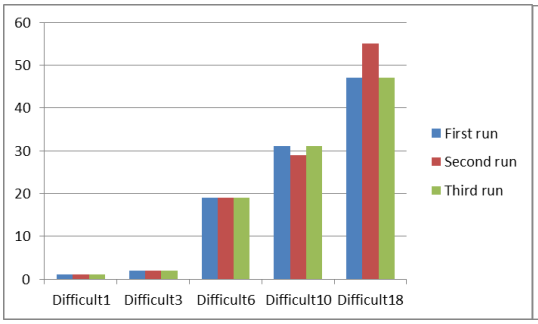


Figure-7 IDS Visited Nodes Performance

IDS's performance shows in Figure-6&7, unlike other three search methods, IDS's performance

also limited by the depth-limit, if the limit is smaller than the smallest depth, it can't get the solution, so if IDS finish all states but haven't find the solution, it should deepen the depth, and do the search again. The searching method is a combination of BFS and DFS, for the reason that, when dealing puzzle like difficulty 10, it will search lots of states to find the solution.



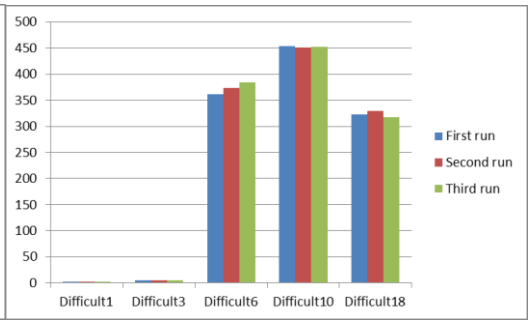
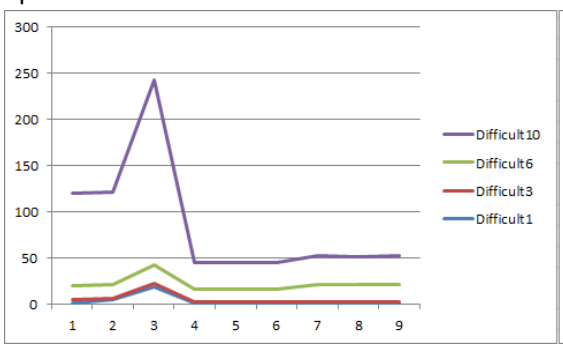


Figure-8 Greedy Step Performance Figure-9 Greedy Visited Nodes Performance

The last one is the Greedy search, from the figure we can see it searches much less than the other three methods, even when dealing with difficulty 18 and difficulty 10. The greedy function did lead the face block towards the goal. However, the solution it gets still may not be the optimal solution.



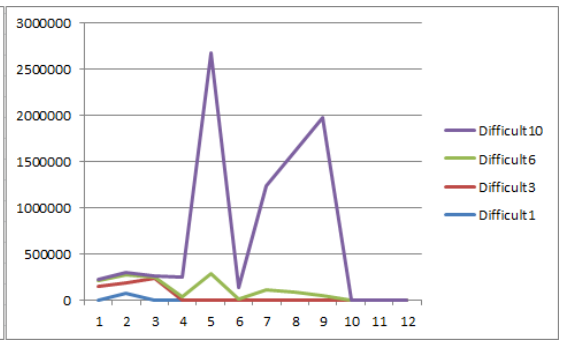


Figure-10 Overall steps comparison (without DFS) Figure-11 Overall visited nodes comparison

Conclusion

The four methods have their own advantages; DFS can give solutions when dealing with big-step-puzzle, but waste too much steps; BFS can find the optimal solution in small step puzzle, however it not good in doing long step puzzle; IDS is an average way to gives a solution but may not be optimal and search too much nodes; Greedy search searches far less states and give the solution, but not really optimal.

Fundamental of AI Coursework 2

Improved Method for 15-puzzle-problem

Lei Zhang
ID: 27678385
Email：lz3g15@soton.ac.uk

Abstract

This part I will pick task3 and introduce the Idea I have to improve the performance of searching on part 1, then gives the result and analysis.

Introduction

In this coursework I choose to try to improve the searching method used in coursework 1, the aim is to get the optimal solution with the least searching nodes. The general idea is to try some other Heuristic searching methods and improve it. So what I chose first is A-star method, then from A-start method, I applied a bidirectional A-star. After that, I got the idea how to use a learning method to do the searching, however I couldn't realize it.

Heuristic searching

When it comes to improve the previous method, there are two ways appears in mind.

First one, improve the evaluation function. Greedy and A-star are using Manhattan distance in their evaluation function, but Manhattan distance is not accurate in this situation, the steps the face block use to move all characters is not the same as the distance between the goal and initial state. So if I want to prove the performance, I may need to calculate the real distance to goal state.

Second ideal is to change the way to use the evaluation function. The biggest problem of heuristic searching is the solution may not be optimal, so if I can apply the evaluation twice or searching several solutions then compare them and get the optimal one.

According to these two ideas I start my improvement.

A-star search

A-star searching generally is a improve method based on greedy search. Greedy search only calculate the distance between current state and goal state, A-star will add the cost from the initial state to current state, the reason to do so, is to make the solution more optimal. But this will cost more time to search states, I will show the result soon after.

Bidirectional A-star search

Bidirectional A-star is to start the search from both the initial state and the goal state (like Figure-1 shows), the aim is to make the two search tree meet in same state, and then we can get an entire route from the initial state to the goal state.

The general flow is, I expand a node, then chose the shortest node towards the current goal, then this node will became the goal of the bidirectional search and so on, when compare the distance, not only compare the current depth but all unexpanded states, and the distance should be recomputed towards the current goal. About re-calculate the distance, I found sometimes, it

much possible to reach the optimal solution without re-calculate, this maybe because the wrong pick avoid the local maximum solution, but I can't prove it, so it's just my opinion.
The key point in this method is how to set the goals for both direction searches. When picking the state from depth 1, the Manhattan distance is compute between current states to the last goal state. Then the goal for the bidirectional search is the state chosen from depth 1, which just been chose to be expand, and the next goal for state in depth 2 is the new goal just been expand. And finally they will meet some where in around the middle.

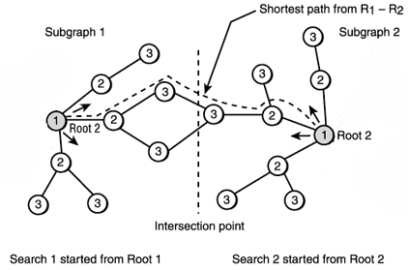


Figure-1 Bidirectional Search Structure

Bidirectional A star with local search

This method is designed to find the optimal solution. Local search is an improvement when one solution has been found, it start the search again from another state, this state can be the neighbor state to the current solution or just pick a random state, trying to find another solution, and compare these two solutions, if the new solution is better, then start the search from this state, until it can't find any better solution, that means, I have got the best one.
So to my situation, I need to make the search keep going, and search for another several times, I believe this could be a solution to find an optimal solution, however it means I need to search much more nodes than current state, which is not I prefer, so finally this method been abandon.

The last method I come to thought is using the learning method to search the puzzle. The principle is to using the Neural Networks to learn the true distance between the current states to the goal state, and then the face block can pick the shortest path to the goal.
The first problem is how to treat the neural network. This requires lots of data, I need to know what the true distance is, how much steps it needs to take from this state to the goal. One way is to using the BFS to do once, and then I can know the true steps. So a 4*4 board have 43680 potential combinations of states. Using BFS to compute the true steps from each state to goal state, this will became the training data of the neural network, then apply the trained neural network to the evaluation function, and done, the function will lead to the shortest path.

Result and Analysis

In this section, I will show the result of A-star and Bidirectional A-star in solving the 15-puzzle-problem and analysis the performance of these two searches. Greedy search will set as the contrast.
The performance is represented by the steps, how much steps it need to take towards the goal, and visited states, how much states it need to search. The best situation is to search the minimal states and get the shortest path.
I prepared 4 initial states this time (shown in figure-2), including the example state. Given the

Manhattan distance can't measure the difficulty well, so this time the difficulty is measured by how much steps it need to take from the start to the end, the steps is measured by A-star (running several times with the same initial and get the shortest path, I want to BFS, but my BFS take incredible long time when dealing long steps problem).here I don't compare the time, the time also limited by the code structure, our aim Is just compare the searching methods.

		B	C
F			A

True step-31

A			
F			B
			C

True step-12

	A		
		B	
	F	C	

True step-24

A	B	C	F

True step-16

Figure-2 Difficulty Chart

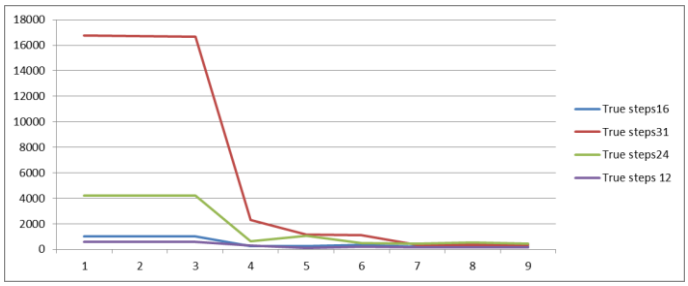


Figure-3 Overall Visited Nodes

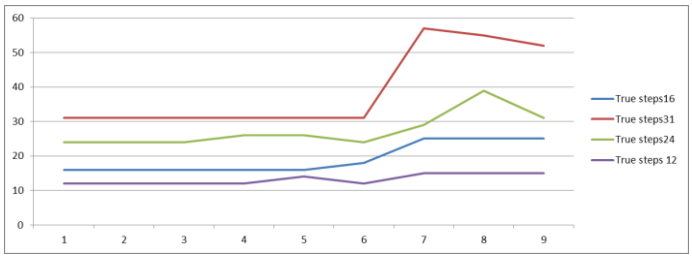


Figure-4 Overall Steps

Figure-3 shows the number of visited nodes each method from different initial state. Figure-4 shows the steps each method gives.

1-3 represents the three run of A-star, 4-6 represents the three run of Bidirectional A-star and the last three is Greedy method.

From Figure-3 we can see, A-star tends to search more states than the other two, especially in long steps search, like 31 steps. However A-star can provide a stable and optimal solution, which can be observed from 1-3 in Figure-5.

Bidirectional A-star searches much less states than A-star, however it not always gives the optimal solution, from Figure-4 4-6 we can see the steps Bidirectional A-star provides is unstable, but the differences in steps is not big compared with A-star, the value is around 1-2 steps.

Greedy method always searches least nodes, but provides more steps than the other two, which is unacceptable.

Conclusion

Bidirectional A-star searches less than A-star, and provides a similar solution, compared with A-star, so I believe Bidirectional A-star is actually better than A-star. But we still can't ignore the disadvantage of it; it can't always provide an optimal solution. However if I try several times like I do in the test, I can still find an optimal solution. Last about the Neural network, I had a consider about that, but since I don't have so much time left, so I can't show the result about that.

Reference

Figure-1 from part2:

http://www.yaldex.com/games-programming/0672323699_ch12lev1sec7.html

Appendix

BFS-Breadth First Search

```
%% set the variables
clear all;
% MapInit=[1 0 0 0;0 0 0 0;0 0 2 0;0 4 3 0];
% MapInit=[0 0 0 0;0 1 0 0;0 2 0 4;0 3 0 0];
% MapInit=[0 0 0 0;0 1 0 0;0 2 0 0;0 4 3 0];
% MapInit=[1 0 0 0;0 0 0 0;4 0 0 2;0 0 0 3];
MapInit=[4 0 2 3;0 0 0 0;0 0 0 0;0 0 0 1];
goal=[0 0 0 0;0 1 0 0;0 2 0 0;0 3 0 4];
Map=MapInit;
Maptemp=Map;
depth=zeros(100,1);
breth=zeros(100,1);
stack=cell(1000000,1);
route=cell(50,1);
route_index=zeros(50,1);
stack{1}=MapInit;
path=zeros(1000000,4);
direction=[-1,0;1,0;0,-1;0,1];
bip=[4,4];%blockinitposition
temp=bip;
score=0;
j=1;
m=0;
%% Main code
 while score==0;
    m=m+1;
    Map=stack{m};
    Maptemp=Map;
%% Random create 4 Directions
    direction_index=randperm(4); % creat a
list of 4 random number
    for l=1:4

b(l,:)=direction(direction_index(:,l),:);
    end
    [row,col]=find(Map==4);
    bip=[row,col];
    temp=bip;
    start=j; % used to judge how many states been
expanded
%%  Expand the Node
    for n=1:4
        bip=bip+b(n,:); % move the block
        moverror=find(bip<1|bip>4); % is the
move out of the board
        if (~isempty(moverror))
            bip=temp;
        elseif b(n,:)==path(m,1:2) % is the move
goback to the previous state
            bip=temp;
        else %creat the new state

Maptemp(bip(1,1),bip(1,2))=Map(temp(1,1),te
mp(1,2));

Maptemp(temp(1,1),temp(1,2))=Map(bip(1,1),b
ip(1,2));
            for x=1:j % is this state has been
expanded before
                if stack{x}==Maptemp
                    bip=temp;
                    Maptemp=Map;
                    break
                else % put the new state to the
stack,wait for the next move
                    j=j+1;
                    stack{j}=Maptemp;
                    path(j,1:2)=-b(n,:);%
record which direction it cames

path(j,3)=path(m,3)+1;%record the depth
                    path(j,4)=m;%record the
position of it's father
                    bip=temp;
                    Maptemp=Map;
                end
            end
        end
    end
%% Compare have it meet the goal
    stop=j;
```

```
    for k=start:stop % compare all expanded
nodes
       if goal==stack{k};
          steps=path(j,3);
          score=1;
          routemp=k;
          for r=1:steps % track the route

route_index(r,:)=path(routemp,4);
             routemp=path(routemp,4);

route{r}=stack{route_index(r,:)};
          end
          return
       else score=0;
       end
    end
 end
```

DFS-Deep First Search (almost the same as BFS but only store one node once)

```
clear all;
MapInit=[0 0 0 0;0 0 0 0;0 0 0 0;1 2 3 4];
% MapInit=[0 0 2 3;0 0 0 0;0 0 0 0;4 0 0 1];
% MapInit=[1 0 0 0;0 0 0 0;4 0 0 2;0 0 0 3];
% MapInit=[0 0 0 0;0 1 0 0;0 0 2 0;0 4 3 0];
goal=[0 0 0 0;0 1 0 0;0 2 0 0;0 3 0 4];
Map=MapInit;
Maptemp=Map;
depth=zeros(100,1);
breth=zeros(100,1);
stack=cell(1000000,1);
route=cell(50,1);
route_index=zeros(50,1);
stack{1}=MapInit;
path=zeros(1000000,4);
direction=[-1,0;1,0;0,-1;0,1];
bip=[4,4];%blockinitposition
temp=bip;
score=0;
j=1;
m=0;
while score==0;
    m=m+1;
    Map=stack{m};
    Maptemp=Map;
    direction_index=randperm(4);
    for l=1:4

b(l,:)=direction(direction_index(:,l),:);
    end
    [row,col]=find(Map==4);
    bip=[row,col];
    temp=bip;
    for n=1:4
       bip=bip+b(n,:);
       moverror=find(bip<1|bip>4);
       if (~isempty(moverror))
          bip=temp;
       elseif b(n,:)==path(m,1:2)
          bip=temp;
       else

Maptemp(bip(1,1),bip(1,2))=Map(temp(1,1),te
mp(1,2));

Maptemp(temp(1,1),temp(1,2))=Map(bip(1,1),b
ip(1,2));
          for x=1:j
             if stack{x}==Maptemp
                bip=temp;
                Maptemp=Map;
             else
                j=j+1;
                stack{j}=Maptemp;
                path(j,1:2)=-b(n,:);
                path(j,3)=path(m,3)+1;
                path(j,4)=m;
                bip=temp;
                Maptemp=Map;
                break %only expand to one
state and store it
             end
          end
          break
       end
```

```
    end
    if goal==stack{m+1};
        steps=path(m,3);
        score=1;
        routemp=m+1;
        for r=1:steps
            route_index(r,:)=path(routemp,4);
            routemp=path(routemp,4);
            route{r}=stack{route_index(r,:)};
        end
        return
    else score=0;
    end
end
```

IDS-Iterative Depth Search

```
clear all;
% MapInit=[1 0 0 0;0 0 0 0;0 0 2 0;0 4 3 0];
% MapInit=[0 0 0 0;0 1 0 0;0 2 0 4;0 3 0 0];
% MapInit=[0 0 0 0;0 1 0 0;0 2 0 0;0 4 3 0];
MapInit=[1 0 0 0;0 0 0 0;4 0 0 2;0 0 0 3];
goal=[0 0 0 0;0 1 0 0;0 2 0 0;0 3 0 4];
Map=MapInit;
Maptemp=Map;
depth=zeros(100,1);
breth=zeros(100,1);
stack=cell(50,1);
set_depth=20; %set the depth limit(it maybe
better to let the program find itself)
visited=0;
stack{1}=MapInit;
path=zeros(1000000,7);
direction=[-1,0;1,0;0,-1;0,1];
bip=[4,4];%blockinitposition
temp=bip;
j=0;
m=0;
score=0;
%% Main code
while score==0;
    m=m+1;
    Map=stack{m};
    Maptemp=Map;
    direction_index=randperm(4);
    for l=1:4

b(l,:)=direction(direction_index(:,l),:);
    end
    [row,col]=find(Map==4);
    bip=[row,col];
    temp=bip;
    for n=1:4
        bip=bip+b(n,:);
        moverror=find(bip<1|bip>4);

visited=find(path(m,1:4)==direction_index(n
));
        if path(m,7)==4;
          path(m:m+1,1:4)=0;
          path(m:m+1,7)=0;
          m=m-2;
%           if m<1
%              return
%           end
          break
        else
            if (~isempty(moverror))
               bip=temp;
               path(m,7)=path(m,7)+1;

path(m,path(m,7))=direction_index(n);
           elseif visited~=0
               bip=temp;
           else

Maptemp(bip(1,1),bip(1,2))=Map(temp(1,1),te
mp(1,2));

Maptemp(temp(1,1),temp(1,2))=Map(bip(1,1),b
ip(1,2));
               stack{m+1}=Maptemp;

t1=find(-b(n,1)==direction(:,1));

t2=find(-b(n,2)==direction(:,2));
               t=intersect(t1,t2);
               path(m,7)=path(m,7)+1;
```

```
path(m,path(m,7))=direction_index(n);
            path(m+1,7)=path(m+1,7)+1;
            path(m+1,path(m+1,7))=t;
            path(m+1,6)=m;
            j=j+1;
            break
        end
    end
end
%% Tell have the research meet the limit
if set_depth==path(m+1,6)
    if goal==stack{m+1};%if the state meet
the goal
        steps=m;
        score=1;
        return
    else
        score=0;
        m=m-1;
    end
else %if the state haven't meet the limit
    if goal==stack{m+1};
        steps=m+1;
        score=1;
        return
    else % keep running
        score=0;
      end
   end
 end
```

Greedy search

```
clear all;
tic;
% MapInit=[0 0 0 0;0 0 0 0;0 0 0 0;1 2 3 4];
% MapInit=[0 0 2 3;0 0 0 0;0 0 0 0;4 0 0 1];
% MapInit=[1 0 0 0;0 0 0 0;4 0 0 2;0 0 0 3];
MapInit=[0 0 0 0;0 1 0 0;0 0 2 0;0 4 3 0];
goal=[0 0 0 0;0 1 0 0;0 2 0 0;0 3 0 4];
Map=MapInit;
Maptemp=Map;
depth=zeros(100,1);
breth=zeros(100,1);
stack=cell(10000,1);
tempstack=cell(4,1);
openstack=cell(10000,1);
openstack{1}=MapInit;
closedstack=cell(10000,1);
closedstack{1}=MapInit;
openstack_index=1;
openpath=zeros(10000,5);
route=cell(50,1);
route_index=zeros(50,1);
stack{1}=MapInit;
score=0;
path=zeros(100000,5);
direction=[-1,0;1,0;0,-1;0,1];
bip=[4,4];%blockinitposition
temp=bip;
m=0;

   while score==0
    m=m+1;
    Map=stack{m};
    Maptemp=Map;
    direction_index=randperm(4);
    tempath=zeros(4,4);
    tempstack=cell(4,1);
    Manhatan=[];
    f=0;
    for l=1:4

b(l,:)=direction(direction_index(:,1),:);
    end
    [row,col]=find(Map==4);
    bip=[row,col];
    temp=bip;
    start=f;
    for n=1:4
        bip=bip+b(n,:);
        moverror=find(bip<1|bip>4);
        if (~isempty(moverror))
            bip=temp;
        elseif b(n,:)==path(m,1:2)
            bip=temp;
        else

Maptemp(bip(1,1),bip(1,2))=Map(temp(1,1),te
```

```
mp(1,2));

Maptemp(temp(1,1),temp(1,2))=Map(bip(1,1),b
ip(1,2));
          f=f+1;
          tempstack{f}=Maptemp;
          tempath(f,1:2)=-b(n,:);
          tempath(f,3)=path(m,3)+1;
          tempath(f,4)=m;
          bip=temp;
          Maptemp=Map;
       end
   end
   stop=f;
   %% Compute the manhattan distance
   %find the 4 characters position
   for c=1:stop-start
       [row,col]=find(tempstack{c}==1);
       signa(c,:)=[row,col];
       [row,col]=find(tempstack{c}==2);
       signb(c,:)=[row,col];
       [row,col]=find(tempstack{c}==3);
       signc(c,:)=[row,col];
       [row,col]=find(tempstack{c}==4);
       signblock(c,:)=[row,col];
       % This is the evalutation function

Manhatan(1,c)=abs(signa(c,1)-2)+abs(signa(c
,2)-2)+abs(signb(c,1)-3)+abs(signb(c,2)-2)+
abs(signc(c,1)-4)+abs(signc(c,2)-2)+abs(sig
nblock(c,1)-4)+abs(signblock(c,2)-4);
       tempath(c,5)=Manhatan(1,c);% store
them in a room
   end
 %% to tell have the new expanded state
appeaered or not before
   for x=1:f
       for o=1:m
          if
closedstack{o}==tempstack{x} %check the
closestack
             visited1=1;
             break
          else
             visited1=0;
          end
       end
       for y=1:openstack_index
          if openstack{y}==tempstack{x} %
check the openstack
             visited2=1;
             break
          else
             visited2=0;
          end
       end
       if visited1==0&&visited2==0
          %if both don't have them before,
store it

openstack_index=openstack_index+1;

openstack{openstack_index}=tempstack{x};

openpath(openstack_index,:)=tempath(x,:);
       end
   end
   %%
   for l=1:f
       if goal==tempstack{l}
          steps=path(m,3);
          score=1;
          routemp=m;
          for r=1:steps

route_index(r,:)=path(routemp,4);
             routemp=path(routemp,4);

route{r}=stack{route_index(r,:)};
          end
          time=toc;
          totalnudes=o+openstack_index-2;
          return
       else score=0;
       end
   end
  %% move the state to the stack and expand
  % pick the state accoding to the Manhatan
```

```
sistance


index=find(openpath(1:openstack_index,5)==m
in(openpath(1:openstack_index,5)));
    stack{m+1}=openstack{index(1,1)};
    path(m+1,:)=openpath(index(1,1),:);
    closedstack{m+1}=openstack{index(1,1)};
    % in order to make the order right, i move
the stored one
    openstack{index(1,1)}=[];
    openpath(index(1,1),:)=[];
    for z=index(1,1):openstack_index-1%
reorderthe openstacks
        openstack{z}=openstack{z+1};
    end
    openstack_index=openstack_index-1;
   end
```

A-star search

```
clear all;
tic;
% MapInit=[0 0 0 0;0 0 0 0;0 0 0 0;1 2 3 4];
% MapInit=[0 0 2 3;0 0 0 0;0 0 0 0;4 0 0 1];
% MapInit=[1 0 0 0;0 0 0 0;4 0 0 2;0 0 0 3];
MapInit=[0 0 0 0;0 1 0 0;0 0 2 0;0 4 3 0];
goal=[0 0 0 0;0 1 0 0;0 2 0 0;0 3 0 4];
Map=MapInit;
Maptemp=Map;
depth=zeros(100,1);
breth=zeros(100,1);
stack=cell(10000,1);
tempstack=cell(4,1);
openstack=cell(10000,1);
openstack{1}=MapInit;
closedstack=cell(10000,1);
closedstack{1}=MapInit;
openstack_index=1;
openpath=zeros(10000,5);
route=cell(50,1);
position=zeros(1,4);
route_index=zeros(50,1);
stack{1}=MapInit;
score=0;
path=zeros(100000,5);
direction=[-1,0;1,0;0,-1;0,1];
bip=[4,4];%blockinitposition
temp=bip;
m=0;

while score==0
    m=m+1;
    Map=stack{m};
    Maptemp=Map;
    direction_index=randperm(4);
    tempath=zeros(4,4);
    tempstack=cell(4,1);
    Manhatan=[];
    f=0;
    for l=1:4

b(l,:)=direction(direction_index(:,l),:);
    end
    [row,col]=find(Map==4);
    bip=[row,col];
    temp=bip;
    start=f;
    for n=1:4
        bip=bip+b(n,:);
        moverror=find(bip<1|bip>4);
        if (~isempty(moverror))
            bip=temp;
        elseif b(n,:)==path(m,1:2)
            bip=temp;
        else

Maptemp(bip(1,1),bip(1,2))=Map(temp(1,1),te
mp(1,2));

Maptemp(temp(1,1),temp(1,2))=Map(bip(1,1),b
ip(1,2));
            f=f+1;
            tempstack{f}=Maptemp;
            tempath(f,1:2)=-b(n,:);
            tempath(f,3)=path(m,3)+1;
            tempath(f,4)=m;
            bip=temp;
            Maptemp=Map;
        end
```

```
    end
    stop=f;
    for c=1:stop-start
        [row,col]=find(tempstack{c}==1);
        signa(c,:)=[row,col];
        [row,col]=find(tempstack{c}==2);
        signb(c,:)=[row,col];
        [row,col]=find(tempstack{c}==3);
        signc(c,:)=[row,col];
        [row,col]=find(tempstack{c}==4);
        signblock(c,:)=[row,col];
        % the difference is i calculate the
distance from the initial to
        % current state

Manhatan(1,c)=tempath(c,3)+abs(signa(c,1)-2
)+abs(signa(c,2)-2)+abs(signb(c,1)-3)+abs(s
ignb(c,2)-2)+abs(signc(c,1)-4)+abs(signc(c,
2)-2)+abs(signblock(c,1)-4)+abs(signblock(c
,2)-4);
        tempath(c,5)=Manhatan(1,c);
    end
    position=find(Manhatan==min(Manhatan));
    for x=1:f
        for o=1:m
            if  closedstack{o}==tempstack{x}
                visited1=1;
                break
            else
                visited1=0;
            end
        end
        for y=1:openstack_index
            if openstack{y}==tempstack{x}
                visited2=1;
                break
            else
                visited2=0;
            end
        end
        if visited1==0&&visited2==0

openstack_index=openstack_index+1;
openstack{openstack_index}=tempstack{x};

openpath(openstack_index,:)=tempath(x,:);
        end
    end
    for l=1:f
        if goal==tempstack{l}
            steps=path(m,3);
            score=1;
            routemp=m;
            for r=1:steps

route_index(r,:)=path(routemp,4);
                routemp=path(routemp,4);

route{r}=stack{route_index(r,:)};
            end
            time=toc;
            return
        else score=0;
        end
    end


index=find(openpath(1:openstack_index,5)==m
in(openpath(1:openstack_index,5)));
    stack{m+1}=openstack{index(1,1)};
    path(m+1,:)=openpath(index(1,1),:);
    closedstack{m+1}=openstack{index(1,1)};
    openstack{index(1,1)}=[];
    openpath(index(1,1),:)=[];
    for z=index(1,1):openstack_index-1
        openstack{z}=openstack{z+1};
    end
    openstack_index=openstack_index-1;
end
```

Bidirectional A-star search

```
clear all;
% MapInit=[0 0 0 0;0 0 0 0;0 0 0 0;1 2 3 4];
% MapInit=[0 0 2 3;0 0 0 0;0 0 0 0;4 0 0 1];
% MapInit=[1 0 0 0;0 0 0 0;4 0 0 2;0 0 0 3];
MapInit=[0 0 0 0;0 1 0 0;0 0 2 0;0 4 3 0];
goal=[0 0 0 0;0 1 0 0;0 2 0 0;0 3 0 4];
```

```
Map=MapInit;
Maptemp=Map;
biMapInit=goal;
bigoal=MapInit;
biMap=goal;
biMaptemp=biMap;
depth=zeros(100,1);
breth=zeros(100,1);
stack=cell(1000,1);
stack{1}=MapInit;
bistack=cell(1000,1);
bistack{1}=biMapInit;
tempstack=cell(4,1);
bitempstack=cell(4,1);
openstack=cell(1000,1);
openstack{1}=MapInit;
biopenstack=cell(1000,1);
biopenstack{1}=biMapInit;
closedstack=cell(1000,1);
closedstack{1}=MapInit;
biclosedstack=cell(1000,1);
biclosedstack{1}=biMapInit;
openstack_index=1;
biopenstack_index=1;
openpath=zeros(1000,5);
biopenpath=zeros(1000,5);
score=0;
path=zeros(1000,5);
bipath=zeros(1000,5);
direction=[-1,0;1,0;0,-1;0,1];
bip=[4,4];%blockinitposition
temp=bip;
m=0;

while score==0
    m=m+1;
    Map=stack{m};
    Maptemp=Map;
    direction_index=randperm(4);
    tempath=zeros(4,4);
    tempstack=cell(4,1);
    Manhatan=[];
    f=0;
    for l=1:4
b(l,:)=direction(direction_index(:,l),:);
    end
    [row,col]=find(Map==4);
    bip=[row,col];
    temp=bip;
    start=f;
    for n=1:4
        bip=bip+b(n,:);
        moverror=find(bip<1|bip>4);
        if (~isempty(moverror))
            bip=temp;
        elseif b(n,:)==path(m,1:2)
            bip=temp;
        else

Maptemp(bip(1,1),bip(1,2))=Map(temp(1,1),te
mp(1,2));

Maptemp(temp(1,1),temp(1,2))=Map(bip(1,1),b
ip(1,2));
            f=f+1;
            tempstack{f}=Maptemp;
            tempath(f,1:2)=-b(n,:);
            tempath(f,3)=path(m,3)+1;
            tempath(f,4)=m;
            bip=temp;
            Maptemp=Map;
        end
    end
    stop=f;
    goal=bistack{m};
    [row,col]=find(goal==1);
    bisigna(1,:)=[row,col];
    [row,col]=find(goal==2);
    bisignb(1,:)=[row,col];
    [row,col]=find(goal==3);
    bisignc(1,:)=[row,col];
    [row,col]=find(goal==4);
    bisignblock(1,:)=[row,col];
    for c=1:stop-start
        [row,col]=find(tempstack{c}==1);
        signa(c,:)=[row,col];
        [row,col]=find(tempstack{c}==2);
```

```
        signb(c,:)=[row,col];
        [row,col]=find(tempstack{c}==3);
        signc(c,:)=[row,col];
        [row,col]=find(tempstack{c}==4);
        signblock(c,:)=[row,col];

Manhatan(1,c)=tempath(c,3)+abs(signa(c,1)-b
isigna(1,1))+abs(signa(c,2)-bisigna(1,2))+a
bs(signb(c,1)-bisignb(1,1))+abs(signb(c,2)-
bisignb(1,2))+abs(signc(c,1)-bisignc(1,1))+
abs(signc(c,2)-bisignc(1,2))+abs(signblock(
c,1)-bisignblock(1,1))+abs(signblock(c,2)-b
isignblock(1,2));
        tempath(c,5)=Manhatan(1,c);
    end
    for x=1:f
        for o=1:m
            if  closedstack{o}==tempstack{x}
                visited1=1;
                break
            else
                visited1=0;
            end
        end
        for y=1:openstack_index
            if openstack{y}==tempstack{x}
                visited2=1;
                break
            else
                visited2=0;
            end
        end
        if visited1==0&&visited2==0

openstack_index=openstack_index+1;

openstack{openstack_index}=tempstack{x};

openpath(openstack_index,:)=tempath(x,:);
        end
    end

    %% Bidirection search
    goal=bistack{m};
index=find(openpath(1:openstack_index,5)==m
in(openpath(1:openstack_index,5)));
    stack{m+1}=openstack{index(1,1)};
    path(m+1,:)=openpath(index(1,1),:);
    closedstack{m+1}=openstack{index(1,1)};
    openstack{index(1,1)}=[];
    openpath(index(1,1),:)=[];
    for z=index(1,1):openstack_index-1
        openstack{z}=openstack{z+1};
    end
    openstack_index=openstack_index-1;

    biMap=bistack{m};
    biMaptemp=biMap;
    direction_index=randperm(4);
    bitempath=zeros(4,4);
    bitempstack=cell(4,1);
    biManhatan=[];
    bif=0;
    for l=1:4

b(l,:)=direction(direction_index(:,l),:);
    end
    [row,col]=find(biMap==4);
    bip=[row,col];
    temp=bip;
    start=bif;
    for n=1:4
        bip=bip+b(n,:);
        moverror=find(bip<1|bip>4);
        if (~isempty(moverror))
            bip=temp;
        elseif b(n,:)==bipath(m,1:2)
            bip=temp;
        else

biMaptemp(bip(1,1),bip(1,2))=biMap(temp(1,1
),temp(1,2));

biMaptemp(temp(1,1),temp(1,2))=biMap(bip(1,
1),bip(1,2));
            bif=bif+1;
            bitempstack{bif}=biMaptemp;
```

```
            bitempath(bif,1:2)=-b(n,:);
            bitempath(bif,3)=bipath(m,3)+1;
            bitempath(bif,4)=m;
            bip=temp;
            biMaptemp=biMap;
        end
    end
    stop=bif;
    bigoal=stack{m+1};
    [row,col]=find(bigoal==1);
    bisigna(1,:)=[row,col];
    [row,col]=find(bigoal==2);
    bisignb(1,:)=[row,col];
    [row,col]=find(bigoal==3);
    bisignc(1,:)=[row,col];
    [row,col]=find(bigoal==4);
    bisignblock(1,:)=[row,col];
    for c=1:stop-start
        [row,col]=find(bitempstack{c}==1);
        signa(c,:)=[row,col];
        [row,col]=find(bitempstack{c}==2);
        signb(c,:)=[row,col];
        [row,col]=find(bitempstack{c}==3);
        signc(c,:)=[row,col];
        [row,col]=find(bitempstack{c}==4);
        signblock(c,:)=[row,col];

biManhatan(1,c)=bitempath(c,3)+abs(signa(c,
1)-bisigna(1,1))+abs(signa(c,2)-bisigna(1,2
))+abs(signb(c,1)-bisignb(1,1))+abs(signb(c
,2)-bisignb(1,2))+abs(signc(c,1)-bisignc(1,
1))+abs(signc(c,2)-bisignc(1,2))+abs(signbl
ock(c,1)-bisignblock(1,1))+abs(signblock(c,
2)-bisignblock(1,2));
        bitempath(c,5)=biManhatan(1,c);
    end
    for x=1:bif
        for o=1:m
            if
biclosedstack{o}==bitempstack{x}
                visited1=1;
                break
            else
                visited1=0;
            end
        end
        for y=1:biopenstack_index
            if biopenstack{y}==bitempstack{x}
                visited2=1;
                break
            else
                visited2=0;
            end
        end
        if visited1==0&&visited2==0

biopenstack_index=biopenstack_index+1;

biopenstack{biopenstack_index}=bitempstack{
x};

biopenpath(biopenstack_index,:)=bitempath(x
,:);
        end
    end
    goal=stack{m+1};

biindex=find(biopenpath(1:biopenstack_index
,5)==min(biopenpath(1:biopenstack_index,5))
);
    bistack{m+1}=biopenstack{biindex(1,1)};

bipath(m+1,:)=biopenpath(biindex(1,1),:);

biclosedstack{m+1}=biopenstack{biindex(1,1)
};
    biopenstack{biindex(1,1)}=[];
    biopenpath(biindex(1,1),:)=[];
    for z=biindex(1,1):biopenstack_index-1
        biopenstack{z}=biopenstack{z+1};
    end
    biopenstack_index=biopenstack_index-1;
    %% check the states in
stack,bistack,openstatck,biopenstack,
    %closestack and biclosestack, the reason
for that because there always
    %one been moved.
    for q=1:biopenstack_index-1
```

```
       for w=1:openstack_index-1
           if (openstack{w}==biopenstack{q})
               score=1;
               steps=openpath(w,3);
               bisteps=biopenpath(q,3);
               %track the route, the path been
stored in route and biroute
               routemp=w;
               biroutemp=q;
               route{1}=openstack{w};
               biroute{1}=biopenstack{q};

route_index(1,:)=openpath(routemp,4);

biroute_index(1,:)=biopenpath(biroutemp,4);
               for r=2:steps

route{r}=stack{route_index(r-1,:)};

route_index(r,:)=path(route_index(r-1,:),4)
;
               end
               for r=2:bisteps

biroute{r}=bistack{biroute_index(r-1,:)};

biroute_index(r,:)=bipath(biroute_index(r-1
,:),4);
               end

totalnudes=2*(m+1)+biopenstack_index+openst
ack_index;
               return
           elseif
(openstack{w}==bistack{m+1})
               score=1;
               steps=openpath(w,3);
               bisteps=biopenpath(q,3);
               routemp=w;
               biroutemp=q;
               route{1}=openstack{w};
               biroute{1}=biopenstack{q};

route_index(1,:)=openpath(routemp,4);

biroute_index(1,:)=biopenpath(biroutemp,4);
               for r=2:steps

route{r}=stack{route_index(r-1,:)};

route_index(r,:)=path(route_index(r-1,:),4)
;
               end
               for r=2:bisteps

biroute{r}=bistack{biroute_index(r-1,:)};

biroute_index(r,:)=bipath(biroute_index(r-1
,:),4);
               end

totalnudes=2*(m+1)+biopenstack_index+openst
ack_index;
               return
           elseif
(stack{m+1}==biopenstack{q})
               score=1;
               steps=openpath(w,3);
               bisteps=biopenpath(q,3);
               routemp=w;
               biroutemp=q;
               route{1}=openstack{w};
               biroute{1}=biopenstack{q};

route_index(1,:)=openpath(routemp,4);

biroute_index(1,:)=biopenpath(biroutemp,4);
               for r=2:steps

route{r}=stack{route_index(r-1,:)};

route_index(r,:)=path(route_index(r-1,:),4)
;
               end
               for r=2:bisteps

biroute{r}=bistack{biroute_index(r-1,:)};
```

```
biroute_index(r,:)=bipath(biroute_index(r-1
,:),4);
              end

totalnudes=2*(m+1)+biopenstack_index+openst
ack_index;
              return
          elseif (stack{m+1}==bistack{m+1})
              score=1;
              steps=openpath(w,3);
              bisteps=biopenpath(q,3);
              routemp=w;
              biroutemp=q;
              route{1}=openstack{w};
              biroute{1}=biopenstack{q};

route_index(1,:)=openpath(routemp,4);

biroute_index(1,:)=biopenpath(biroutemp,4);
              for r=2:steps

route{r}=stack{route_index(r-1,:)};

route_index(r,:)=path(route_index(r-1,:),4)
;
              end
              for r=2:bisteps

biroute{r}=bistack{biroute_index(r-1,:)};

biroute_index(r,:)=bipath(biroute_index(r-1
,:),4);
              end

totalnudes=2*(m+1)+biopenstack_index+openst
ack_index;
              return
          end
       end
   end
end
```