Cairo University
Faculty of Computers & Artificial Intelligence
Department of Computer Science
Artificial Intelligence Course

Lab 5: Informed Search

In search problems, there may be many possible ways to get to the goal state, but we want to get the best possible path for our search. So, we use **informed search** when we have information about the closeness of the goal, and this narrows down the results precisely.

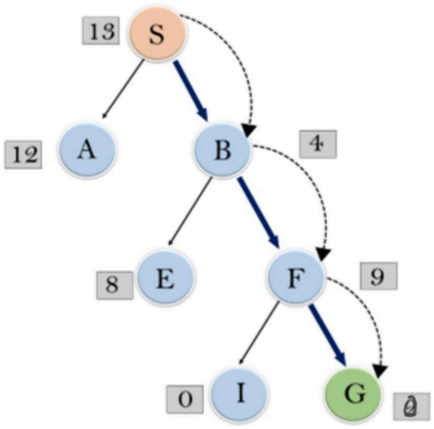
Informed search algorithms use a **heuristic function *"h"*** which is a function used to measure how close the current state is to the goal state. In this lab, we will use heuristic functions that **estimate the cost/distance** from the current state to the goal.

Informed Search Algorithms:

- ### Greedy Best-first Search

  In this algorithm, each node ***n*** has a value ***h(n)*** evaluated from the heuristic function and the search algorithm **expands nodes according to their heuristic values.** The node with the lowest heuristic value (i.e. lowest estimated cost) will be expanded first and so on.

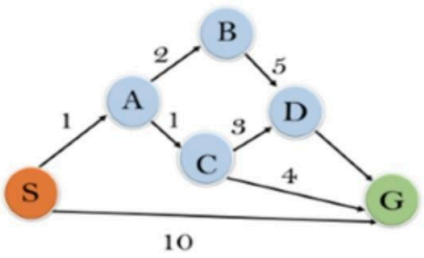
  It can be implemented using a **priority queue based on *h(n)*.**



- **A* Search:**

  This algorithm **combines both UCS and greedy** best-first search by expanding nodes which have the **lowest total value (cumulative cost + estimated cost)** first.

  It can be implemented using a **priority queue based on *f(n)*** *where f(n)=g(n)+h(n)*.



State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0

The Formal Informed Search Algorithm:

☆ We can use the same steps from the previous lab with few changes.

☆ **The decision of whether to use *h* or *f* when getting the best state determines the informed search algorithm.**

☆ For simplicity, a **node** can be represented as a **list containing the state, its parent, its actual cost from the start, its estimated cost to the goal (heuristic), and its total cost.**

We can now update the "search.pl" from the previous lab as follows:

```
search(Open, Closed, Goal):-
    getBestState(Open, [CurrentState,Parent,G,H,F], _),     % Step 1
    CurrentState = Goal,      % Step 2
    write("Search is complete!"), nl,
    printSolution([CurrentState,Parent,G,H,F], Closed), !.

search(Open, Closed, Goal):-
    getBestState(Open, CurrentNode, TmpOpen),
    getAllValidChildren(CurrentNode,TmpOpen,Closed,Goal,Children), % Step 3
    addChildren(Children, TmpOpen, NewOpen),     % Step 4
    append(Closed, [CurrentNode], NewClosed),     % Step 5.1
    search(NewOpen, NewClosed, Goal).          % Step 5.2
```

```
% Implementation of step 3 to get the next states
getAllValidChildren(Node, Open, Closed, Goal, Children):-
    findall(Next, getNextState(Node,Open,Closed,Goal,Next),
Children).


getNextState([State,_,G,_,_],Open,Closed,Goal,[Next,State,NewG,NewH,NewF]):-
    move(State, Next, MoveCost),
    calculateH(Next, Goal, NewH),
    NewG is G + MoveCost,
    NewF is NewG + NewH,
    not(member([Next,_,_,_,_], Open)),
    not(member([Next,_,_,_,_], Closed)).


% Implementation of addChildren and getBestState
addChildren(Children, Open, NewOpen):-
     append(Open, Children, NewOpen).

getBestState(Open, BestChild, Rest):-
     findMin(Open, BestChild),
     delete(Open, BestChild, Rest).

% Implementation of findMin in getBestState determines the search
alg.
% Greedy best-first search
findMin([X], X):- !.

findMin([Head|T], Min):-
     findMin(T, TmpMin),
     Head = [_,_,_,HeadH,HeadF],
     TmpMin = [_,_,_,TmpH,TmpF],
     (TmpH < HeadH -> Min = TmpMin ; Min = Head).


% Instead of adding children at the end and searching for the best
% each time using getBestState, we can make addChildren add in the
% right place (sorted open list) and getBestState just returns the
% head of open.


% Implementation of printSolution to print the actual solution path
printSolution([State, null, G, H, F],_):-
     write([State, G, H, F]), nl.

printSolution([State, Parent, G, H, F], Closed):-
     member([Parent, GrandParent, PrevG, Ph, Pf], Closed),
     printSolution([Parent, GrandParent, PrevG, Ph, Pf], Closed),
     write([State, G, H, F]), nl.
```

Notice that **"calculateH"** is problem-specific. The heuristic function we use must be **admissible such that it never overestimates the cost** from the state to the goal. It must be **consistent** as well.

If we wanted to **implement A*,** we would need to change the comparison line in **"findMin"** (highlighted in pink) but we would also have to change the **"not(member)" checks** to be as follows:

```
findMin([X], X):- !.

findMin([Head|T], Min):-
      findMin(T, TmpMin),
      Head = [_,_,_,HeadH,HeadF],
      TmpMin = [_,_,_,TmpH,TmpF],
      (TmpF < HeadF -> Min = TmpMin ; Min = Head).


getNextState([State,_,G,_,_],Open,Closed,Goal,[Next,State,NewG,NewH,NewF]):-
     move(State, Next, MoveCost),
     calculateH(Next, Goal, NewH),
     NewG is G + MoveCost,
     NewF is NewG + NewH,
     ( not(member([Next,_,_,_,_], Open)) ; memberButBetter(Next,Open,NewF) ),
     ( not(member([Next,_,_,_,_],Closed));memberButBetter(Next,Closed,NewF)).

memberButBetter(Next, List, NewF):-
     findall(F, member([Next,_,_,_,F], List), Numbers),
     min_list(Numbers, MinOldF),
     MinOldF > NewF.


% If the node is already a member with greater cost, the new node will
% be valid (and better) and it will be added to the children  but  the
% old version will remain in the  list  for  simplicity.  In  reality,
% the old version should be removed or replaced.
```

The 8-Puzzle Problem with Informed Search:

Implementation:

We can use the code from the search module and just determine the state representation and implement the problem-specific predicates "move" (with MoveCost) and "calculateH".

1. **State Representation:**

   We can represent the state as a list that contains 9 elements. Each element represents a tile in the 8-puzzle. We can use any symbol, say #, to define the empty tile.

   For example the initial and goal states of 8-puzzle may be as follows:

   Initial: [b, d, c, a, #, e, g, h, f]

   Goal: [a, b, c, d, e, f, g, h, #]

2. **Moves:**

```
move(State, Next, 1):- % All moves in this problem have equal cost (1)
     left(State, Next); right(State, Next);
     up(State, Next); down(State, Next).

left(State, Next):-
     nth0(EmptyTileIndex, State, #),
     not(0 is EmptyTileIndex mod 3),
     NewIndex is EmptyTileIndex - 1,
     nth0(NewIndex, State, Element),
     % Swap
     substitute(#, State, x, TmpList1),
     substitute(Element, TmpList1, #, TmpList2),
     substitute(x, TmpList2, Element, Next).

right(State, Next):-
     nth0(EmptyTileIndex, State, #),
     not(2 is EmptyTileIndex mod 3),
     NewIndex is EmptyTileIndex + 1,
     nth0(NewIndex, State, Element),
     % Swap
     substitute(#, State, x, TmpList1),
     substitute(Element, TmpList1, #, TmpList2),
     substitute(x, TmpList2, Element, Next).

up(State, Next):-
     nth0(EmptyTileIndex, State, #),
     EmptyTileIndex > 2,
     NewIndex is EmptyTileIndex - 3,
     nth0(NewIndex, State, Element),
     % Swap
     substitute(#, State, x, TmpList1),
     substitute(Element, TmpList1, #, TmpList2),
     substitute(x, TmpList2, Element, Next).
```

```
down(State, Next):-
     nth0(EmptyTileIndex, State, #),
     EmptyTileIndex < 6,
     NewIndex is EmptyTileIndex + 3,
     nth0(NewIndex, State, Element),
     % Swap
     substitute(#, State, x, TmpList1),
     substitute(Element, TmpList1, #, TmpList2),
     substitute(x, TmpList2, Element, Next).

substitute(Element, [Element|T], NewElement, [NewElement|T]):- !.
substitute(Element, [H|T], NewElement, [H|NewT]):-
     substitute(Element, T, NewElement, NewT).
```

3. **Heuristic Function:**

   There are different heuristic functions that we can use:

   - We can simply calculate the heuristic value by counting the number of tiles that are out of place as follows:

```
calculateH([], [], 0):- !.

calculateH([Head|T1], [Head|T2], Hvalue):-
    !, calculateH(T1, T2, Hvalue).

calculateH([_|T1], [_|T2], Hvalue):-
    calculateH(T1, T2, Count),
    Hvalue is Count + 1.
```

   - We can use a better heuristic which is the sum of the distances of each tile from its goal position.
   - More heuristics exist!

Run using the query:

```
?- search([[[b,d,c,a,#,e,g,h,f],null,0,x,0]], [],  [a,b,c,d,e,f,g,h,#]).
```