

Chapter 12

Best-First Search

Graph search typically leads to the problem of *combinatorial complexity*, when there are too many alternatives at each step of the way. Thus, we have to use some heuristics to cut the searching space.

One way to do it is to calculate the *heuristic* value of the nodes in the state space to find out how promising that node is. If it is promising enough, we will extend it, otherwise, we will drop it for something else.

The general idea

The best-first search can be derived from BFS. It also starts from the start node and keeps a set of candidate paths. While BFS always expands a shortest path, the best-first search will compute a heuristic estimate for each candidate and expand the *best* one.

We assume that a cost function, such that $c(n, n')$ represents the cost to go from node n to n' . We also assume that we have an heuristic estimator, $f(n)$, which estimates, for each node, n , in the state space, the *difficulty* of further working with n .

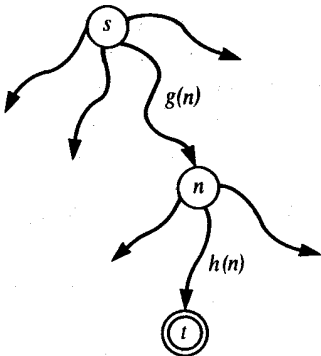
The most promising node n is the one with a minimum $f(n)$.

We begin by constructing $f(n)$ to estimate the cost of a best solution path from the start node s to a goal node t , which goes through n .

Assume there is such a t , we have that

$$f(n) = g(n) + h(n),$$

where $g(n)$ estimates the cost of an optimal path from s to n , (How difficult was it to get *here*?) and $h(n)$ estimates the cost of an optimal path from n to t , (How difficult will it be from *here*?)



How to get $g(n)$ and $h(n)$?

When the searching process sees n , it already finds a path from s to n , thus the cost of such path. This cost may not be the optimal one, but it can be used as an estimate (an upper bound) for such an optimum.

On the other hand, since we have yet to explore the path from n to t , the $h(n)$ at best can only be an (educated) guess, based on the knowledge of the problem.

There is no universal method to decide h . It depends on the specific problem. Thus, we assume that h is given.

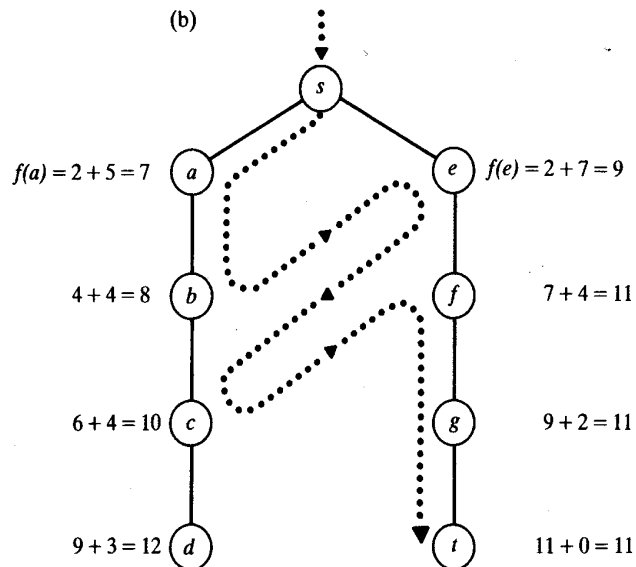
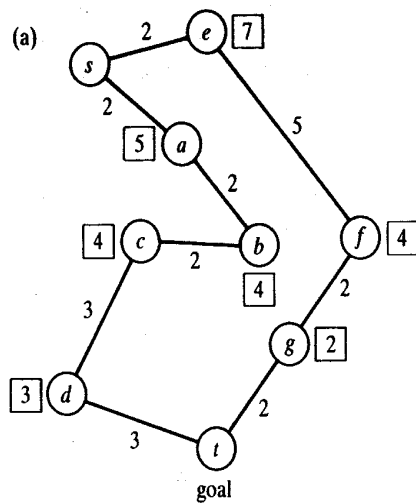
The general process

The search process consists of a number of competing subprocesses, each of them exploring its subtrees. Only one of them is active, the one exploring the currently most promising one. The rest simply stay put, until and unless, the current f -estimates changes, and one of the waiting one becomes more promising than the currently active one, and *it* then become the active one.

The processes keep on exploring until the active one reports a solution when a goal node is found.

An example

Let $n_0 = s$, and $n_h = n$, $g(n) = \sum_{i=0}^h c(n_i, n_j)$; and \square next to n shows $h(n)$.



Turned into a program

When exploring a search tree, we can represent it in one of two forms:

1) Use $l(N, F/G)$ to denote a leaf, where N is a node, G stands for $g(N)$, and F is $f(N) = g(N) + h(N)$, i.e., the total cost of including N in the tree.

2) Use $t(N, F/G, Subs)$ to denote a tree with non-empty subtrees, where N denotes the root of the subtree, and $Subs$ denotes a list of its subtrees, ordered by the increasing f -values of the subtrees. The pair F/G represents the best estimate of including N , where G denotes the cost of getting to N , and F reflects the one of its most promising children.

For example, in the above case, when s is seen, it has two children, with their F/G values being $7/2$ and $9/2$, respectively. Thus, we have $t(s, 7/0, [1(a, 7/2), 1(e, 9/2)])$.

Now, we expand its most promising child, i.e., a , until the best estimate of the cost of including a into the tree becomes more than that of e , where the tree developed becomes $t(s, 9/0, [1(e, 9/2), t(a, 10/2, [t(b, 10/4, [1(c, 10/6)])])])$.

Notice $g(s) = 0$, and $f(s) = 9$ is the best estimate of including s at this point, via e , and the two subtrees of s are sorted by the f value of their roots.

We will thus explore e as the next one.

Update estimation

Notice that at the end, the estimation of s has been updated to 9, while that of a updated to 10. This mechanism is certainly a necessity if we want to let the searching process recognize the most promising node during the process. This update can be characterized as following:

1) For a leaf, we still have

$$f(n) = g(n) + h(n).$$

2) For a tree, T , with subtrees, S_1, S_2, \dots , we have that

$$f(T) = \min_i \{f(S_i)\}.$$

A bunch of names

The key relation is $\text{expand}(P, T, \text{Bound}, T1, \text{Solved}, \text{Sol})$, where P is the path between the start node and T , rooted at N .

T is the current search tree.

Bound is the f -limit for expansion of T , i.e., the best estimate of T can't be more than Bound .

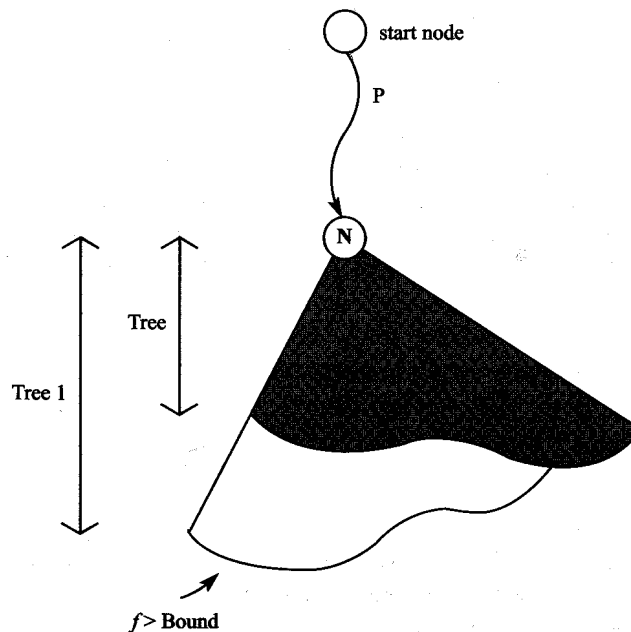
$T1$ is a T expanded within Bound . Eventually, the f -value of $T1$ will be larger than that of T , unless a goal is met.

Solved is either 'yes', 'no', or 'never'.

Sol provides the solution if Solved is "yes".

The gist

Let's look at a picture which demonstrates what the relation `expand` is intended to do.



T1 is expanded until either it finds a goal or its f -value exceeds Bound, where we pick up another more promising lead of N for further expansion.

Three possible scenarios

In `expand(P,T,Bound,T1,Solved,Sol)`, if the returned value of

1. `Solved` is 'yes', then `Sol` brings back a solution path found by expanding `T` within bound; while `T1` does not have a value.
2. `Solved` is 'no', then `T1` is expanded such that its f -value exceeds `Bound`. But, it might come back, so its value is not dropped. In this case, `Sol` does not have a value.
3. `Solved` is 'never', and both `T1` and `Sol` are uninstantiated. This shows that `T` is a dead end, when its f -value is less than `Bound`, but none of its leaf is expandable.

Get out the f -value

The relation $f(_,F)$ extracts the f -value of either a leaf or a tree.

```
f(l(_,F/_),F).
```

```
f(t(_,F/_,_),F).
```

The relation $bestf(TList,F)$ extracts the f -value of the very first tree of $TList$, after another relation $succlist$ is called to order all the trees in $TList$ by its f -values.

```
bestf([T|_],F):-f(T,F).
```

```
bestf([],9999).
```

Thus $bestf$ returns the best f -value of all the trees in $TList$.

Put things into order

The relation `insert(T,Tlist,NTList)` inserts `T` into `TList` according to its f -value, while the relation `succlist(G0,Succ,T)` sorts all the successors M , of N in `Succ` with $g(N) = G_0$, into `T`.

```
succlist(_, [], []).
```

```
succlist(G0, [M/C|NCs], Ts):-  
  G is G0+C, h(M,H), F is G+H,  
  succlist(G0, NCs, Ts1), %Ts1 is now sorted  
  insert(l(M, F/G), Ts1, Ts).
```

```
insert(T, Ts, [T|Ts]):-  
  f(T, F), bestf(Ts, F1), F=<F1, !.
```

```
%[T|Ts] is sorted by F, so T1 is the one  
% with the best F
```

```
insert(T, [T1|Ts], [T1|Ts1]):-  
  insert(T, Ts, Ts1).
```

The process

Initially, we start with the start node. We are looking for the solution path. No tree has been expanded yet.

We are willing to go anywhere, with any bound, and will accept the solution, if the value returned to `Solved` is a “yes”.

```
bestfirst(Start,Solution):-  
    expand([],l(Start,0/0),9999,_,yes,Solution).
```

We are done as soon as we see a goal node, and return the path, which is in the reverse order.

```
expand(P,l(N,_),_,_,yes,[N|P]):-goal(N).
```

The leaf case

When seeing a leaf N , we have constructed a partial path, P , with N being the last node of P .

If the f -value of this leaf is less than Bound , we will try to expand all its children to Succ , then sort them out into Ts , and pick up the f -value of its most promising child, $F1$, as the one for N , and continue from there.

```
expand(P,l(N,F/G),Bound,Tree1,Solved,Sol):-  
    F=<Bound,  
    (bagof(M/C,(s(N,M,C),not(member(M,P))),Succ),!,  
    succlist(G,Succ,Ts),bestf(Ts,F1),  
    expand(P,t(N,F1/G,Ts),Bound,Tree1,Solved,Sol)  
    ;  
    Solved = never %not expandable, dead end.  
    ).
```

Notice that this N is added into P in the next case.

The expandable tree case

When seeing a non-leaf node N with a bunch of subtrees, if its f -value is less than Bound , we set a new bound, make an expansion, and continue from there with the current search tree T .

Based on the value of $S1$ sent back by this expansion, we will let `continue` tell us what to do next. Notice, in the following, since $[T|Ts]$ is ordered in their f values, F is the f value of T .

```
expand(P,t(N,F/G,[T|Ts]),B,T1,S,Sol):-  
  F=<B,bestf(Ts,BF),min(B,BF,Bound1),  
  expand([N|P],T,Bound1,T1,S1,Sol),  
  continue(P,t(N,F/G,[T1|Ts]),B,T1,S1,S,Sol).
```

Here, with `Bound1`, we make sure that the bound for the current node being expanded is not more than that of the next one in line.

The other cases

In the next case, we have a non-leaf node, but no subtrees. This is a dead end, thus, we drop it.

```
expand(_,t(_,_,[ ]),_,_,never,_):-!.
```

In the last case, the f -value is larger than the Bound, we have to stop expanding along this path for now, but still keep this branch, and go back to the currently most promising one.

```
expand(_,Tree,Bound,Tree,no,_):-  
    f(Tree,F),F>Bound.
```

How about continue?

1. If a previous `expand` finds the solution, we are done.

```
continue(_,_,_,_,yes,yes,Sol).
```

2. If the previous *expand* does not work out, because its *f*-value exceeds the `Bound`, we put it into the right place in the candidate trees. Find the currently best one, then continue from there.

```
continue(P,t(N,F/G,[T1|Ts]),B,T1,no,S,Sol):-  
    insert(T1,Ts,NTs),bestf(NTs,F1),  
    expand(P,t(N,F1/G,NTs),B,T1,S,Sol).
```

3. If the current search tree eventually leads to a dead end, drop it completely, then continue with the other candidate trees.

```
continue(P,t(N,F/G,[_|Ts]),B,T1,never,S,Sol):-  
    bestf(Ts,F1),  
    expand(P,t(N,F1/G,Ts),B,T1,S,Sol).
```

Some general results

The best-first search algorithm just presented is a variant of the famous A^* algorithm, which does the best-first search.

A search algorithm is *admissible* if it is guaranteed to produce an optimal solution provided a solution exists. This property is often called *completeness*. Both BFS and DFSID are complete, or admissible.

Let $h^*(n)$ denote the cost of the optimal path from n to a goal node. A theorem states that an A^* algorithm is admissible if it uses a heuristic function h such that for all nodes n in its state space,

$$h(n) \leq h^*(n),$$

i.e., the heuristic function it uses is a lower bound of the real one.

Practically speaking

This result is important since even if we don't know h^* , the only thing we need is a lower bound.

For example, the trivial estimate $h(n) = 0$, for all n , will do, although it has no heuristic value.

Further more, if we let $c(n, n')$ be 1 for adjacent n and n' , such an A^* algorithm reduces to BFS, since now the one with the less depth from the start node will have a less value in its h value, thus explored first.

On the other hand, if we know h^* , and we use it as h , then we will get the optimal result without any backtracking.

Apply the search strategy

When applying the best-first search strategy to solve a problem, we have to represent that problem first. Particularly, we have to specify some problem specific knowledge, including:

1. represent the state space;
2. identify the start and goal state(s); and
3. represent the transition function: $s(N1, N2, Cost)$, i.e. there is an arc from $N1$ to $N2$ with $Cost$ being cost;
4. define the heuristic function: $h(Node, H)$, i.e., the estimation of the cost from $Node$ to a goal node is H .
5. Apply the searching method

```
bestfirst( Start, Solution) :-  
    expand( [], 1( Start, 0/0), 9999, _, yes, Solution).
```

Homework: 12.1.

The eight puzzle problem

The eight puzzle problem is a well-known and difficult problem, even for human being. Let's see how to solve it by applying the best-first search algorithm.

1. We represent a general state of the game with a list of positions: position of the empty, position of 1, ..., position of 8. For example, we represent the following state, with [2/2, 1/3, 2/3, 3/3, 3/2, 3/1, 2/1, 1/1, 1/2].

1	2	3
8		4
7	6	5

2. The above state is also the only goal state. Thus, we have that

`goal([2/2, 1/3, 2/3, 3/3, 3/2, 3/1, 2/1, 1/1, 1/2]).`

3. We can start with anything.

4. Conceptually, we can make a move by swapping the empty with any of the adjacent tiles. For example, if we swap the empty and 2, we will get the following:

1		3
8	2	4
7	6	5

5. To define the heuristic function, $h(n, H)$, we use an auxiliary relation, $\text{mandist}(S1, S2, D)$, which specifies the *Manhattan distance* between two tiles $N1$ and $N2$, namely, the gap between them in the horizontal direction plus that of the vertical direction.

For example, the Manhattan distance between 3 and 8 in the following state is $2+1=3$.

1	2	3
8		4
7	6	5

Heuristics

We define $h(\text{Pos}, H)$ in terms of two measures:
1) *totdist* is the total distance of the eight tiles in Pos from their positions as defined in the goal state.

The value of *totdist* for the following is 4, as indicated with the superscripts, respectively.

$$\begin{array}{ccc} 1_2^0 & 3_0^1 & 4_0^1 \\ 8_2^0 & & 2_2^2 \\ 7_0^0 & 6_0^0 & 5_0^0 \end{array}$$

Obviously, the smaller *totdist* is, the closer a state is to the goal state.

Another consideration is that in the goal state, i is always followed by $i + 1$. The more we have such i in a state, the better. Thus, ...

...another piece

2) seq is the sequence score that measures the degree to which the tiles are already ordered in Pos relative to the required one in the goal state.

More specifically, a) a center tile scores 1;

b) a non-center tile scores 0, if it is followed by its proper successor;

c) otherwise, it scores 2.

Thus, the value of seq for the above is 6.

Finally, we set the H value of any state to be $totdist + 3 * seq$, 23 for the above example.

State transitions

A move can be made from `[Empty|Tiles]` to `[Tile|Tiles1]` with a cost of 1, if for a `Tile` in `Tiles` which can swap its position with that of `Empty`.

```
s([Empty|Tiles],[Tile|Tiles1],1):-  
  swap(Empty,Tile,Tiles,Tiles1).
```

The state `[Empty|Tile,Tiles]` can change to `[Tile|Empty,Tiles]` if the position of `Tile` is adjacent to that of `Empty`.

```
swap(Empty,Tile,[Tile|Ts],[Empty|Ts]):-  
  mandist( Empty, Tile, 1).
```

Homework 1: Use the first swap clause to show that `[2/2, 2/3, 1/2, 1/3, 3/2, 3/3, 3/1, 1/1, 2/1]` can change to `[2/3, 2/2, 1/2, 1/3, 3/2, 3/3, 3/1, 1/1, 2/1]`.

Otherwise, `Empty` is not adjacent to the next piece in the list, called `T1`. We thus keep `T1` in the current position and look for `Tile`, a piece that *is* adjacent to `Empty` in `Ts`, the rest of the list, make a swap between `Empty` and `Tile` in `Ts`, and come up with a new state `Ts1`. Now the whole list is represented as `[T1|Ts1]`. The corresponding code is the following:

```
swap(Empty,Tile,[T1|Ts],[T1|Ts1):-  
    swap(Empty,Tile,Ts,Ts1).
```

Thus, the state `[Empty, T1|Ts]` changes to `[Tile, T1|Ts1]`.

Homework 2: Find out a configuration `Next` that makes `s([2/2, 1/3, 1/2, 2/3, 3/2, 3/3, 3/1, 1/1, 2/1], Next, 1)` succeed.

Manhattan distance

This is easy

```
mandist(X/Y,X1/Y1,D):-  
  dif(X,X1,Dx),dif(Y,Y1,Dy),  
  D is Dx+Dy.
```

```
dif(A,B,D):-D is A-B,D>=0,!  
  ;  
  D is B-A.
```

Heuristics

Find out the total distance between Tiles and their final destination as specified in the goal, then find out their scores, finally sum them up as mentioned before.

```
h([Empty|Tiles],H):-  
    goal([Empty1|GoalSquares] ),  
    totdist(Tiles,GoalSquares,D),  
    seq(Tiles,S),H is D+3*S.
```

Find out the Manhattan distance between respective tiles, then add them up.

```
totdist([],[],0).  
totdist([Tile|Tiles],[Square|Squares],D):-  
    mandist(Tile,Square,D1),  
    totdist(Tiles,Squares,D2),  
    D is D1+D2.
```

Get the scores

We first use a bunch of facts to specify the scores, which state that if n is in X/Y , and $n+1$ follows it in $X1/Y1$, the n gets a score 0.

```
%A center tile always gets 1.
```

```
score(2/2,_,1):-!.
```

```
score(1/3,2/3,0):-!.
```

```
score(2/3,3/3,0):-!.
```

```
score(3/3,3/2,0):-!
```

```
score(3/2,3/1,0):-!.
```

```
score(3/1,2/1,0):-!.
```

```
score(2/1,1/1,0):-!.
```

```
score(1/1,1/2,0):-!.
```

```
score(1/2,1/3,0):-!.
```

Otherwise, n scores 2.

```
score(_,_,2).
```

For example, if 1 is in 1/3 and 2 is in 2/3, then 1 gets a score of 0, otherwise, 1 is not followed by 2, hence, 1 gets a 2.

The only other tricky point is to keep a record for the position of the tile 1, so that at the end, we can get a score between the tile 1, and tile 8.

```
seq([First|OtherTiles],S):-  
    seq([First|OtherTiles],First,S).
```

```
seq([Tile1,Tile2|Tiles],First,S):-  
    score(Tile1,Tile2,S1),  
    seq([Tile2|Tiles],First,S2),  
    S is S1+S2.
```

```
seq([Last],First,S):-  
    score(Last,First,S).
```

Homework 3: Run the code to find out the heuristic value for the state as shown in pp. 26.

Print out the solution

We are done when the list is empty.

```
showsol([]).
```

If we have something to print, print the rest first, then print the first. The key is that the solution is reversed.

```
showsol([P|L]):-  
  showsol(L),nl,  
  write('---'),showpos(P).
```

Print a board

To print the content of the board as a result of a move, we print out the content of 1/3, 2/3, 3/3,..., until the content of 3/1 is printed.

```
showpos([S0,S1,S2,S3,S4,S5,S6,S7,S8]):-  
  member(Y,[3,2,1]),  
  nl,member(X,[1,2,3]),  
  member(Tile-X/Y,  
    [' '-S0,1-S1,2-S2,3-S3,4-S4,  
     5-S5,6-S6,7-S7,8-S8]),  
  write(Tile),fail  
  ;  
  true.
```

Notice for each value of Y, it will move to the next line, and whenever a value is printed, it will move over to the next column.

An example

For the following state:

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

we have that $S_0=2/3$, $S_1=1/3$, $S_2=2/2$, $S_3=3/3$, etc. Thus, when $X/Y=1/3$, the program tries to do a match of `Tile-1/3` with the values in the list of [`' '`- $2/3$, $1-1/3$, $2-2/2$, $3-3/3$, ...] and will match with $1-1/3$. In particular, `Tile=1`. Hence, it will print out 1 in the position of $1/3$.

It then take the value of X/Y to be $2/3$, and do a match of `Tile-2/3` with the list, and find out `Tile=' '`, and print out the empty space. It then print out a 3 in the position of $3/3$, etc..

Notice a `nl` moves to the next line, and after every `write`, it moves one position to the right.

Time and space complexity

Best-first search typically cuts the search space to only a small part of the state space. More specifically, a best-first search effectively cuts the branching factor from the original b to b' , which is usually significantly smaller than b .

But, the time, and space, complexity of A^* in general, and our implementation in particular, is still exponential in terms of the depth of a search. This is true since the algorithm holds all the nodes generated so far.

Several variants have been developed that save space at the cost of time. The basic idea is similar to the iterative deepening version of DFS, which cuts the space to only linear. The price is to regenerate some of the nodes that have already been generated before.