Branch & Bound:

The term branch & bound refers to all state space search methods in which all children of ~~the~~ a given node are generated before any other live node become active.

We have already seen two graph search strategies BFS & DFS. Here the exploration of a new node cannot begin until the node currently being explored is fully explored. Both of these generalized to branch and bound strategies. In branch & bound terminology a BFS like state-space search will be called FIFO (First In First Out) search as the list of live nodes is a first in first out list. A DFS is a state space search which uses LIFO (Last-In First Out) as the list of live nodes is a last in first out list.

Along with these search strategies an ~~the~~ state-space-tree ~~(a bounding function is)~~ generated by branching of the node, ~~the~~ the bounding function is used to help avoid the generation of subtrees that do not contain an answer node.

Branch & Bound: 16 puzzle Problem:

1	3	4	15
2		5	12
7	6	11	14
8	9	10	13

Initial Configuration

$\Rightarrow$

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Final configuration

We all are familier with 15 puzzle problem, here the bord consists of 15 ~~tiles~~ movable tiles numbered from 1 to 15 & there is one empty spot. The tiles can move in four direction (up, down, left, right).

"The puzzle refers to obtaining the final configuration from a given initial configuration." In general this is a permutation type of problem. Therefore the total possible number of solution can be $16! = \underline{2.09 \times 10^{13}}$, which is a staggering number, & if we try to solve it using the traditional brute-force mechanism the state-space tree may become unmanageable, & for any initial configuration half of the intermediate states can be reached.

To avoid this complexity we will consider solving the problem by using some heuristic methodology.

1	3	4	15
2		5	12
7	6	11	14
8	9	10	13

Initial configuration

up | Down | right | left

1		4	15
2	3	5	12
7	6	11	14
8	9	10	13

configuration-1

1	3	4	15
2	6	5	12
7		11	14
8	9	10	13

configuration-2

1	3	4	15
2	5		12
7	6	11	14
8	9	10	13

configuration-3

1	3	4	15
	2	5	12
7	6	11	14
8	9	10	13

configuration-4.

Lets take a look at different board configuration. In the above example it is shown that we can have four different board configuration from a given initial configuration by applying up, down, right & left movement of the empty space.

Up move → empty spot will go up (swap the number) & the rest of the board position remain same.

Once we have a particular move we should not use it's opposite move in the immediately next step. for example -

1	3	4	15
2		5	12
7	6	11	14
8	9	10	13

Initial config.

UP ⟹

1		4	15
2	3	5	12
7	6	11	14
8	9	10	13

Config-1

DOWN ⟹

1	3	4	15
2		5	12
7	6	11	14
8	9	10	13

Config-2.

Here we first applied <u>'up'</u> move & got config-1 then we applied <u>'DOWN'</u> move next. & we got config-2. which is same as the initial config.

As discussed earlier that to reduce the size of the problem we will apply some heuristics so that we are not going to branch all the possible configurations, we should bound some of them which are not promising towards the goal configuration. In this way the game state-space tree need not to be expanded to each & every node available. which increases the time & space complexity.

One such heuristic formula is -

$$\hat{c}(x) = f(x) + \hat{g}(x) \quad \text{where,-}$$

$\hat{c}(x)$ = Estimated minimum cost to reach to the goal node.

$f(x)$ = Length of the path from the root to node 'x'

$\hat{g}(x)$ = Is an estimate of length of a shortest path from 'x' to a goal node in the subtree with root 'x'.

Here $\hat{g}(x)$ = Number of non-blank tiles not in their goal position.

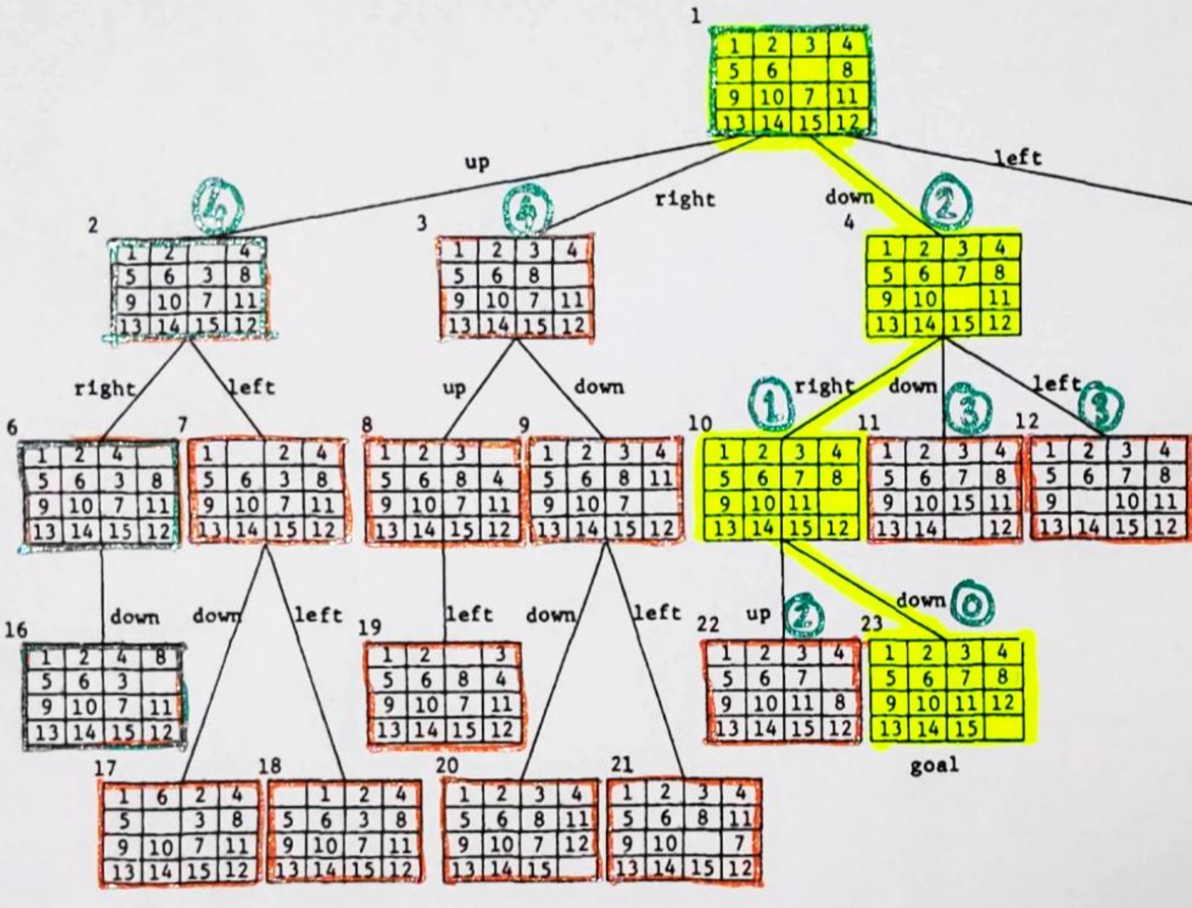
Algorithm:

The Algorithm below uses two function Least() and Add(x) to respectively delete & add a live node from or to the list of live nodes. Least() finds a live node with least estimated cost. Add(x) adds the new live node x to the list of live nodes. This list of live nodes usually implemented using a min-heap.

The algorithm outputs the path from the answer node it finds to the root node t.

```
Struct listnode {
    Struct listnode * next;
    Struct listnode * prev;
    float cost
}.
LC Search ( Struct listnode *t)
{
    Struct listnode *x, *E, * least();
    if (*t is an answer node)
        output *t and return;
    E = t
    initialize the list of live nodes to be empty.
    do {
        for (each chield x of N) {
            if (x is an answer node)
                output the path from x to t and return.
            Add (x);
            x -> parent = N;
        }
        if (there are no more live nodes) {
            output (( no answer node)) & return;
        }
        N = least();
    } while (1);
}.
```

Algorithm for 15-puzzle problem.



edges are labeled according to the direction in which the empty space moves

Figure 8.3(a) Part of the state space tree for the 15-puzzle

Node which has least disimilarity & degrees to the Solution.

bounded node

dissimilarity index