Algorithms and Lab (CSE130)

State Space Search Techniques : Branch and Bound

Muhammad Tariq Mahmood

tariq@koreatech.ac.kr
School of Computer Science and Engineering

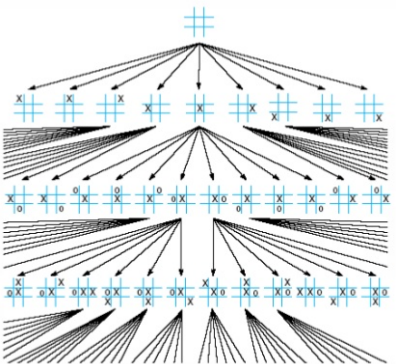
Note: These notes are prepared from the following resources.

- (main text)Foundations of Algorithms, by Richard Neapolitan and Kumarss Naimipour
- Python Algorithm (파이썬 알고리즘) by Y.K. Choi (2021) (Korean)
- Introduction to the Design and Analysis of Algorithms by Anany Levitin
- Introduction to Algorithms, by By Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein
- https://www.geeksforgeeks.org

Contents

State Space Tree

- A problem can be represented by a tree structure(named as **State Space Tree**)
- Few terminologies are provided related to **state space** formulation of a problem
- **State Space**: Set of All states reachable from the initial state and it forms a graph (Tree) in which the nodes are states and the arcs are actions
- **A Start State**: The state from where the search begins.
- **Path**: A path in the state space is a sequence of states connected by a sequence of actions
- **Solution**:
  - A path from the initial state to another state (the goal state)
  - A state/node
- **Optimal Solution**: Has lowest path cost amongst all solutions



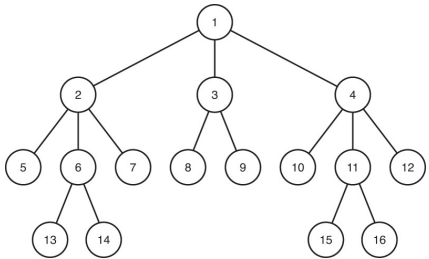
Backtracking

- **Backtracking** is a modified ***depth-first search(DFS)*** of a tree.
- **Backtracking** is the procedure whereby, after determining that a node can lead to nothing but dead ends, we go back ("backtrack") to the node's parent and proceed with the search on the next child.
- We call a **nonpromising** node if when visiting the node we determine that it cannot possibly lead to a solution. Otherwise, we call it **promising node**.
- In other words, backtracking consists of doing a depth-first search of a state space tree, checking whether each node is promising, and, if it is nonpromising, backtracking to the node's parent.
- This is called **pruning the state space tree**, and the subtree consisting of the visited nodes is called the **pruned state space tree**.
- A general algorithm for the backtracking approach is as follows:

```
1:  procedure backtrack2(v)
2:      Node u
3:      for (each child u of v) do
4:          if (promising(v)) then
5:              if (there is a solution at v) then
6:                  write the solution
7:              else
8:                  backtrack2(u)
9:              end if
10:         end if
11:     end for
12: end procedure
```

Breadth First Search

- A breadth-first search consists of visiting the root first, followed by all nodes at level 1, followed by all nodes at level 2, and so on.



- Unlike depth-first search, there is no simple recursive algorithm for breadth-first search. However, it can be implemented using a queue data structure.

```
1:  procedure BFS(Tree T)
2:      Queue Q
3:      Node u, v
4:      initialize(Q)
5:      v=root of T
6:      visit v
7:      enqueue(Q, v)
8:      while (!empty(Q)) do
9:          v=dequeue(Q)
10:         for ((each child u of v)) do
11:             visit u
12:             enqueue(Q, u)
13:         end for
14:     end while
15: end procedure
```

Branch and Bound Technique

- The branch-and-bound design strategy is very similar to backtracking in that a state space tree is used to solve a problem.
- The differences are that the branch-and-bound method (1) does not limit us to any particular way of traversing the tree and (2) is used only for optimization problems
- A branch-and-bound algorithm computes a number (bound) at a node to determine whether the node is promising.
- If that bound is no better than the value of the best solution found so far, the node is nonpromising. otherwise, it is promising.
- Using the bound to determine whether a node is promising, we can compare the bounds of promising nodes and visit the children of the one with the best bound.
- In this way we often can arrive at an optimal solution faster than we would by methodically visiting the nodes in some predetermined order (such as a depth-first search).
- This approach is called best-first search with branch-and-bound pruning. The implementation of the approach is a simple modification of another methodical approach called breadth-first search with branch-and-bound pruning.

Branch and Bound Technique (cont...)

Branch and Bound Algorithm

- This algorithm is a modification of the breadth-first search (BFS) algorithm.
- We expand beyond a node (visit a node's children) only if its *bound* is better than the value of the current *best* solution.
- The functions **bound** and value are different in each application of breadth first branch_and_bound.
- There are basically three types of nodes involved in Branch and Bound
  1. **Live node** is a node that has been generated but whose children have not yet been generated.
  2. **E-node** is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.
  3. **Dead node** is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.

- The general algorithm for breadth-first search with branch-and-bound pruning.

```
 1: procedure BnB(Tree T)
 2:     Queue Q
 3:     Node u, v
 4:     initialize(Q)
 5:     v=root of T
 6:     enqueue(Q, u)
 7:     best = value(v)
 8:     while (!empty(Q)) do
 9:         v = dequeue(Q)
10:         for ((each child u of v)) do
11:             if (value(u) is better than best) then
12:                 best = value(u)
13:             end if
14:             if (bound(u)) is better than best then
15:                 enqueue(Q, u)
16:             end if
17:         end for
18:     end while
19: end procedure
```

Branch and Bound Technique (cont...)

Branch and Bound with Best First Search:

- In general, the breadth-first search strategy has no advantage over a depth-first search (backtracking).
- However, we can improve our search by using our bound to do more than just determine whether a node is promising. After visiting all the children of a given node, we can look at all the promising, unexpanded nodes and expand beyond the one with the best bound.
- Recall that a node is promising if its bound is better than the value of the best solution found so far.
- In this way we often arrive at an optimal solution more quickly than if we simply proceeded blindly in a predetermined order.
- This ordered depth search is named as **best-first search**
- **best-first search** may reduce the number of nodes in the space tree

- Algorithm for best-first search is provided here

```
1:  procedure BnBwithBestFirstSearch(Tree T)
2:      PriorityQueue PQ
3:      Node u, v
4:      initialize(PQ)
5:      v=root of T
6:      enqueue(PQ, u)
7:      best = value(v)
8:      while (!empty(PQ)) do
9:          v = dequeue(PQ)
10:         for ((each child u of v)) do
11:             if (value(u) is better than best) then
12:                 best = value(u)
13:             end if
14:             if (bound(u)) is better than best then
15:                 enqueue(PQ, u)
16:             end if
17:         end for
18:     end while
19: end procedure
```

Coin Change-making problem

- Give change for amount n using the minimum number of coins from a set of coins $\{d_1 < d_2 < d_3 < ... < d_m\}$

Dynamic programming approach

- For the general case, assuming availability of unlimited quantities of coins for each of the m denominations
- **Problem**: find the minimum number of coins of denominations $\{d_1 < d_2 < d_3 < ... < d_m\}$ that add up to a given amount n
- **Inputs**: Positive integer n and array $D[1..m]$ of increasing positive integers indicating the coin denominations where $D[1] = 1$
- **Outputs**: The minimum number of coins that add up to n
- Recursive Property

$$F(n) = min_{j:n \geq d_j} \{F(n - d_j)\} + 1, n > 0$$
$$F(n) = 0, n = 0$$

Coin Change-making problem (cont...)

- **Pseudo-code:**

```
1:  procedure ChangeMaking(integer n, D[1..m])
2:      F[0]=0
3:      for ((i = 1; i <= n; i++)) do
4:          temp=inf, j=0
5:          while (j ≤ m) && (i ≥ D[j]) do
6:              temp = min(F[i-D[j]], temp)
7:              j = j+1
8:          end while
9:          temp = temp+1
10:     end for
11:     return F[n]
12: end procedure
```

- Time complexity : $T(nm) \in \Theta(nm)$
- Application of ChangeMaking to amount $n = 6$ and coin denominations 1, 3, and 4.

$F[0] = 0$

n	0	1	2	3	4	5	6
F	0						

$F[1] = \min\{F[1-1]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1					

$F[2] = \min\{F[2-1]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2				

$F[3] = \min\{F[3-1], F[3-3]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1	2	1			

$F[4] = \min\{F[4-1], F[4-3], F[4-4]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1	2	1	1		

$F[5] = \min\{F[5-1], F[5-3], F[5-4]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2	1	1	2	

$F[6] = \min\{F[6-1], F[6-3], F[6-4]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2	1	1	2	**2**

Coin Change-making problem (cont...)

Greedy approach

- In a greedy algorithm for change-making problem , Each iteration consists of the following components
  1. **selection procedure:** The criterion for deciding which coin is the best (locally optimal) is the coin having the highest value. (sorted coins)
  2. **feasibility check** If adding a coin to the change would make the total value of the change exceed the amount owed.
  3. **solution check** If the value of the change is now equal to the amount owed.

- **Pseudo-code:**

```
1:  procedure change(coins,owed amount)
2:      while (there are more coins and the instance is not solved) do
3:          grab the largest remaining coin                          ▷ selection procedure
4:          if (adding the coin makes the change exceed the amount owed) then
5:              reject the coin                                      ▷ feasibility check
6:          else
7:              add the coin to the change
8:          end if
9:          if (the total value of the change equals the amount owed) then
10:             the instance is solved                               ▷ solution check
11:         end if
12:     end while
13: end procedure
```

Coin Change-making problem (cont...)

- The greedy algorithm for instance is shown in Figures

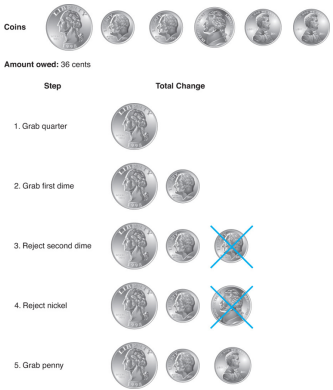


Figure 1: A greedy algorithm for giving change

- The greedy algorithm is not optimal if a 12-cent coin is included as shown in Figure 2.

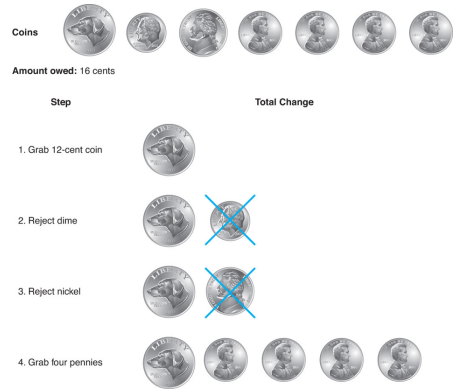
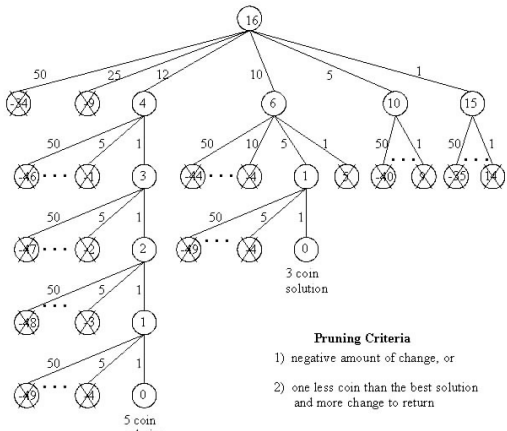


Figure 2: The greedy algorithm is not optimal if a 12-cent coin is included

Coin Change-making problem (cont...)

Backtracking approach

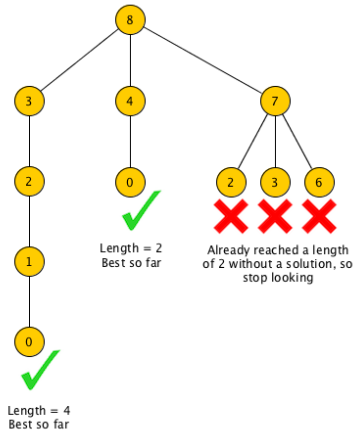
- Use the best solution found so far to prune partial solutions that are not "promising,", i.e., cannot lead to a better solution than one already found.
- The goal is to prune enough of the state-space tree (exponential is size) that the optimal solution can be found in a reasonable amount of time.
- State-Space Tree for 16-cents coins $\{1, 5, 10, 12, 25, 50\}$



Coin Change-making problem (cont...)

Branch-and-Bound approach

- Let's try to find the minimum number of coins to make change from 8 cents, given that our coin denominations are 5 cents, 4 cents, and 1 cent:
- We used a branch and bound approach. We started with 8 cents at the root. We branched by trying a new coin. We bounded our search whenever we found that there was no reason to keep going down our current path, since we had already reached (or passed) the best result we found so far.
- Search the "state-space tree" using no specific search pattern. Instead, expand the tree node that appears to lead to the best solution - node with the best bound



Assignment Problem

- Let there are n people who need to be assigned to execute n jobs, one person per job. (That is, each person is assigned to exactly one job and each job is assigned to exactly one person.)
- The cost that would accrue if the *ith* person is assigned to the *jth* job is a known quantity $C[i, j]$ for each pair $i, j = 1, 2, ..., n$.
- A small instance of this problem follows, with the table entries representing the assignment costs $C[i, j]$.
- It is easy to see that an instance of the assignment problem is completely specified by its cost matrix C.
- The problem is to find an assignment with the minimum total cost.

$$C = \begin{array}{c} \begin{array}{cccc} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} \end{array} \\ \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \end{array} \begin{array}{l} \text{person } a \\ \text{person } b \\ \text{person } c \\ \text{person } d \end{array}$$

- In terms of this matrix, the problem is to select one element in each row of the matrix so that all selected elements are in different columns and the total sum of the selected elements is the smallest possible.

Assignment Problem (cont...)

- Solution 1: Brute Force/exhaustive-search
  - We can describe feasible solutions to the assignment problem as n-tuples $j_1, ..., j_n$ in which the ith component, $i = 1, ..., n$, indicates the column of the element selected in the ith row (i.e., the job number assigned to the ith person).
  - the exhaustive-search approach to the assignment problem would require generating all the permutations of integers $1, 2, ..., n$, computing the total cost of each assignment by summing up the corresponding elements of the cost matrix, and finally selecting the one with the smallest sum.
  - We generate $n!$ possible job assignments and for each such assignment, we compute its total cost and return the less expensive assignment. Since the solution is a permutation of the n jobs, its complexity is $O(n!)$.

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

<1, 2, 3, 4>	cost = 9 + 4 + 1 + 4 = 18
<1, 2, 4, 3>	cost = 9 + 4 + 8 + 9 = 30
<1, 3, 2, 4>	cost = 9 + 3 + 8 + 4 = 24
<1, 3, 4, 2>	cost = 9 + 3 + 8 + 6 = 26
<1, 4, 2, 3>	cost = 9 + 7 + 8 + 9 = 33
<1, 4, 3, 2>	cost = 9 + 7 + 1 + 6 = 23

  - First few iterations of solving a small instance of the assignment problem by exhaustive search.

Assignment Problem (cont...)

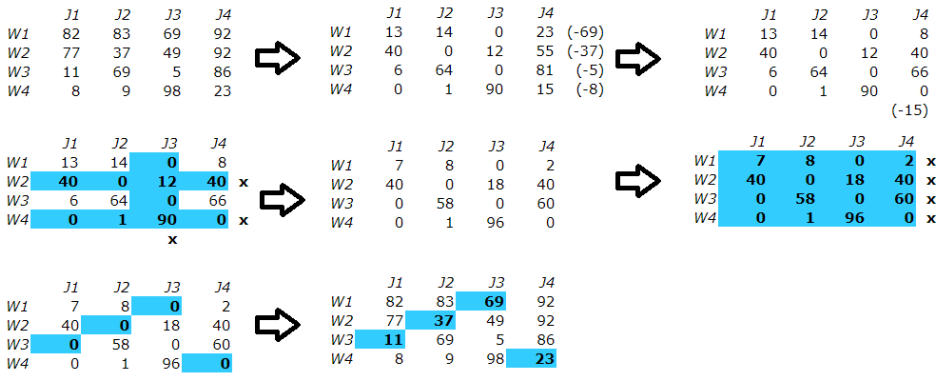
- Solution 2: Hungarian Algorithm : The optimal assignment can be found using the Hungarian algorithm. The Hungarian algorithm has worst case run-time complexity of $O(n^3)$.

- The Hungarian algorithm, aka Munkres assignment algorithm, utilizes the following theorem for polynomial runtime complexity (worst case $O(n^3)$) and guaranteed optimality:
  If a number is added to or subtracted from all of the entries of any one row or column of a cost matrix, then an optimal assignment for the resulting cost matrix is also an optimal assignment for the original cost matrix.
  We reduce our original weight matrix to contain zeros, by using the above theorem. We try to assign tasks to agents such that each agent is doing only one task and the penalty incurred in each case is zero.

- The Hungarian algorithm : The Hungarian algorithm consists of the four steps below.
  1. Step 1: Subtract row minima : For each row, find the lowest element and subtract it from each element in that row.
  2. Step 2: Subtract column minima : Similarly, for each column, find the lowest element and subtract it from each element in that column.
  3. Step 3: Cover all zeros with a minimum number of lines: Cover all zeros in the resulting matrix using a minimum number of horizontal and vertical lines. If n lines are required, an optimal assignment exists among the zeros. The algorithm stops. If less than n lines are required, continue with Step 4.
  4. Step 4: Create additional zeros : Find the smallest element (call it k) that is not covered by a line in Step 3. Subtract k from all uncovered elements, and add k to all elements that are covered twice.

Assignment Problem (cont...)

- Example:

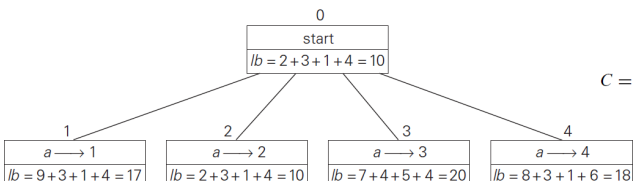


https : //www.hungarianalgorithm.com/examplehungarianalgorithm.php

Assignment Problem (cont...)

Solution 3: Optimal Solution using Branch and Bound

- Start with the root that corresponds to no elements selected from the cost matrix.
- There are two approaches to calculate the cost function(bound).
  1. For each worker, we choose job with minimum cost from list of unassigned jobs (take minimum entry from each row).
  2. For each job, we choose a worker with lowest cost for that job from list of unassigned workers (take minimum entry from each column).



$$C = \begin{array}{c} \begin{array}{cccc} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} \end{array} \\ \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \end{array} \begin{array}{l} \text{person } a \\ \text{person } b \\ \text{person } c \\ \text{person } d \end{array}$$

Figure 3: Levels 0 and 1 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm. The number above a node shows the order in which the node was generated. A node's fields indicate the job number assigned to person a and the lower bound value, lb, for this node.

Assignment Problem (cont...)

- State space tree

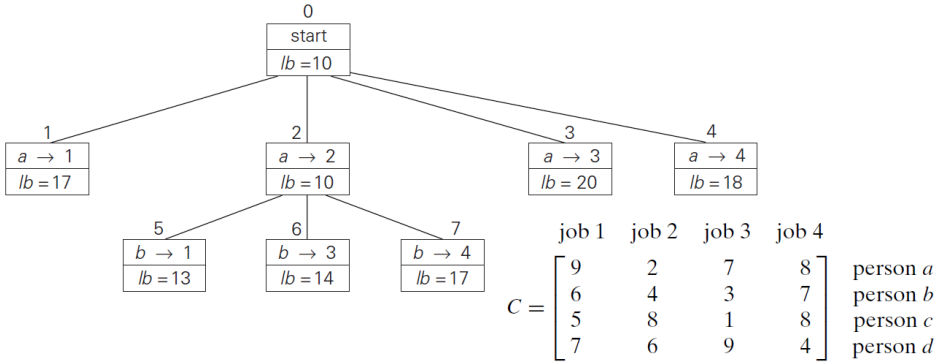


Figure 4: Levels 0, 1, and 2 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm.

Assignment Problem (cont...)

- State space tree

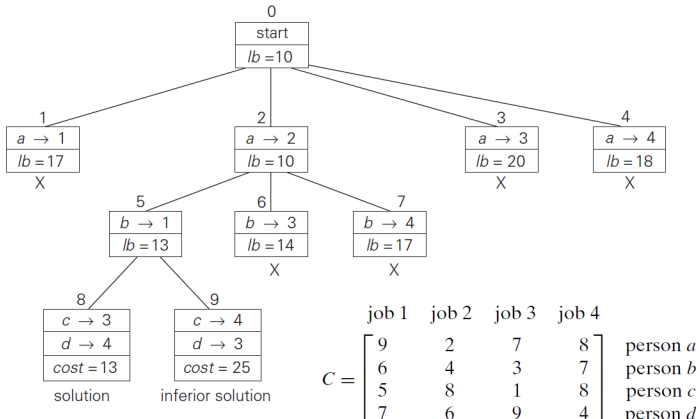


Figure 5: Complete state-space tree for the instance of the assignment problem solved with the best-first branch-and-bound algorithm.

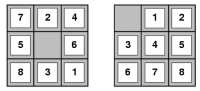
Assignment Problem (cont...)

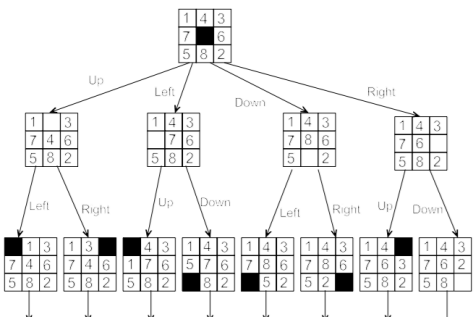
- Outlines of the algorithm are provided here

```
1:  procedure MinCost(costMatrix mat[][])
2:      PriorityQueue PQ
3:      Node root: a Dummy node
4:      calculate its lower bound
5:      enqueue(PQ, root)
6:      while (!empty(PQ)) do
7:          NodeE = dequeue(PQ)                ▷ Node E contains job with min cost
8:          Assign the job to work i
9:          if all workers are assigned a job then
10:             Print Solution
11:             return
12:         end if
13:         i++
14:         for (each job j) do
15:             create a new tree node C
16:             calculate its lower bound
17:             enqueue(PQ, C)
18:         end for
19:     end while
20: end procedure
```

8-puzzle Problem

1. A typical instance of the 8-puzzle problem is shown in Figure
2. **States:** A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
3. **Initial state:** Any state can be designated as the initial state.
4. **Actions:** The simplest formulation defines the actions as movements of the blank space Left, Right, Up, or Down. Different subsets of these are possible depending on where the blank is.
5. **Transition model:** Given a state and action, this returns the resulting state.
6. **Goal test:** This checks whether the state matches the goal configuration
7. **Path cost:** Each step costs 1, so the path cost is the number of steps in the path.

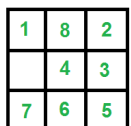


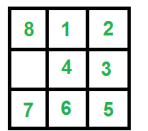


8-puzzle Problem (cont...)

How to find if given state is solvable?

- It is not possible to solve an instance of 8 puzzle if number of inversions is odd in the input state.





- **What is inversion?** A pair of tiles form an inversion if the the values on tiles are in reverse order of their appearance in goal state. For example, (8, 6) and (8, 7) are two inversions.

- In the examples given in above figure, the first example has 10 inversions, therefore solvable. The second example has 11 inversions, therefore unsolvable.

8-puzzle Problem (cont...)

- if a board has an odd number of inversions, then it cannot lead to the goal board by a sequence of legal moves because the goal board has an even number of inversions (zero).

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

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

 inversions = 1   inversions = 1   inversions = 3   inversions = 3   inversions = 3
       (8-7)            (8-7)      (6-5 8-5 8-7)    (6-5 8-5 8-7)    (6-5 7-5 8-5)
```

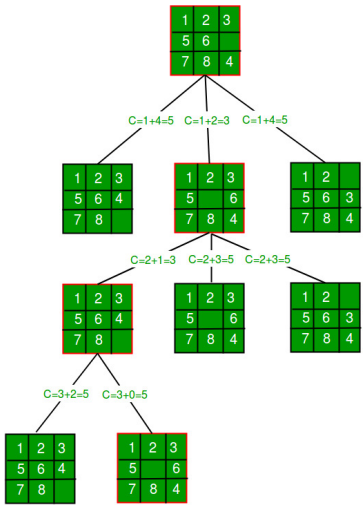
- If the board size an odd integer, then each legal move changes the number of inversions by an even number

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

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

 inversions = 4   inversions = 4   inversions = 2   inversions = 2   inversions = 0
(3-2 4-2 7-6 8-6) (3-2 4-2 7-6 8-6)      (7-6 8-6)        (7-6 8-6)
```

8-puzzle Problem (cont...)

Bound (cost) function for the 8-Puzzle problem

- Each node X in the search tree is associated with a cost. The cost function is useful for determining the next E-node.
- The next E-node is the one with least cost. The cost function is defined as,
  $C(X) = g(X) + h(X)$ where
  $g(X)$ =cost of reaching the current node from the root
  $h(X)$ = cost of reaching an answer node from X.
  (Heuristic function)
- Heuristics function: Heuristic is a function which is used in Informed Search, and it finds the most promising path. Heuristic function estimates how close a state is to the goal. It is represented by h(n), and it calculates the cost of an optimal path between the pair of states.



8-puzzle Problem (cont...)

- There are several priority functions can be used to calculate the $h(n)$ including:
  - **Hamming Distance:** The number of bits which differ between two binary strings. More formally, the distance between two strings A and

$$d_h(A, B) = \sum_{i=1}^{n} |A_i - B_i|$$

The number of tiles not in their goal position(the number of misplaced tiles)

  - **Manhattan Distance:** The sum of the vertical and horizontal distances from each tile to their goal position.

$$d_m(p_1, p_2) = |x_2 - x_1| + |y_2 - y_1|$$

- Example

```
8  1  3        1  2  3     1  2  3  4  5  6  7  8     1  2  3  4  5  6  7  8
4     2        4  5  6     ----------------------     ----------------------
7  6  5        7  8        1  1  0  0  1  1  0  1     1  2  0  0  2  2  0  3

initial          goal          Hamming = 5 + 0          Manhattan = 10 + 0
```

8-puzzle Problem (cont...)

BnB algorithm for the 8-Puzzle problem

- A high level branch and bound algorithm for 8-puzzle problem is provided below

```
1:  procedure solvePuzzle(int[][]initialState, int[][]finalState)
2:      PriorityQueue PQ
3:      State root                                              ▷ initial state
4:      calculate Cost (bound) for root state/node
5:      PQ.enqueue(s))
6:      while (!empty(Q)) do
7:          State current = dequeue(PQ)
8:          if (current.cost == 0) then                         ▷ If solution found, print it
9:              print path
10:             return
11:         end if
12:         for ((each child C of current state)) do
13:             calculate Cost (bound) for C
14:             PQ.enqueue(C)
15:         end for
16:     end while
17: end procedure
```

Summary