Project Report

CSC 7336 : Advanced Software Engineering

Project : The 8-puzzle implementation

Master Of Science CCN 16/02/2018

Written by: **Sanae SEBAI**
Sisihlia Yuniyarti

Supervised by: **M. J Paul Gibson**

I. Artificial Intellegence on Android development

We have 3 fitness functions: Mismatch, Manhattan and BFS.

String str = ***"208635147"***;

String goal = ***"123456780"***;

We apply those 3 different approach for same position (str) and get different result based on each complexity.

I/System.out: Mismatch

I/System.out: Solution found in 75 step(s)

I/System.out: Node generated: 904

I/System.out: Unique Node generated: 551

I/System.out: Solution found is 123456780 in 75 step(s)

I/System.out: Fitness value is 0.00

I/System.out: time duration is 0.046810273 s

I/System.out: Actual Memory used 416384

I/System.out: BFS

I/System.out: Solution found in 17 step(s)

I/System.out: Node generated: 47853

I/System.out: Unique Node generated: 26170

I/System.out: Result is 123456780

I/System.out: Fitness value is 0.00

I/System.out: time duration is 0.269670446 s

I/System.out: Actual Memory used 3004216

I/System.out: Manhattan

I/System.out: Solution found in 17 step(s)

I/System.out: Node generated: 53

I/System.out: Unique Node generated: 35

I/System.out: State found is 123456780 in 17 step(s)

I/System.out: Fitness value is 0.00

I/System.out: time duration is 0.025325297 s

I/System.out: Actual Memory used 61936

==>the smallest time duration is given by Manhattan function with the least memory used. So when it is solvabale, the code generates the steps to for users.

I/System.out: Move 0 DOWN

I/System.out: Move 1 LEFT

I/System.out: Move 2 DOWN

I/System.out: Move 3 RIGHT

I/System.out: Move 4 RIGHT

I/System.out: Move 5 UP

I/System.out: Move 6 UP

I/System.out: Move 7 LEFT

I/System.out: Move 8 LEFT

I/System.out: Move 9 DOWN

I/System.out: Move 10 DOWN

I/System.out: Move 11 RIGHT

I/System.out: Move 12 RIGHT

I/System.out: Move 13 UP

I/System.out: Move 14 LEFT

I/System.out: Move 15 DOWN

I/System.out: Move 16 RIGHT

II. Parallel Programming :

Mechanisms:

1- Search Tree

• Each node represents a problem or subproblem

• Root of tree: initial problem to be solved

• Children of a node created by adding constraints (one move from the father)

• AND node: to find solution, must solve problems represented by all children

• OR node: to find a solution, solve any of the problems represented by the children

• AND tree : Contains only AND nodes , Divide-and-conquer algorithms

• OR tree : Contains only OR nodes , Backtrack search and branch and bound

• AND/OR tree : Contains both AND and OR nodes, game trees

2- Divide and Conquer methodology:

• Partition a problem into subproblems

• Solve the subproblems

• Combine solutions to subproblems

• Recursive: sub-problems may be solved using the divide-and-conquer methodology

• Subproblems must be distributed among memories of individual processors

3- Backtrack Search:

• OR tree : Contains only OR nodes

• Uses depth-first search to consider alternative solutions to a combinatorial search problem

• Recursive algorithm

• Backtrack occurs when

• A node has no children (“dead end”)

• All of a node’s children have been explored

Parallel Backtrack Search

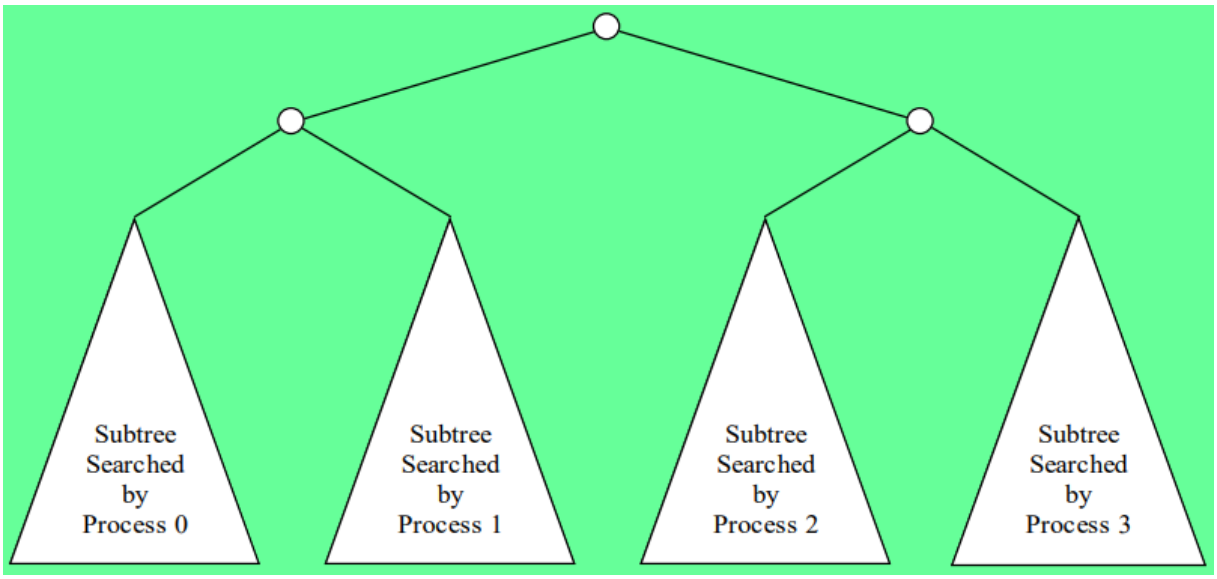
• First strategy: give each processor a subtree

• Suppose p = bk

• A process searches all nodes to depth k

• It then explores only one of subtrees rooted at level k

• If d (depth of search) > 2k, time required by each process to traverse first k levels of state space tree is negligible



What If p ≠ bk ?

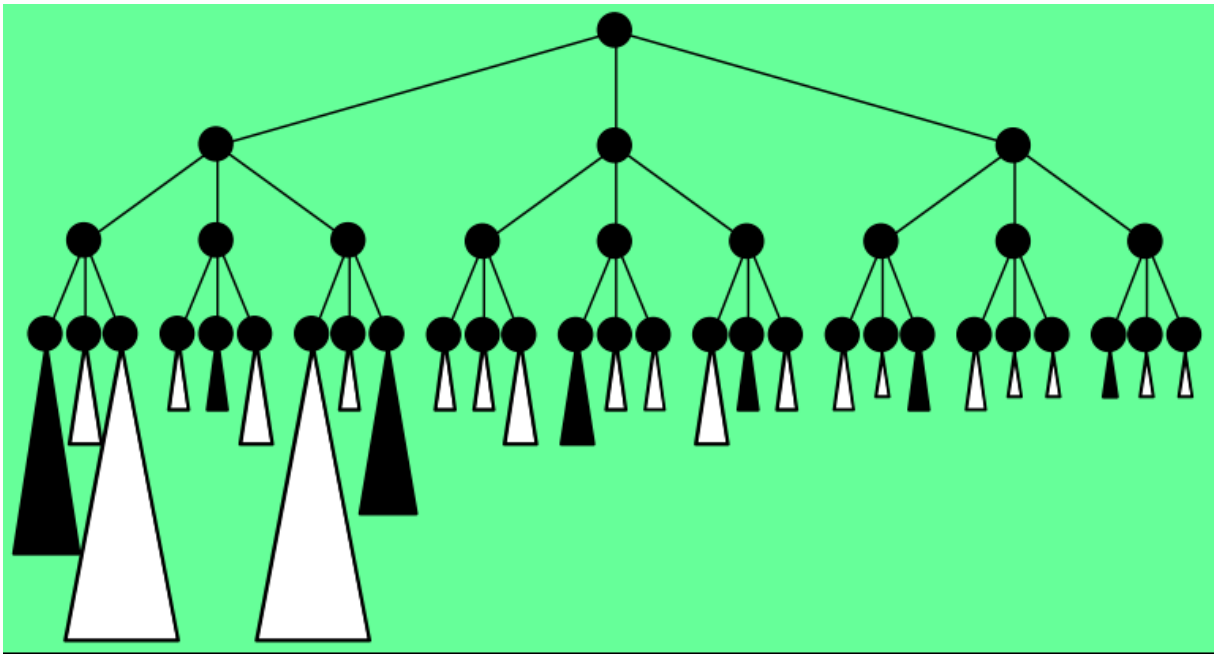
• A process can perform sequential search to level m (where bm > p) of state space tree

• Each process explores its share of the subtrees rooted by nodes at level m

• As m increases, there are more subtrees to divide among processes, which can make workloads more balanced

• Increasing m also increases number of redundant computations

Allocating Many Subtrees per Process



b = 3; p = 4; m = 3; allocation rule ◊ (subtree nr) % p == rank

Backtrack Algorithm

cutoff_count – nr of nodes at cutoff_depth

cutoff_depth – depth at which subtrees are divided among processes

depth – maximum search depth in the state space tree

moves – records the path to the current node (moves made so far)

p, id – number of processes, process rank

Parallel_Backtrack(node, level)

if (level == depth)

if (node is a solution)

Print_Solution(moves)

Else

if (level == cutoff_depth)

cutoff_count ++

if (cutoff_count % p != id)

return

possible_moves = Count_Moves(node) // nr of possible moves from current node

for i = 1 to possible_moves

node = Make_Move(node, i)

moves[level] = i

Parallel_Backtrack(node, level+1)

node = Unmake_Move(node, i)

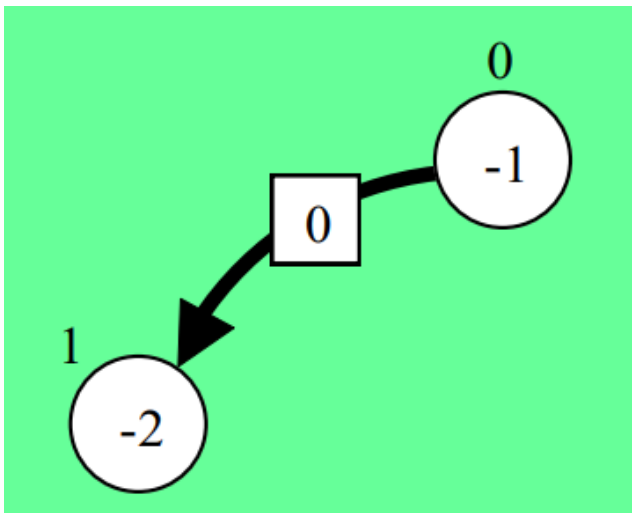
return

Dijkstra et al.'s Algorithm

• Each process has a color and a message count

• Initial color is white • Initial message count is 0

• A process that sends a message turns black and increments its message count



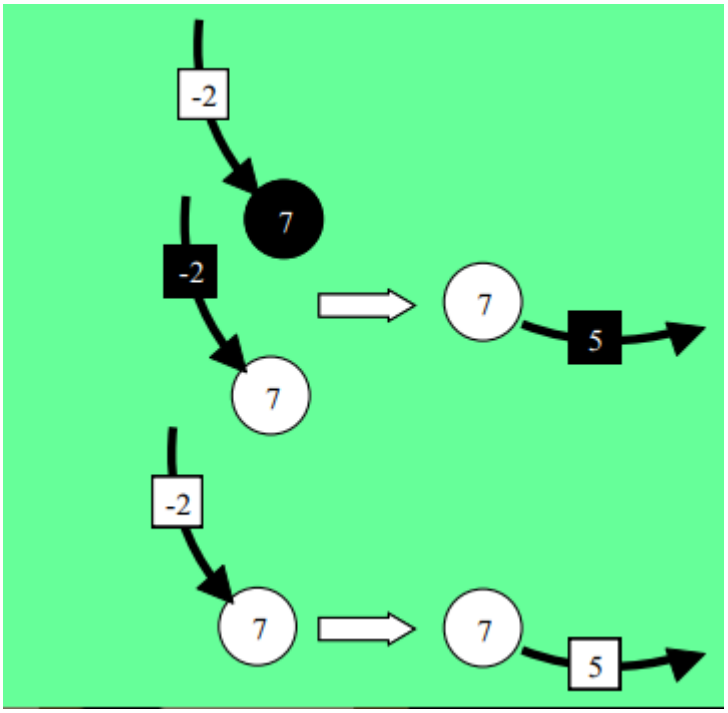
• A process that receives a message turns black and decrements its message count

• If all processes are white and sum of all their message counts are 0, there are no pending messages and we can terminate the processes

• Organize processes into a logical ring

•Process 0 passes a token around the ring

• Token also has a color (initially white) and count (initially 0)



• A process receives the token

• If process is black

• Process changes token color to black

• Process changes its color to white

• Process adds its message count to token's message count

• A process sends the token to its successor in the logical ring

• Process 0 receives the token

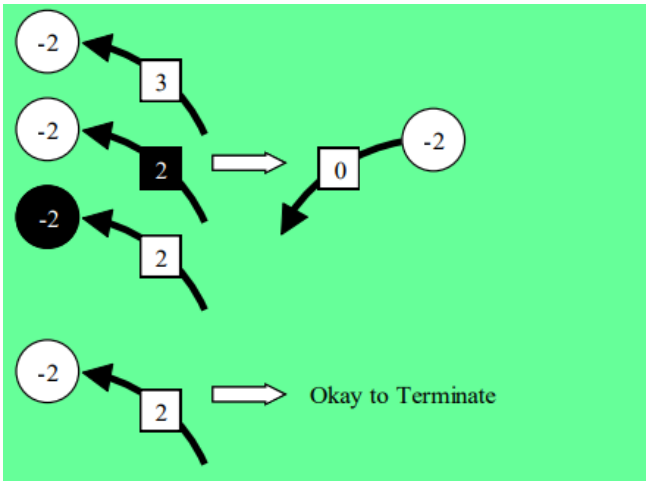
• Safe to terminate processes if

• Token is white

• Process 0 is white

• Token count + process 0 message count = 0

• Otherwise, process 0 must probe ring of processes again



4- Branch and Bound

• Variant of backtrack search

• OR tree : Contains only OR nodes

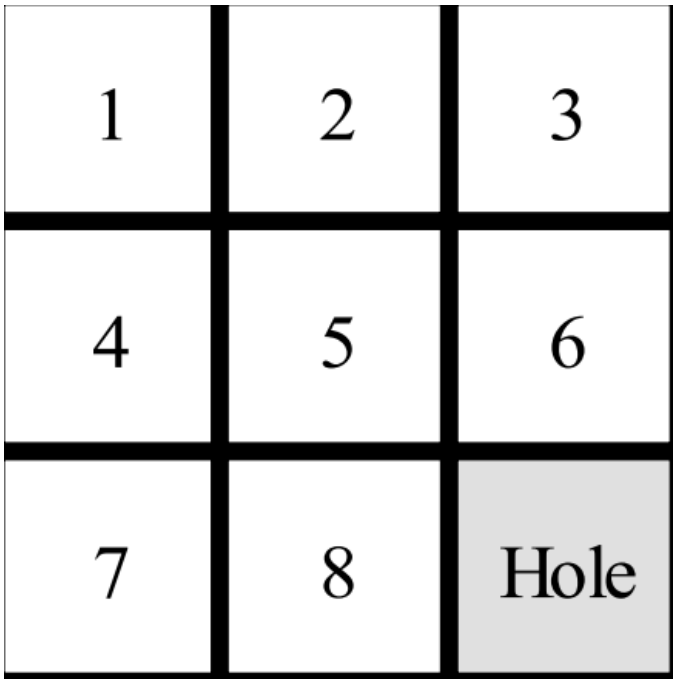
• Takes advantage of information about optimality of partial solutions to avoid considering solutions that cannot be optimal

Application: (8-Puzzle)

8-Puzzle is an OR tree, so we will use Backtrack search and branch and bound

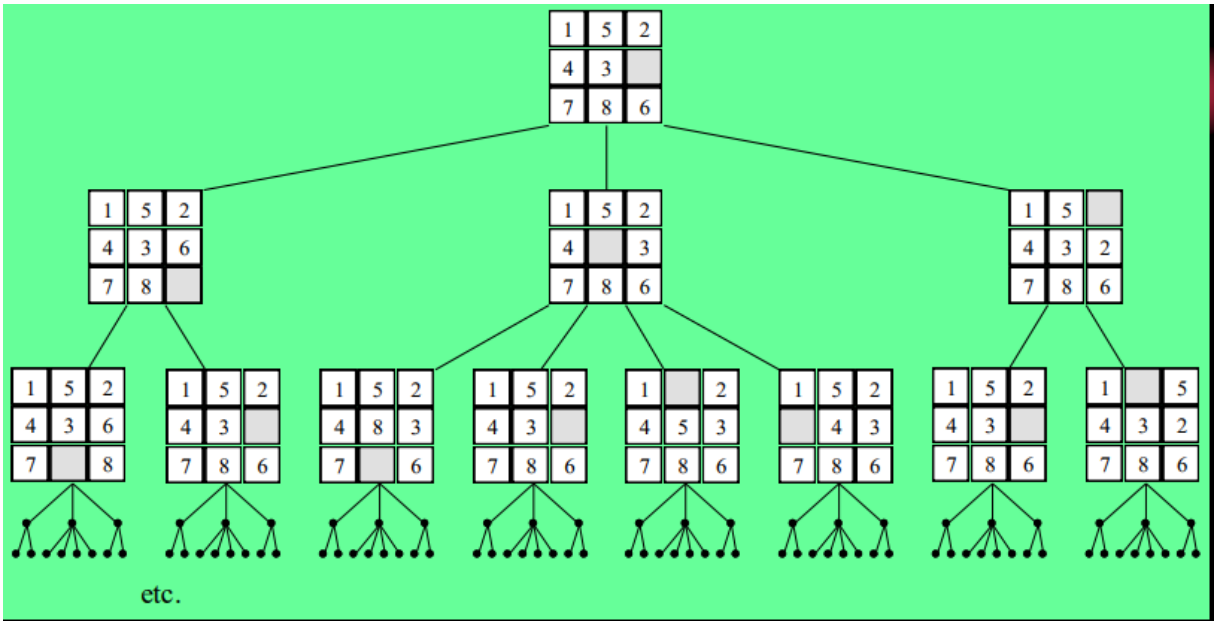
This is the solution state.

Tiles slide up, down, or sideways into hole.



State Space Tree :

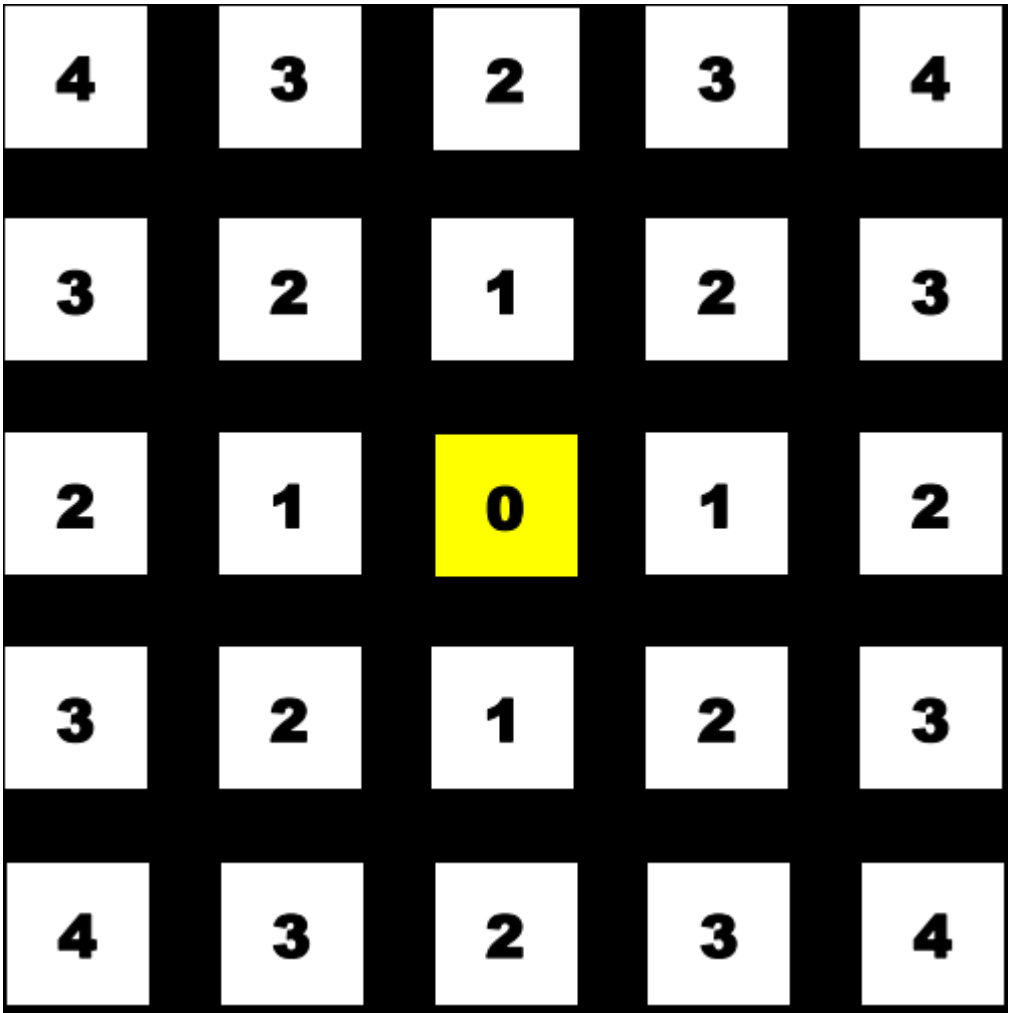
These are all possible moves :



To solve this Puzzle problem, we should impement the “Branch and Bound” Methodology:

1- pursuing breadth-first search of state space tree
2- examine as few nodes as possible
3- speed search if we associate with each node an estimate of minimum number of tile moves needed to solve the puzzle, given moves made so far, by the “Manhattan (or City Block) Distance”

Manhattan Distance from the



yellow intersection

4- A Lower Bound Function:

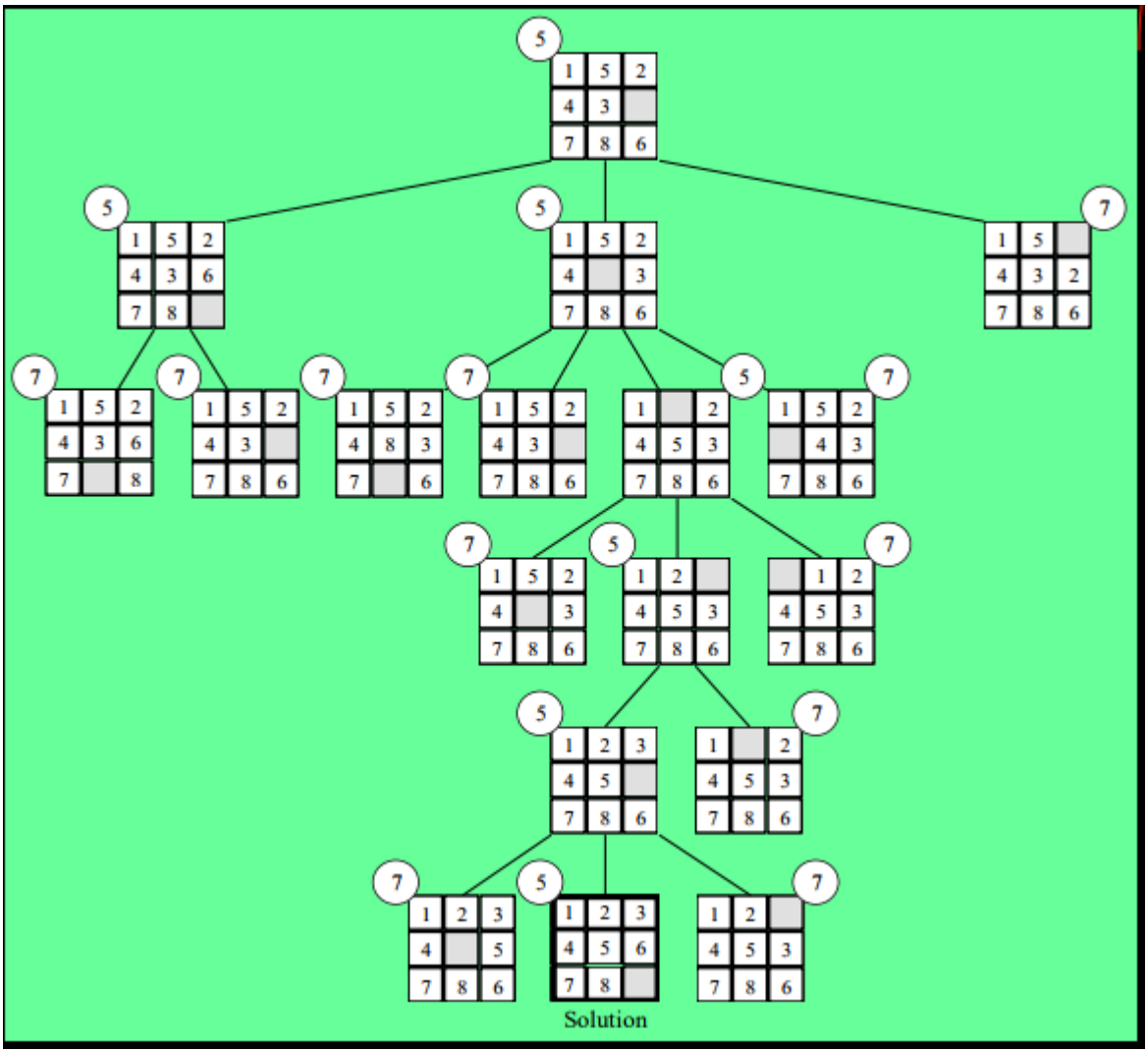
• a lower bound on number of moves needed to solve puzzle is sum of Manhattan distance of each tile's current position from its correct position

• Depth of node in state space tree indicates number of moves made so far

• Adding two values gives lower bound on number of moves needed for any solution, given moves made so far

• We always search from node having smallest value of this function (best-first search)

➔Best-first Search of 8-puzzle :



Sequential Algorithm :

//initial - initial problem

//q - priority queue

//u, v - nodes of the search tree

Initialize (q)

Insert (q, initial)

Repeat

u ← Delete_Min (q)

if u is a solution then

Print_Solution (u)

Halt

Else

For i ← 1 to possible_constraints (u) do

Add constraint i to u, creating v

Insert (q, v)

Time and Memory Complexity :

• In worst case, lower bound function causes function to perform breadth-first search

• Suppose average branching factor in state space tree is b

• Searching a tree of depth k requires examining

$1+ b+ b^2+ ...+ b^k = ((b^{k+1} - b) / (b-1)) + 1 = \Theta (b^k)$

nodes in the worst case (exponential time)

• Worst-case time complexity is $\Theta (b^k)$

• On average, b nodes inserted into priority queue every time a node is deleted

• Worst-case space complexity is $\Theta (b^k)$

• Memory limitations often put an upper bound on the size of the problem that can be solved

Parallel Branch and Bound

• We will develop a parallel algorithm suitable for implementation on a multicomputer or distributed multiprocessor

• Conflicting goals :

- Want to maximize ratio of local to non-local memory references

- Want to ensure processors searching worthwhile portions of state space tree

1- Single Priority Queue

• Maintaining a single priority queue not a good idea

• Communication overhead too great

• Accessing queue is a performance bottleneck

• Does not allow problem size to scale with number of processors

2- Multiple Priority Queues

• Each process maintains separate priority queue of unexamined subproblems

• Each process retrieves subproblem with smallest lower bound to continue search

• Occasionally processes send unexamined subproblems to other processes

Start-up Mode

• Process 0 contains original problem in its priority queue

• Other processes have no work

• After process 0 distributes an unexamined subproblem, 2 processes have work

• A logarithmic number of distribution steps are sufficient to get all processes engaged

Efficiency

• Conditions for solution to be found and guaranteed optimal

• At least one solution node must be found

• All nodes in state space tree with smaller lower bounds must be explored

• Execution time dictated by which of these events occurs last

• This depends on number of processes, shape of state space tree, communication pattern

• Sequential algorithm searches minimum number of nodes (never explores nodes with lower bounds greater than cost of optimal solution)

• Parallel algorithm may examine unnecessary nodes because each process searching locally best nodes • Exchanging subproblems

• promotes distribute of subproblems with good lower bounds, reducing amount of wasted work

• increases communication overhead

Modifications to DTP Algorithm

• Process turns black if it manipulates an unexamined subproblem with lower bound less than cost of best solution found so far

• Add additional fields to termination token

• Cost of best solution found so far

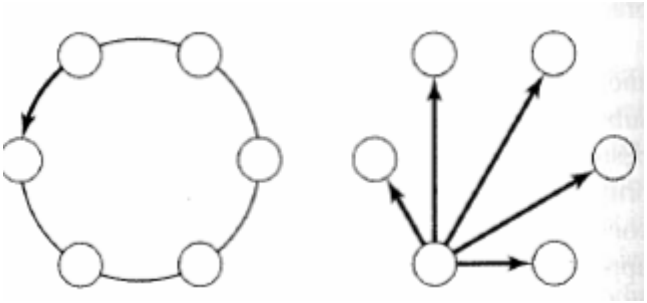
• Solution itself (i.e., moves made to reach solution)

Actions When Process Gets Token

• Updates token's color, count fields

• If locally found solution better than one carried by token, updates token

• If lower bound of first unexamined problem in priority queue ≥ best solution found so far, empties priority queue



References:

- http://parallelcomp.github.io/CombinatorialSearch.pdf

Assignment :

Part I : Sisihlia YUNIYARTI

Part II : Sanae SEBAI