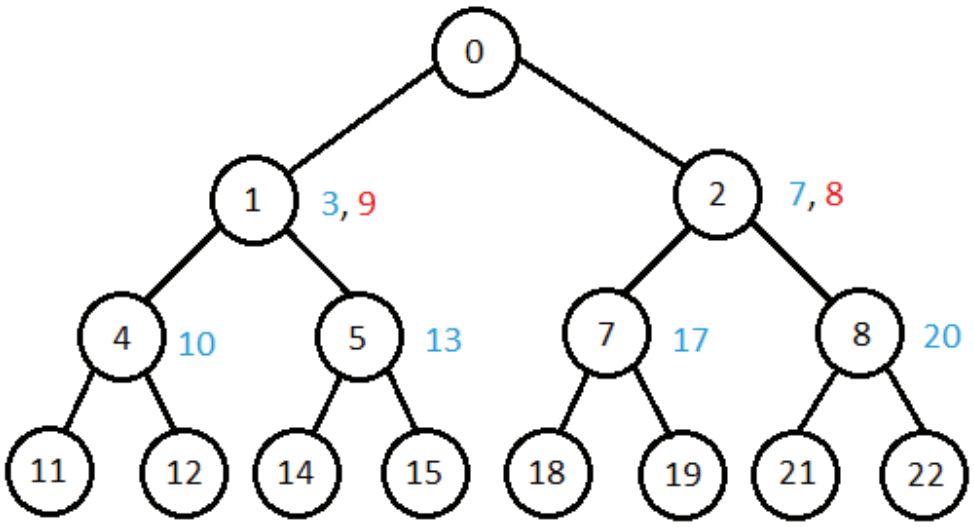


Comparison of Various Algorithms Complexities

Let us see the performance of algorithms based on various criteria:

Criterion	Breadth First	Depth First	Bidirectional	Uniform Cost	Iterative Deepening
Time	b^d	b^m	$b^{d/2}$	b^d	b^d
Space	b^d	b^m	$b^{d/2}$	b^d	b^d
Optimality	Y	N	Y	Y	Y
Completeness	Y	N	Y	Y	Y

Informed (Heuristic) Search Strategies

To solve large problems with large number of possible states, problem-specific knowledge needs to be added to increase the efficiency of search algorithms.

Heuristic Evaluation Functions

They calculate the cost of optimal path between two states. A heuristic function for sliding-tiles games is computed by counting number of moves that each tile makes from its goal state and adding these number of moves for all tiles.

Pure Heuristic Search

It expands nodes in the order of their heuristic values. It creates two lists, a closed list for the already expanded nodes and an open list for the created but unexpanded nodes.

In each iteration, a node with a minimum heuristic value is expanded, all its child nodes are created and placed in the closed list. Then, the heuristic function is applied

to the child nodes and they are placed in the open list according to their heuristic value. The shorter paths are saved and the longer ones are disposed.

A* Search

It is best-known form of Best First search. It avoids expanding paths that are already expensive, but expands most promising paths first.

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

- g(n) the cost (so far) to reach the node
- h(n) estimated cost to get from the node to the goal
- f(n) estimated total cost of path through n to goal. It is implemented using priority queue by increasing f(n).

Greedy Best First Search

It expands the node that is estimated to be closest to goal. It expands nodes based on f(n) = h(n). It is implemented using priority queue.

Disadvantage: It can get stuck in loops. It is not optimal.

Local Search Algorithms

They start from a prospective solution and then move to a neighboring solution. They can return a valid solution even if it is interrupted at any time before they end.

Hill-Climbing Search

It is an iterative algorithm that starts with an arbitrary solution to a problem and attempts to find a better solution by changing a single element of the solution incrementally. If the change produces a better solution, an incremental change is taken as a new solution. This process is repeated until there are no further improvements.

function Hill-Climbing (problem), returns a state that is a local maximum.

inputs: problem, a problem

local variables: *current*, a node

neighbor, a node

current ←Make_Node(Initial-State[problem])

loop

do *neighbor* ← a highest_valued successor of *current*

if Value[*neighbor*] ≤ Value[*current*] then

return State[*current*]

current ← *neighbor*

end

Disadvantage: This algorithm is neither complete, nor optimal.

Local Beam Search

In this algorithm, it holds *k* number of states at any given time. At the start, these states are generated randomly. The successors of these k states are computed with the help of objective function. If any of these successors is the maximum value of the objective function, then the algorithm stops.

Otherwise the (initial *k* states and *k* number of successors of the states = 2*k*) states are placed in a pool. The pool is then sorted numerically. The highest *k* states are selected as new initial states. This process continues until a maximum value is reached.

function BeamSearch(*problem, k*), returns a solution state.

start with *k* randomly generated states

loop

generate all successors of all *k* states

if any of the states = solution, then return the state

else select the *k* best successors

end

Simulated Annealing

Annealing is the process of heating and cooling a metal to change its internal structure for modifying its physical properties. When the metal cools, its new structure is seized, and the metal retains its newly obtained properties. In simulated annealing process, the temperature is kept variable.

We initially set the temperature high and then allow it to 'cool' slowly as the algorithm proceeds. When the temperature is high, the algorithm is allowed to accept worse solutions with high frequency.

Start

5. Initialize k = 0; L = integer number of variables;
6. From i -> j, search the performance difference Δ.
7. If Δ <= 0 then accept else if exp(-Δ/T(k)) > random(0,1) then accept;
8. Repeat steps 1 and 2 for L(k) steps.
9. k = k + 1;

Repeat steps 1 through 4 till the criteria is met.

End

Travelling Salesman Problem

In this algorithm, the objective is to find a low-cost tour that starts from a city, visits all cities en-route exactly once and ends at the same starting city.

Start

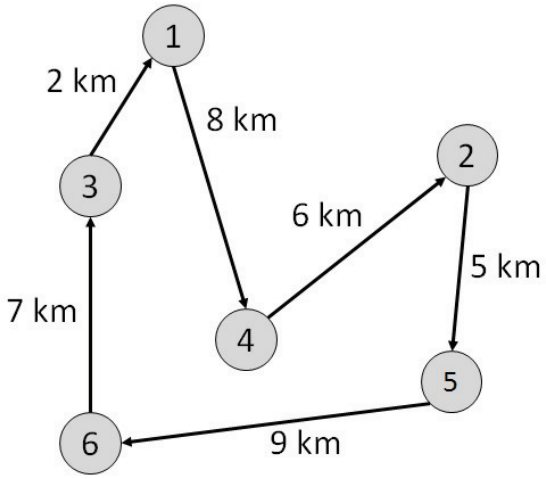
Find out all (*n* -1)! Possible solutions, where *n* is the total number of cities.

Determine the minimum cost by finding out the cost of each of these (*n* -1)!

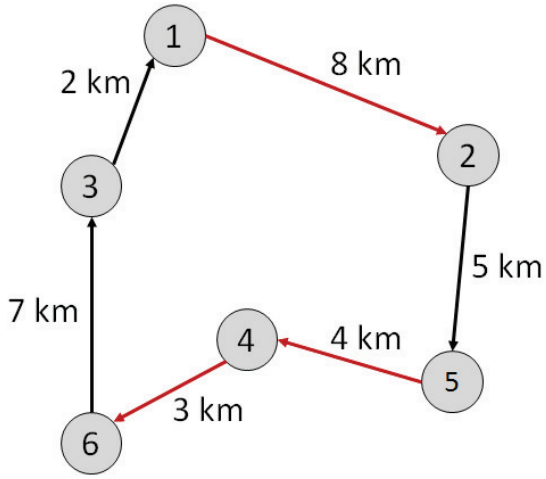
solutions.

Finally, keep the one with the minimum cost.

end



Total Distance = 37km



Total Distance = 31km