SOLVING NP-HARD COMBINATORIAL OPTIMIZATION PROBLEMS

Partha P Chakrabarti

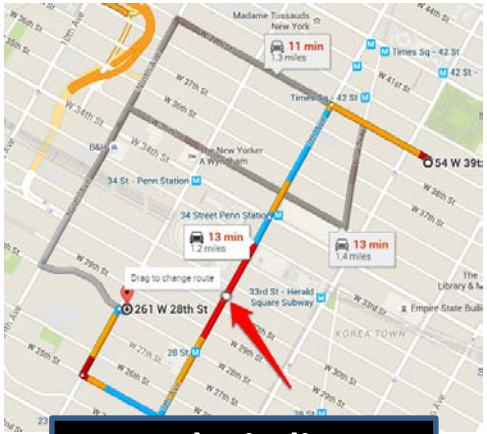
Indian Institute of Technology Kharagpur

Overview of Algorithm Design

1. Initial Solution
   a. Recursive Definition – A set of Solutions
   b. Inductive Proof of Correctness
   c. Analysis Using Recurrence Relations
2. Exploration of Possibilities
   a. Decomposition or Unfolding of the Recursion Tree
   b. Examination of Structures formed
   c. Re-composition Properties
3. Choice of Solution & Complexity Analysis
   a. Balancing the Split, Choosing Paths
   b. Identical Sub-problems
4. Data Structures & Complexity Analysis
   a. Remembering Past Computation for Future
   b. Space Complexity
5. Final Algorithm & Complexity Analysis
   a. Traversal of the Recursion Tree
   b. Pruning
6. Implementation
   a. Available Memory, Time, Quality of Solution, etc

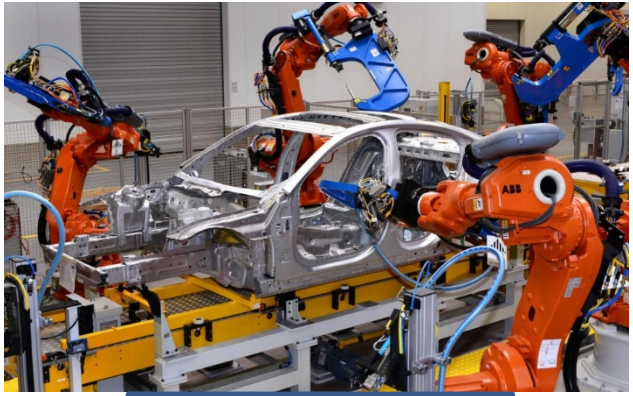
1. Core Methods
   a. Divide and Conquer
   b. Greedy Algorithms
   c. Dynamic Programming
   d. Branch-and-Bound
   e. Analysis using Recurrences
   f. Advanced Data Structuring
2. Important Problems to be addressed
   a. Sorting and Searching
   b. Strings and Patterns
   c. Trees and Graphs
   d. Combinatorial Optimization
3. Complexity & Advanced Topics
   a. Time and Space Complexity
   b. Lower Bounds
   c. Polynomial Time, NP-Hard
   d. Parallelizability, Randomization

COMPLEX PROBLEMS

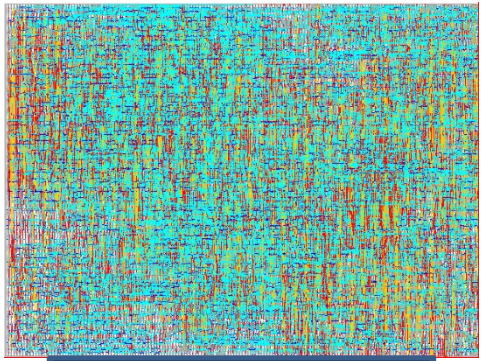


Path Finding

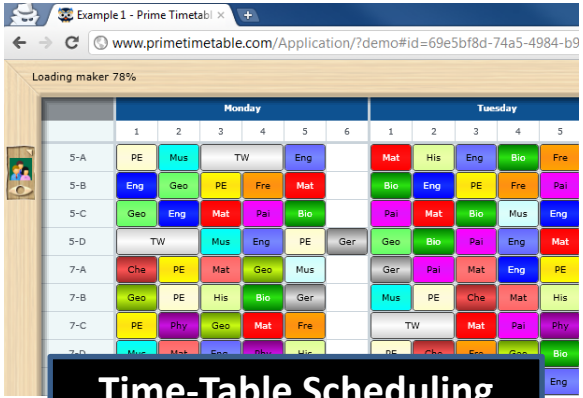
Chess Playing



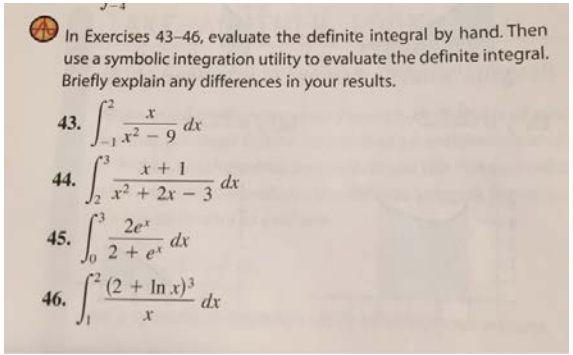
Robot Assembly



VLSI Chip Design



Time-Table Scheduling



Symbolic Integration

Revisiting the Coins Problem

Given a set C of n coins having denomination values $\{c_1, c_2, \ldots, c_n\}$ and a desired final value of V, find the minimum number of coins to be chosen from C to get an exact value of V from the sum of denominations of the chosen subset

example: $C = \{8, 6, 5, 2, 1\}$, $V = 11$

$S_1 = \{8, 3, 1\}$, $S_2 = \{6, 5\}$ ← minimum

Coins (S, T, x, z, n)

S: set of coins selected till now

T: remaining set of coins from which we can select

x: value of set S

z: remaining value desired to be chosen from T

n: the number of coins selected

Coins (NULL, C, 0, V, 0)

→ Base condition

→ Recursive condition

Revisiting the Coins Problem: Non-Deterministic Choice

$\langle P, d \rangle$ = coins (S, T, x, z, n)

{ Let $S = \{s_1, s_2, \ldots, s_n\}$

$T = \{t_1, t_2, \ldots, t_m\}$

BASE CONDITIONS

if $(z = 0)$ return $(\langle S, n \rangle)$

if $(z < 0)$ return $(\langle NULL, \infty \rangle)$

if $(T = NULL)$ return $(\langle NULL, \infty \rangle)$

$P_{min} = NULL$

$min = \infty$

RECURSIVE CONDITION

for $(i = 1$ to $m)$ do

{ $W = S + \{t_i\}$

$U = T - \{t_i\}$

$\langle P', d' \rangle$ = coins $(W, U, x + t_i, z - t_i, n + 1)$

if $(d' < min)$

{ $min = d'$

$P_{min} = P'$

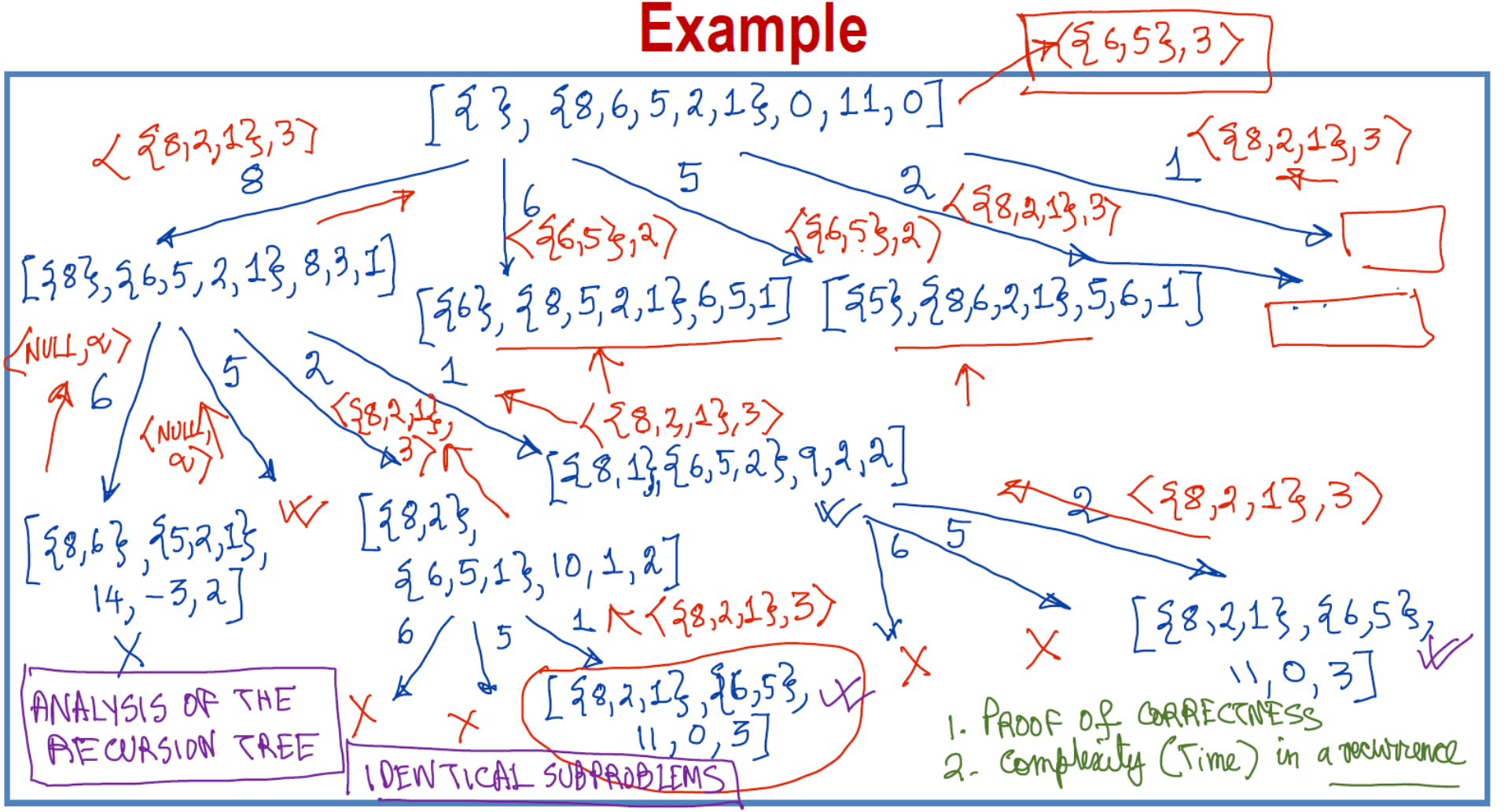
}

}

return $(\langle P_{min}, min \rangle)$

}

Example


[{}, {8,6,5,2,1}, 0, 11, 0]
⟨{6,5}, 3⟩
⟨{8,2,1}, 3⟩
[{8}, {6,5,2,1}, 8, 3, 1]
[{6}, {8,5,2,1}, 6, 5, 1]
[{5}, {8,6,2,1}, 5, 6, 1]
[{8,1}, {6,5,2}, 9, 2, 2]
[{8,6}, {5,2,1}, 14, -3, 2]
[{8,2}, {6,5,1}, 10, 1, 2]
[{8,2,1}, {6,5}, 11, 0, 3]
[{8,2,1}, {6,5}, 11, 0, 3]
⟨NULL, ∞⟩
ANALYSIS OF THE RECURSION TREE
IDENTICAL SUBPROBLEMS
1. PROOF of CORRECTNESS
2. Complexity (Time) in a recurrence

Structure of Recursive Definition

f(X)

{ 1. Base Condition B(X)

if B(X) return (J(X))

2. Decomposition D(X)

$\langle X_1, X_2, \ldots, X_K \rangle = D(X)$

3. Recursive Call

for each i, $Y_i = f(X_i)$

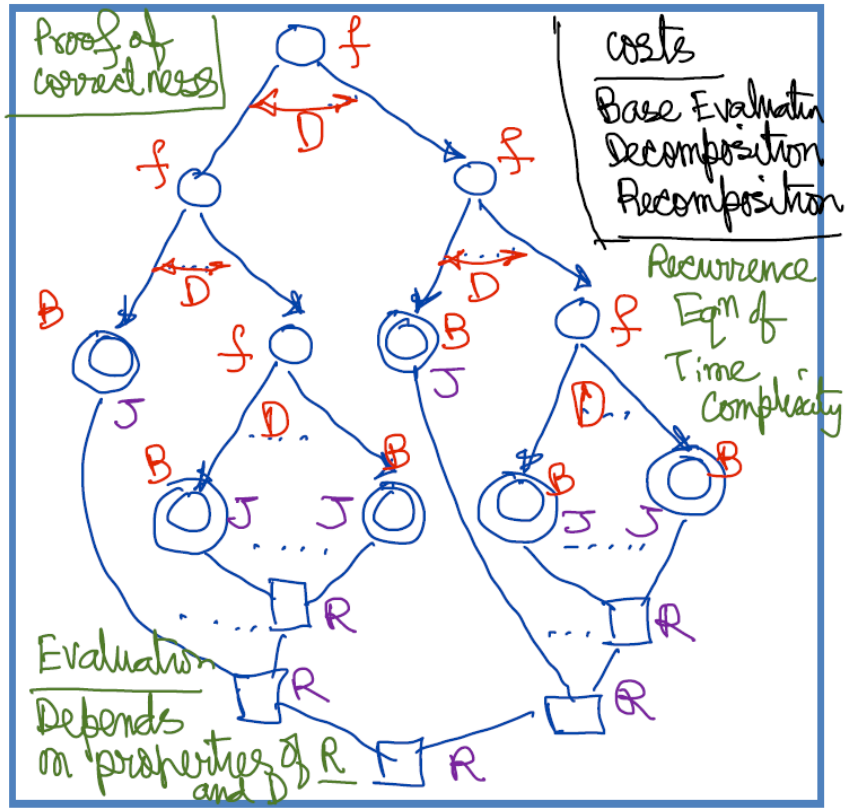
4. Recomposition R(Y)

$J = R(Y_1, Y_2, \ldots, Y_K)$

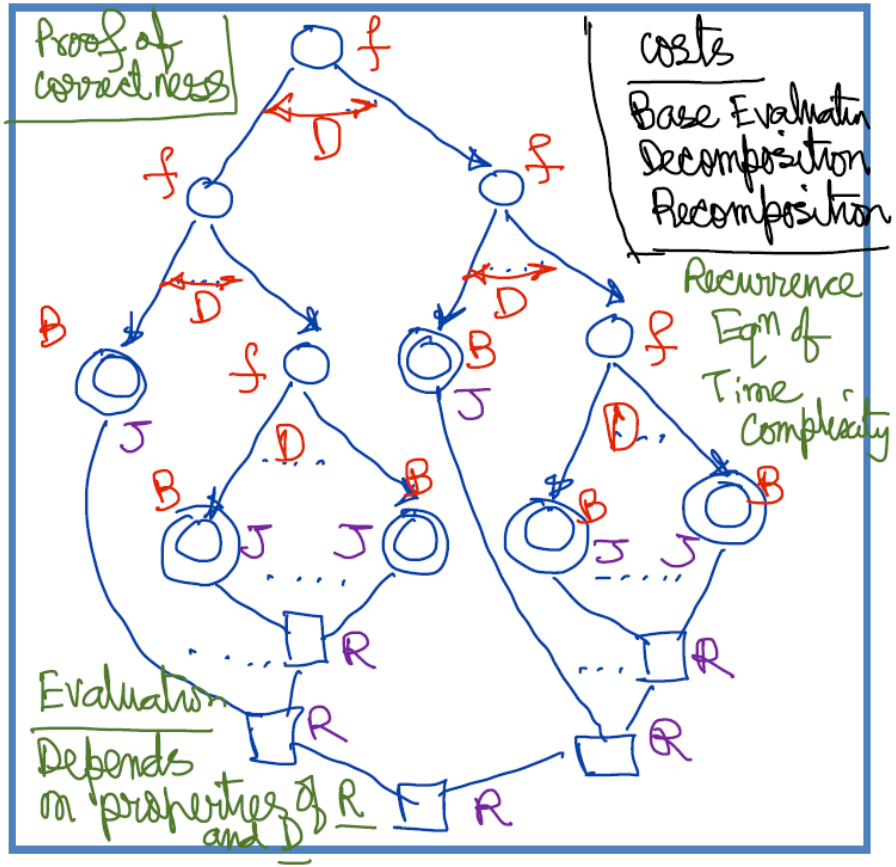
5. Returning the Result

Return (J)

}



Evaluation Methods



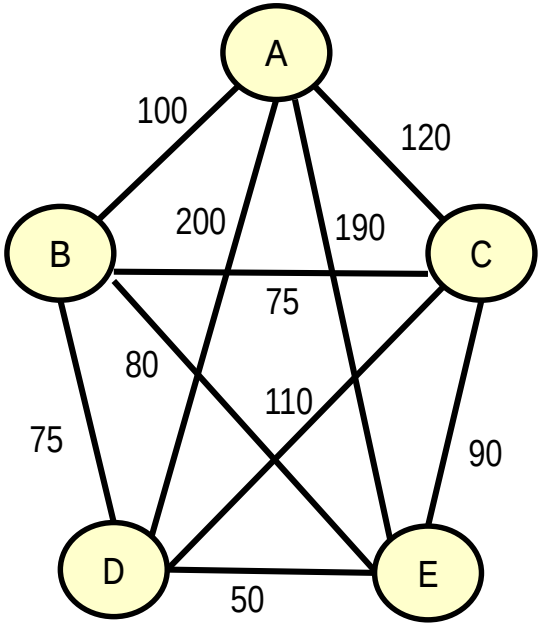
Evaluation by Traversal or Search of the Recursion Tree (also known as the State Space):

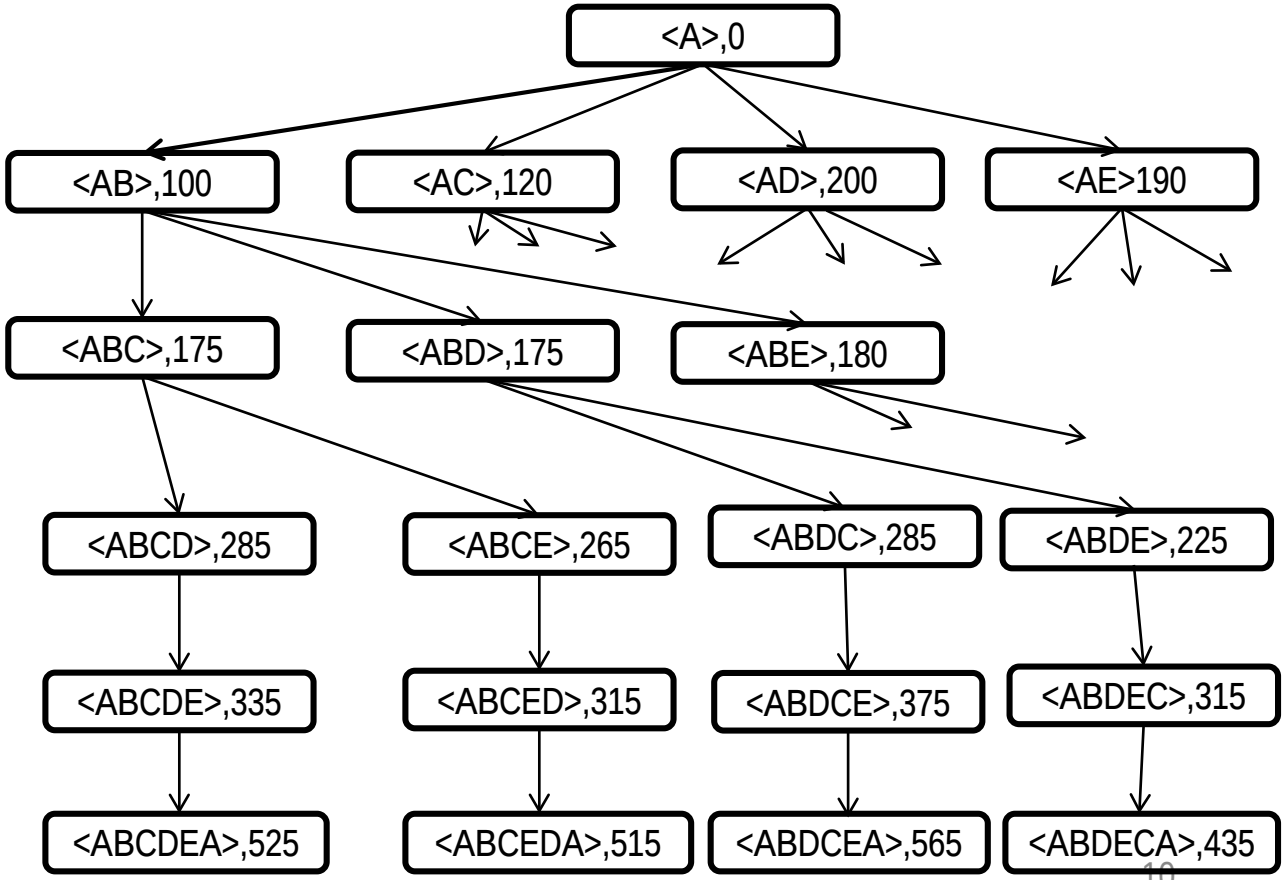
- BASIC Algorithms: Depth-First (DFS), Breadth-first (BFS), Iterative Deepening (IDS)
- COST-based Algorithms: Depth-First Branch-and-Bound, Best First Search, Best-First Iterative Deepening
- Widely Used Algorithms: A* and IDA* (Or Graphs), AO* (And/Or Graphs), Alpha-beta Pruning (Game-Trees)

Recursive Definitions and State Spaces

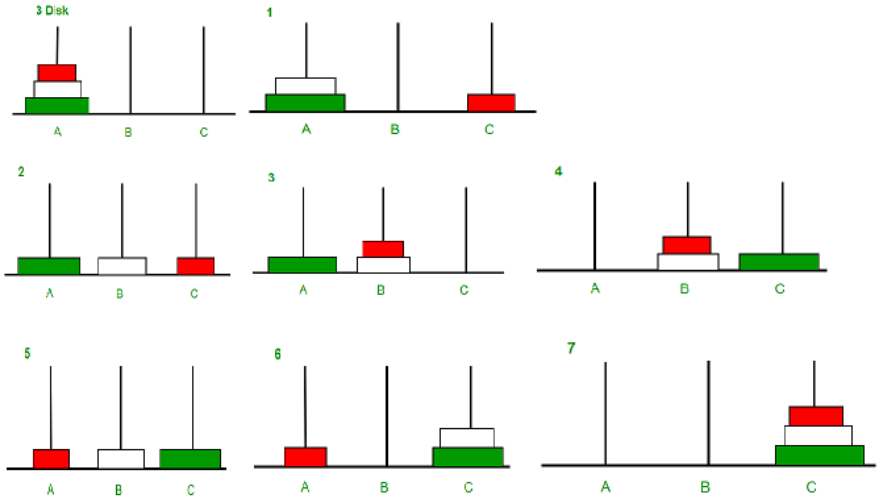
- STATE or CONFIGURATION:
  - A set of variables which define a state or configuration
  - Domains for every variable and constraints among variables to define a valid configuration
- STATE TRANSFORMATION RULES or MOVES:
  - A set of RULES which define which are the valid set of NEXT STATE of a given State
  - It also indicates who can make these Moves (OR Nodes, AND nodes, etc)
- STATE SPACE or IMPLICIT GRAPH
  - The Complete Graph produced out of the State Transformation Rules.
  - Typically too large to store. Could be Infinite.
- INITIAL or START STATE(s), GOAL STATE(s)
- SOLUTION(s), COSTS
  - Depending on the problem formulation, it can be a PATH from Start to Goal or a Sub-graph of And-ed Nodes
- SEARCH ALGORITHMS
  - Intelligently explore the Implicit Graph or State Space by examining only a small sub-set to find the solution
  - To use Domain Knowledge or HEURISTICS to try and reach Goals faster

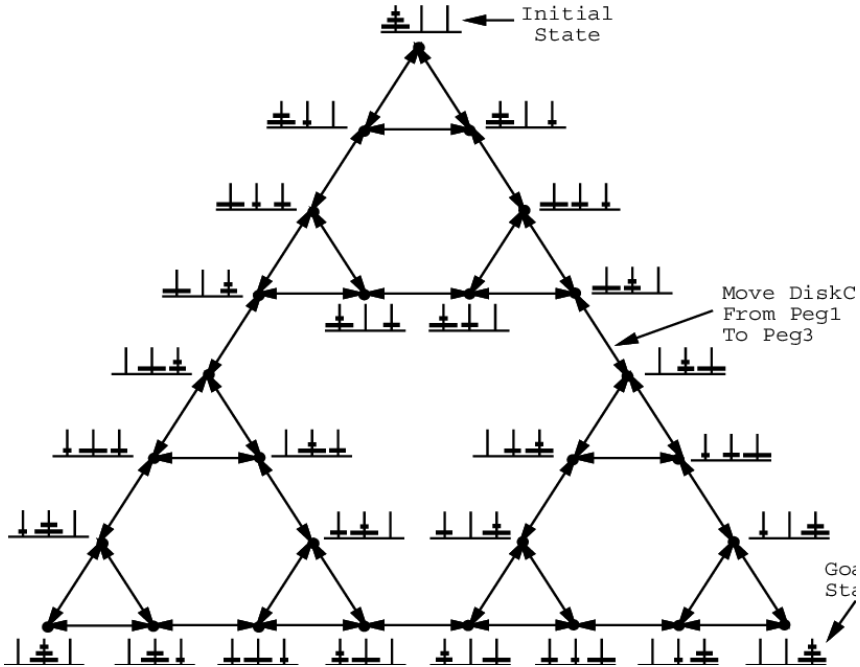
OR-Graph: TRAVELLING SALESPERSON PROBLEM



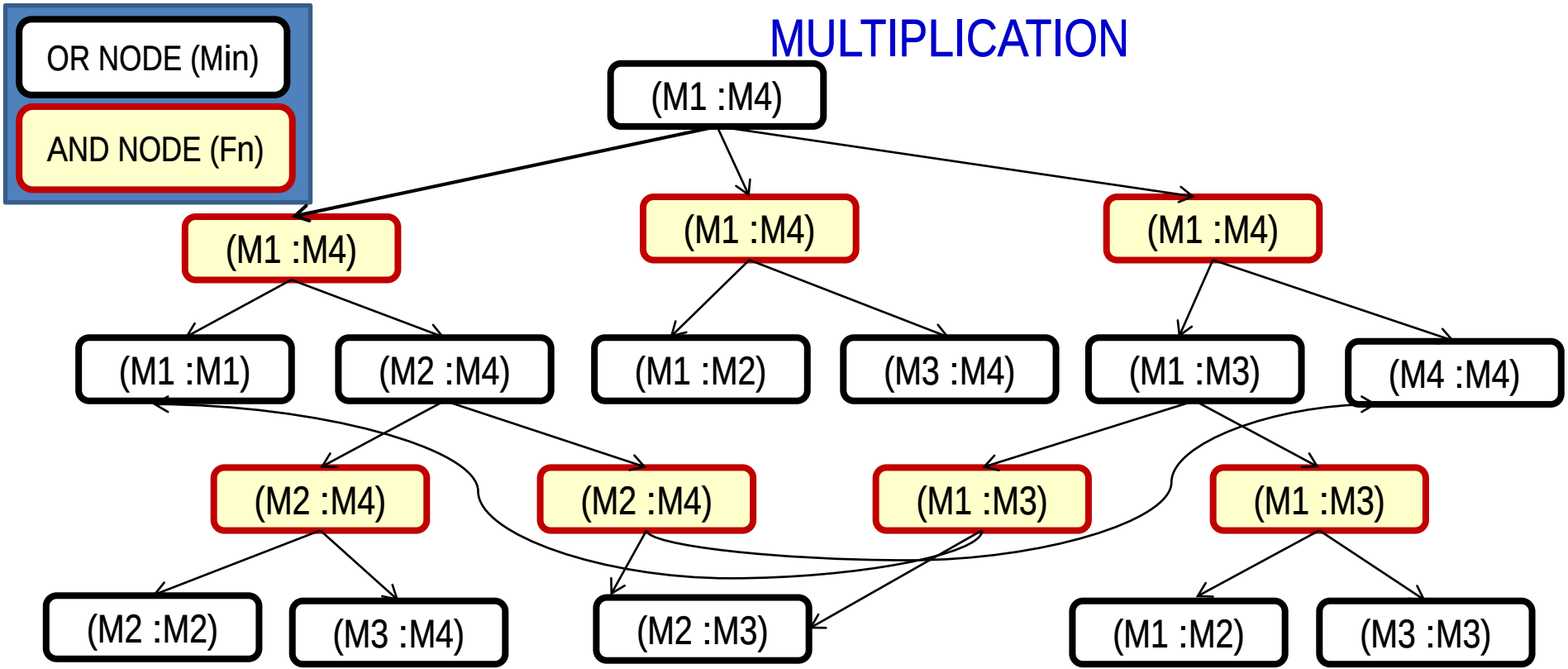


3 DISK, 3 PEG TOWER of HANOI STATE SPACE



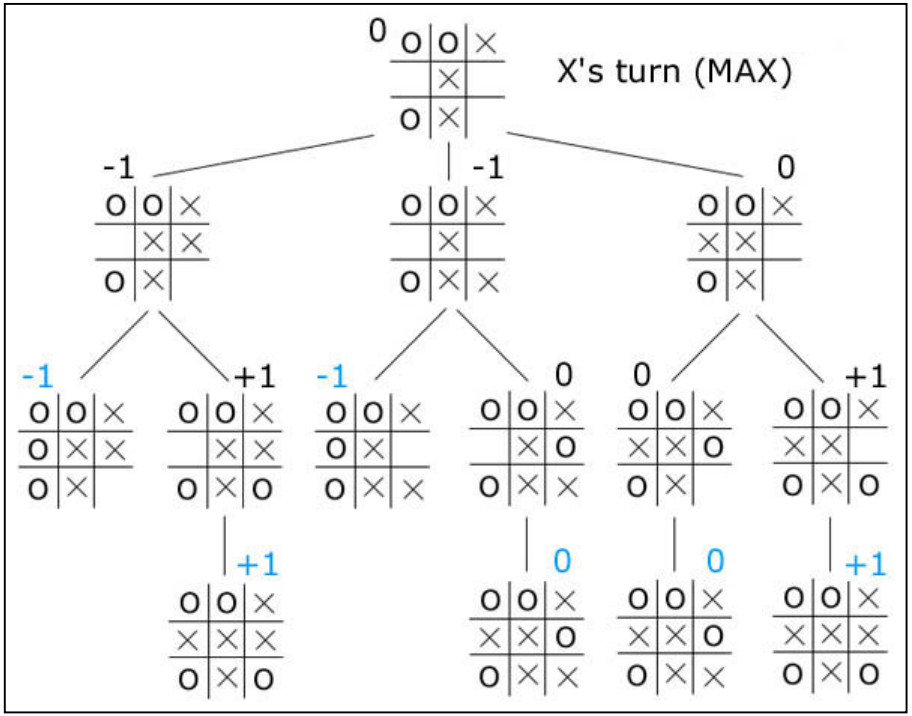


COMPOSITIONAL AND/OR GRAPHS: MATRIX CHAIN MULTIPLICATION



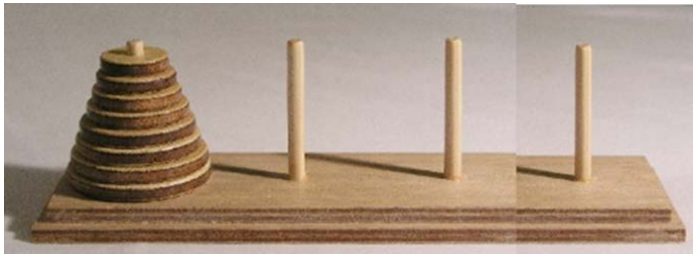
(M1 X (M2 X (M3 X M4))) = ((M1 x M2) X (M3 X M4)) = (((M1 x M2) X M3) X M4) = (M1 X (M2 X M3)) X M4)

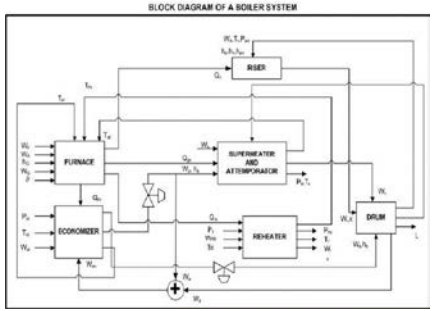
GAME GRAPHS: ADVERSARIAL

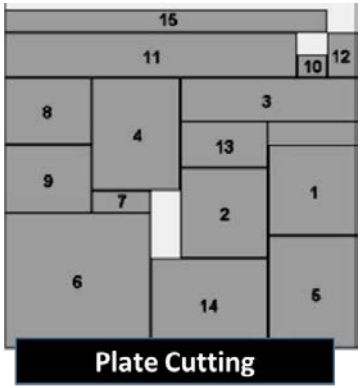


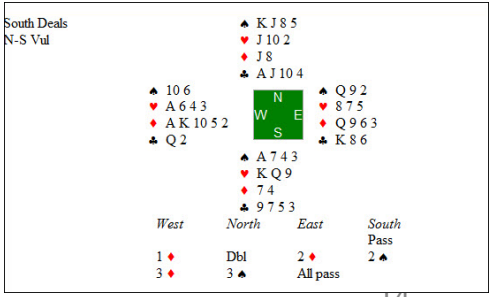
STATES, SPACES, SOLUTIONS, SEARCH

- States
  - Full / Perfect Information and Partial Information States
- State Transformation Rules
  - Deterministic Outcomes
  - Non-Deterministic / Probabilistic Outcomes
- State Spaces As Generalized Games
  - Single Player: OR Graphs
  - Multi-Player: And / Or, Adversarial, Probabilistic Graphs
- Solutions
  - Paths
  - Sub-graphs
  - Expected Outcomes
- Costs
- Sizes
- Domain Knowledge
- Algorithms for Heuristic Search





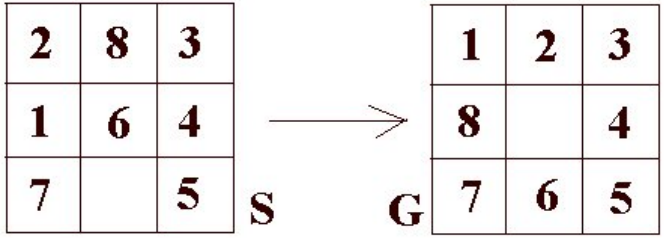




BASIC SEARCH ALGORITHM

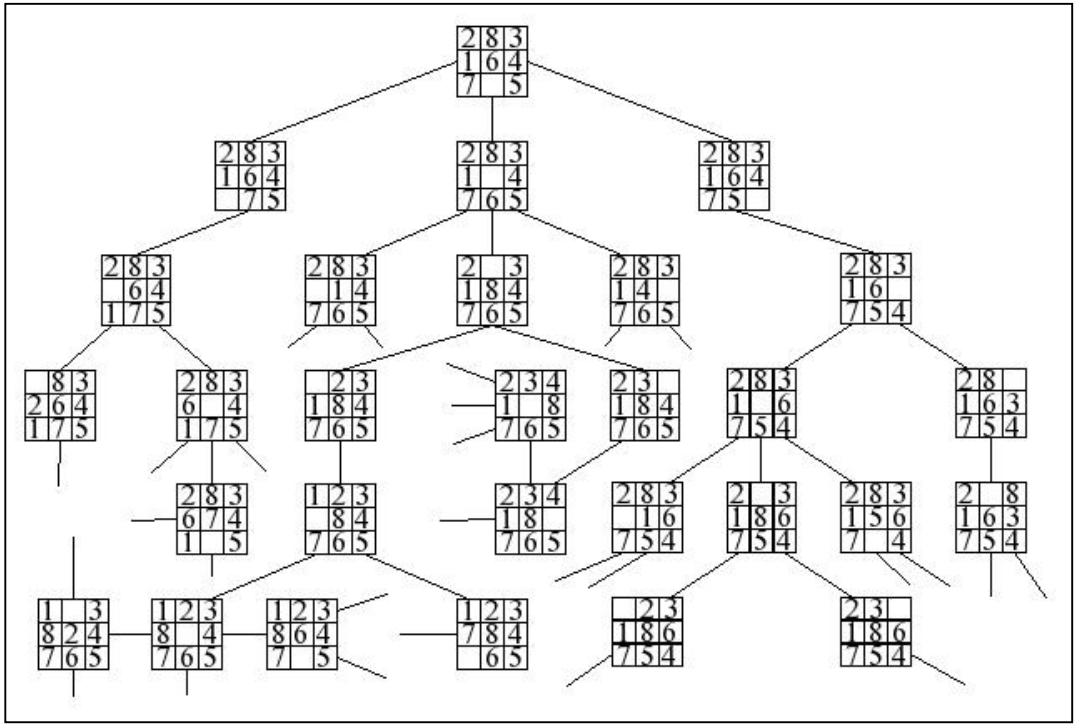
1. [Initialize] Initially the OPEN List contains the Start Node s. CLOSED List is Empty. Set f(s) by applying heuristic evaluation function. Set CB = Initial Best Cost
2. [Select] Select the a Suitable Node n on the OPEN List. If OPEN is empty, Terminate
3. [Goal Test] If n is Goal, then decide on Termination or Continuation with / without Cost Update
4. [Expand]
   a) Generate the successors n_1, n_2, ... n_k, of node n, based on the State Transformation Rules
   b) Put n in LIST CLOSED
   c) For each n_i, not already in OPEN or CLOSED List, put n_i in OPEN List. Compute the f(n_i) of each node and PRUNE if applicable based on CB.
   d) For each n_i already in OPEN or CLOSED decide based on cost of the paths. Update f(n_i) and bring back from CLOSED to OPEN if needed.
5. [Continue] Go to Step 2

EXAMPLE OF HEURISTICS: 8 PUZZLE PROBLEM

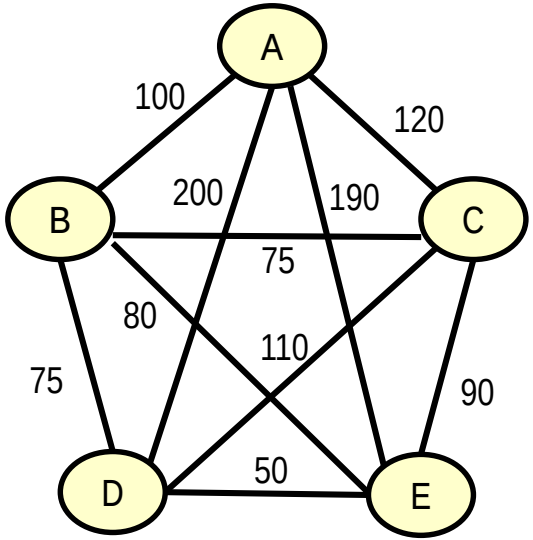


HEURISTIC 1: NUMBER OF MISPLACED TILES

HEURISTIC 2: SUM OF TILE DISTANCES FROM GOAL

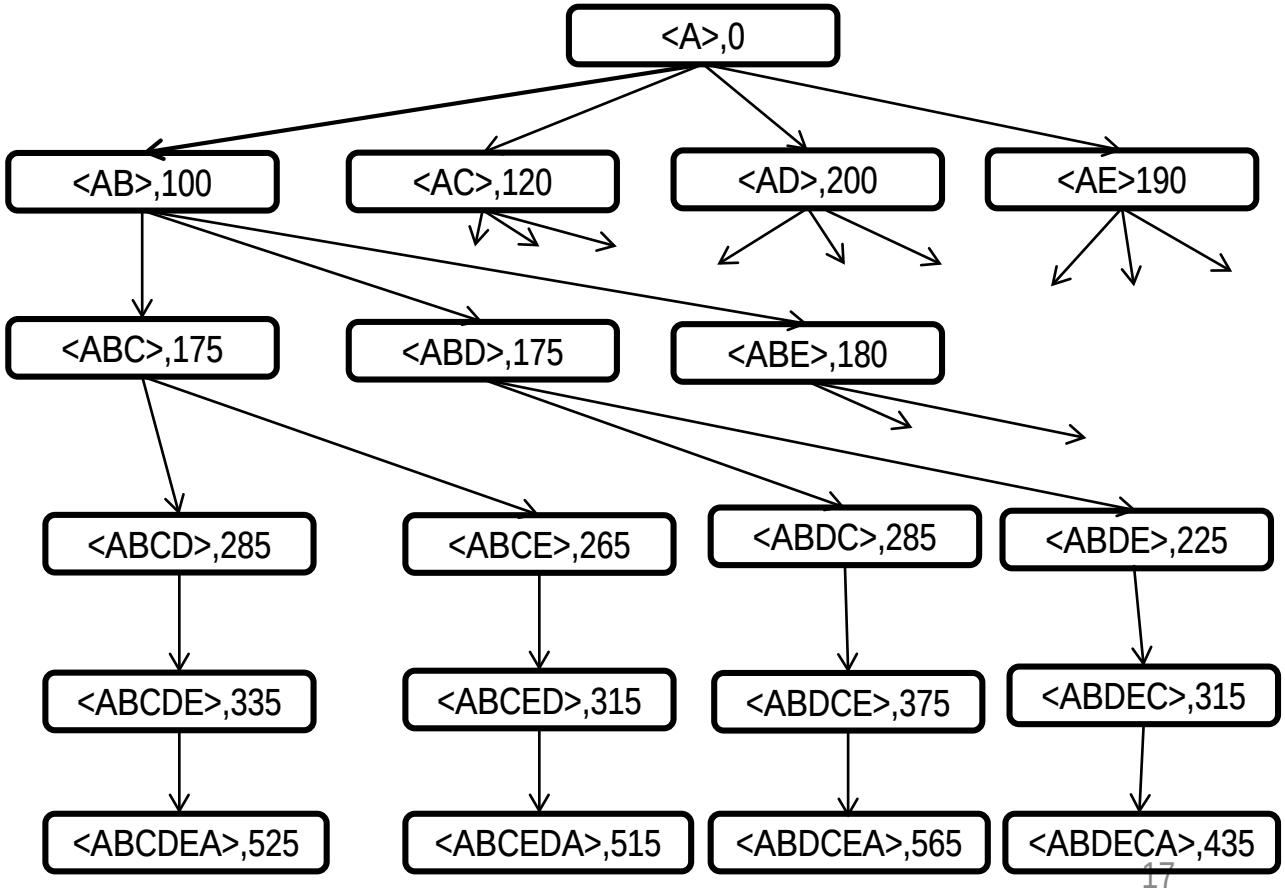


TRAVELLING SALESPERSON PROBLEM

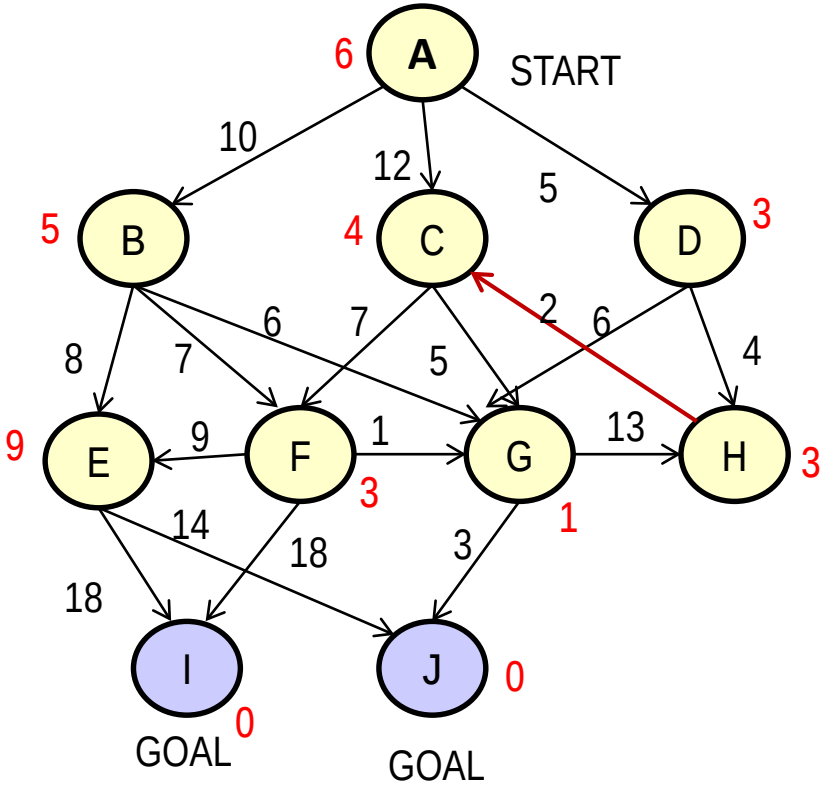


HEURISTIC 1: COST OF SHORTEST PATH FROM CURRENT NODE TO START THROUGH REMAINING NODES ONLY

HEURISTIC 2: COST OF MINIMUM COST SPANNING TREE OF REMAINING NODES

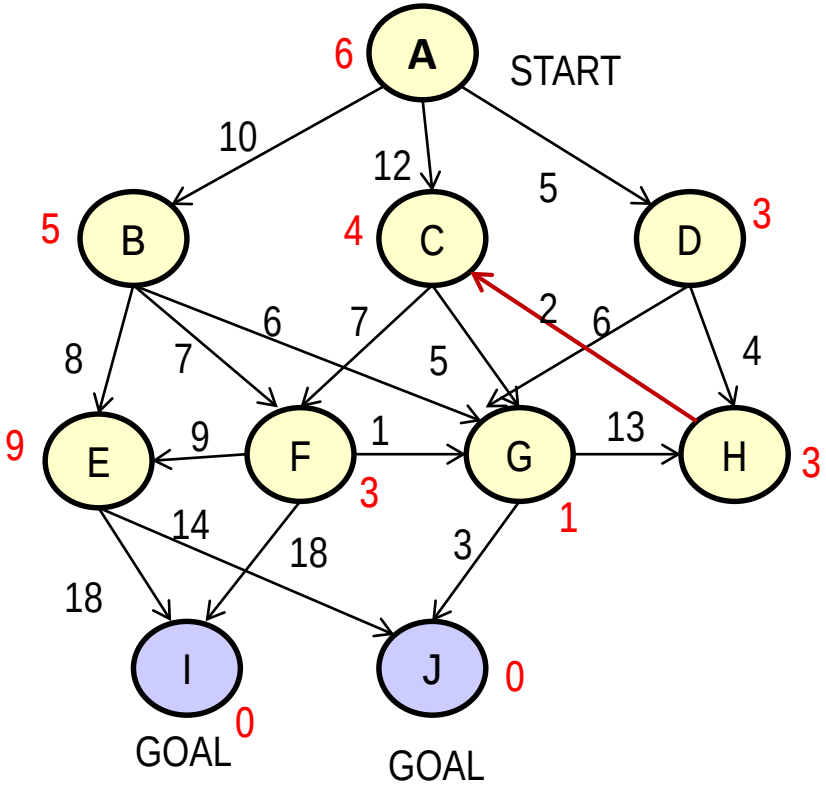


HEURISTIC SEARCH



1. [Initialize] Initially the OPEN List contains the Start Node s. CLOSED List is Empty. Set f(s) by applying heuristic evaluation function. Set CB = Initial Best Cost
2. [Select] Select the a Suitable Node n on the OPEN List. If OPEN is empty, Terminate
3. [Goal Test] If n is Goal, then decide on Termination or Continuation with / without Cost Update
4. [Expand]
   a) Generate the successors n_1, n_2, ... n_k, of node n, based on the State Transformation Rules
   b) Put n in LIST CLOSED
   c) For each n_i, not already in OPEN or CLOSED List, put n_i in OPEN List. Compute the f(n_i) of each node and PRUNE if applicable based on CB.
   d) For each n_i already in OPEN or CLOSED decide based on cost of the paths. Update f(n_i) and bring back from CLOSED to OPEN if needed.
5. [Continue] Go to Step 2

SEARCHING STATE SPACES WITH EDGE COSTS, HEURISTIC ESTIMATES



- HEURISTIC SEARCH ALGORITHMS:
  - DFBB
  - A*: Best First Search,
  - IDA*: Iterative Deepening A*
  - Every edge (n, m) in the graph has a cost c(n,m) > 0.
  - **HEURISTIC Estimates: h(n) ≥ 0 at** every node is the estimated cost of the minimum cost path from node n to goal

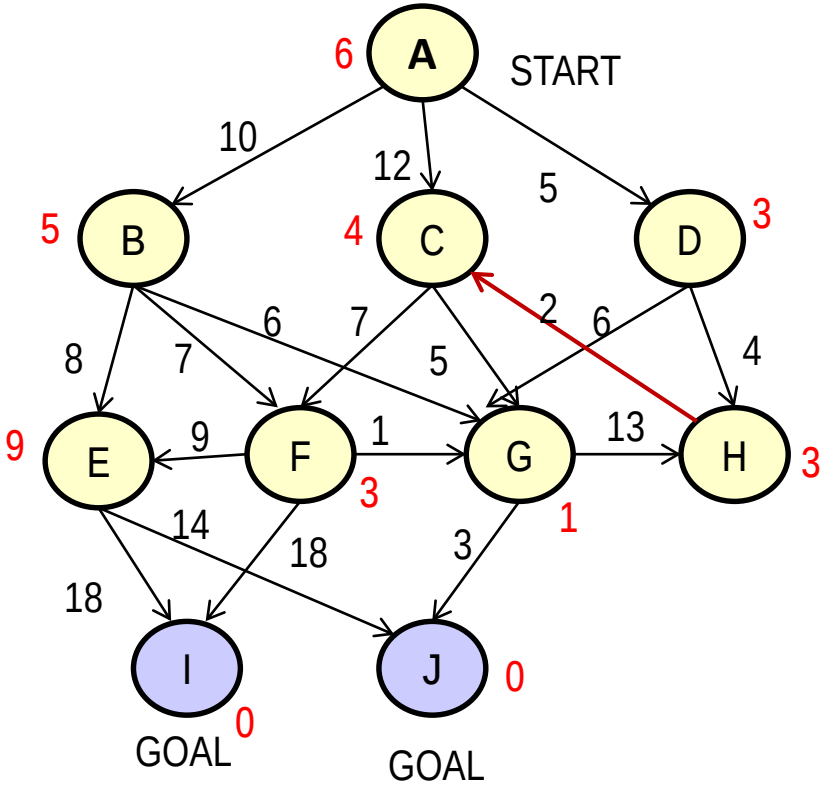
- PROPERTIES
  - SOLUTION GUARANTEES
  - MEMORY REQUIREMENTS

ALGORITHM A* (BEST FIRST SEARCH IN OR GRAPHS)

Each Node n in the algorithm has a cost g(n) and a heuristic estimate h(n), f(n) = g(n) + h(n).
Assume all c(n,m) > 0

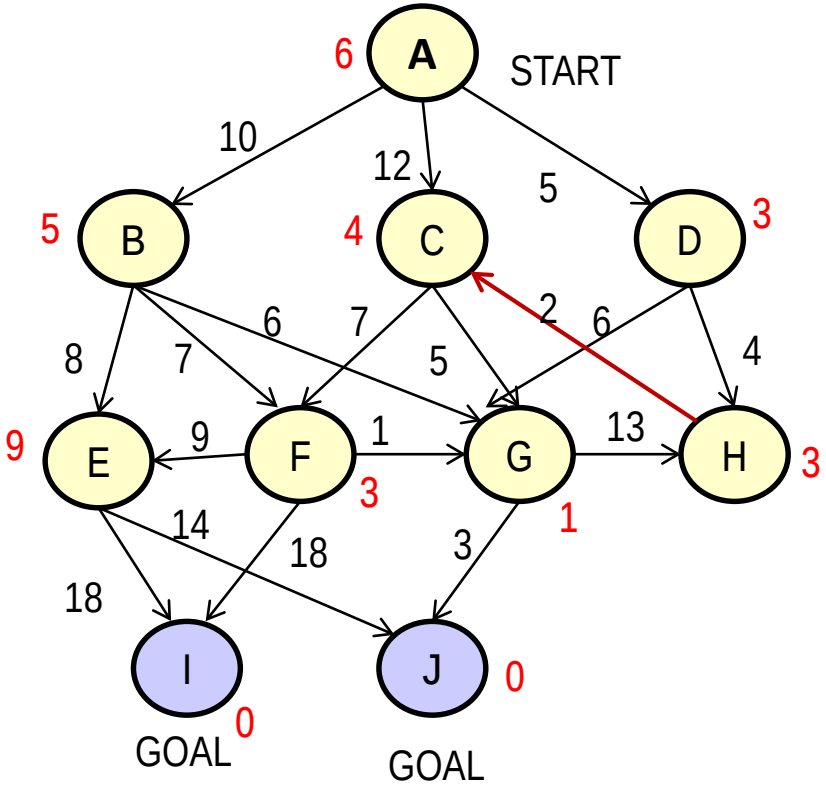
1. [Initialize] Initially the OPEN List contains the Start Node s. g(s) = 0, f(s) = h(s). CLOSED List is Empty.
2. [Select] Select the Node n on the OPEN List with minimum f(n). If OPEN is empty, Terminate with Failure
3. [Goal Test, Terminate] If n is Goal, then Terminate with Success and path from s to n.
4. [Expand]
   a) Generate the successors n_1, n_2, …. n_k, of node n, based on the State Transformation Rules
   b) Put n in LIST CLOSED
   c) For each n_i, not already in OPEN or CLOSED List, compute
      a) g(n_i) = g(n) + c(n, n_i), f(n_i) = g(n_i) + h(n_i) , Put n_i in the OPEN List
   d) For each n_i already in OPEN, if g(n_i) > g(n) + c(n,n_i), then revise costs as:
      a) g(n_i) = g(n) + c(n, n_i), f(n_i) = g(n_i) + h(n_i)
5. [Continue] Go to Step 2

EXECUTION OF ALGORITHM A*



1. OPEN = {A[0,6,6]}, CLOSED = {}
2. OPEN = {B[10,5,15,A], C[12,4,16,A], D[5,3,8,A]}, CLOSED = {A}
3. OPEN = {B[10,5,15,A], C[12,4,16,A], G[11,1,12,D], H[9,3,12,D]}, CLOSED = {A,D}
4. OPEN = {B[10,5,15,A], C[11,4,15,H], G[11,1,12,D]}, CLOSED = {A,D,H}
5. OPEN = {B[10,5,15,A], C[11,4,15,H], J[14,0,14,G]}, CLOSED = {A[],D[A],H[D],G[D]}
6. Goal J found. Terminate with cost 14 and path A,D,G,J.

PROPERTIES OF ALGORITHM A*



IF HEURISTIC ESTIMATES ARE NON-NEGATIVE LOWER BOUNDS AND EDGE COSTS ARE POSITIVE:

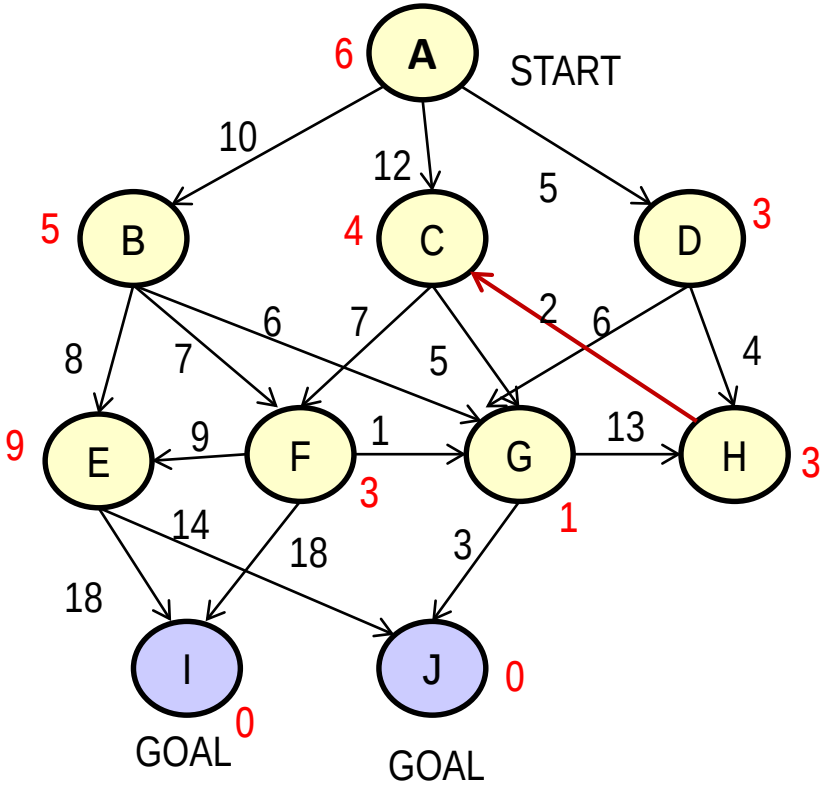
- FIRST SOLUTION IS OPTIMAL
- NO NODE IN CLOSED IN EVER REOPENED
- WHENEVER A NODE IS REMOVED FROM OPEN ITS MINIMUM COST FROM START IS FOUND
- EVERY NODE n WITH f(n) LESS THAN OPTIMAL COST IS EXPANDED
- IF HEURISTICS ARE MORE ACCURATE THEN SEARCH IS LESS

ALGORITHM DFBB

DEPTH FIRST BRANCH AND BOUND (DFBB)

1. Initialize Best-Cost to INFINITY
2. Perform DFS with costs and Backtrack from any node n **whose f(n) ≥ Best**-Cost
3. On reaching a Goal Node, update Best-Cost to the current best
4. Continue till OPEN becomes empty

EXECUTION OF DFBB

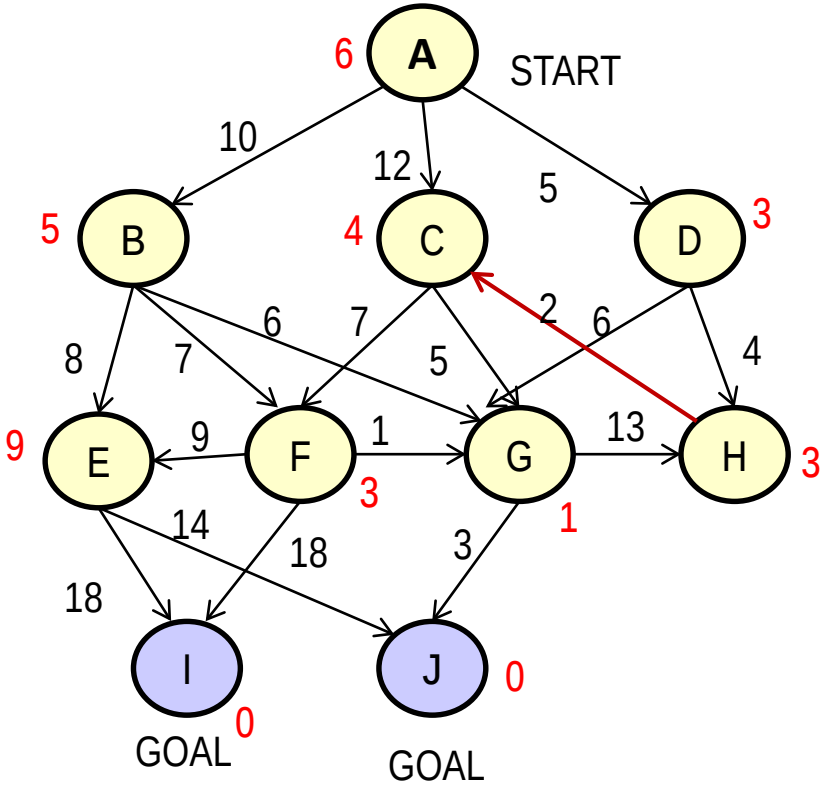


ALGORITHM IDA*

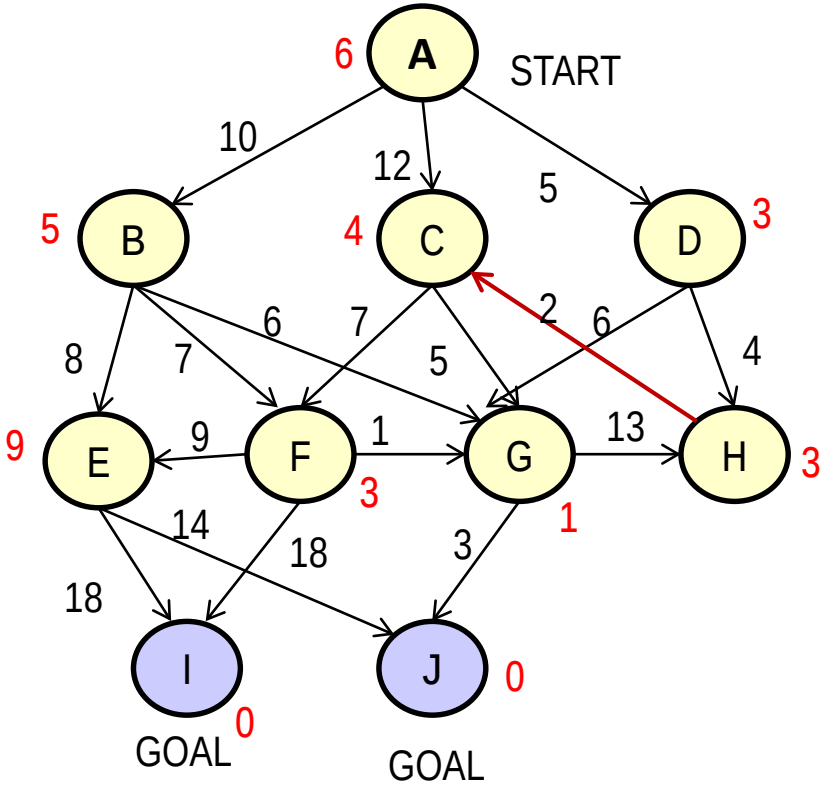
ITERATIVE DEEPENING A* (IDA*)

1. Set Cut-off Bound to f(s)
2. Perform DFBB with Cut-off Bound. Backtrack from any node whose f(n) > Cut-off Bound.
3. If Solution is Found, at the end of one Iteration, Terminate with Solution
4. If Solution is not found in any iteration, then update Cut-off Bound to the lowest f(n) among all nodes from which the algorithm Backtracked.
5. Go to Step 2
6. PROPERTIES OF DFBB AND IDA*: Solution Cost, Memory, Node expansions, Heuristic Accuracy, Performance on Trees / Graphs

EXECUTION OF IDA*



COMPARING A*, DFBB & IDA*



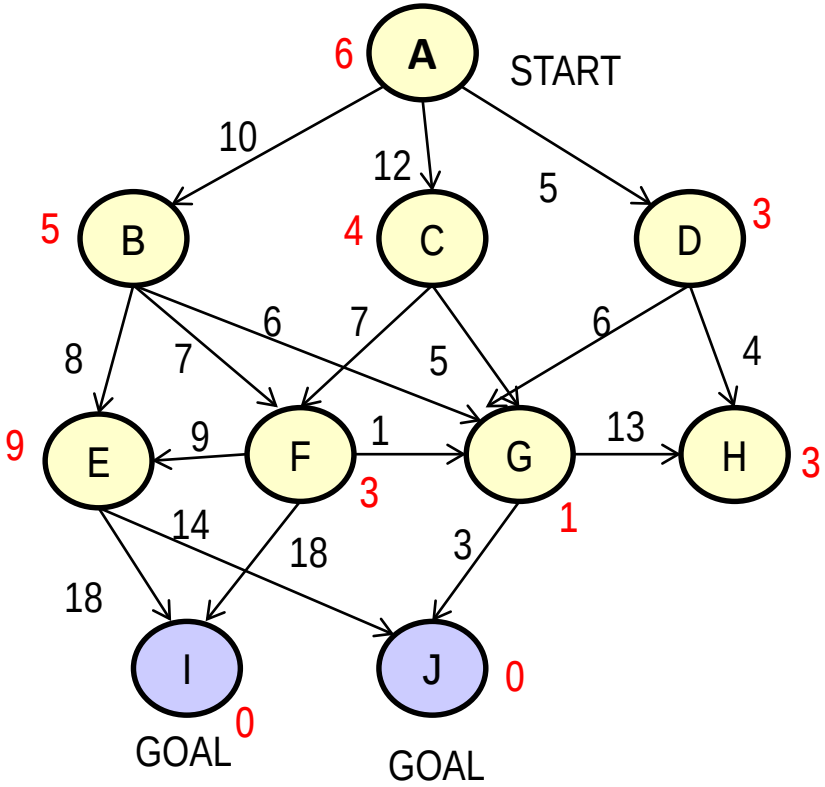
1. TERMINATION ISSUES
2. MEMORY AND NODE EXPANSIONS
3. HEURISTIC ACCURACY EFFECT
4. MAXIMIZATION PROBLEMS

KNAPSACK PROBLEM: MAXIMIZATION

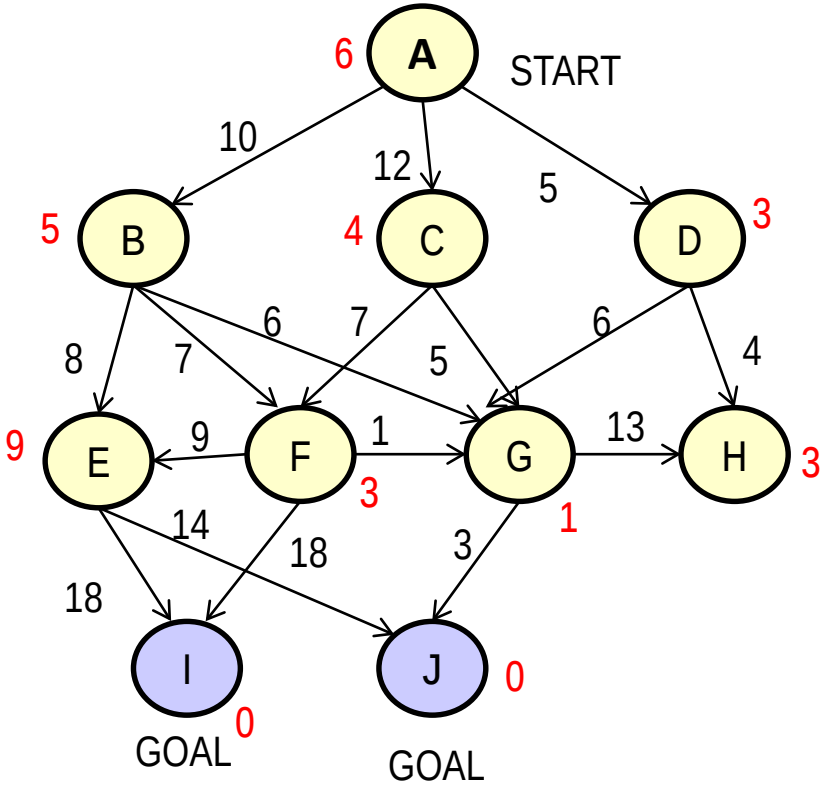
BASIC HEURISTIC SEARCH ALGORITHM: m-DFS Version

1. [Initialize] Initially the OPEN List contains the Start Node s. CLOSED List is Empty. Set f(s) by applying heuristic evaluation function. Set CB = Initial Best Cost
2. [Select] Select the a Suitable Node n on the OPEN List. If OPEN is empty, Terminate
3. [Goal Test] If n is Goal, then decide on Termination or Continuation with / without Cost Update
4. [Expand]
   a) Generate the successors n_1, n_2, …. n_k, of node n, based on the State Transformation Rules
   b) Put n in LIST CLOSED
   c) For each n_i, not already in OPEN or CLOSED List, put n_i in OPEN List. Compute the f(n_i) of each node and PRUNE if applicable based on CB.
   d) For each n_i already in OPEN or CLOSED decide based on cost of the paths. Update f(n_i) and bring back from CLOSED to OPEN if needed.
5. [Continue] Go to Step 2

EXECUTION OF 2-DFS (Minimization)



EXECUTION OF 3-DFS (Minimization)



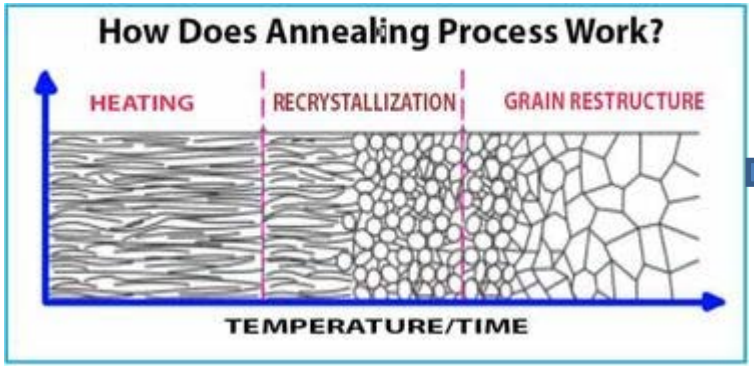
BEAM SEARCH

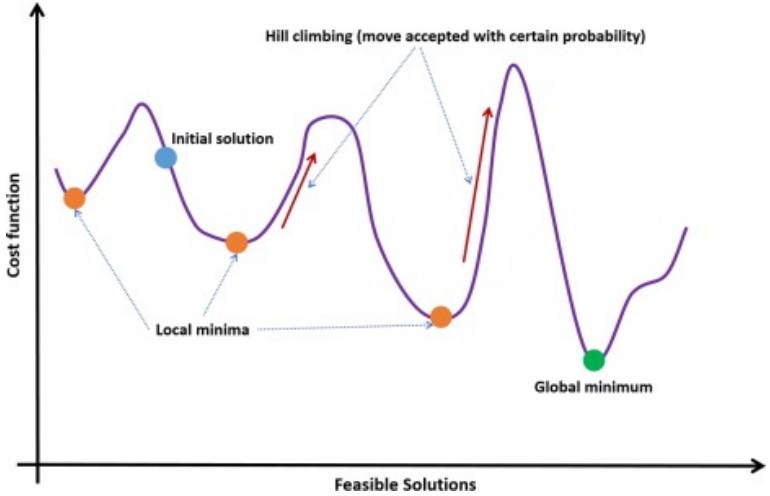
1. [Initialize] Initially the OPEN List contains the Start Node s. CLOSED List is Empty. Set f(s) by applying heuristic evaluation function. Set CB = Initial Best Cost
2. [Select] Select the a Suitable Node n on the OPEN List. If OPEN is empty, Terminate
3. [Goal Test] If n is Goal, then decide on Termination or Continuation with / without Cost Update
4. [Expand]
   a) Generate the successors n_1, n_2, ... n_k, of node n, based on the State Transformation Rules
   b) Put n in LIST CLOSED
   c) For each n_i, not already in OPEN or CLOSED List, put n_i in OPEN List. Compute the f(n_i) of each node and PRUNE if applicable based on CB.
   d) For each n_i already in OPEN or CLOSED decide based on cost of the paths. Update f(n_i) and bring back from CLOSED to OPEN if needed.
5. [Continue] Go to Step 2

ANYTIME SEARCH

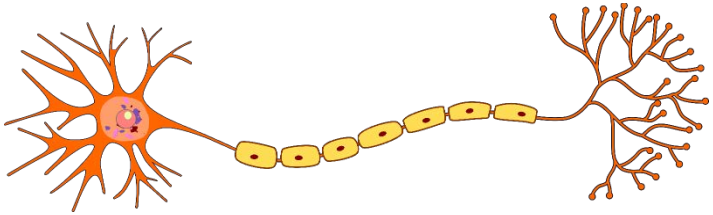
1. [Initialize] Initially the OPEN List contains the Start Node s. CLOSED List is Empty. Set f(s) by applying heuristic evaluation function. Set CB = Initial Best Cost
2. [Select] Select the a Suitable Node n on the OPEN List. If OPEN is empty, Terminate
3. [Goal Test] If n is Goal, then decide on Termination or Continuation with / without Cost Update
4. [Expand]
   a) Generate the successors n_1, n_2, ... n_k, of node n, based on the State Transformation Rules
   b) Put n in LIST CLOSED
   c) For each n_i, not already in OPEN or CLOSED List, put n_i in OPEN List. Compute the f(n_i) of each node and PRUNE if applicable based on CB.
   d) For each n_i already in OPEN or CLOSED decide based on cost of the paths. Update f(n_i) and bring back from CLOSED to OPEN if needed.
5. [Continue] Go to Step 2

Inspiration from Nature's Optimization

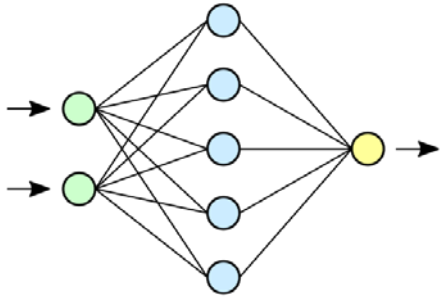




Inspiration from Life Processes

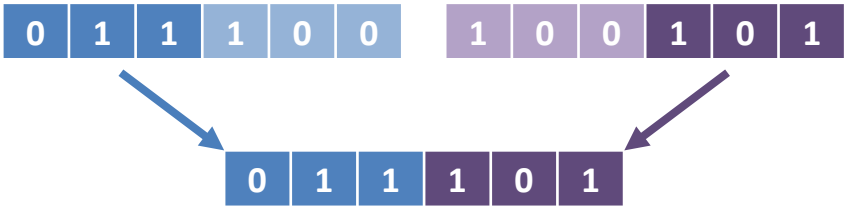


Neurons



Neural Networks

Genes



Genetic Algorithm

Search by Solution Transformation

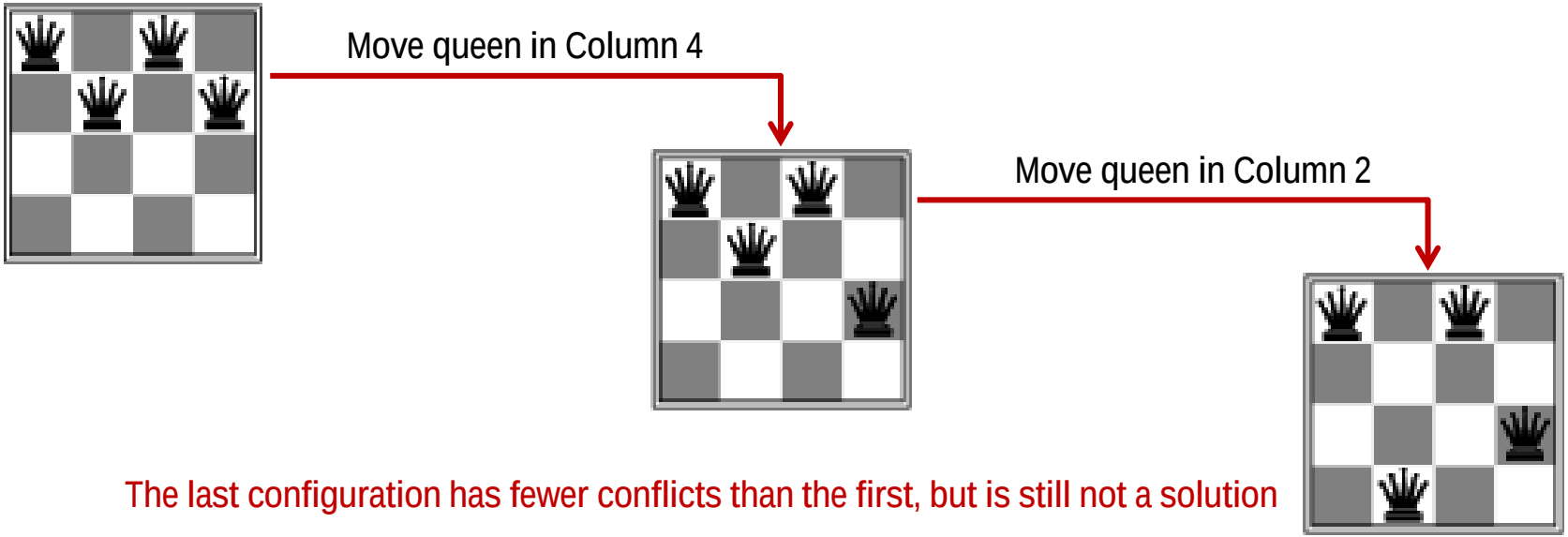
- Local Search – Gradient Descent, Hill Climbing
- Simulated Annealing
- Genetic Algorithms
- Tabu Search

Local Search

- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution
  - Local search: widely used for *very big* problems
  - Returns good but *not optimal* solutions in general
- The state space consists of "complete" configurations
  - For example, every permutation of the set of cities is a configuration for the traveling salesperson problem
- The goal is to find a “close to optimal” configuration satisfying constraints
  - Examples: n-Queens, VLSI layout, exam time table
- Local search algorithms
  - Keep a single "current" state, or small set of states
  - Iteratively try to improve it / them
  - Very memory efficient since only a few states are stored

Example: *4*-queens

- Goal: Put *4* queens on an *4* × *4* board with no two queens on the same row, column, or diagonal
- State space: All configurations with the queens in distinct columns
- State transition: Move a queen from its present place to some other square in the same column
- Local Search: Start with a configuration and repeatedly use the moves to reach the goal



The last configuration has fewer conflicts than the first, but is still not a solution

Hill-climbing: *A greedy approach*

THE IDEA: *Make a move only if the neighboring configuration is better than the present one*

```
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
```

The dual of Hill Climbing is Gradient Descent. Hill climbing is for maximizing, Gradient Descent is for minimizing

Source: Artificial Intelligence – A Modern Approach, Peter Norvig and Stuart Russell, Prentice Hall

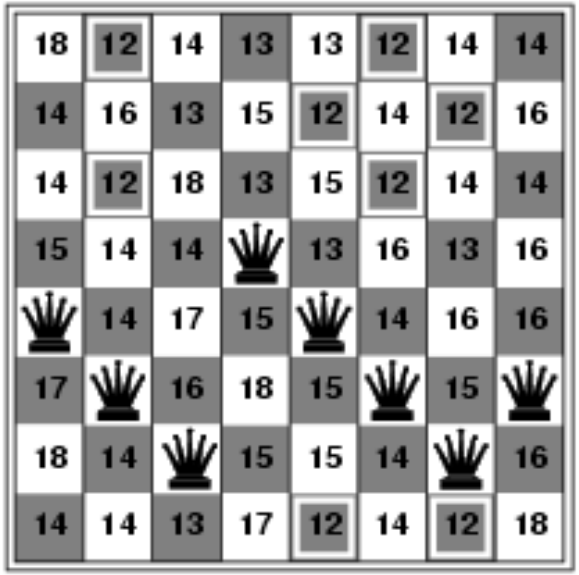
Gradient Descent in 8-queens

Value[state] = The numbers pairs of queens that are attacking each other, either directly or indirectly.

Value[state] = 17 for the state shown in the Fig.

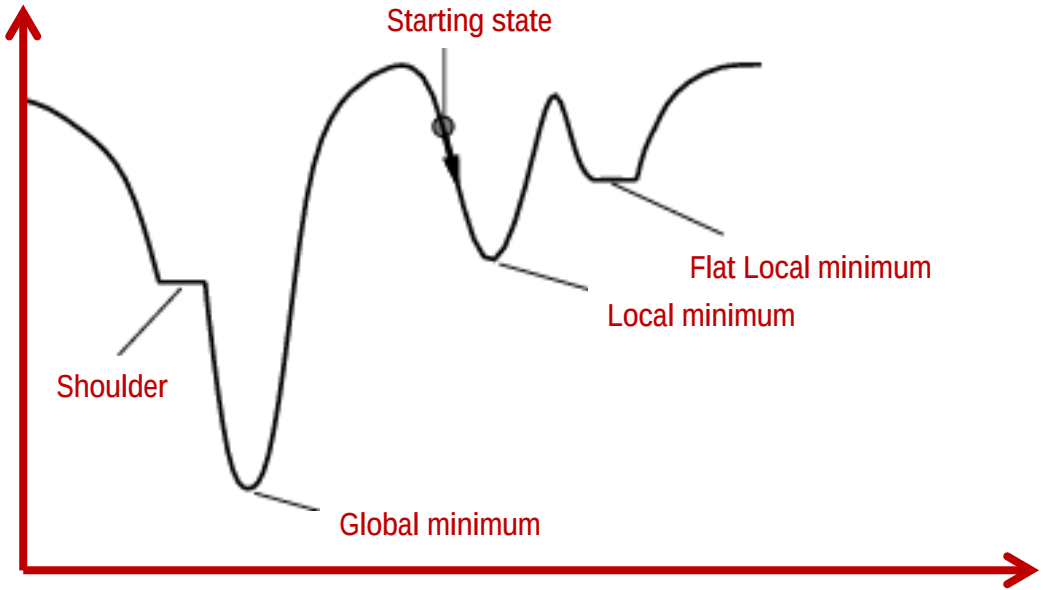
- The number in each square is the value of state if we move the queen in the same column to that square.

- Therefore the best greedy move is to move a queen to a square labeled with 12.
    - There are many such moves. We choose one of them at random.



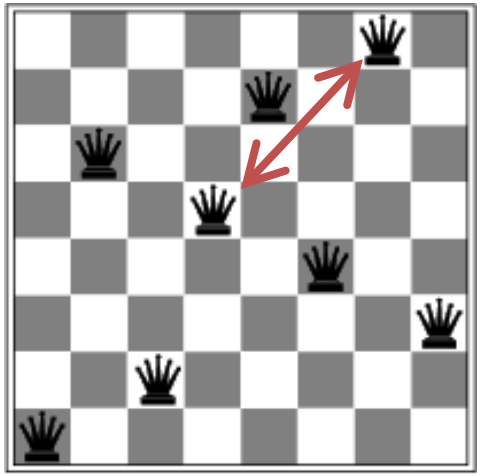
Gradient descent can get stuck in local minima

- Each neighbor of a minimum is inferior with respect to the minimum
- No move in a minimum takes us to a better state than the present state



Local minimum in 8-queens

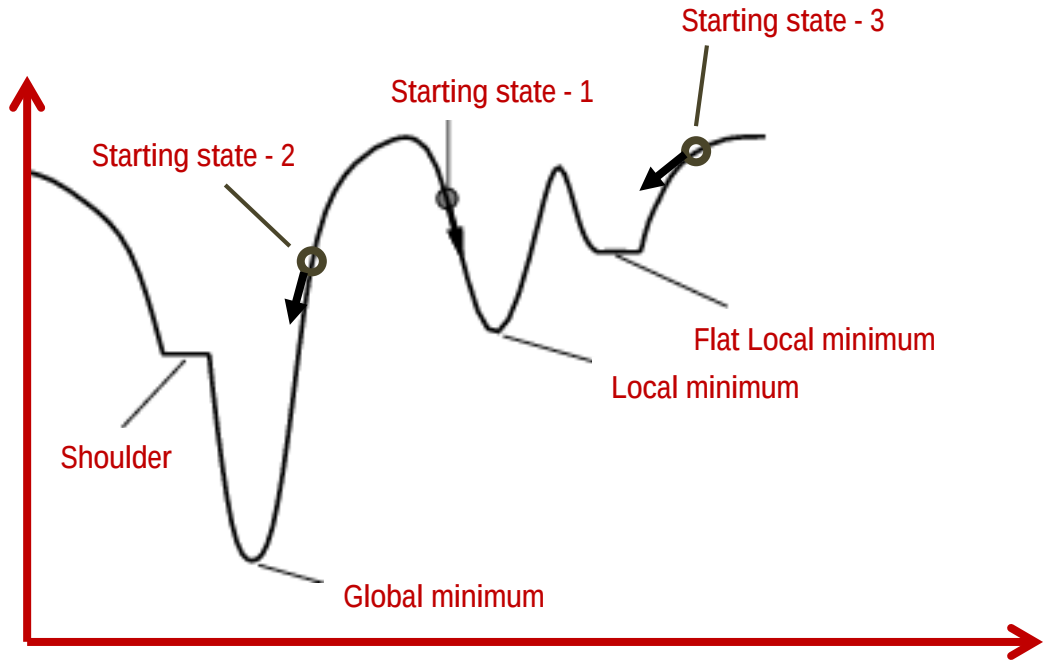
- A local minimum with only one conflict
- All one-step neighbors have more than one conflict



How to get out of local minima?

Idea-1: Gradient Descent with Random Restart

- Using many random restarts improves our chances
- Restart a random initial state, *many times*
  - Report the best result found across *many* trials



Idea-2: Allow moves to inferior neighbors

- To get out of a local minimum, we must allow moves to inferior neighbors
- However, we must ensure that we do not oscillate among a set of states

- IDEAS
- Simulated Annealing: Allow moves to inferior neighbors with a probability that is regulated over time. We discuss this in more details later

- Tabu Search: Add recently visited states to a tabu-list.
  - These states are temporarily excluded from being visited again
  - Forces solver away from explored regions
  - Avoid getting stuck in local minima (in principle)

Simulated annealing search

- IDEA: Escape local minima by allowing some "bad" moves but gradually decrease their probability
  - The probability is controlled by a parameter called *temperature*
  - *Higher temperatures allow more bad moves than lower temperatures*
  - Annealing: Lowering the temperature gradually Quenching: Lowering the temperature rapidly

function SIMULATED-ANNEALING(*problem, schedule*)

** *current* ← INITIAL-STATE[*problem*]**

** for *t* ← 1 to ∞ do**

** *T* ← *schedule*[*t*]**

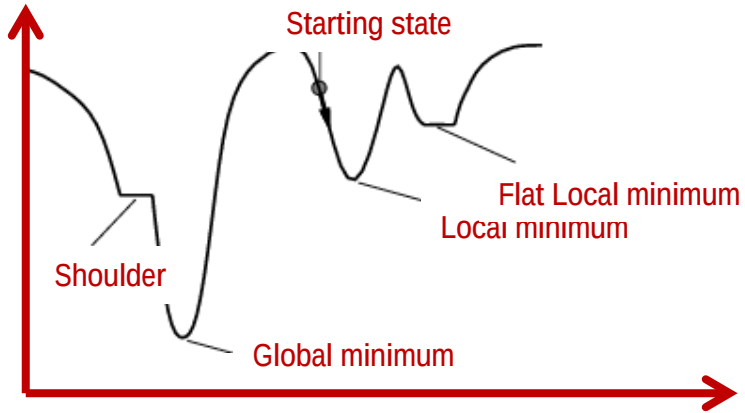
** if *T* = 0 then return *current***

** *next* ← a randomly selected successor of current**

** Δ*E* ← VALUE[*next*] – VALUE[*current*]**

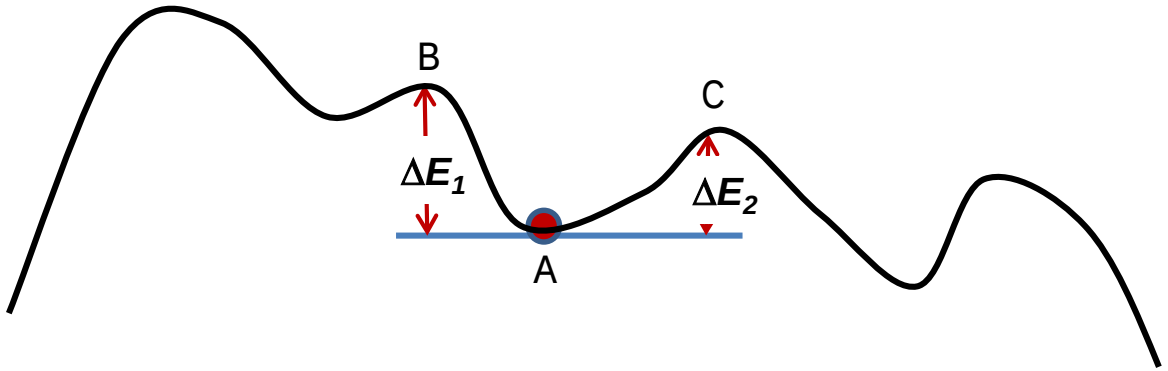
** if** Δ*E* < 0 then *current* ← *next*

  else *current* ← *next* with probability $e^{-\Delta E/T}$



How simulated annealing works

Probability of making a bad move = $e^{-\Delta E/T} = \frac{1}{e^{\Delta E/T}}$



Since $\Delta E_1 > \Delta E_2$ moving from A to C is exponentially more probable than moving from A to B

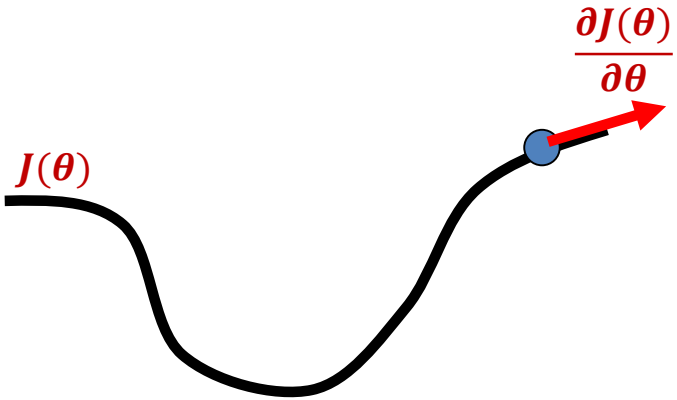
Properties of Simulated Annealing

- It can be proven that:
  - If *T* decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1
  - Since this can take a long time, we typically use a temperature schedule which fits our time budget and settle for the sub-optimal solution

- Simulated annealing works very well in practice
- Widely used in VLSI layout, airline scheduling, etc.

Hill Climbing in Continuous Multi-variate State Spaces

- Denote “state” as μ; cost as J(μ)



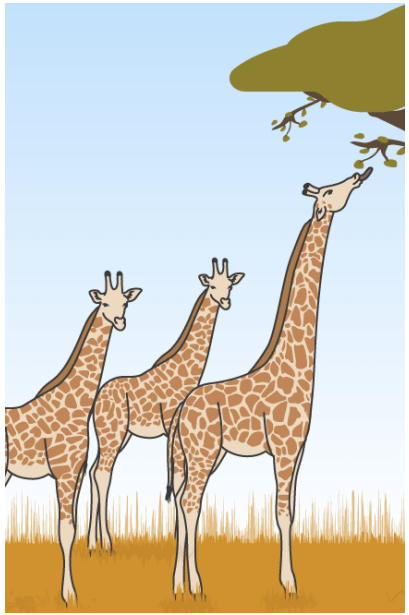
- Choose a direction in which J(μ) is decreasing
- Derivative: $\frac{\partial J(\theta)}{\partial \theta}$
  - Positive derivative means increasing
  - Negative derivative means decreasing
- Move: A short uphill step in the chosen direction

Local Search with Multiple Present States

- Instead of working on only one configuration at any time, we could work on multiple promising configurations concurrently
- LOCAL BEAM SEARCH
- Maintain *k* states rather than just one. Begin with *k* randomly generated states
- In each iteration, generate all the successors of all *k* states
- Stop if a goal state is found; otherwise Select the *k* best successors from the complete list and repeat

- GENETIC ALGORITHMS
- States are strings over a finite alphabet (genes). Begin with *k* randomly generated states (population).
- Select individuals for next generation based on a fitness function.
- Two types of operators for creating the next states:
  - Crossover: Fit parents to yield next generation (offspring)
  - Mutation: Mutate a parent to create an offspring randomly with some low probability

Genetic Algorithm (GA)



- GA is inspired by evolutionary biology
- Natural selection of the best performing individuals
- Suited for problems with many feasible solution but finding the optimal is difficult
- Examples of problems which can be solved with it:
  - Knapsack problem
  - Travelling salesman problem
  - Antenna design
  - Efficient space utilization in computer chip

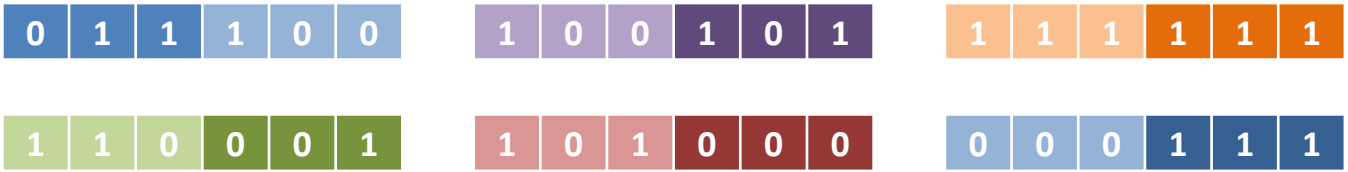
Basic Genetic Algorithm

- Encode possible solutions in population
- Initialize population – first generation
- Repeat for each generation
  - Crossover
  - Mutation
  - Fitness computation
  - Select next generation

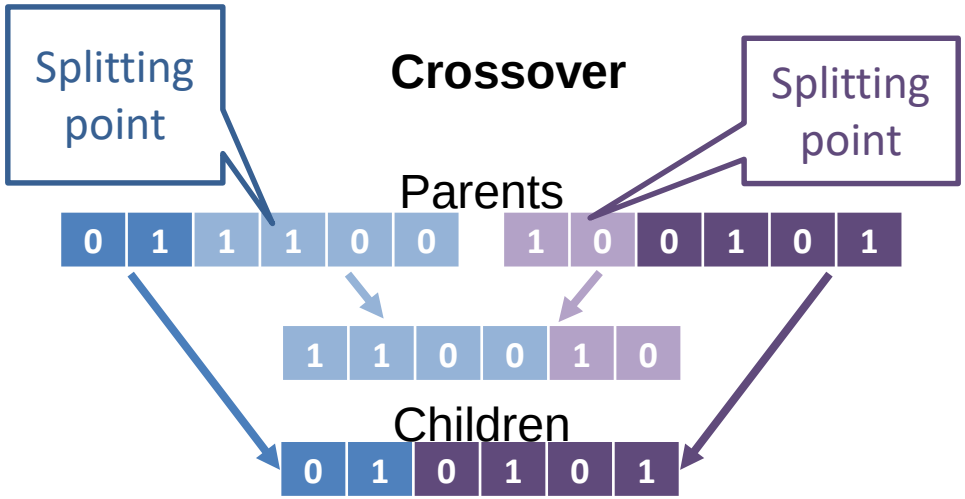
Encode and initialize First Generation

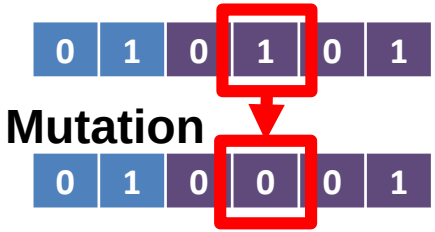
- A feasible solution is represented a string

- String should be able to represent all feasible solutions
- **Individual** – One feasible solution represented as a string
- **Population** – Set of all individuals in a generation
- Initial population can be selected randomly



Crossover and Mutation





Crossover

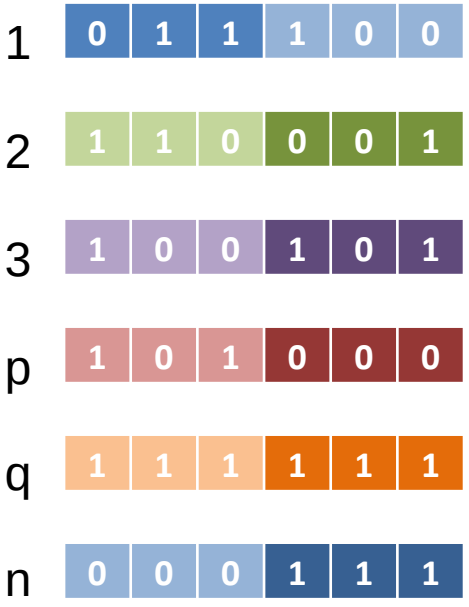
- Select two or more individuals in a population - **parents**
- Randomly select a splitting point
- Split each 'parent' string at the splitting point
- Recombine the halves of each parent string with the other half of the other parent string – **children**
- Instead of splitting, any other, more involved, operation can also be performed here which gives children using multiple parents

Mutation

- Random small changes in the string

Fitness Function

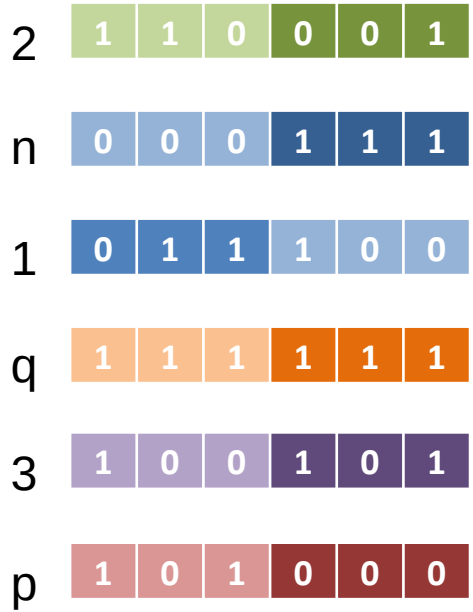
Population of a generation in the sequence it is generated



Fitness function evaluates how good each solution is for the given problem

$f(\cdot)$

Ranked individual as per the fitness function



Example of GA

- Four factories supplying good to four cities
- Each factory can supply only one city
- Each city can get supplies from exactly one factory
- Distances between factories and cities are given in table
- Objective is to minimise distance

	City 1	City 2	City 3	City 4
Factory A	20	25	17	22
Factory B	14	9	19	6
Factory C	13	15	23	30
Factory D	8	25	12	7

Distance between each factory and city in kilometres

Adapted with thanks to Prof Biswajit Mahanty, Dept of Industrial and Systems Engineering, IIT Kharagpur

Encoding : Permutation Encoding

- ‘d’ in the 1st location denotes that Factory D will be supplying to City 1 and so on
- Each string will have only one occurrence of each of the letters – a, b, c or d
- Total distance for this individual is 8+9+23+22 = 62km

- Initial random population of 4 individuals

Fitness Function

	City 1	City 2	City 3	City 4
Factory A	20	25	17	22
Factory B	14	9	19	6
Factory C	13	15	23	30
Factory D	8	25	12	7

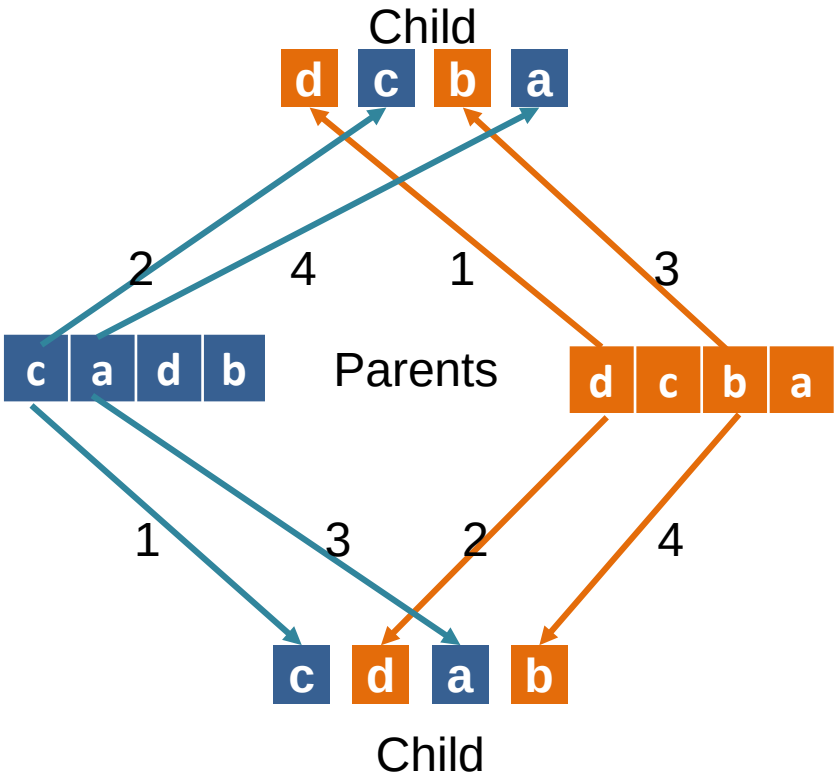
- Objective is to minimise total distance D
- Alternatively : maximise (100 – D)
- Fitness function : f = (100 – D)
- Selection : expected count = f/f_{avg} : count for pool for crossover
- For initial population:

String	f = 100 – D	f/f_{avg} (expected count)	Comments
abcd	100 – (20+9+23+7) = 41	1.21 (≈ 1)	Occurs once in pool for crossover
bdac	100 – (14+25+17+30) = 14	0.41 (≈ 0)	Not included in the pool for crossover
cadb	100 – (13+25+12+6) = 44	1.30 (≈ 2)	Occurs twice in the pool for crossover
dcba	100 – (8+15+19+22) = 36	1.07 (≈ 1)	Occurs once in the pool for crossover

$f_{avg} = 33.75$

Crossover

	City 1	City 2	City 3	City 4
Factory A	20	25	17	22
Factory B	14	9	19	6
Factory C	13	15	23	30
Factory D	8	25	12	7



	Pool for crossover
1	abcd
2	cadb
3	cadb
4	dcba

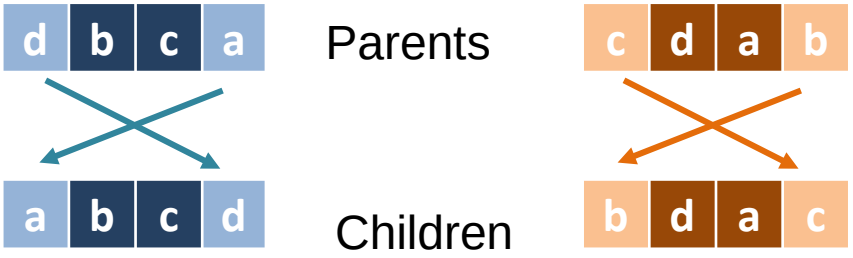
replicated

	After crossover
1	abcd
2	cadb
3	dcba
4	cdab

crossover

Mutation …

	City 1	City 2	City 3	City 4
Factory A	20	25	17	22
Factory B	14	9	19	6
Factory C	13	15	23	30
Factory D	8	25	12	7



	Pool for crossover
1	abcd
2	cadb
3	cadb
4	dcba

replicated

mutation

	After crossover
1	abcd
2	cadb
3	acbd
4	badc

Iteration 2

	City 1	City 2	City 3	City 4
Factory A	20	25	17	22
Factory B	14	9	19	6
Factory C	13	15	23	30
Factory D	8	25	12	7

String	f = 100 – D	f/f_{avg} (expected count)	Comments
abcd	100 – (20+9+23+7) = 41	1.19 (≈ 1)	Occurs once in pool for crossover
cadb	100 – (13+25+12+6) = 44	1.28 (≈ 2)	Occurs twice in the pool for crossover
acbd	100 – (20+15+19+7) = 39	1.02 (≈ 1)	Occurs once in pool for crossover
bdac	100 – (14+25+17+30) = 14	0.41 (≈ 0)	Not included in the pool for crossover

$f_{avg} = 34.5$

Iteration 1: f_{avg} =33.75

Iteration 2: f_{avg} =34.5

Best string till now = cabd

Best (f) till now = 44

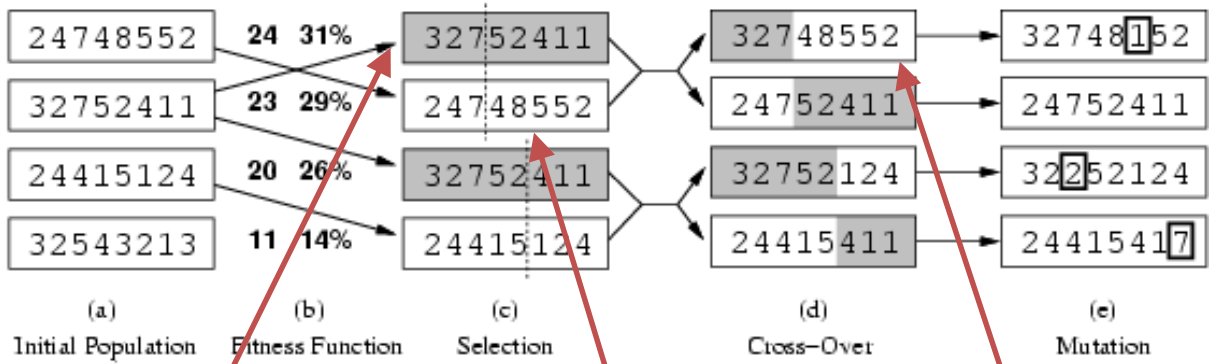
Min distance till now = 56 km

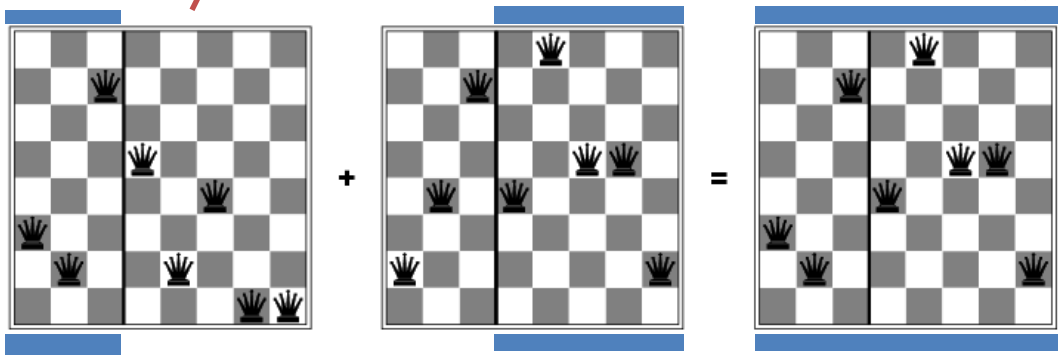
Genetic Algorithm for 8 Queens

Fitness function:

non-attacking pairs

(min = 0, max = 8×7 / 2 = 28)



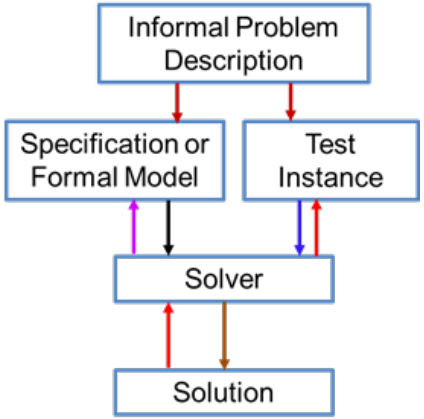


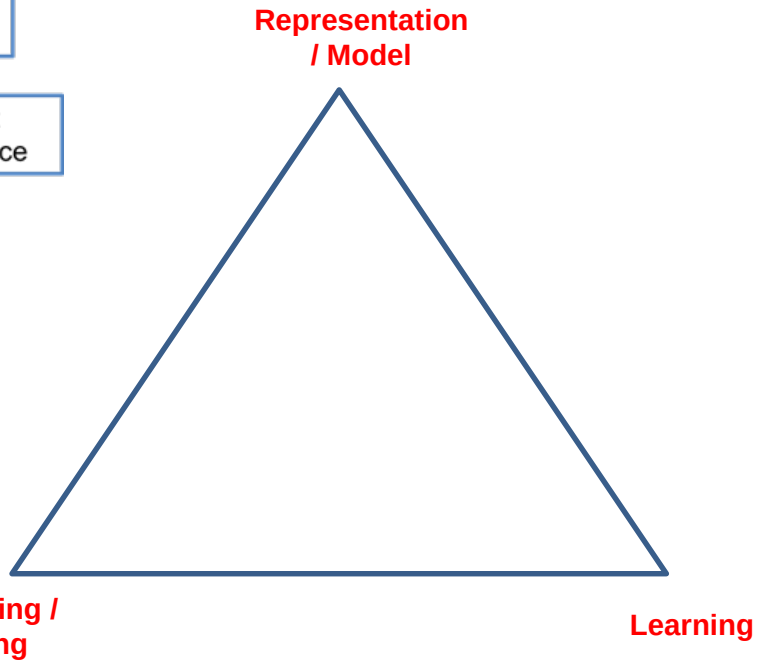
Population fitness = 24+23+20+11 = 78

P(Gene-1 is chosen)
= Fitness of Gene-1 / Population fitness
= 24 / 78 = 31%

P(Gene-2 is chosen)
= Fitness of Gene-2 / Population fitness
= 23 / 78 = 29%

Machine Learning & Its Integration with Algorithms





Problem Types:

- ❑ Full instance
- ❑ Partial Information
- ❑ Incremental
- ❑ Dynamic
- ❑ Resource Limited

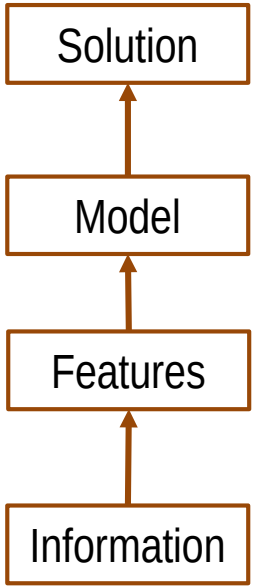
Metrics of Goodness of the Solver:

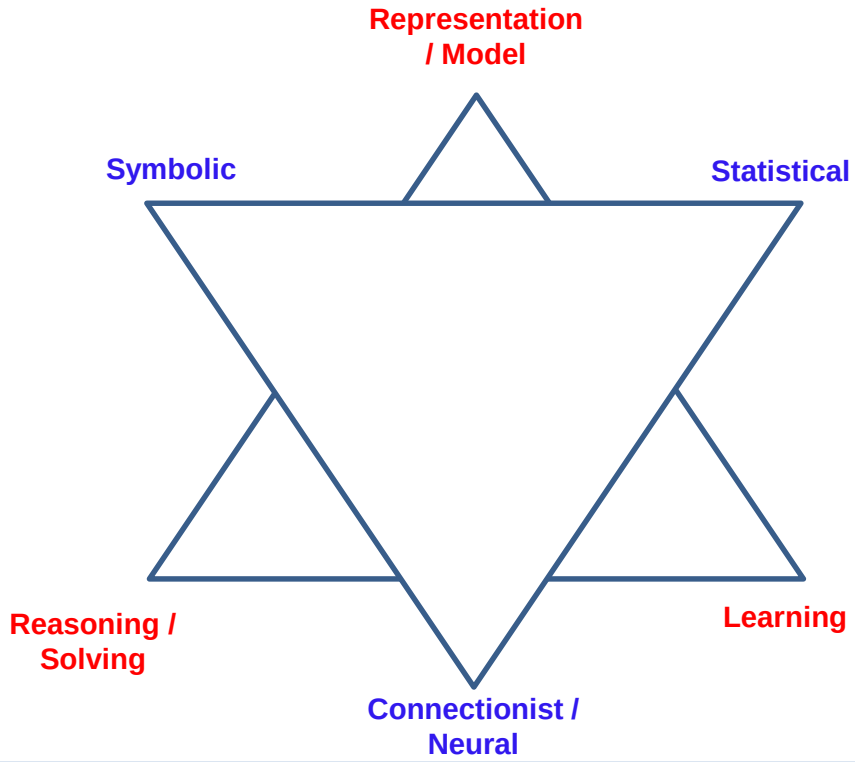
- Correctness (Eventually Probably Approximately Correct)
- Efficiency

Sample Problems:

1. Top k-elements
2. Finding a Match
3. Solving a Puzzle
4. Making a Budget
5. Buying / Selling Shares

Algorithms and AI/ML Pipeline





Sample Problems:
1. Top k-elements
2. Finding a Match
3. Solving a Puzzle
4. Making a Budget
5. Buying / Selling Shares

Problem Types:
- ❑ Full instance
- ❑ Partial Information
- ❑ Incremental
- ❑ Dynamic
- ❑ Resource Limited

Metrics of Goodness of the Solver:
- Correctness (Eventually Probably Approximately Correct)
- Efficiency

Thank you

Any Questions?