Week1 – History & Philosophy

- The main challenge with AI algorithms is CombEx. The exponential nature of space/time complexity.
- We expect all intelligent agents to be persistent, autonomous, proactive and goal-directed. It's required to carry a model of the world with it, of which it's part.
- Signal processing, Motor control, Neuro-fuzzy systems, Symbolic reasoning are what every intelligent system must have . Of these, symbolic reasoning is the largest part of the whole, as far as human cognition is concerned.
- Collections of interconnected neurons (typically, sigmoids) can do complex computations.
- Perceptron, a binary classifier, started the journey of artificial neural networks. The next step was inclusion of a hidden layer into the scheme of things. Backpropagation helped with the learning algorithm.
- AI involves symbolic knowledge representation (modelling) and problem solving, whereas ML tries to make sense of the data. AI also is helpful at arriving at inferences - plausible (probabilistic) and deductive inferences.
- Semiotics is a study of signs and symbols, which are abstract representation for something else altogether.
- Biosemiotics is an interdisciplinary field that combines biology and semiotics, focusing on the role of signs and communication in living organisms. For example, ants leaving pheromones to signal source of food or danger comes under the preview of biosemiotics.
- The Pascaline (the arithmetic machine) by Pascal, Leibnitz wheel (stepped drum) by Leibnitz, Arithmometer by Thomas de Colmar, Difference Engine by Charles Babbage, Jaquard Looms by Jaquard, Analytic Engine by Chales Babbage, ENIAC by John Mauchy and team were all precursors to the modern day computer.
- John McCarthy, Marvin Minsky and Claude Shannon organized the DartMouth conference on AI in 1956. Coining of the name AI is credited to John McCarthy, who also created the Lisp programming language.
- Every being (living or non-living) is created out of fundamental particles. It's the interactions between them that's giving them the shape, structure and could be even behaviour of those beings. Human body is made of close to 10^{27} atoms.
- The two kinds of Problem Solving techniques are
  - Model based reasoning, which starts with first principles and solves problems through trial and error. Study of *Search methods* (this course) belong to this category.
  - Memory based reasoning, which utilizes knowledge and experience to solve problems.

Week2 – State based search and algos

- Map coloring problem can be solved in any of the following methods - Brute force, Informed Search or General (Constraint satisfaction or State space) search.
- States are often represented as a graph. S represents the start state and G the goal state. $MoveGen$ function returns all moves (neighbors) that can be made in a given state. $GoalTest$ function returns $True$ if the given state is the goal.
- The state space is implicit, since the state space graph isn't available to start with. Given the start state, the graph can be *generated* using $MoveGen$ function.
- Note that certain transitions are reversible, but that's not true for all transitions.
- The general approach to the problems is to develop domain-independent algorithms to perform state-space search, and tie them up to appropriate domain-specific functions.
- In general, the algorithm proceeds like this.
  1. Start at the start node
  2. add its neighbors to OPEN list, unless they are already in CLOSED list.
  3. Add the current node to CLOSED list, and remove from the OPEN list.
  3. pick *next* node in the OPEN list, check if its the goal.
  4. Stop if it's the goal, else go to 2.

  NOTE: This algorithm is called $GenerateAndTest$
- Above algorithm could go into cycles, if the CLOSED list is not kept.
- In a variation of the above algorithm, the step 2 is modified to
  - add its neighbors to OPEN list, unless they are already in CLOSED list or OPEN list.
- All of the above algorithms seeks the goal state, but not the path to it. These are termed configuration problems. Examples of such problems are N-queens, Crossword, Sudoku, Map Coloring, SAT etc.
- Problems that seek the path to the goal state are termed Planning problems. Examples of such problems are River-crossing problems, route-finding problems, Rubik's cube, 8/15/24 puzzle, Cooking a dish etc.
- In planning problems, we need to store $NodePairs$, containing the node and its parent, so that when we find the goal state, we can reconstruct the path to it.
- The above algorithms haven't defined which node should be picked from the OPEN list. These algorithms are called non-deterministic algorithms for this reason.
- In the deterministic versions of these algorithms, LIST is used instead of SET to keep the OPEN list, and during each iteration the first (head) node is picked from the OPEN list. Note that HEAD represents the first node in the list, and TAIL represents rest of the list (except the head).
- Semicolon (:) is used as an append operation. Thus,
  $[a,\ b,\ c]\ :\ [d,\ e]\ =\ [[a,\ b,\ c],\ d,\ e]$.
- ++ is used as an list extend operation. Thus, $[a,\ b,\ c]\ ++\ [d,\ e]\ =\ [a,\ b,\ c,\ d,\ e]$.
- $first, second, third$ functions will return the corresponding elements from a tuple.
- $(a,\ h\ :\ t,\ c)\ \leftarrow\ (1,\ [101,\ 102,\ 103],\ null)$ will assign $null$ to c, 1 to a, and

$[101, 102, 103]$ will be split between h and t. Thus head of the list (101) gets assigned to h, and tail of the list ($[102, 103]$) to t.

- The following pseudo-code represents the DFS algorithm.

```
DFS(S)
 1  OPEN ← (S, null) : [ ]
 2  CLOSED ← empty list
 3  while OPEN is not empty
 4      nodePair ← head OPEN
 5      (N, _) ← nodePair
 6      if GoalTest(N) = true
 7          return ReconstructPath(nodePair, CLOSED)
 8      else CLOSED ← nodePair : CLOSED
 9          children ← MoveGen(N)
10          newNodes ← RemoveSeen(children, OPEN, CLOSED)
11          newPairs ← MakePairs(newNodes, N)
12          OPEN ← newPairs ++ (tail OPEN)
13  return empty list
```

The algorithm starts at the start node S. Note that each node (including the start node) is represented by constructing a $nodePair$, containing the node itself and its parent. This is required such that we're able to reconstruct the path, when goal state is found. Extract the first among the $nodePairs$ in the OPEN list, and from that $nodePair$ extract the node N. If this node is the goal state, then we reconstruct the path using the CLOSED list. Else, we obtain the child (neighbor) nodes for this node, remove those that are present in OPEN list or CLOSED list, build $nodePairs$ for each of them, and add to the beginning of the OPEN list. It's added to the beginning of the list, so that the OPEN list is maintained as a stack data structure.

- The following pseudo-code represents the BFS algorithm.

```
BFS(S)
 1  OPEN ← (S, null) : [ ]
 2  CLOSED ← empty list
 3  while OPEN is not empty
 4        nodePair ← head OPEN
 5        (N, _) ← nodePair
 6        if GoalTest(N) = TRUE
 7              return ReconstructPath(nodePair, CLOSED)
 8        else CLOSED ← nodePair : CLOSED
 9              children ← MoveGen(N)
10              newNodes ← RemoveSeen(children, OPEN, CLOSED)
11              newPairs ← MakePairs(newNodes, N)
12              OPEN ← (tail OPEN) ++ newPairs
13  return empty list
```

Note that the only difference between DFS and BFS is the line #12, where $newPairs$ is added at the start of the OPEN list in DFS, and at the end of the OPEN list in BFS. Thus, the OPEN list is a **stack** in the case of DFS, whereas it's a **queue** in the case of BFS.

- In BFS, search remains close to the start node and guarantees to find the shortest path, whereas DFS moves as far away from the start node, and almost always will not find the shortest path. Thus, if the goal state is closer to the start state, BFS will find it quicker than DFS.
- If the goal state occurs in the left-side of the tree at some depth d, the number of nodes inspected by DFS (N_{DFS}) and BFS (N_{BFS}) are as follows:

$$N_{DFS} = d + 1$$
$$N_{BFS} = \left(b^d - 1\right) / (b - 1) + 1 \quad (1)$$

- If the goal state occurs in the right-side of the tree at some depth d, the number of nodes inspected by DFS (N_{DFS}) and BFS (N_{BFS}) are as follows:

$$N_{DFS} = N_{BFS} = \left(b^{d+1} - 1\right) / (b - 1) \quad (2)$$

NOTE: b represents the number of child states for each state, also called branching factor. d represents the depth at which the goal state occurs in the graph.

- Above calculations can be performed simply by adding $1 + b + b^2 + ... + b^{d-1} + 1$ in the first case, and $1 + b + b^2 + ... + b^d$ in the second case.
- On average, $N_{DFS} = \dfrac{(d+1) + \left(b^{d+1} - 1\right) / (b - 1)}{2} = \dfrac{b^d}{2}$. Hence, the time complexity is exponential in nature.
  Similarly, on average,

$$N_{BFS} = \frac{\left(b^d - 1\right) / (b-1) + 1 + \left(b^{d+1} - 1\right) / (b-1)}{2} \approx \frac{b^d(b+1)}{2(b-1)}$$

Thus, $N_{BFS} = \frac{b+1}{b-1} N_{DFS}$. For example, if the branching factor in a state space is 10, then number of nodes visited by BFS is $\frac{11}{9}$ times that of DFS.

- Speaking about space complexity, we can consider the number of open nodes. Number of open nodes increases by $(b-1)$ at each level, in the case of DFS. THis is linear growth. However, in BFS, the number of open nodes increases by b^d. This is exponential growth.
- The following table compares DFS and BFS.

	Depth First Search	Breadth First Search
Time	Exponential	Exponential
Space	Linear	Exponential
Quality of solution	No guarantees	Shortest path
Completeness	Not for infinite search space	Guaranteed to terminate if solution path exists

- BFS and DFS both have exponential time complexities, with BFS guaranteeing us the shortest paths, and DFS not making any such guarantees. THe advantage of DFS is its linear space complexity.
- To get the best of both the worlds, we can use the Depth-bounded DFS (DBDFS). DBDFS restricts the search within a bound.

```
DB-DFS-2(S, depthBound)
 1  count ← 0
 2  OPEN ← (S, null, 0) : [ ]
 3  CLOSED ← empty list
 4  while OPEN is not empty
 5      nodePair ← head OPEN
 6      (N, _, depth) ← nodePair
 7      if GoalTest(N) = TRUE
 8          return (count, ReconstructPath(nodePair, CLOSED))
 9      else CLOSED ← nodePair : CLOSED
10          if depth < depthBound
11              children ← MoveGen(N)
12              newNodes ← RemoveSeen(children, OPEN, CLOSED)
13              newPairs ← MakePairs(newNodes, N, depth + 1)
14              OPEN ← newPairs ++ tail OPEN
15              count ← count + length newPairs
16          else OPEN ← tail OPEN
17  return (count, empty list)
```

Question: Why count nodes?

Most part of the algorithm remains the same as DFS, except that it doesn't probe deeper than the *depthBound*. It also keeps track of the number of *nodePairs* generated during each iteration. These features are used in another algorithm called Depth-First Iterative Deepening (DFID) as follows.

```
DFID(S)
1  count ← −1
2  path ← empty list
3  depthBound ← 0
4  repeat
5      previousCount ← count
6      (count, path) ← DB-DFS-2(S, depthBound)
7      depthBound ← depthBound + 1
8  until (path is not empty) or (previousCount = count)
9  return path
```

Why?

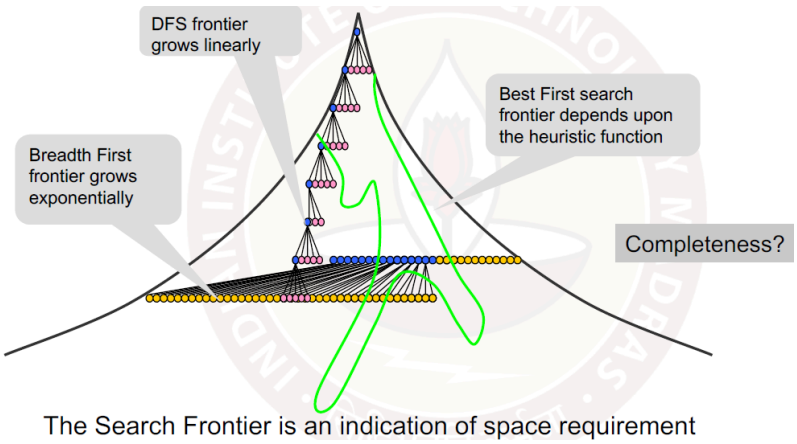
Note that in the above algorithm, the DFS is performed with increasing *depthBound* multiple times, until there is a valid path to the goal state or the number of *nodePairs* doesn't change between iterations.

- Keeping a count of the nodes visted during each cycle of DFID is necessary to prevent the algorithm from getting into an infinite loop on a FINITE graph when the goal node does not exist in the connected component
- DFID may not find shortest path to the goal state, unless we allow addition of nodes from CLOSED list to the OPEN list. However, ensure that we still maintain the CLOSED list, so that *ReconstructPath* function still works as expected. Alternatively, store the entire path to the node in *nodePair*, instead of only the node.

- In the case of DFID, at every level, the number of nodes inspected includes the leaf nodes L and the internal nodes I, whereas in the case of BFS, at every level, the number of nodes inspected includes only the leaf nodes. Since $L = (b-1) * I + 1$, we can say $N_{DFID} = N_{BFS} \frac{(L+I)}{L} \approx N_{BFS} \frac{b}{(b-1)}$. For example, if the branching factor in a state space is 10, then number of nodes visited by DFID is $\frac{10}{9}$ times that of BFS. This is not a significant cost, given that DFID ensures shortest path and space complexity.
- One of the disadvantages of BFS/DFS is that the addition of nodes always occur blindly, and proceed in the same manner each time these algorithms are run. Hence both these algorithms are called *blind/uninformed*.

Week3 – Heuristics

- Heuristics provides search with a sense of direction.
- Heuristic function $h(N)$ takes a state or a node as input and computes estimated distance of that node from the goal. Search algorithm now picks the node with the least estimated distance from the OPEN list. Note that $h(goal) = 0$.
- This leads to an algorithm called *Best First Search,* which instead of "blindly" adding neighbors of the current node to the OPEN list, will also *sort* the resultant list afterwards. Alternatively, *sort* only the neighbors and *merge* it with the current OPEN list.
- Nodes are maintained as *nodeTriples*, instead of nodePairs, since the algorithm needs to keep track of its heuristic value, in addition to the node and its parent.
- OPEN list is practically implemented using a *priority queue*, and stores the nodes in an increasing order of estimated distance from the goal node. Thus, the first node in the OPEN list is the nearest to the goal node, and the last node is the farthest away from the goal node.
- Perfect heuristic function is hard to find. Choose a function that produces the best estimates of the distance from the goal state.
- Heuristic functions are assumed to be static in nature, which implies that they're assumed to have constant time complexity.
- It's not mandatory that the heuristic function applied to the new state must return a value lesser than the current state, though ideally it should. The current state must transition to the new state, nevertheless.
- Two commonly used heuristics used in the case of 8-puzzle problem, are Hamming distance (number of tiles out of place) and Manhattan distance (sum of the number of moves each tile should make to reach the goal state). In certain cases, you might also choose to use Euclidean distance as the heuristic.
- Search frontiers for BFS, DFS and *Best First Search* are depicted below.



The Search Frontier is an indication of space requirement

In the case of BFS, the frontier consists of all nodes at the current level n, and is given by 2^n. In the case of DFS, it is linear. In the case of *Best First Search*, it depends on the heuristic function chosen.

- *BFS* guarantees to find the shortest path in a search space - finite or infinite.
- *DFS* provides a reasonable guarantee to find a path (not necessarily shortest) in a finite search space, but doesn't guarantee a path in an infinite search space.
- *Best First Search* doesn't give a guarantee to find the shortest path in a search space - finite or infinite.
- Like BFS/DFS, Best First Search also takes exponential time.
- *Hill Climbing* is an algorithm that focus on searching for the highest point (goal) of a hill, whose terrain is defined by the heuristic function.

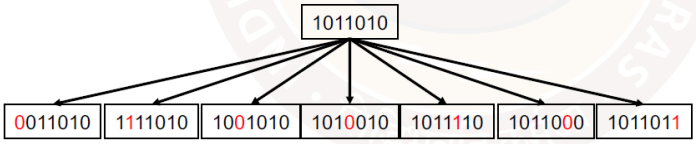
```
Hill Climbing
   node ← Start
   newNode ← head(sort_h moveGen (node)))
   While h(newNode) < h(node) Do
         node ← newNode
         newNode ← head(sort_h (moveGen (node))))
   endWhile
   return node
End
```

Note that the algorithm doesn't maintain OPEN list, but uses *MoveGen* on the current node, picks the node with the lowest heuristic from among its neighbors. These steps are repeated until the next "best" node is not better than the current node. It's also worth noting that this algorithm doesn't use a *GoalTest* function. *Hill Climbing* is a local search algorithm and can get stuck at the local maxima and will not be able to reach the global maxima.

- The advantage of *hill climbing* algorithm is that it is a constant space algorithm and a linear time complexity, but it gives no guarantee of finding a path in a search space - finite/infinite.
- In general, *climbing* is associated with ascending, but in the context of *hill climbing* algorithm, it could also descend. So, it can get stuck on a local maxima (in the case of

ascending) or a local minima (in the case of descending), and thus miss the global optima.

- *Variable Neighborhood Descent* is an extension of *hill climbing* algorithm, which attempts to escape from local optima by using a different *MoveGen* function which is more dense.
- In Synthesis/constructive methods, the final state are built piece-by-piece starting with the initial state. In perturbative methods, the final state can be subject to permutation to arrive at other candidate solutions.
- In SAT problems, a formula represented in CNF and containing multiple clauses (connected using $\wedge$) each of which use a set of propositional boolean variables must be solved to find the assignments of the individual variables such that the original formula evaluates to True or False. Each clause could use $\vee$ and ~ operation.
- For a SAT formula that uses N variables, there could be 2^N perturbations. Given a N-bit string, if the *MoveGen* function changes 1 out of N bits, this could result N perturbations. For example, 7 states could be constructed by varying each of the N bits in a 7-bit string.



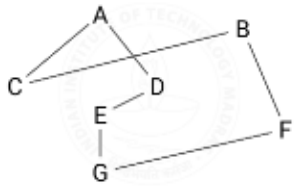
- Travelling Salesman Problem (TSP) is defined as a problem wherein, given a set of cities and given a distance measure between each pair of cities, it's required to find a Hamiltonian cycle visiting each city exactly once at the least cost.
- TSP is NP-hard problem, which implies that the solution needs at least exponential time and cannot be verified. SAT is NP-complete, which implies that the solution needs at least exponential time, but can be verified in polynomial time.
- Possible solutions to TSP:
  - Nearest neighbor tour: Start at some city, move to the nearest neighbor as long as it doesn't close the loop prematurely.
  - Greedy tour: Sort the edges. Add shortest available edge to the tour, as long as it doesn't close the loop prematurely, or create a branch/fork. Repeat this, until the goal is start city is reached or number of iterations exceeds the number of cities.
  - Savings tour:
    * Select two nodes $i,\ j$ and construct edges between each of them and a *designated* starting node D and also between each other.
    * Now, calculate $s(i, j) = d(D, i) + d(D, j) - d(i, j)$, where $d(D, i)$ and $d(D,\ j)$ are distances of the edges between the chosen nodes and the designated node, and $d(i,\ j)$ is direct distance between the chosen nodes.

$s(i,\ j)$ is called the savings value between the nodes i and j.

* Once all nodes have been covered and savings values computed for each pair $(i,\ j)$, choose the pair that has the highest savings value and construct a direct edge between them.
* Repeat this until no more merges are possible or beneficial.

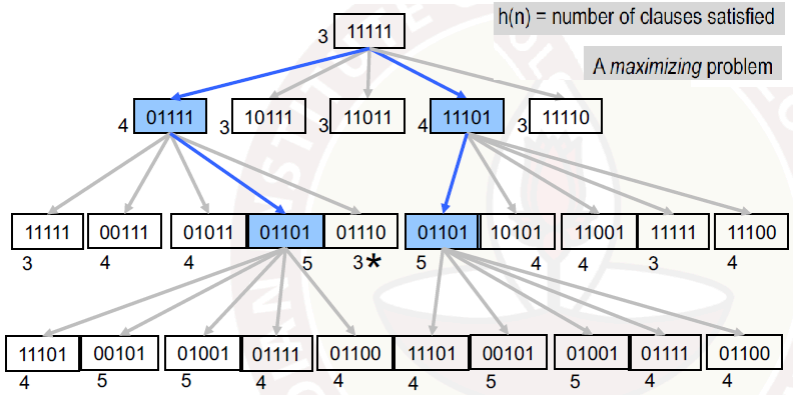
– On a candidate solution, construct perturbations by exchange of cities or edges (2/3/4). 4-edges exchanges is a case of exchange of 2 cities.

- Let's suppose, in the following TSP, we want to exchange the cities B and E.



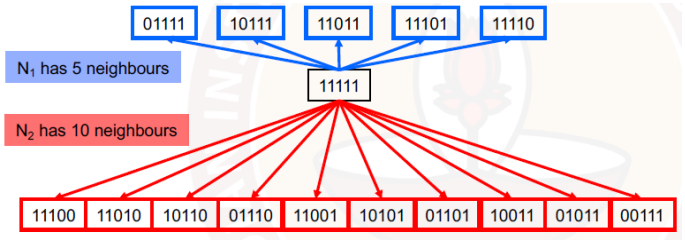
List edges with B $(BC,\ BF)$. In these edges, change the node other than B to E, thus creating $EC,\ EF$. Now, list edges with E $(EG,\ ED)$. In these edges, change the node other than E to B, thus creating $BG,\ BD$.

- Let's suppose, in the same TSP above, we want to exchange the edges BC and DE. From each of these edges, pick the first node and create a new edge BD. Similarly, pick the second node from these edges and create a new edge CE.
- TSP typically requires factorial time $(N!)$ to solve it, while SAT takes exponential (2^N) time to solve it. Thus, in the case a SAT problem with 20 variables, it takes 2^{20} operations. In the case of TSP with 20 cities, it takes $20!$ operations.
- Unlike *Hill climbling*, *Beam search* doesn't stop at identifying the best move, but looks for b best options. For example, if the beam width is set to 2, it starts by finding the 2 best *first* moves, and then goes on to find best *second* moves for each of the first moves etc.



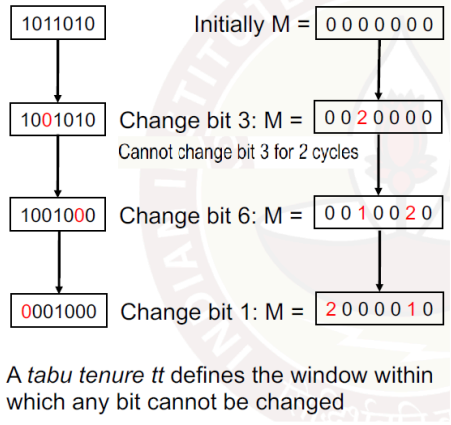
Beam search also could have different *MoveGen* (neighborhood) function that changes 2 bits instead of one bit during each move. Thus, N_1 neighborhood function

changes 1 out of 5 bits in the input state, and results in 5 $(5C_1)$ neighboring states. However, N_2 neighborhood function changes 2 out of 5 bits in the input state, and results in 10 $(5C_2)$ neighboring states.



Likewise, N_{12} can change either 1 or 2 out of 4 bits, and results in 15 neighboring states. At an extreme, N_{1-n} implies that any of the n bits (1-n) can change, thus resulting in the densest neighborhood function with 2^N neighbors.

- In a variation to *Hill climbing*, *Best Neighbor* algorithm will move to the best neighbor, even if it's not better than the current node. The algorithm will terminate on reaching an external condition. The algorithm doesn't get stuck at the local optima, but then it moves back to the same local optima in the following step.
- This can be avoided by tabu search algorithm, which restricts from making frequent moves to certain states. This is achieved by defining a tabu tenure tt and incrementing a predefined memory array M to keep track of the moves to various states, and disallowing moving back to the states already moved into for tt times.



- Typically, Tabu Search has an aspiration criteria that allows a move, if it's better than the best move so far, even if the move isn't allowed as per tt.

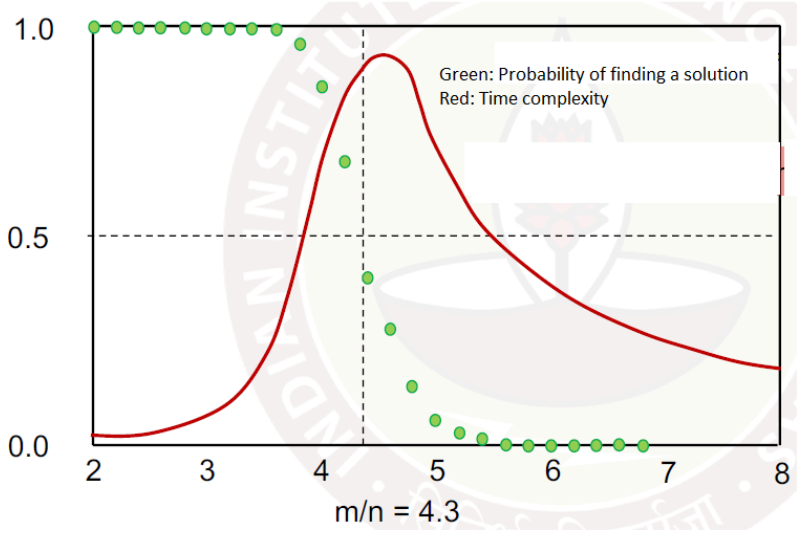
```
Best Neighbour
   N ← Start
   bestSeen ← N
   Until some termination criterion
            N ← best(allowed(moveGen (N))
            IF N better than bestSeen
                     bestSeen ← N
   return bestSeen
End
```

Move to the best neighbour anyway, *but only if it is not tabu*

- When the start nodes in *hill climbing* is randomly chosen, then it results in a stochastic algorithm called *iterated hill climbing*.
- *iterated hill climbing* can be used for configuration problems, and not for planning problems, because the latter type of problems require a well-defined start node and goal description.

Week4 – Local Search

- Any propositional/boolean formula could be converted to CNF. In CNF, each clause is connected by $\wedge$. THe clause should not have $\wedge$.
- In 2SAT problem, each clause will have 2 variables. 2SAT can be solved in polynomial time.
- 3SAT problem is solveable in non-deterministic polynomial time (NP-complete), for which worst case is exponential. mSAT has a much higher time compexity.
- Assuming that SAT problem has m clauses and n variables in each clause, probability of finding a solution reduces drastically after $\frac{m}{n} = 4.3$. At the same time, the complexity peaks. Very interestingly, after this point, it's increasingly evident that the problem cannot be solved and hence complexity reduces.



- Iterated hill climbing chooses start nodes at random for *hill climbing* over multiple iterations. There could be more than one start node in the state space that can reach

the goal.

```
ITERATED-HILL-CLIMBING(N)
1  bestNode ← random candidate solution
2  repeat N times
3       currentBest ← HILL-CLIMBING(new random candidate solution)
4       if h(currentBest) is better than h(bestNode)
5            bestNode ← currentBest
6  return bestNode
```

- *RandomWalk* function given below selects the best neighbor from a set of *n* random moves.

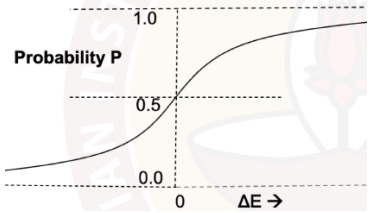
```
RandomWalk()
1  node ← random candidate solution or start
2  bestNode ← node
3  for i ← 1 to n
4     do node ← RandomChoose(MoveGen(node))
5        if node is better than bestNode
6           then bestNode ← node
7  return bestNode
```

- *Hill Climbing* does pure exploitation (of the gradient), but *RandomWalk* does pure exploration (of the gradient)
- We can combine both methods into *Stochastic hill climbling.* It moves to the neighbor (v_n) from the current neighbor (v_n) with a probability defined by $\Delta E = eval(v_n) - eval(v_c)$. If this quantity is positive and high, the probability that it'll move to v_n is high. So, a good random move is accepted with a high probability, and a bad random move is accepted with a low probability.
- Note that in *stochastic hill climbing*, we're not finding the best of all neighbors like in the case of *hill climbing*. Instead, we're moving to a *randomly chosen* neighbor if the probability calculated from the heuristic (eval) value is high.
- Sigmoids are typically used to arrive at the probability, given ΔE. To clarify,

$$P = \frac{1}{1 + e^{-\Delta E/T}}$$

$$NOTE: \ \Delta E = eval(v_n) - eval(v_c) \ for \ a \ maximization \ problem, \ and \ T \ is \ a \ parameter.$$

- When we use sigmoid to decide the probability, probability varies as shown below.



- Annealing is a method commonly used in materials to increase the stability of the materials, by way of increasing the temperature quickly and then slowly allowing it to cool down. This allows the dislocations in materials to reduce and stabilize.

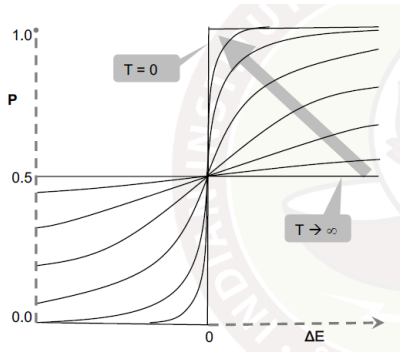
- Following is the *simulated annealing* algorithm used on *stochastic hill climbing* to arrive at the best node.

```
SimulatedAnnealing()
    node ← random candidate solution or start
    bestNode ← node
    T ← some large value
    for time ← 1 to numberOfEpochs
      do while some termination criteria     /* M cycles in a simple case */
        do
          neighbour ← RandomNeighbour(node)
          ΔE ← Eval(neighbour) – Eval(node)
          if Random(0, 1) < 1 / (1+e^(-ΔE/T))
            then node ← neighbour
              if Eval(node) > Eval(bestNode)
                then bestNode ← node
      T ← CoolingFunction(T, time)
    return bestNode
```

Stochastic Hill Climbing

Thus, in *simulated annealing* we run the stochastic hill climbing for multiple epochs, each time reducing the value of T (which increases the probability of choosing the next neighbor).

- When the temperature is the highest (say, infinity), $-\Delta E / T \approx 0$ and hence the probability is 0.5. As the temperature reduces, $\Delta E / T$ increases ($-\Delta E / T$ decreases) and hence probability increases and gets closer to 1, until at $T = 0$, it behaves like a step function.



- In genetic algorithm, out of N candidate solutions in the initial population P, select a certain number of members into a subset S based on their fitness values. Partition S into two halves and randomly mate (use cross-over method) two members selected from each half. In addition, mutate some offsprings, and replace k weakest members from P with k strongest offsprings. Repeat the entire process, and on meeting some termination criteria return the best member of P.
- For example, splitting the two parents P_1 (010110) and P_2 (111010) in the middle and performing a cross-over generates C_1 (010010) and C_2 (111110). Of this C_2 is the solution for the SAT with a heuristic value of 6.
- In the case of generic algorithms, parents separated widely by fitness function values

will gravitate towards local optimum. Cross-over, mutation and replacement of weaker members by stronger children will help towards reaching global optimum.

- Following is an example on how this algorithm works with a population of 4 nodes.

Initial	Binary	f(x)	Prob.	Expected	Actual	Selected
01101	13	169	0.14	0.58	1	01101
11000	24	576	0.49	1.97	2	11000
01000	8	64	0.06	0.22	0	11000
10011	19	361	0.31	1.23	1	10011

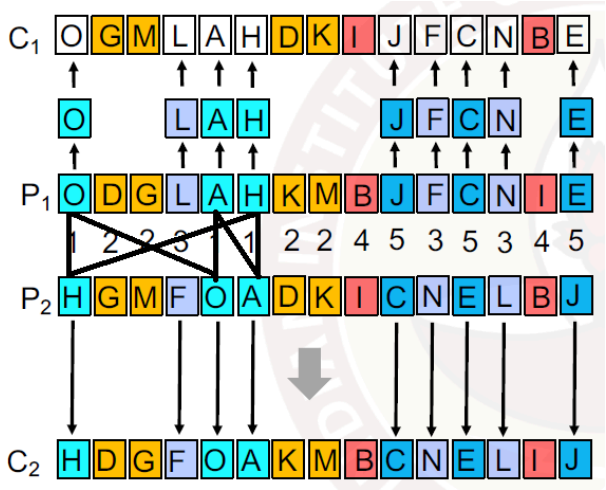
Total: 1170, Avg: 293

Selected	Crossed-over	Binary	f(x)
01101	01100	12	144
11000	11001	25	625
crossover			
11000	11011	27	729
10011	10000	16	256

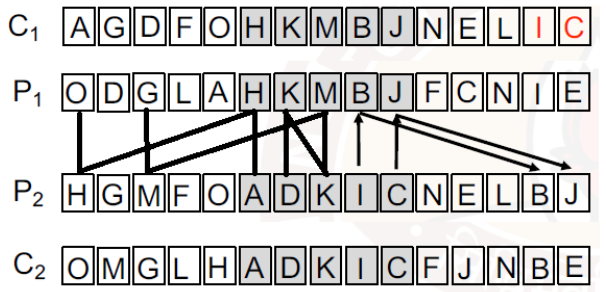
Total: 1754, Avg: 439

NOTE: Here, the fitness function $f(x)$ squares the given string.
$Probability = f(x) / \Sigma f(x)$. $Expected = f(x) / mean(f(x))$. $Actual = round(Expected)$.
The nodes from the initial population are copied $< Actual >$ number of times.

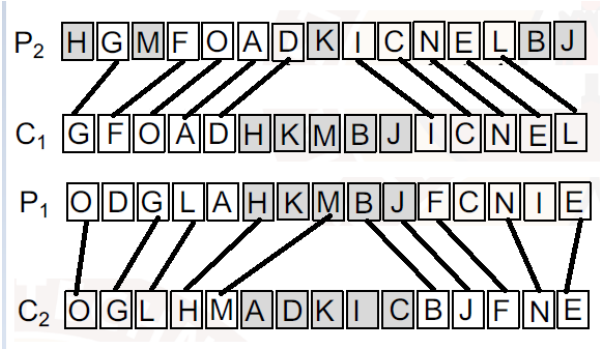
- In the above example, the population has become fitter by applying the fitness function $f(x)$, as evident from the increasing $Average$ value. During the next iteration too, the population will become fitter, but less diverse. See the lecture slides for details about next iteration.
- A large diverse population is critical to the performance of the algorithm.
- In the case of TSP, it's possible to use the genetic algorithms with some modification. Note that single-point cross-over will not work, unlike with SAT, because it could cause repetition of same city multiple times in the tour, which is NOT allowed.
- In the first version to the algorithm (called cycle crossover) used for TSP,
  - identify the cycles among the parents as follows. Observe that O from P_1 maps vertically to H from P_2, then diagonally to H in P_1, vertically to A in P_2 and so on, until it completes the cycle. Mark all nodes in cycle-1 as 1. Similarly, find the cycle-2. Mark all nodes in cycle-2 as 2. Repeat this for all nodes.
  - C_1 get odd numbered cycles from P_1 and even numbered cycles from P_2
  - C_2 get even numbered cycles from P_1 and odd numbered cycles from P_2



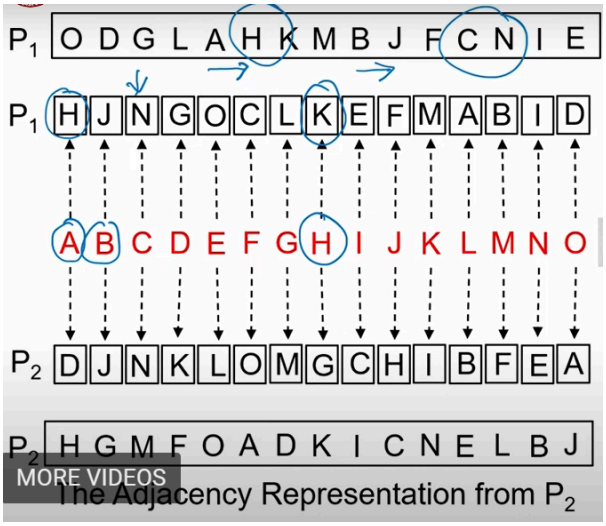
- In the second version of this algorithm (called partial crossover) for TSP,
  - find a subtour from P_1 and copy to C_1
  - Copy rest of the nodes from P_2, but some of these locations in C_1 already have copied nodes from P_1. Hence, we must create a mapping in similar manner as with the previous algorithm.
  - find a subtour from P_2 and copy to C_2
  - Copy rest of the nodes from P_1, but some of these locations in C_2 already have copied nodes from P_2. Hence, we must create a mapping in similar manner as with the previous algorithm.



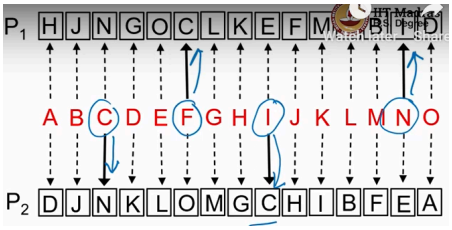
- In the third version of this algorithm (called order crossover) for TSP,
  - find a subtour from P_1 and copy to C_1
  - Copy rest of the nodes from P_2 in the same order they occur.
  - find a subtour from P_2 and copy to C_2
  - Copy rest of the nodes from P_1 in the same order they occur.



- In adjacency representation of the tours, cities could be arranged based on where they come from in the tour w.r.t the index.

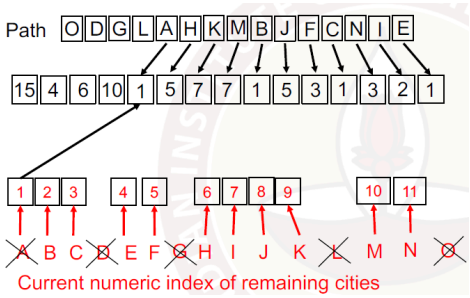


- In the Alternating edges algorithm for TSP,
  - Assume that the parents are in adjacency representation.
  - While creating C_1, choose the next node from alternate parents.



NOTE: Be careful about this, since it can lead to an invalid tour. For example, in the above case, if we start with F, we get $CNIC$ (not a valid tour).

- In ordinal representation, we use numbers (instead of index labels) and map the nodes. One thing to note is once a label is mapped, it's removed from the index list.



One advantage of ordinal representation is that single-point crossover always produces a valid offspring.

- Collection of simple entities organize themselves and a larger more sophisticated entity emerges leading to the concept of emergent systems. Game of Life is an example, which is governed by a few rules and causes illusion of continuous movement. Another example is fractals, which are never ending patterns and self-similar across different scales.
- In ANNs, there could be several hidden layers apart from the input and output layer. Each layer will have several neurons connected to neurons in another layer using weights in a criss-cross manner. These weights are learnt using an algorithm called *BackPropagation*.

Week5 – B&B and A*

- Solutions with the least number of steps may not be *the* optimal solution.
- *Brute Force/British museum procedure* explores the entire search space and returns the optimal solution.
- Branch and Bound (B&B) starts by defining an intial solution set containing *all* solutions. In each step, divide the set into two smaller sets, until you pick a solution that's fully specified. Each solution set needs to have an estimated (positive) cost, that serves as a *lowest-bound (LB)* on the actual cost. If a fully refined solution is cheaper than a partially refined one, discard (prune) the partially refined solution.
- For the initial *LB* computation, locate 2 lowest costs from each each row in the table, sum them up (2 per row) and divide by 2.

	Chennai	Goa	Mumbai	Delhi	Bangalore
Chennai	0	800	1280	2190	360
Goa	800	0	590	2080	570
Mumbai	1280	590	0	1540	1210
Delhi	2190	2080	1540	0	2434
Bangalore	360	570	1210	2434	0

$$LB = \frac{(360 + 800) + (570 + 590) + (590 + 1210) + (1540 + 2080) + (360 + 570)}{2}$$

$$= \frac{8670}{2} = 4335$$

Note that it's not necessary that this method will be the final tour, or even help at identifying the final tour for the following two reasons.

  - It could turn out that it's impossible to construct a tour without choosing a particular longer edge (or more than one) that hasn't participated in the above *LB* computation.
  - In the above LB computation, we included 3 edges from Bangalore, which itself renders the tour invalid.

  It's not necessary to consider the above points at this time and proceed with *pruning* the tour by including/excluding specific segments instead. This might cause a longer search, but will eventually find the optimal tour.
- As segments get fixed in the tour, their known costs are included in the *LB* computation. This makes the estimate more accurate, but could invalidate certain previously selected segments, if it creates a fork/loop with the newly added segments. This would require to include cities with next higher cost from each row of the table, followed by *LB* computation again.
- Note that B&B always expands the cheapest leaf node.
- While using B&B in searching through a state space, following this procedure:
  - Create a search tree with the start node at the root.
  - List all its neighbors and identify the best (nearest from source) node and its distance from the root.
  - Expand the cheapest node, list its neighbors, and again identify the best node and its distance from the root.
  - Repeat this until the goal state is reached.
  - It's possible that there exists multiple paths to G, explore all of them.
- How to solve B&B problems?
  Consider the following TSP problem (May '23 quiz2)

The distance matrix for 5 cities (A to E) is provided below. From each city the distances to other cities are listed in ascending order. For example, the distance from A to C is 56, from A to D is 80, and so on.

A	→	C:56	D:80	B:110	E:118
B	→	D:40	E:40	C:80	A:110
C	→	D:40	A:56	B:80	E:106
D	→	B:40	C:40	E:72	A:80
E	→	B:40	D:72	C:106	A:118

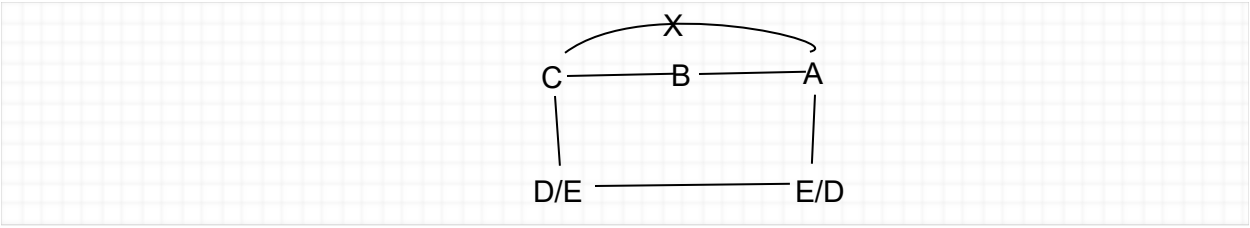
Solve the sub-questions using the TSP Branch-and-Bound algorithm.

Attention: Infer as much as possible (and as early as possible) about the permanent edges in the partial solutions. The semantics of the permanent edges (which are bidirectional) are as discussed in the practice assignments.

Lower bound on the cost of the tours (S0) is calculated by taking the average of multiple edges (two lowest cost edges per node).

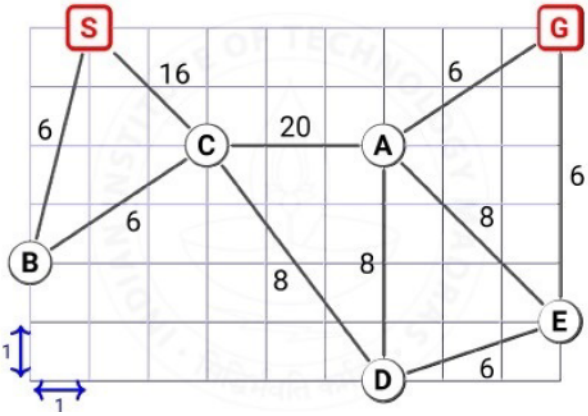
$$\frac{(AC + AD) + (BD + BE) + (CD + CA) + (DB + DC) + (EB + ED)}{2} = 252$$

To infer all PI edges in the node $(S0, \neg BD, \neg BE)$ in the B&B tree, do the following. Since B must have two edges from it, and cannot be to $D\ or\ E$, they must be $C\ and\ A$. In addition to BD and BE, AC is an additional excluded edge, since it would make a sub-tour. Next, C will be connected to either $D\ or\ E$ and A will be connected to either $D\ or\ E$. In either case, DE will be a mandatory edge. Thus, PI will have 3 edges - $BA,\ BC\ and\ DE$. Similarly, PE will have 3 edges - $BD,\ BE\ and\ AC$.

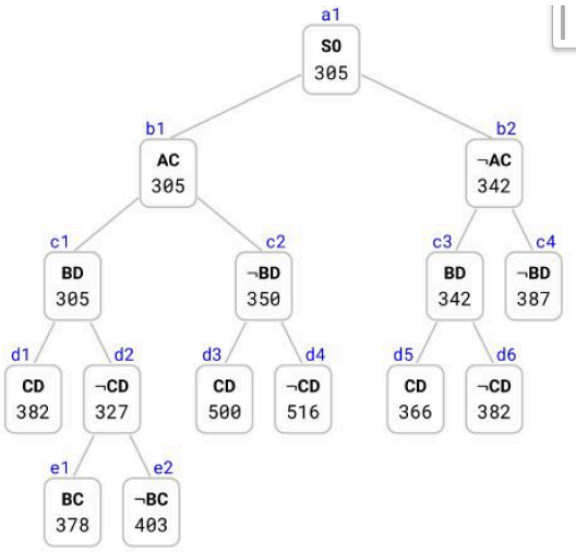


- Djsktra's algorithm, unlike B&B, doesn't add explore as many paths.
  - Assign ∞ to all nodes except the start node.
  - Calculate the distance to all neighbors from the source node. Update the marked distance of the nodes if the calculated distance is lesser than the marked distance.
  - Repeat this process, until the goal state is reached.
- B&B (as well as Djsktra) has no sense of direction and recursively explores the nearest nodes first, before exploring nodes that are farther (one-hop).
- Unlike B&B, Djkstra keeps only one copy of each node and adjusts the parent pointer when it finds a better path to a node.
- Algorithm **A*** combines the best features of B&B and Best first (heuristic-based) search.
- In **A*** algorithm, each candidate is tagged with an estimated cost of the complete solution $f(n) = g(n) + h(n)$, where $g(n)$ is the known cost of path found from the start node to node n (as defined by B&B or Djkstra's algorithm) and $h(n)$ is the estimated cost from the node n to the goal node.
- The algorithm works as follows:
  - Maintain a priority queue of node sorted on f-values.
  - Pick the node with the lowest f-value and if it's goal return the path
  - else, generate its neighbors and compute f-values.
  - Insert new nodes into the priority queue.
  - For existing nodes, check for better path.
- **A*** handles closed nodes differently than Djsktra's. While Djkstra doesn't revisit closed nodes, **A*** revisits them and if necessary, propagates improvement to all its neighbors, recursively.

- Inspite of revisiting closed paths to discover newer paths to the goal state, **A*** guarantees to find the cheapest path to goal state. This is because, it works with an estimated cost $h(n)$, while Djkstra works with known cost $g(n)$
- If $g^*(n)$ is the optimal path cost from S to n, $g^*(n) \leqslant g(n)$ because the algorithm may not have found the optimal path yet.
- If $h^*(n)$ is the optimal heuristic cost from n to G, either the actual heuristic underestimates ($h(n) \leqslant h^*(n)$) or overestimates ($h(n) \geqslant h^*(n)$) the distance to the goal.
- It can be proved that underestimating the $h(n)$ will help in finding the optimal path to goal node.
- A search algorithm is said to be admissible if it's guaranteed to return an optimal solution if there exists one. **A*** is admissible. The conditions for admissibility are:
  - Branching factor is finite. Note that the state space could still be infinite.
  - Heuristic function $h(n)$ should underestimate the cost.
- Given a state space graph, the given heuristic (say Manhattan distance) is considered admissible if it never overestimates the true cost to reach the goal from any node. In other words, the heuristic value ($h(n)$) for a node should always be less than or equal to the actual shortest path cost from that node to the goal. For example, in the following graph, the shortest path to reach G from C is $C \rightarrow A \rightarrow G$, which costs 26. h(C) = 8. Thus, $h(C) < actual(C)$. Repeat this for all nodes in the graph. If all nodes follow the rule $h(n) < actual(n)$, then the given heuristic is admissible.



- If the heuristic function is more informed (closer to the optimal heuristic $h^*(n)$), the search will be faster.
- If a path exists to the goal node, then the OPEN list always contains a node n from an optimal path. Moreover $f(n)$ is not greater that the optimal cost.
- **A*** algorithm finds the optimal path (and least cost) always, even if the state space is infinite.
- For the TSP B&B tree given below, answer the following questions.



- Find the number of cities. To solve this, find the longest path in the tree: b1-c1-d2-e1 and b1-c1-d2-e2. Of these, the first one corresponds to A-C-B-D and the second corresponds to B-D-A-C. In the first one, there is no further requirement, but the second one requires that there is city E, so that BC is excluded. Thus, we can conclude that there are 5 cities.
- After S0, identify the next 4 nodes (2nd to 5th node) that are refined by the TSP BnB algorithm. To solve this, arrange the nodes in increasing order of the associated cost. The nodes 2-5 in this ordered list is the answer. Thus, in this case, b1, c1, d2,b2.
- Which node represents the optimal tour? To solve this, locate the leaf node with the lowest associated cost. In this case, d5.

Week6 - Variations of A*

- In weighted A*, $f(n) = g(n) + w * h(n)$. When $w = 0$, it's similar to B&B which uses only g(n) to search and no heuristic. When $w = \infty$, $g(n)$ is effectively zero and is similar to Best First Search.
- Best First Search and *wA** algorithms are more focused on the goal, and move towards it faster. However, *A** and B&B algorithms take a conservative approach and keep looking out for paths with a lesser cost.
- *wA** could find a path to the goal faster than *A** and by exploring fewer nodes, but at a higher cost. Hence *wA** doesn't find optimal path like *A**.
- Best First Search is the fastest algorithm, but finds the path with the highest cost among all algorithms.
- For the state space given in Fig.1, the next 4 figures show the search for the goal state conducted using different algorithms, and the table below that summarizes the result of each.

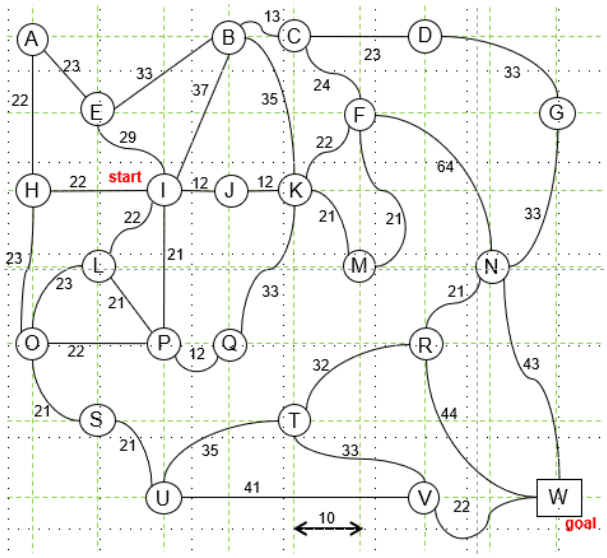


Figure 1: State space

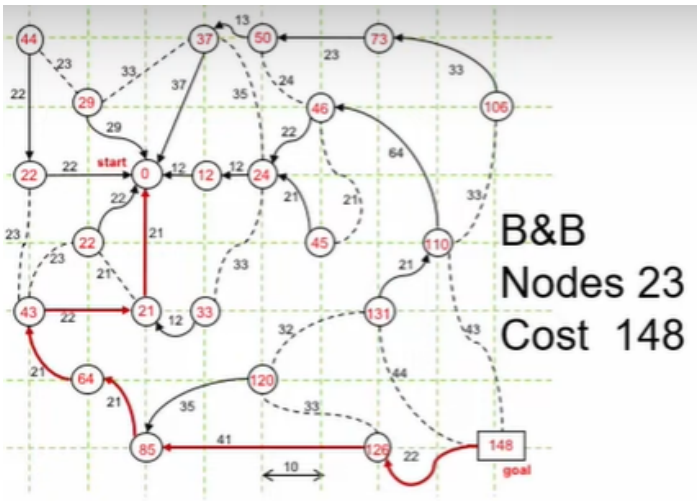


Figure 2: B&B

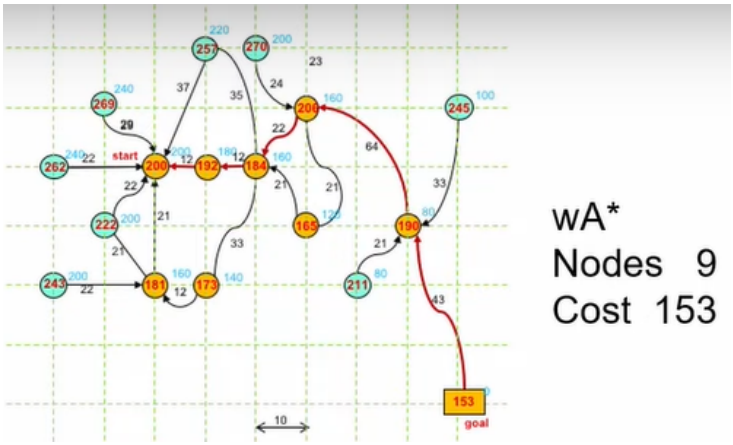


Figure 3: wA*

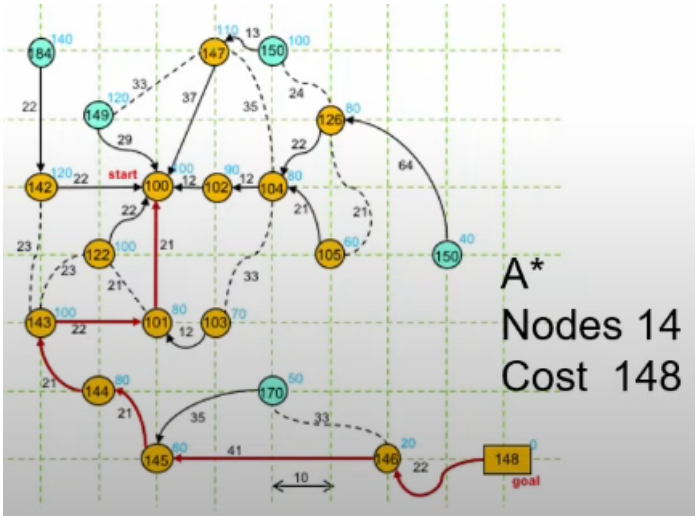


Figure 4: A*

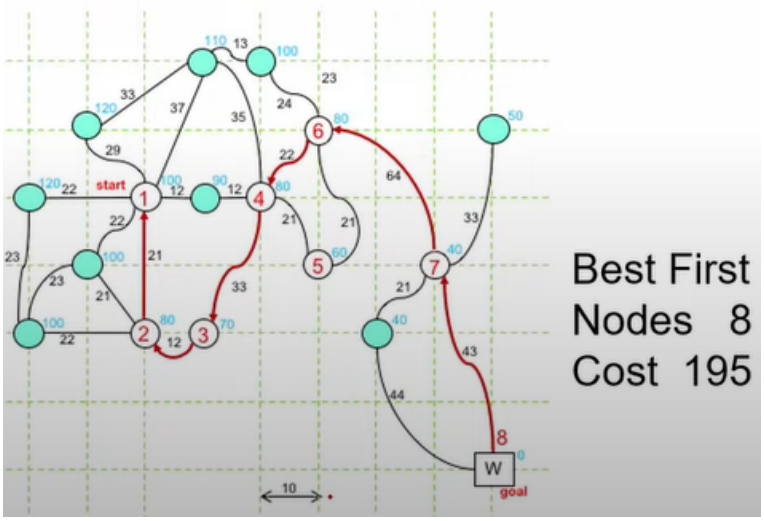
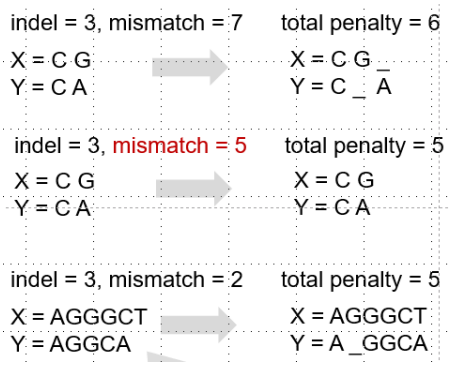


Figure 5: Best First Search

Weight w	Algorithm	Nodes generated	Nodes inspected	Cost of path
0	Branch & Bound	23	23	148
1	A*	19	14	148
2	Weighted A*	17	9	153
∞[1]	Best First Search	17	8	195

- Note that f-values of the nodes increases as you go move towards the goal, in the case of *A**. This is because, if f-value was lesser than the previous node, it would've explored a different path. In the case of *wA*,* f-values of successive nodes don't increase, but might decrease for higher weights.
- A more informed (or accurate) heuristic gives a closer estimate to the actual cost, which can significantly reduce the number of nodes explored.
- A well-informed heuristic provides tighter bounds on the cost, allowing the algorithm to prioritize paths more effectively. With better estimates, the algorithm can avoid exploring paths that are less likely to lead to the optimal solution, focusing on the most promising paths. This reduces the overall computational effort, as fewer nodes are expanded, leading to faster convergence to the optimal solution.
- In *wA*,* we can further reduce the search space, but at the risk of losing admissibility.
- The monotone (consistency) property for a heuristic function says that for a node n that is a successor to node m on a path to the goal being constructed by *A** using heuristic function $h(x)$, $h(m) - h(n) \leqslant k(m, n)$, where k represents the the edge cost between the two nodes. This is also referred to as the triangle inequality, where sum of $h(n)$ and $k(m, n)$ is less than the third side of the triangle $h(m)$. For *A**, one of the consequences of this condition is that every node it picks for exploration happens to be in the optimal path.

- In terms of space complexity, Hill Climbing < Simulated annealing < DFS < DFID < Best Neighbor Search < Beam Search < Best First Search < A* < wA* < BFS < B&B < Tabu Search < Generic algos.
- In terms of time complexity, Hill Climbing < Simulated annealing < Beam Search < Best Neighbor Search < DFS < DFID < BFS < Best First Search < wA* < A* < B&B < Tabu Search < Generic algos.
- *Iterative Deepening A* (IDA*)* adopts DFID and attempt to reduce the space complexity of *A** further. The only difference is that instead of using depth as the parameter, *IDA** uses f-values to determine how far should DFS go. *IDA** initially sets the bound to f(S) and in each iteration gets updated to the *f(N)* of the cheapest unexpanded node on OPEN.
- In a variation of *IDA*,* instead of incrementing the bound to the next *f(N)*, it could be incremented by a constant δ.
- RBFS explores new nodes in the best-first order recursively, and expands fewer nodes than IDA*, but rolls back to the second-best node at the first level. At any level, if it encounters a neighbor with a higher heuristic than the second best, it recursively updates its parent to the lowest among the neighbors until the first level.
- Sequence alignment problems are fundamental to bioinformatics. It's used to compare amino-acids.
- Objective of sequence alignment is to maximize similarity between two sequences with gaps inserted into them. Cost of algnment between two given strings is computed using the following penalties/costs: Mismatch (between the characters), Indel (inserting a gap). THus, the following cases demonstrate possible reduction of total penalties by changing the strings X and Y.



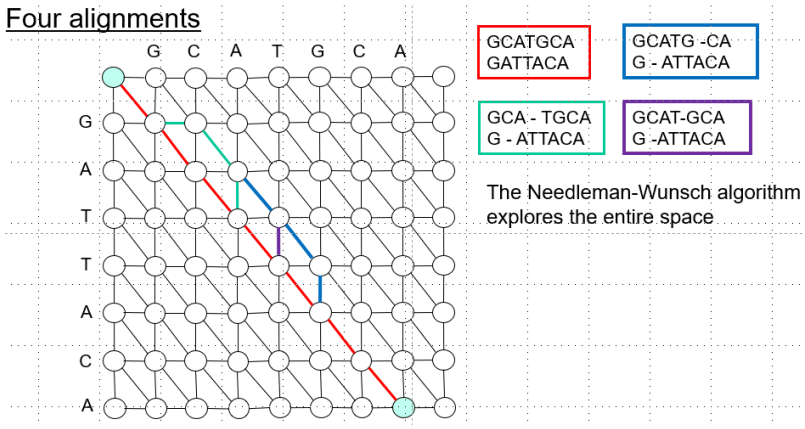
- In a fine-grained similarity function, highest weight of aligning a character with itself between two strings could be different than in the case of another character. For example, suppose the alignment weights are represented by the following table. Assuming that the indel penalty I is -5.

	A	G	C	T
A	10	-1	-3	-4
G	-1	7	-5	-3
C	-3	-5	9	0
T	-4	-3	0	8

In this case, the alignment between the strings $AGACTAGTTAC$ and $CGA---GACGT$ is given by the following computation.

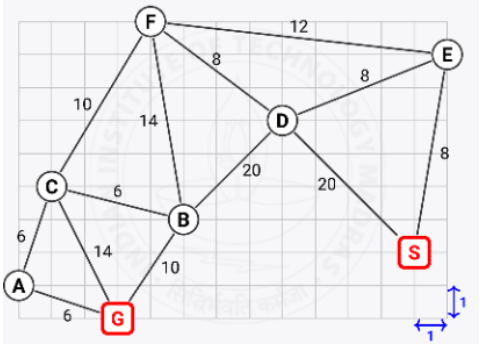
$$S(A,C) + S(G,G) + S(A,A) + 3 * I + S(G,G) + S(T,A) + S(T,C) + S(A,G) + S(C,T)$$
$$= -3 + 7 + 10 + (3 * -5) + 7 + (-4) + 0 + (-1) + 0$$
$$= 1$$

- Similarity and distance functions are inverses of each other.
- Sequence alignment can be represented as a graph search problem, where we account for 3 possibilities:
  - Align X with Y (diagonal move)
  - Insert gap before X (vertical move)
  - Insert gap before Y (horizontal move)

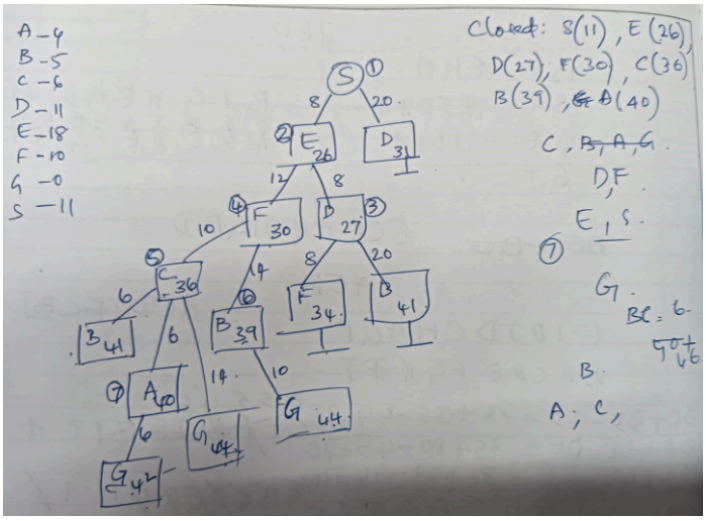


- Solving sequent alignment problem using A* will result in $(N+M)!\,/\,(N!M!)$ number of ways that gaps can be inserted, moving only horizontally or vertically.
- OPEN grows linearly, and CLOSED quadratically in a sequence alignment problem with A*. Hence, it's important to prune CLOSED in order to reduce space.
- *A**, in general, has exponential time complexity. $O\left(b^d\right)$ is the time it takes to explore the nodes, where b is the branching factor, d is the depth of the solution. Accuracy of the heuristic - specifically when admissible and consistent (satisfies the triangle inequality), however, can significantly reduce the number of nodes explored.
- One of the purposes of keeping the CLOSED list at each step of the search is to avoid visiting the *closed* nodes again and thus ending up in an infinite loop. Instead, maintain a barred (tabu) list for each node that contains a list of nodes that'll not be added as its neighbors. In this algorithm, we need only OPEN and hence called frontier search.

- However, we can't entirely remove the concept of CLOSED because it helps in reconstructing path to the goal, when we've found one. One alternative is to keep one or more RELAY layers, at fixed depths, using a divide and conquer approach. In order to reconstruct the path, two recusive calls are made - one from *Start* to *Relay* and another from *Relay* to *Goal*.
- *Modified Beam Search can* be used in the context of finding the goal node, wherein the algorithm will choose the w best successors of a node, even if they're not better. In this case, we'll terminate at finding the goal.
- *Modified Beam search* is guaranteed to find the path to goal, but may not be optimal. We'll fix an upper bound U on the cost of the optimal path, and hence all nodes with $f(n) > U$ can be pruned.
- Total space required by beam search = *width * depth.* Note that beam search is not admissible.
- In *Beam Stack Search*, at level k, only those nodes are permissible that are between f_{min} and f_{max}. From the next level $k+1$, it can backtrack to a previous level to generate nodes at level $k+1$ between a different f_{min} and f_{max}. Note that, new f_{min} at level $k+1$ = f_{max} at level k.
- Perform SMGS on the following state space and answer the questions below



Q1.Construct the search space.



See PA2 for Week6. For more details, refer to https://discourse.onlinedegree.iitm.ac.in/t/week-6-pa-2-smgs-question-6/140321/3

- How to find boundary/kernel nodes after inspecting a particular node?
  After inspection of each node, append it to the CLOSED list.
  CLOSED = BOUNDARY + KERNEL. Note that BOUNDARY and KERNEL sets are disjoint. Boundary nodes comprise of neighbors (from the state space) of currently open nodes, from the CLOSED set. Kernel nodes are remaining nodes from the CLOSED set.
- Comparison of all search algorithms we've seen so far.
  - DFS, BFS are purely based on MoveGen. No edge costs or heuristic. Selection of next neighbor is based on alphabetical order. Note that if a node is present in Open/Closed list already, it's not considered again.
  - Given a finite state space with unit edge costs, and a heuristic function whose properties are not known, the following algorithms are suitable to find the optimal path - BFS, Djkstra, B&B. Without admissible heuristics, the following algorithms cannot find the optimal path - A*, wA*, SMGS. DFS, with or without heuristic, cannot find optimal path.
  - Given a state space with Euclidean edge costs, and with the best admissible heuristics, the following algorithms are suitable to find the optimal path - Djkstra, B&B, A*, SMGS. Note that BFS can find optimal path, only if the edge cost is 1.
  - Depth First Iterative Deepening (DFID) guarantees an optimal solution, when it inspects new as well as closed nodes
  - Best First Search is based on heuristic.
  - B&B is based on edge costs. No heuristic. Closed nodes shouldn't be added to the priority list again. A*, wA* is based on edge costs and heuristic.
  - In B&B, even after the goal state is reached, continue exploring more paths to see if there are any with lesser cost.

Week7 – Games

- Games typically involve multiple agents. It's the science of strategy, or at least the optimal decision-making of independent and competing actors in a strategic setting.
- Each player's payoff is contingent on the strategy implemented by the other player.
- The basic assumption is that the players are rational and will strive to maximize their payoffs.
- Nash equilibrium is an outcome, which when reached in a game, no player can increase payoff by changing decisions unilaterally.
- Games can be either zero sum, positive sum or negative sum games. In the first type, some players may gain while other lose. Typical board games like chess are of this type. In the second, most or all players gain due to the cooperation between them. Some strategy-based games are of this type. In the third, most or all players lose. A good example are price wars between competitors.
- Certain games can have incomplete information, thus leading to uncertainty. Card games, or games that use dice to decide the number of moves are the best examples.
- *Minimax* value is the outcome of a game when both players play rationally. *Minimax* is a depth-first algorithm and computes the highest among the heuristic values of its children for the *Max* player, and lowest among the heuristic values of its children for the *Min* player. THis is done recursively along the depth, until *Minimax* value is computed for the root node of the tree. This is also known as the Nash equilibrium value of the tree.
- A strategy for a player is a subtree of the game tree that completely specifies the choices for that player. The algorithm below constructs a strategy for *Max:*

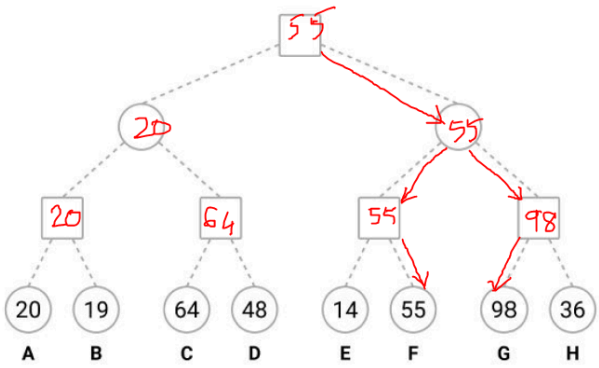
```
CONSTRUCT-STRATEGY(MAX)
1  traverse the tree starting at the root
2  if level is MAX
3       choose one branch below it
4  if level is MIN
5       choose all branches below it
6  return the subtree constructed
```

- Since it's impossible to analyse a game like chess (with a branching factor of 35 and depth of ≈ 40, which will take 35^{2*40} computations) entirely, we will analyse it by looking ahead for say, 4 plys, at a time. Once the *Max* moves, we'll wait for the *Min* to move and then repeat these steps until the game is over.
- Evaluation function h in a game is typically computed by adding up features capturing material strength, along with features representing positional strength. It's in a range [$-\infty, +\infty$]. Positive values are considered good for *Max*, negative ones are considered good for *Min* and 0 is a draw.
- *Max* nodes are also called *alpha* nodes, and *Min* nodes are also called *beta* nodes. Alpha value is the value found so far for the *alpha* node, and it's a lower bound on the

value of the node since it cannot be less than the current *alpha* value. Beta value is the value found so far for the beta node, and it's a upper bound on the value of the node since it cannot be higher than the current beta value.

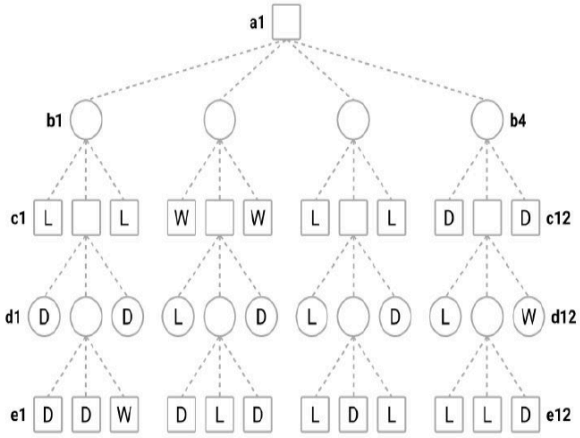
- In a game tree, for a *Min* parent, leaf node will influence the root only if $\alpha_{leaf} < \beta_{parent}$, or in general $\alpha < \beta$. α-cutoff is induced by an α-bound from a Max ancestor below a β-node. If $\alpha \geqslant \beta$, then return the lower bound α.
- In a game tree, for a *Max* parent, leaf node will influence the root only if $\beta_{leaf} > \alpha_{parent}$, or in general $\beta > \alpha$. β-cutoff is induced by an β-bound from a Min ancestor below a α-node. If $\beta \leqslant \alpha(\alpha \geqslant \beta)$, then return the upper bound β.
- With alpha-beta pruning, the number of nodes inspected is lesser than MiniMax. So, it could take lesser time. For a simple explanation of how to perform alpha-beta pruning, watch https://www.youtube.com/watch?v=_i-lZcbWkps
- Both Minimax and alpha-beta pruning algorithms are blind algorithms and always proceed in the same direction. SSS* is an algorthm with a direction as provided by the heuristic.
- There exists 8 strategies for in 4-ply game tree. Assuming that the root node represents *Max,* for each of its children (L and R), there exists 4 strategies - LL, LR, RL, RR. Each of the 16 leaf nodes in a 4-ply search represents two strategies each. Thus, in the below figure, $e3$ (highlighted node with the heuristic value of 15) represents a cluster of two strategies - LRL and LRR. Note that $e3$ has the upper bound of the strategies associated with it.
- SSS* was devised by George Stockman in 1979 and uses best first search. It refines the best looking partial solution in a loop, until the best solution is fully refined. A solution in a game tree is a strategy and partial solution is a partial strategy and stands for a cluster of strategies.
- There are two steps involved in SSS*.
  - Add root node to priority queue and continue till root is SOLVED and at the head.
  - Construct the initial clusters by recursively choosing all children at Max level, and left-most child at Min level.
  - Add the leaf nodes that represent initial clusters to a priority list in the descending order of heuristic value and pop them out one at a time. Thus, pop out the node with highest heuristic each time.
  - Next, refine the strategies in which the chosen node participates. Refinement is done for a Max node by selecting its sibling and adding to the priority list before popping it out, until there are no more siblings, and backing up one level. For a Min node, just pass on the node value to the parent without bothering about siblings. If $\alpha \geqslant \beta$, cut off the corresponding branch.
  - Add more clusters as required and refine the strategies, by recursively choosing all children at Max level, and left-most child at Min level.
- How to find the horizon nodes for the best strategy?

Consider the following graph



In the given graph, backup all minimax values. Now, choose best node from Max and all nodes from Min. At $a1$, the best next node is $b2$. At $b2$, choose both $c3$ and $c4$. At $c3$, best next node is $d6$ (F), and at $c4$, best next node is $d7$ (G). Thus, answer to the question is nodes F and G.

- How to find the don't care values among graph nodes?
  Consider the following graph. Evaluate the game tree in depth-first left-to-right order, for this evaluation order identify the don't care nodes in level c, nodes c1 to c12.



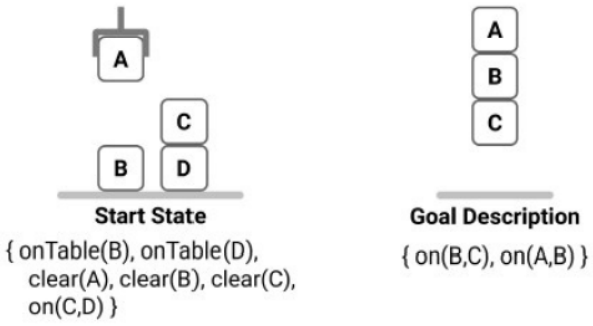
c1 is L, which means b1 will be L. So, c2 and c3 are don't cares.
c4 is W, c5 is D (found by backing up minimax values below the level), c6 is W. None of them are don't cares.
c7 is L, c8 and c9 are don't cares.
c10 is D. c11 is W (found by backing up minimax values below the level), c12 is D. None of them are don't cares.

Week8 – Planning

- FSSP algorithm is shown below.

```
FSSP(S0, G)
1  S ← S0
2  π ← empty plan
3  loop
4      if G ⊆ S
5          return π
6      actions ← set of applicable actions in S
7      if actions is empty
8          return failure
9      choose a ∈ actions          ▷ choose an action
10     S ← γ(S, a)                 ▷ progress to new state
11     π ← π ∘ a                   ▷ update plan
```

- FSSP
  - lists the necessary steps for generating a plan.
  - operates over a search space where each node represents a *state*.
  - doesn't specify MoveGen and search strategy
  - searches for applicable actions from the start state S ($pre(a) \subseteq S$).
  - constructs the plan π from the start state, where $S_n = \gamma(S_0, \pi)$. Note that each interim state $S_i = \gamma(S_{i-1}, a_i) = \left\{S_{i-1} \cup effects^+(a_i) \setminus effects^-(a_i)\right\}$ where a_i is the interim action.
  - the first action found is the first action in the plan
  - suffers from high branching since S is a full description
- *Applicable actions* are those actions that are possible in the current state. *Relevant actions* are those actions that helps the plan reach the (sub)goal from the previous state. Consider the following example:



  In this case, applicable actions include Stack(A, B), Stack(A, C) and Putdown(A), since these are the only possible actions in the given (start) state. Now, relevant actions at the goal state include Stack(A,B) and Stack(B, C), because these are the only actions that could reach the goal state from its previous state.
- BSSP algorithm is given below.

```
BSSP(S0, G)
1  G' ← G
2  π ← empty plan
3  loop
4      if G' ⊆ S0
5          if G ⊆ γ(S0, π) then return π
6          else return failure
7      actions ← set of relevant actions for G'
8      if actions is empty
9          return failure
10     choose a ∈ actions          ▷ choose an action
11     G' ← γ^-1(G', a)            ▷ regress to new goal
12     π ← a ∘ π                   ▷ update plan
```

- BSSP
  - operates over a search space where each node represents a *goal*.
  - searches for relevant actions from the goal G. Action a is considered as relevant if $\{effect^{+}(a) \cap G\} \neq \phi \wedge \{effects^{-}(a) \cap G = \phi\}$
  - construct the plan π from the the goal, where $G_0 = \gamma^{-1}(G_n, \pi)$. Note that each interim/sub goal $G' = \gamma^{-1}(G, a) = \{G \setminus effects^{+}(a) \cup pre(a)\}$, where a is the interim action
  - the first action found is the last one in the plan
  - suffers from spurious sub-goals
- GSP algorithm is given below

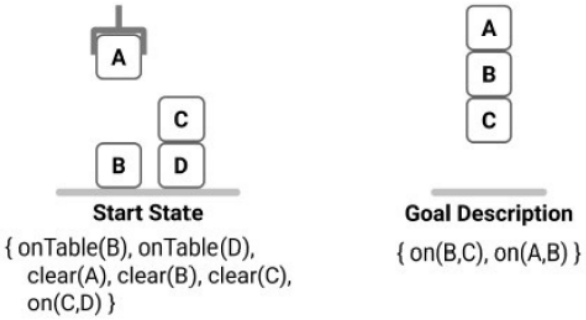
```
GSP(givenState, givenGoal, actions)
1  π ← empty plan
2  stack ← empty stack
3  S ← givenState
4  PUSHSET(givenGoal, stack)
5  while stack is not empty
6      X ← pop stack
7      if X is an action a
8          π ← π ∘ a
9          S ← PROGRESS(S, a)
10     else if X is a compound goal G, and G is not true in S
11         PUSHSET(G, stack)
12     else if X is a goal g, and g is not true in S
13         a ← choose a relevant action that achieves g
14         if a is null then return failure
15         push a to stack
16         PUSHSET(pre(a), stack)
17 return π

PUSHSET(G, stack)
1  push G to stack
2  for each goal g in G
3      push g to stack
4  return

PROGRESS(S, a)
1  return (S ∪ add-effects(a)) \ del-effects(a)
```

- Goal Stack Planning (GSP)
  - uses a stack to maintain the linear nature of the plan, and uses $PushSet()$ to push actions/goals into it.
  - breaks up a compound goal into individual goals, and solve them one at a time linearly.
  - first pushes a compound goal $G = \{g_1, g_2, ..., g_n\}$ *or* $(g_1 \wedge g_2 \wedge ... \wedge g_n)$, followed by each of the individual goals in some order, and solves them in *LIFO* order.
  - When a goal g_i is popped from the stack, if $g \in S$ no action is taken irrespective of whether the goal is compound or atomitc.
  - When a goal g_i is popped from the stack, if $g \notin S$ and g is a compound goal, push it into the stack. if $g \notin S$ and g is atomic, an action (chosen non-deterministically and with backtracking) that achieves the goal is pushed into stack.
  - When an action is popped, adds it to the plan and applied to the current state.
  - searches in a goal directed backward manner

  - has low branching
  - construct plans in a forward manner, with each action added to the end of the current plan.
- How to solve a problem using GSP?
  - Push relevant action to meet the (sub)goal
  - Push the pre-conditions
  - Unstack the entries in the stack one by one in LIFO manner. If the unstacked entry is a (sub)goal, add to the plan. If the unstacked entry is a pre-condition already met by the current state, ignore. If not, repeat from first step.
  - If the initial compound goal is in the *wrong* order, GSP won't give an *optimal* plan. For example, consider the following start state and goal state.



The given goal state $\{on(B, C),\ on(A, B)\}$ will result in a stack that has $on(A, B)$ at the top. This will add $stack(A, B)$ to the plan first. This is not optimal, as is evident from a manual inspection. In the optimal plan, the following actions must be used in the given order: $putdown(A),\ pickup(B),\ stack(B, C),\ pickup(A),\ stack(A, B)$

- Listed below are the pre-conditions for each of the actions
  - $stack(A,\ B) :\ clear(B),\ holding(A)$
  - $unstack(A,\ B) :\ on(A,\ B),\ armEmpty,\ clear(A)$
  - $pickup(A) :\ onTable(A),\ clear(A),\ armEmpty$
  - $putdown(A) :\ holding(A)$
- Listed below are the after-effects for each of the actions
  - $stack(A,\ B) :\ on(A, B),\ \neg clear(B),\ armEmpty$
  - $unstack(A,\ B) :\ clear(B),\ holding(A),\ \neg armEmpty,\ \neg on(A, B)$
  - $pickup(A) :\ holding(A),\ \neg armEmpty,\ \neg onTable(A)$
  - $putdown(A) :\ \neg holding(A),\ armEmpty,\ onTable(A)$

  NOTE: All these cases are with respect to a one-armed robot. In the case of a multi-armed robot, $clear(A)$ is an after-effect to $stack(A,\ B)$ and $putdown(A)$. Similarly, for a multi-armed robot, $\neg clear(A)$ is an after-effect to $unstack(A,\ B)$ and $pickup(A)$.
- Plan Space Planning (PSP)
  - Constructs a partial plan. A partial plan is a 4-tuple <A, O, L, B>, where A is a set

of partially actions in the plan, O is a set of ordering links, L is a set of causal links, B is a set of binding constraints.
  - Parial plan could have two kinds of flaws - open flaws, wherein an action is unsupported by a causal link and threats, wherein one action might delete an effect before it's consumed by another action.
  - Threats can be resolved by separation (where we disallow the variable to refer to specific value), promotion (where we plan for an action before another) or demotion (where we plan for an action after another).

Week9 – Graph plan

- The concept of mutex (mutual exclusion) in planning graphs is used to determine when two actions (or propositions) cannot be executed simultaneously. Following are 4 reasons for actions a and b to be considered mutex.
  - Competing needs: pre-conditions of actions a and b are mutex.
  - Inconsistent effects: a adds a proposition (effect+), but b deletes it.
  - Interference: Negative effect of a removes a pre-condition of b.
  - Resource conflict: a and b has same pre-condition.
- The term “Means” in the Means-Ends-Analysis (MEA) refers to the actions or operators that can be used to reduce the difference between the current state and the goal state.

Week10 - Expert systems

- Depicted below is the working of *inference engine* in expert system.