Artificial Intelligence

Fahad Sherwani

Course Contents

- Introduction
- Intelligent Agents
- Search
  - Problem solving through search
  - Uninformed search
  - Informed search
  - Local search and constraint Satisfaction
- Games
  - games as search problems
- Knowledge and Reasoning
  - reasoning agents
  - propositional logic
  - predicate logic
  - planning
  - knowledge-based systems
  - uncertain knowledge and reasoning
- Machine Learning
  - learning from observation
  - reinforcement learning
  - neural networks
  - deep learning
- (Natural Language Processing)
- (Robotics)
- (Philosophical, Ethical, Social Issues with AI)
- Conclusions

AI

↳ aims to replicate human intelligence

↳ Reasoning — how knowledge is acquired, represented and stored

↳ Learning — learn and store things, so won't make same mistake; how intelligent behaviour is generated and learned

↳ Problem Solving

↳ Perception — to reason about past and plan for future

Strong AI

↳ machine that superseeds human intelligence

↳ if it can

- do typical human tasks
- apply a wide range of background and knowledge
- has some degree of self-conciousness

↳ it is indistinguishable from that of a human

Weak AI

↳ use of software to accomplish tasks that do not encompass the full range of human cognitive abilities

↳ does not achive self-awarness

↳ demonstrates few of human level cognitive abilities

AI Goals

↳ replicate human intelligence

↳ solve knowledge intensive tasks

↳ make intelligent connection b/w perception and action

- Enhance human-human, human-computer and computer to computer interaction / communication.
- **Engineering based AI Goal**
- Develop concepts, theory and practice of building intelligent machines
- Emphasis is on system building.
- **Science based AI Goal**
- Develop concepts, mechanisms and vocabulary to understand biological intelligent behavior.
- Emphasis is on understanding intelligent behavior.

Cognitive Science

Think human like

- Goal is not just to produce human-like behavior but to produce a sequence of steps of the reasoning process, similar to the steps followed by a human in solving the same task.

Laws of Thought

Think Rationally

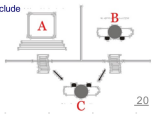
- Goal is to formalize the reasoning process as a system of logical rules and procedures for inference.

Turing Test → act like human

- The art of creating machines that perform functions requiring intelligence when performed by people; that it is the study of, how to make computers to do things which at the moment people do better.
  - A Behaviorist approach, is not concerned with how to get results but to the similarity to what human results are…

Example : Turing Test

- 3 rooms contain: a person, a computer, and an interrogator.
- The interrogator can communicate with the other 2 by teletype (to avoid the machine imitate the appearance or voice of the person).
- The interrogator tries to determine which is the person and which is the machine.
- The machine tries to fool the interrogator to believe that it is the human, and the person also tries to convince the interrogator that it is the human.
- If the machine succeeds in fooling the interrogator, then conclude that the machine is intelligent.
- Goal is to develop systems that are human-like.



20

1. What is the common definition of "AI"? Do you agree?
   Artificial Intelligence (AI) is the simulation of human intelligence processes by machines, especially computer systems. Yes, I agree with this definition.

2. Do you know any AI application?
   One example of an AI application is virtual personal assistants like Siri, Alexa, or Google Assistant.

3. Should artificial intelligence simulate natural intelligence?
   Yes, the goal of AI is often to simulate or replicate aspects of natural human intelligence.

4. What are the criticisms on the AI research? Do you agree?
   Some criticisms include concerns about job displacement, ethical implications, biases in algorithms, and potential misuse of AI technologies. Yes, I agree with these criticisms.

5. What is the relation between AI and logic? AI and philosophy? Logic and philosophy?
   AI often relies on logical reasoning as a foundational component, and there are philosophical implications regarding the nature of intelligence and consciousness in AI research.

6. Explain the meaning of logic? reasoning? ontology?
   Logic: The study of principles of correct reasoning. Reasoning: The mental process of drawing conclusions or making inferences from available information. Ontology: The branch of philosophy that deals with the nature of being, existence, and reality.

7. What is Natural Language Processing? And how is it related to AI?
   Natural Language Processing (NLP) is a subfield of AI that focuses on the interaction between computers and humans through natural language. It involves tasks such as text understanding, language generation, and machine translation.

8. Why and how are Probabilistic and statistical methods used in AI?
   Probabilistic and statistical methods are used to deal with uncertainty and incomplete information in AI systems. They help in decision-making, pattern recognition, and learning from data.

9. What are the major research approaches/schools in AI? Which one do you think is more productive?
   Some major approaches include symbolic AI (based on logic and rules), connectionism/neural networks, evolutionary computation, and Bayesian networks. Each has its strengths and weaknesses, but the neural network approach has seen significant productivity and success in recent years.

CHP 2

Agents

↳ anything that perceives enviorment through sensors
↳ and acts upon that enviorment through actuators

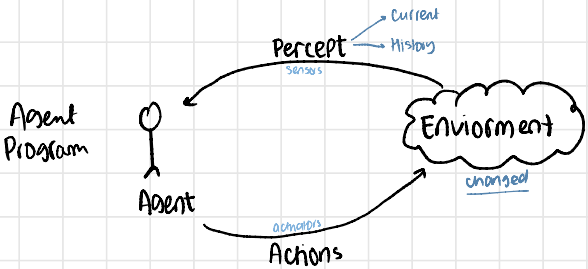
Human agent: Eyes, ears, and other organs for sensors; hands, legs, mouth, and other body parts for actuators

Robotic agent: Cameras and infrared range finders for sensors; various motors for actuators.

Software agent: Software agent is a computer program that acts for a user or other program: an agreement to act on one's behalf.

Agent Programs

↳ help make actions based on perceptions ← check if current P similar to any previous P if not then what to do if did then what to do



Agent → Percepts → Decision → Actions

The agent function maps from percept histories to actions:
[f: P* → A]

The agent program runs on the physical architecture to produce f
Agent = Architecture + Program

when designing an agent we keep in mind

P E A S
Performance, Enviorment, Actuators, Sensors

Goals of Agents

↳ High Performance
↳ Optimized Result
↳ Rational Action
right action

Agent: automated taxi driver

- Performance measure: *Safe, fast, legal, comfortable trip, maximize profits*
- Environment: *Roads, other traffic, people and objects in/around the street*
- Actuators: *Steering wheel, accelerator, brake, signal, horn*
- Sensors: *Cameras, sonar, speedometer, GPS, odometer, engine sensors, keyboard*

Agent: Part-picking robot

- Performance measure: Percentage of parts in correct bins
- Environment: Conveyor belt with parts, bins
- Actuators: Jointed arm and hand
- Sensors: Camera, joint angle sensors

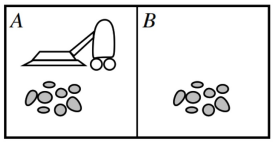
Agent: Interactive English tutor

- Performance measure: Maximize student's score on test
- Environment: Set of students
- Actuators: Screen display (exercises, suggestions, corrections)
- Sensors: Keyboard

Agent: Medical diagnosis system

- Performance measure: Healthy patient, minimize costs, lawsuits
- Environment: Patient, hospital, staff
- Actuators: Screen display (questions, tests, diagnoses, treatments, referrals)
- Sensors: Keyboard (entry of symptoms, findings, patient's answers)

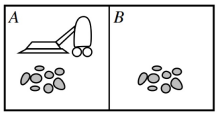
Vacuum-cleaner world



Percepts: location and contents, e.g., [A, Dirty]

Actions: *Left*, *Right*, *Suck*, *DoNothing*

A vacuum-cleaner Agent



Tabulation of an agent function of the vacuum-cleaner

Percept sequence	Action
[A, Clean]	Right
[A, Dirty]	Suck
[B, Clean]	Left
[B, Dirty]	Suck
[A, Clean], [A, Clean]	Right
[A, Clean], [A, Dirty]	Suck
⋮	⋮

```
function REFLEX-VACUUM-AGENT([location,status]) returns an action
  if status = Dirty then return Suck
  else if location = A then return Right
  else if location = B then return Left
```

Rational Agents

An agent should strive to "do the right thing", based on what it can perceive and the actions it can perform. The right action is the one that will cause the agent to be most successful.

Performance measure: an objective criterion for success of an agent's behavior.

E.g., *performance measure* of a vacuum-cleaner agent could be amount of dirt cleaned up, amount of time taken, amount of electricity consumed, amount of noise generated, etc.

Rational Agent: For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.

An agent is autonomous if its behavior is determined by its own experience (with ability to learn and adapt).

Enviorment Types

1. Fully Observable

↳ sensors capture all relavent info from the enviorment

e.g chess, tic tac toe

board → pieces → moves

know all details of enviorments

VS

2. Partially Observable

e.g self driving cars

- how many cars on road ?
- speed of other cars ?
- will a pedestrian accidentally come infront of our car ?

dont know all details of enviorment

can see all moves

cant see others cards

Tic Tac Toe is Fully Observable. Cards are Partially Observable.

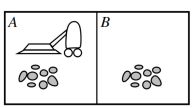
3. Deterministic

↳ changes in the enviorment are predictable

↳ the next state is completely determined by the current state

VS

4. Stochastic → Non deterministic



dirty → clean

?

Vacuum is deterministic. Taxi driver is stochastic.

5. Episodic

↳ independent perceiving-acting episodes

↳ agents experience is divided into atomic 'episodes' → consists of 1. agent perceiving 2. agent performing a single action

↳ the choice of action in each episode depends on the episode itself

↳ takes one step

VS

6. Sequential → non Episodic

Robot is Episodic. Taxi driver is sequential.

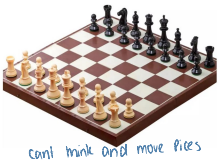
7. Static

↳ no changes in the enviorment while the agent is thinking

VS

8. Dynamic

can think and drive



cant think and move pices together

Taxi driver is dynamic. Chess is static.

9. Discrete

↳ limited no of dishinct percepts/actions

VS

10. Continous

discreete

Continous

Chess has a finite number of distinct states. thus it is discrete; however the Taxi-driving is not. ??

24

11. Single Agent

↳ interaction and collaboration among agents

↳ competitive, cooperative

↳ an agent operating by itself in an enviorment

VS

12. Multiagents

2 Player

1 Player

Crossword is Single agent, while Chess is a multi-agent environment.

2

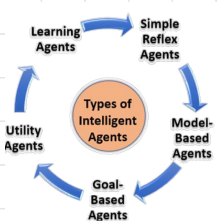
Environment Types

Task Environment	Observable	Agents	Deterministic	Episodic	Static	Discrete
Crossword puzzle	Fully	Single	Deterministic	Sequential	Static	Discrete
Chess with a clock	Fully	Multi	Deterministic	Sequential	Semi	Discrete
Poker	Partially	Multi	Stochastic	Sequential	Static	Discrete
Backgammon	Fully	Multi	Stochastic	Sequential	Static	Discrete
Taxi driving	Partially	Multi	Stochastic	Sequential	Dynamic	Continuous
Medical diagnosis	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
Image analysis	Fully	Single	Deterministic	Episodic	Semi	Continuous
Part-picking robot	Partially	Single	Stochastic	Episodic	Dynamic	Continuous
Refinery controller	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
Interactive English tutor	Partially	Multi	Stochastic	Sequential	Dynamic	Discrete

Agent design

- **The environment type largely determines the agent design**
  - ***Partially observable*** => agent requires ***memory*** (internal state)
  - ***Stochastic*** => agent may have to prepare for ***contingencies***
  - ***Multi-agent*** => agent may need to behave ***randomly***
  - ***Static*** => agent has time to compute a rational decision
  - ***Continuous time*** => continuously operating ***controller***

Agent Types



1. Simple Reflex Agents

→ immediately

↳ agent act only on the basis of current perception

↳ ignores rest of the percept history → hence very limited knowledge

↳ If-else rules

↳ Partially observable enviorment → game smashing sonic it's fully observable

e.g sneeze, alarm clock

change

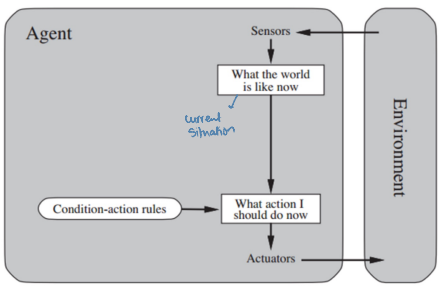
- Enviorment — alarm ringing
- Percept
- Current Situation
- If then else — if time 6:00am
- Action — ring alarm

CONS

↳ infinite loops are unavoidable

↳ very limited knowledge

↳ no perception about prev or next state



2. Model Based Reflex Agents

→ knowledge based

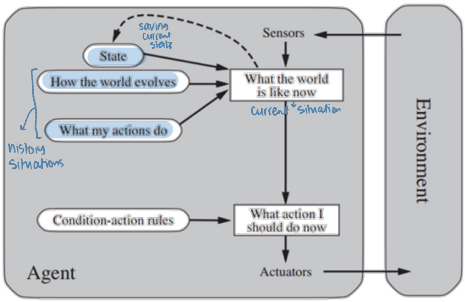
↳ agent acts on the basis of current perception + perception history

↳ state is evaluated based on how it changed from previous state

↳ Partially Observable Enviorment

↳ stores Percept history

e.g cleaner Robot, telephone answering machine



- Know how world evolves
  - Overtaking car gets closer from behind
- How agents actions affect the world
  - Wheel turned clockwise takes you right
- Model base agents update their state

3. Goal based Agents

→ Expansion of model based agent

↳ known goal → agent decides actions based on known goal

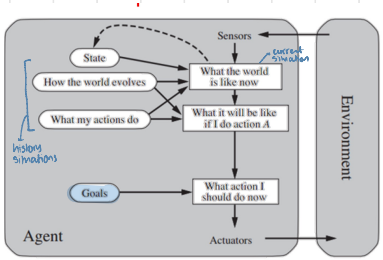
↳ Focuses on Goal

↳ Searching and Planning

- find best route
  - safe and fun route
- how to get to their
  - which bike
  - when to stop
  - where fuel stations

↳ Partially Observable Enviorment

eg GPS system to find a path to a certain destination



4. Utility based Agents

→ Expansion of model based agent

↳ agent acts on the basis of utilities / Preferences

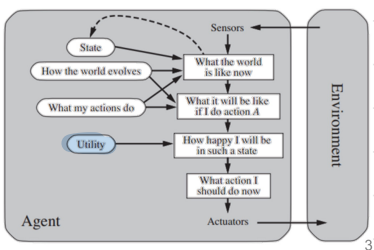
↳ Focuses on Utility not goal

↳ Utility function → happy state / unhappy state

↳ finds Optimally better state for agent

↳ Partially Observable Enviorment

eg GPS system finding fastest path to a certain destination



5. Learning Agents

↳ adapts its actions based on feedback → not only sensors

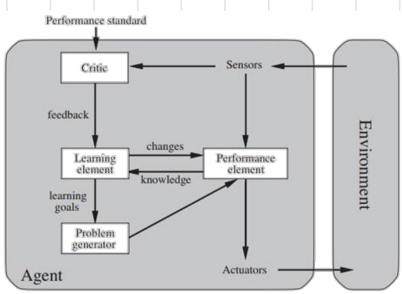
↳ Learning element: makes improvements from past events

↳ Performance element: takes percepts (all percepts) and then decide on actions

↳ Critic: gives learning element feedback → what needs to be modified

↳ Problem generator: suggests actions that lead to new and informative experiences

eg human agent

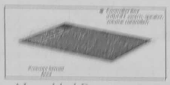


Q: 1, 3, 6, 8

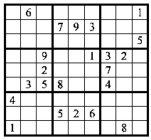
A) Define the followings: [8]

1- Define AI agent.
2- What is back propagation and why do we use it?
3- How supervised and reinforcement learning are related?
4- Define the role of activation function and give the name of popular activation function.
5- Differentiate between Markov decision process and reinforcement learning.
6- Given the figure below, determine the followings:
   a) Whether A and B are conditionally independent given C is observable (NO)
   b) Whether A and B are conditionally independent given C is unobservable (YES)

7- What is the concept of gradient decent in neural network?
8- Explain the difference between impurity and abnormality of a Dataset?

Applications	PEAS	Environment Type	Agent type
 AI-enabled Prayer mat			
 Ambulance drone			
 AI-enabled smart shoes			
 Autonomous car			

Sudoku is a logic-based number placement puzzle. The objective is to fill a 9x9 grid so that each column, each row, and each of the nine 3x3 boxes contains the digits from 1 to 9. Each digit can only appear once per column, row, and 3x3 box. A sample Sudoku puzzle to use for this question is given in the figure below.



a) Classify the characteristics of the environment for Sudoku according to the following properties and explain the analyzed reason for each of your five choices.

- Fully observable/partially observable
- Deterministic/stochastic
- Episodic/sequential
- Static/dynamic/semi-dynamic
- Discrete/continuous

b) Specify the task environment of Sudoku puzzle?

c) Are Game theoretic problems necessarily multi agent problems? Give an example of a non-zero-sum game.

Q1) Classify the following properties for a Sudoku puzzle and choose the appropriate classification

1. **Fully observable/partially observable:**
   - **Fully observable**: In Sudoku, all information about the state of the puzzle is available to the player at all times. There is no hidden information.

2. **Deterministic/stochastic:**
   - **Deterministic**: The outcome of placing a number in any given cell is completely determined by the rules of Sudoku. There is no element of chance or randomness in how the numbers are placed.

3. **Episodic/sequential:**
   - **Sequential**: Each move in Sudoku (placing a number in a cell) affects future moves, and the player must consider the sequence of moves to solve the puzzle correctly.

4. **Static/dynamic/semi-dynamic:**
   - **Static**: The Sudoku puzzle does not change while the player is thinking about the next move. The puzzle only changes when the player makes a move.

5. **Discrete/continuous:**
   - **Discrete**: The Sudoku puzzle involves a finite set of distinct values (the numbers 1 through 9) and discrete placement options (the cells of the grid).

puzzle

Zero sum games
↳ draw jahan hojaye
e.g chess draw
tic tac toe draw
rock paper scissor draw

Summary

- Agents interact with environments through actuators and sensors
- The agent function describes what the agent does in all circumstances.
- The performance measure evaluates the environment sequence.
- A perfectly rational agent maximizes expected performance.
- Agent programs implement (some) agent functions PEAS descriptions define task environments.
- Environments are categorized along several dimensions:
  - Observable? deterministic? episodic? static? discrete? single-agent?
- Several basic agent architectures exist:
  - Simple Reflex, Model based, goal-based, utility-based, Learning

CHP 3

Searching

↳ falls under AI

Searching in AI

- Optimization. Search algorithms are the basis for many optimization and planning methods.
- Solve a problem by searching for a solution. Search strategies are important methods for many approaches to problem-solving.
- Problem formulation. The use of search requires an abstract formulation of the problem and the available steps to construct solutions.

Problem Formulation

To solve a problem by search, we need to first formulate the problem.

HOW?

Our textbook suggest the following schema to help us formulate problems:

1. State
2. Initial state
3. Actions or Successor Function
4. Goal Test
5. Path Cost
6. Solution

Searching for Solutions

Traversal of the search space
- From the initial state to a goal state.
- Legal sequence of actions as defined by successor function.

General procedure
- Check for goal state
- Expand the current state
  - Determine the set of reachable states
  - Return "failure" if the set is empty
- Select one from the set of reachable states
- Move to the selected state

A search tree is generated
- Nodes are added as more states are visited

Search Terminology

Search Tree
- Generated as the search space is traversed
  - The search space itself is not necessarily a tree, frequently it is a graph
  - The tree specifies possible paths through the search space
- Expansion of nodes
  - As states are explored, the corresponding nodes are expanded by applying the successor function
    - this generates a new set of (child) nodes
  - The *fringe* (frontier/queue) is the set of nodes not yet visited
    - newly generated nodes are added to the fringe
- Search strategy
  - Determines the selection of the next node to be expanded
  - Can be achieved by ordering the nodes in the fringe
    - e.g. queue (FIFO), stack (LIFO), "best" node w.r.t. some measure (cost)

State space search

→ no of states in which the problem can go

S: { S, A, Action(s), Result(s,a), Cost(s,a) }

S: start state, goal state, intermediate states

A: all possible actions (eg for 8 Puzzel Problem action: up, down, left, right)

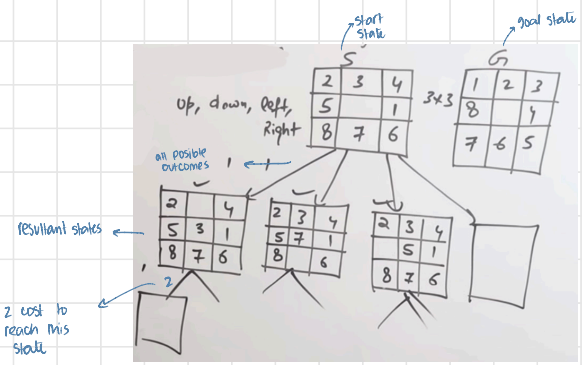
Actions(s): the choosen actions

Result(s,a): the resultant state

Cost(s,a): get minimum cost

↳ Precisely

↳ analyse



8 Puzzle problem

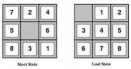
↳ Blind Search → Uninformed

↳ BFS

↳ 4 moves (up, down, left, right)

Problem Formulation (The 8- Puzzle Example)

State: The location of the eight tiles, and the blank



Initial State: {(7,0), (2,1), (4,2), (5,3), (_,4), (6,5), (8,6), (3,7), (1,8)}

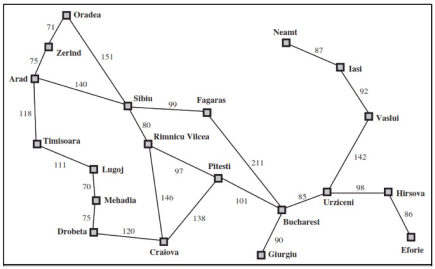
Successor Function: one of the four actions (blank moves Left, Right, Up, Down).

Goal Test: determine a given state is a goal state.

Path Cost: each step costs 1

Solution: {(_,0),(1,1),(2,2) ,(3,3) ,(4,4) ,(5,5) ,(6,6) ,(7,7) ,(8,8)}

Problem Formulation (The Romania Example)



Solution: a sequence of actions leading from the initial state to a goal state
{Arad →Sibiu → Rimnicu Vilcea → Pitesti → Bucharest}

Problem Formulation (Real-life Applications)

Route Finding Problem

Car Navigation | Airline travel planning | Routing in Computer networks | Military operation planning

- States
  - locations
- Initial state
  - starting point
- Successor function (operators)
  - move from one location to another
- Goal test
  - arrive at a certain location
- Path cost
  - may be quite complex
    - money, time, travel comfort, scenery,

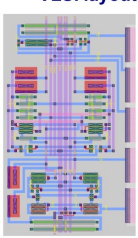
Problem Formulation (Real-life Applications)

Travel Salesperson Problem

- States
  - locations / cities
  - illegal states
    - each city may be visited only once
    - visited cities must be kept as state information
- Initial state
  - starting point
  - no cities visited
- Successor function (operators)
  - move from one location to another one
- Goal test
  - all locations visited
  - agent at the initial location
- Path cost
  - distance between locations

Problem Formulation (Real-life Applications)

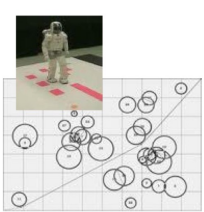
VLSI layout Problem



- States
  - positions of components, wires on a chip
- Initial state
  - incremental: no components placed
  - complete-state: all components placed (e.g. randomly, manually)
- Successor function (operators)
  - incremental: place components, route wire
  - complete-state: move component, move wire
- Goal test
  - all components placed
  - components connected as specified
- Path cost
  - maybe complex
    - distance, capacity, number of connections per component

Problem Formulation (Real-life Applications)

Robot Navigation



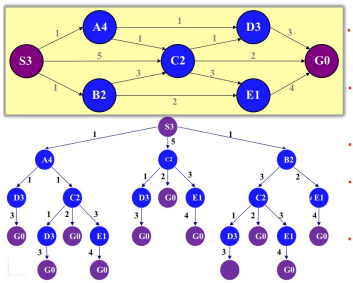
- States
  - locations
  - position of actuators
- Initial state
  - start position (dependent on the task)
- Successor function (operators)
  - movement, actions of actuators
- Goal test
  - task-dependent
- Path cost
  - maybe very complex
    - distance, energy consumption

Problem Formulation (Real-life Applications)

Automatic Assembly Sequencing

- States
  - location of components
- Initial state
  - no components assembled
- Successor function (operators)
  - place component
- Goal test
  - system fully assembled
- Path cost
  - number of moves

Traversing a Graph as Tree



- A tree is generated by traversing the graph.
- The same node in the graph may appear repeatedly in the tree.
- The arrangement of the tree depends on the traversal strategy (search method)
- The initial state becomes the root node of the tree
- In the fully expanded tree, the goal states are the leaf nodes.
- Cycles in graphs may result in infinite branches

Prim's Vs Kruskal's

Prim's Algorithm	Kruskal's Algorithm
It starts to build the Minimum Spanning Tree from any vertex in the graph.	It starts to build the Minimum Spanning Tree from the vertex carrying minimum weight in the graph.
It traverses one node more than one time to get the minimum distance.	It traverses one node only once.
Prim's algorithm has a time complexity of $O(V^2)$, V being the number of vertices and can be improved up to O(E + log V) using Fibonacci heaps.	Kruskal's algorithm's time complexity is O(E log V), V being the number of vertices.
Prim's algorithm gives connected component as well as it works only on connected graph.	Kruskal's algorithm can generate forest (disconnected components) at any instant as well as it can work on disconnected components
Prim's algorithm runs faster in dense graphs.	Kruskal's algorithm runs faster in sparse graphs.

1. Prims Algo

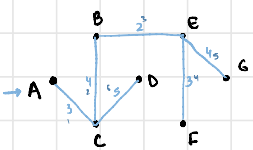
node based → greedy Algo

↳ start with any node

↳ connectedly move to smallest neighbour

↳ no cycles

↳ edges = n-1



MST = 21

2. Kushkal Algo

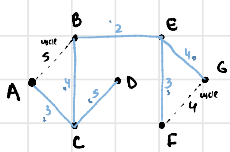
edge based → greedy Algo

↳ stral with smallest edge

↳ disconnectedly mark smallest edge

↳ no cycles

↳ edges = n-1



MST = 21

Prim's Algorithm

Prim's Algorithm includes 4 Steps:

Step 1: Draw an n by n (n x n) vertices' matrix and label them as V_1, V_2…..V_n along with the given weights of the edges.

Step 2: Starting from vertex V_1 and connect it to its nearest neighbor by searching in row 1.

Step 3: Consider V1 and Vi as one subgraph and connect as step 2 while do not forming any circuit.

Step 4: Continue this process until we get the MST having n vertices and (Vi-1) edges.

Kruskal's Algorithm

Kruskal's Algorithm includes 4 Steps:

Step 1: List all the edges of the graph in order of increasing weights.

Step 2: Select the smallest edge of the graph.

Step 3: Select the next smallest edge that do not makes any circuit.

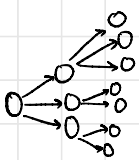
Step 4: Continue this process until all the Vertices are explored and (V_n-1) edges have been selected

Searching

Uniformed search
Blind Search

- no of steps unknown
- search without info (start state, goal state) → like brute force
- Time Consuming
- More Complexity
- Always Optimal solution

- BFS
- Uniformed cost search
- DFS
- Depth limited search
- Iterative Deeping
- Bi-directional search

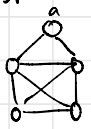


Informed search
Heuristic Search

- agent has background info of the problem
- search with info
- Quick solution
- Less complexity
- Not always optimal

- Best First Search
- Heuristics
- Memory Bounded search
- Iterative improvement search
- A*

TSP $(n-1)!$



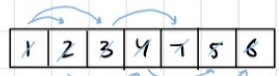
QUEUE FIFO

1. Breadth-First-Search (BFS)

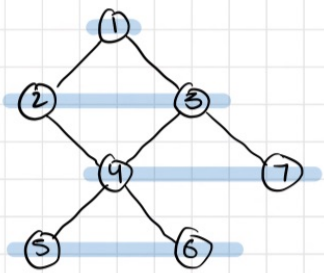
moves

Uniformed search

- Start from top, put in Queue
- Put adj vertices in Queue
- take out vertice when explored (take it out of queue)
- move to next vertice in Queue
- Repeat till all vertices explored



1 2 3 4 7 5 6



CON

Not suitable for searching large graphs

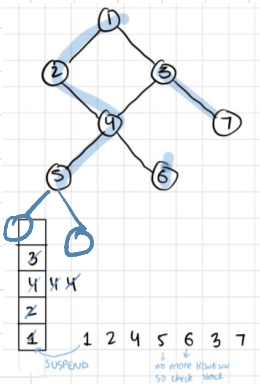
Algorithm (Informal)

1. Enqueue the root/initial node (Queue Structure).
2. Dequeue a node and examine it.
   1. If the element sought is found in this node, quit the search and return a result.
   2. Otherwise enqueue any successors (the direct child nodes) that have not yet been discovered.
3. If the queue is empty, every node on the graph has been examined – quit the search and return "not found".
4. Repeat from Step 2.

STACK LIFO

2. Depth First Search (DFS)

- Start from top, write it
- move to adj vertex
- suspend prev vertex to stack
- Repeat till no adj vertex
- backtrack by checking stack
- Repeat till no vertices left



Depth-First Search

1. Start
2. Push Root Node to Stack
   - Mark Root Node as Visited
   - Print Root Node as Output
3. Check Top of the Stack If Stack is Empty, Go to Step 6
4. Else, Check Adjacent Top of the Stack
   - If Adjacent is not Visited
   - Push Node to Stack
   - Mark Node as Visited
   - Print Node as Output
   - Else Adjacent Visited
5. Go to Step 3
6. Stop

Similar to BFS
↳ an evaluation of cost for each reachable node

3. Uniformed-Cost-Search

↳ visits next node which has least total cost from root

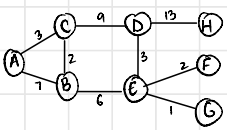
↳ until a goal state is reached

Steps

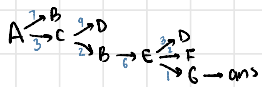
1. Start from start node
2. find min cost node from that node
3. Repeat 2 until
   - ↳ Goal node found
   - ↳ Goal node not found, but reached last node
   - ↳ backtrack

repeat

↳ checks all
↳ gets smallest



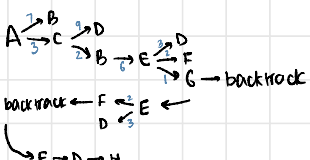
Case 1: Goal node G



ANS: A → C → B → E → G

= 12

Case 2: Goal node H

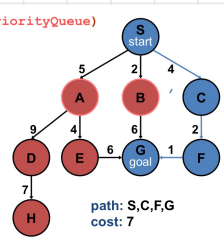


ANS: A → C → B → E → G → E → F → E → D → H

= 33

generalSearch(problem, priorityQueue)
# of nodes tested: 6, expanded: 5

Closed Node	Open Node
	{S}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C	{A:5,F:6,G:8}
A	{F:6,G:8,E:9,D:14}
F	{G:7,G:8,E:9,D:14}
G	{G:8,E:9,D:14}



Similar to DFS
but with a limit

4. Depth-Limited Search

↳ solves DFS' infinite path problem

- Sometimes a depth limit can be inferred or estimated from the problem description
  - In other cases, a good depth limit is only known when the problem is solved
- must keep track of the depth

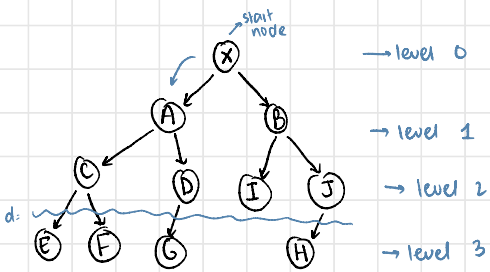
Termination conditions

↳ failure value: There is no solution

↳ cutoff failure: Terminates on reaching predetermined depth

Steps

1. Same as DFS
2. dont move past predetermined depth



predetermined depth = 2

Case 1: Goal node J

ANS: X → A → C → D → B → I → J

Case 2: Goal node H

ANS: no solution as H is in level 3 and predetermined depth is 2

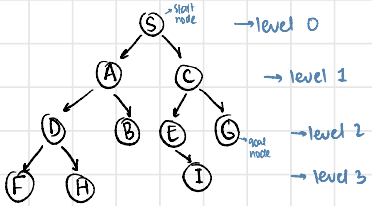
5. Interative Deepening DFS

↳ applies LIMITED-DEPTH with increasing depth limit

↳ combines adv of BFS and DFS

Steps

1. DFS to depth 1
2. Increase depth, apply DFS
3. Repeat 2 till goal node found



Iteration	d	[S]
1	0	S
2	1	S→A→C
3	2	S→A→D→B→C→E→G → found

PRO

↳ includes benefits of bom DFS BFS

↳ fast searching

↳ less memory req → uses only linear space

CON

↳ Repeats Process → as many state are expanded multiple times

function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution, or failure
  inputs: *problem*, a problem
  for *depth* ← 0 **to** ∞ **do**
    result ← DEPTH-LIMITED-SEARCH(*problem*, *depth*)
    if *result* ≠ cutoff **then return** *result*

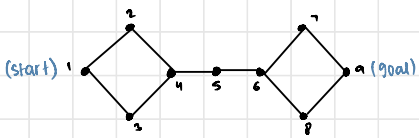
6. Bi-directional search

↳ bom search forward from initial state and backwards from goals

Motivation: $b^{d/2} + b^{d/2} = 2b^{d/2}$ (BFS from start, BFS from goal) is much less than b^d (BFS/DFS)

Steps

1. Start BFS/DFS from start and goal node
2. When bom searches meet stop
   ↳ solution found



Evaluation of Search Strategies

↳ defined by picking the order of node expansion

↳ evaluated along the following dimensions

Properties	Breadth First	Depth First	Uniform Cost	Iterative Deeping	Depth Limited	Bi-directional	
Complete (→ might not give solution)	Yes	NO (→ fails for loops)	Yes	Yes	NO	Yes (→ if BFS)	b: branching factor d: depth of goal m: max depth of state space
Time Complexity (→ how long to find solution)	$O(b^{d+1})$	$O(b^m)$		$O(b^d)$	$O(b^L)$	$2b^{d/2}$	
Space Complexity (→ memory req)	$O(b^{d+1})$	$O(bm)$		$O(bd)$	$O(bL)$		
Optimality (→ best solution)	Yes	NO	Yes	Yes	NO (→ if L<d)	Yes (→ if BFS)	

When to use what

BFS

↳ shallow solutions (→ of little depth)

UCS

↳ actions have varying costs

↳ least cost solution is req

Iterative Deeping DFS

↳ large search spaces

↳ unknown depth

Depth First Search

↳ many solutions exist

↳ know/estimate the depth of solution

Breadth-first search (BFS) and depth-first search (DFS) are the foundation for all other search techniques.

Properties of Uniform-cost Search (UCS)

Completeness Yes (if *b* is finite, and step cost is positive)

Time Complexity much larger than b^d, and just b^d if all steps have the same cost.

Space Complexity: as above

Optimality: Yes

Criterion	Uniform-Cost
Complete?	Yes
Time	$O(b^{\lfloor C^*/\epsilon \rfloor})$
Space	$O(b^{\lfloor C^*/\epsilon \rfloor})$
Optimal?	Yes

b Branching Factor

d Depth of the goal/tree

Requires that the goal test being applied when a node is removed from the nodes list rather than when the node is first generated while its parent node is expanded.

only uniformed search that worries about cost

Breadth First search VS Uniform Cost Search

- ↳ special case of uniformed cost search
- ↳ when all edge costs are positive and identical
- ↳ expands the shallowest node → only optimal if all costs equal
- ↳ only optimal if all costs equal

- ↳ considers overall path cost
- ↳ optimal for any cost function
  - ↳ non-zero, positive
- ↳ Gets stuck down in trees
  - ↳ low path cost, but no goal node

both are complete for non-extreme problems

- ↳ finite no. of branches
- ↳ strictly positive search function

Depth First VS Breadth First

- ↳ goes off into one branch until it reaches leaf node
- ↳ not good if goal is on another branch
- ↳ neither complete or optimal
- ↳ uses lesser space than breadth first

For a very large tree

- ↳ may take excessively long time to find even a very nearby goal

- ↳ checks all alternative
- ↳ complete and optimal
- ↳ memory intensive

- ↳ memory req maybe excessive

Iterative Deepening Search

Based on [4]

When searching a binary tree to depth 7:
- DFS requires searching 255 nodes
- Iterative deepening requires searching 502 nodes
- Iterative deepening takes only about twice as long

When searching a tree with branching factor of 4 (each node may have four children):
- DFS requires searching 21845 nodes
- Iterative deepening requires searching 29124 nodes
- Iterative deepening takes about 4/3 = 1.33 times as long

The higher the branching factor, the lower the relative cost of iterative deepening depth first search

→ low branch factor
↳ worse IDS

Iterative Deepening Search

The nodes in the bottom level (level d) are generated once, those on the next bottom level are generated twice, and so or

$$N_{IDS} = (d)b + (d-1)b^2 + \ldots + (1)\,b^d$$

Time complexity = b^d

Compared with BFS:

$$N_{BFS} = b + b^2 \ldots + b^d + (b^{d+1} - b)$$

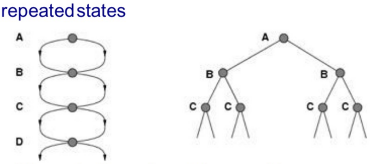
- Suppose $b = 10$, $d = 5$,

N_{IDS} = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456
N_{BFS} = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111

→ IDS behaves better in case the search space is large and the depth of goal is unknown.

Improving Search Methods

Make algorithms more efficient
- avoiding repeated states



Use additional knowledge about the problem
- properties ("shape") of the search space
  - more interesting areas are investigated first
- pruning of irrelevant areas
  - areas that are guaranteed not to contain a solution can be discarded

CHP 4

Heuristics → aka Rule of Thumb

(To discover)

↳ a technique designed to solve a problem quickly

↳ comes from experience

↳ helps you think through things (e.g. process of elimination, process of trial and error)

↳ will give quick solution
But not necessarily optimal

Heuristic function

↳ are problem specific

↳ mostly employs straight line distance
for route finding and similar problems

↳ often better than DFS
although worse time complexities are
equally bad or worser

Properties	Greedy Best First search	A*	SMA*	RBFS
Complete		Yes	Yes (if memory is high enough to store shallowest solution)	
Time Complexity		Exponential		
Space Complexity		keeps all nodes in memory		Linear in depth
Optimality		Yes (if memory is high enough to store shallowest solution)	Yes (if memory is high enough to store shallowest solution)	Yes

Greedy best-first search

```
function GREEDY-BEST-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :  /* Cost f(n) = h(n) */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

A* Algorithm

```
function A-STAR-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :  /* Cost f(n) = g(n) + h(n) */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

SMA* pseudocode

Base

```
function SMA*(problem) returns a solution sequence
  inputs: problem, a problem
  static: Queue, a queue of nodes ordered by f-cost

  Queue ← MAKE-QUEUE({MAKE-NODE(INITIAL-STATE[problem])})
  loop do
      if Queue is empty then return failure
      n ← deepest least-f-cost node in Queue
      if GOAL-TEST(n) then return success
      s ← NEXT-SUCCESSOR(n)
      if s is not a goal and is at maximum depth then
          f(s) ← ∞
      else
          f(s) ← MAX(f(n),g(s)+h(s))
      if all of n's successors have been generated then
          update n's f-cost and those of its ancestors if necessary
      if SUCCESSORS(n) all in memory then remove n from Queue
      if memory is full then
          delete shallowest, highest-f-cost node in Queue
          remove it from its parent's successor list
          insert its parent on Queue if necessary
      insert s in Queue
  end
```

Greedy Best First Search

$O(b^m)$

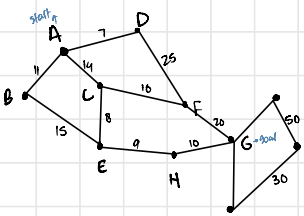
↳ expands the node that appears to be closest to goal heuristically

↳ estimate of cost from n to goal using heuristic function

$h_{SLD}(n)$ = straight line distance from n to Bucharest

Steps

1. Start from start node
2. Find min h_{SLD} node from that node
3. Repeat 2 until
   ↳ Goal node found



use heuristic values to compare

h_{SLD}

A→G : 40
B→G : 32
C→G : 25
D→G : 35
E→G : 19
F→G : 17
H→G : 10
G→G : 0

A → B → 32, C → 25, D → 35

A, C, B, D (32, 35)

C → E → 19, F → 17

A, C, B, D, F, E (32, 35, 17, 19)

F → G → 0 → reached goal state

closed list
A
C
F

A → C → F → G

A* Search

combines greedy and UCS

$O(b^d)$

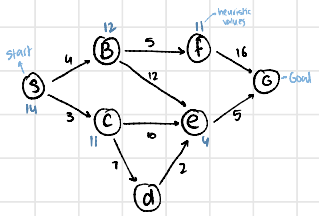
↳ avoids expanding paths that are already expensive

$f(n) = g(n) + h(n)$ (g(n): path cost; h(n): estimated cost to goal)

↳ Heuristics must be admissible

Steps

1. Start from start node
2. Use f(n) function to find min heuristic cost node
3. Repeat 2 until
   ↳ Goal node found



↳ f(S) = 0 + 14 = 14

↳ S → B : 4 + 12 = 16
S → C : 3 + 11 = 14

SB → f : 4 + 5 + 11 = 20
SB → e : 4 + 12 + 4 = 20

↳ SC → d : 3 + 7 + 6 = 16
SC → e : 3 + 10 + 4 = 17 → smallest so use this

↳ SCD → e : 3 + 7 + 2 + 4 = 16

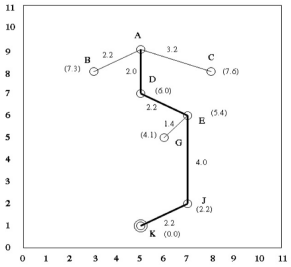
↳ SCDe → G : 3 + 7 + 2 + 5 = 17

CONS

↳ complexity problems
↳ huge memory req

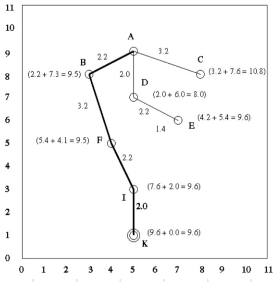
Solution to Greedy Best-First Exercise

Node	Coordinates	h(n)
A	(5,9)	8.0
B	(3,8)	7.3
C	(8,8)	7.6
D	(5,7)	6.0
E	(7,6)	5.4
F	(4,5)	4.1
G	(6,5)	4.1
H	(3,3)	2.8
I	(5,3)	2.0
J	(7,2)	2.2
K	(5,1)	0.0



Solution to A* Exercise

Node	Coordinates	h(n)
A	(5,9)	8.0
B	(3,8)	7.3
C	(8,8)	7.6
D	(5,7)	6.0
E	(7,6)	5.4
F	(4,5)	4.1
G	(6,5)	4.1
H	(3,3)	2.8
I	(5,3)	2.0
J	(7,2)	2.2
K	(5,1)	0.0



Admissible Heuristics

↳ $h(n)$ is admissible

if every node n, $h(n) \leq h^*(n)$ → underestimation

($h^*(n)$: true cost to reach the goal state from n)

↳ it is optimistic → never overestimates the cost to reach the goal

Theorem-1: If *h(n)* is admissible, A* using TREE-SEARCH is optimal.

Optimality of A* (proof Admissible)

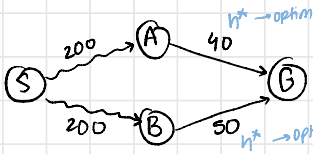
Recall that $f(n) = g(n) + h(n)$

Now, suppose some suboptimal goal G_2 has been generated and is in the fringe. Let n be an unexpanded node in the fringe such that n is on a shortest path to an optimal goal G.

We want to prove:
$f(n) < f(G_2)$
(then A* will prefer n over G_2)

$f(G_2) = g(G_2)$ since $h(G_2) = 0$
$g(G_2) > g(G)$ since G_2 is suboptimal
$f(G) = g(G)$ since $h(G) = 0$
Then $f(G_2) > f(G)$ from above
$h(n) \leq h^*(n)$ since h is admissible
$g(n) + h(n) \leq g(n) + h^*(n)$
Then $f(n) \leq f(G)$
Thus, A* will never select G_2 for expansion

In other words:
$f(G_2) = g(G_2) + h(G_2) = g(G_2) > C^*$,
since G_2 is a goal on a non-optimal path (C^* is the optimal cost)
$f(n) = g(n) + h(n) \leq C^*$, since h is admissible
$f(n) \leq C^* < f(G_2)$, so G_2 will never be expanded
→ A* will not expand goals on sub-optimal paths

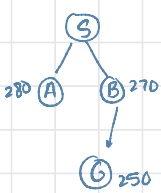


g(A): 200
g(B): 200

$h(n) \geq h^*(n)$ → overestimation

Q) $h(A) = 80$, $h(B) = 70$] $> h^*$

$f(A) = 200 + 80 = 280$
$f(B) = 200 + 70 = 270$



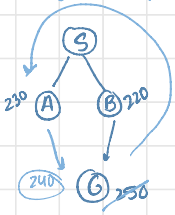
$f(G) = g(G) + h(G)$
$= \overbrace{200 + 50}^{g(G) + h^*(G)} + \overbrace{0}^{h}$
$= 250$

$h(n) \leq h^*(n)$ → underestimation (hence admissible)

Q) $h(A) = 30$, $h(B) = 20$] $< h^*$

$f(A) = 200 + 30 = 230$
$f(B) = 200 + 20 = 220$

check this path



$f(G) = g(G) + h(G)$
$= \overbrace{200 + 50}^{g(G) + h^*(G)} + \overbrace{0}^{h} = 250$

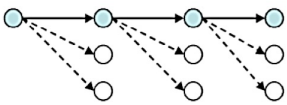
$f(G) = g(G) + h(G)$
$= \overbrace{200 + 40}^{g(G) + h^*(n)} + \overbrace{0}^{h} = 240$

$g(n) + h(n) < g(n) + h^*(n)$
$230 < 250$

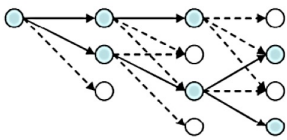
so want expand on B route

Local Beam Search

- Start with k randomly generated states (ß= 2 or 3)
- At each iteration, all the successors of all k states are generated
- If any one is a goal state, stop; else select the k best successors from the complete list and successors from the complete list and repeat



Greedy Local



Local Beam

Memory Bounded Heuristic Search

↳ To solve memory problem of A*

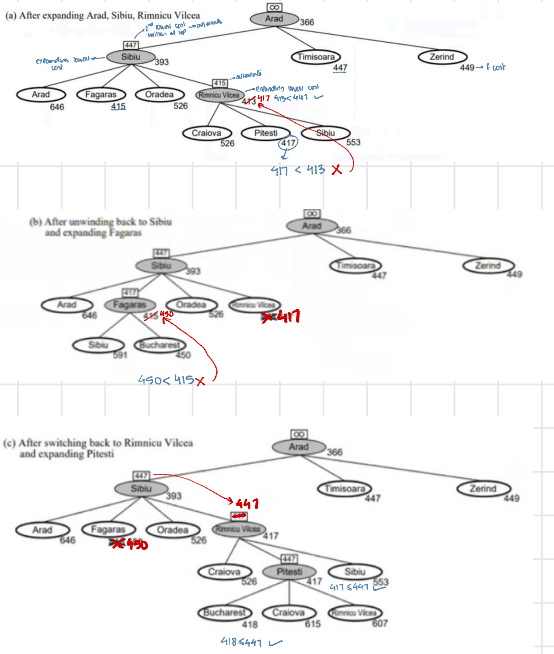
↳ Two types of Memory Bounded Heuristic Search

↳ Recursive BFS

↳ SMA*

Recursive Best First Search (RBFS)

↳ BFS using only linear space



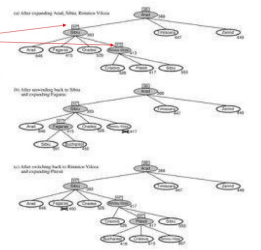
best alternative over fringe nodes, which are not children: *do I want to back up?*

RBFS changes its mind very often in practice.

This is because the $f=g+h$ become more accurate (less optimistic) as we approach the goal. Hence, higher level nodes have smaller f-values and will be explored first.

Problem? If we have more memory we cannot make use of it.

Ay idea to improve this?



Ans: Arad → Sibiu → Rim → Pitesti → Bucharest

Simple Memory Bounded A* (SMA*)

↳ finds optimal reachable solution given the memory constraint → but time can still be exponential

↳ if there is a tie → equal f-values
we first delete the oldest nodes first

How it works:

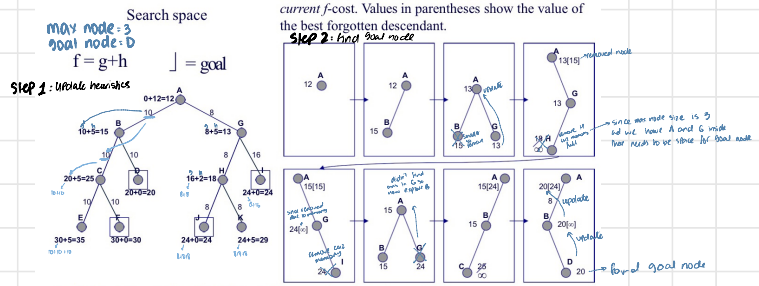
- Like A*, it expands the best leaf until memory is full.
- Drops the worst leaf node- the one with the highest f-value.
- SMA* then backs up the value of the forgotten node to its parent.

- This is like A*, but when memory is full we delete the worst node (largest *f*-value).
- Like RBFS, we remember the best descendent in the branch we delete.

Simple Memory-bounded A* (SMA*)

(Example with 3-node memory)

Progress of SMA*. Each node is labeled with its *current f*-cost. Values in parentheses show the value of the best forgotten descendant.



∞ is given to nodes that the path up to it uses all available memory.
Can tell when best solution found within memory constraint is optimal or not.

35

The Algorithm proceeds as follow

Let MaxNodes = 3
•Initially B and G are added to open list, then hit max.
•B is larger f value, so discard but save f(B)=15 at parent A
Add H, but f(H)=18. This is not a goal and we can never go deeper, so set f(H)=infinity and save at G.
•Generate next child I with f(I)=24, bigger other child of A. Now we have seen all children of G, so reset f(G) to 24.
•Regenerate B and child C. This is not a goal so f(C) is reset to infinity.
•Generate second child D with f(D)=20, backing up value to ancestors.
•D is a goal node, so search terminates.

PROS

↳ uses bounded memory

↳ optimal and complete if memory is high enough to store shallowest solution

↳ avoids repeated states if memory allows it

↳ will use all memory

↳ higher the memory, higher the speed

Relaxed Problems

↳ problems with fewer restrictions on actions

The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem.

If the rules of the 8-puzzle are relaxed so that a tile can move anywhere, then $h_1(n)$ gives the shortest solution.

If the rules are relaxed so that a tile can move to any near square, then $h_2(n)$ gives the shortest solution.

Simple Hill Climbing Algo

→ local search

CHP 5

↳ local search Algo → not Global

↳ Greedy Approach → only explores best state, ignores others → smallest ANS

↳ No backtrack

↳ only keeps a singular current state → no need to maintain tree or graph

↳ moves in the direction of increased elevation to find best solution to problem

↳ Terminates when

↳ Peak value reached

↳ and no neighbour has higher value

STEPS

↳ Evaluate initial state

↳ LOOP until solution found OR there are no operations left

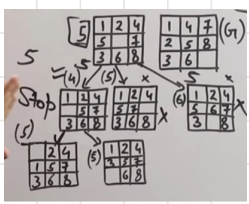
↳ select and apply a new operator

↳ Evaluate new state

↳ if goal then quit

↳ if new state < curr

curr = new state



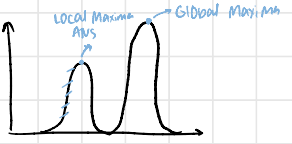
CONS

1. Local Maximum

local peak state

↳ as no backtrackers

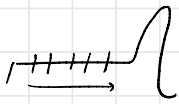
↳ There may be a better maxima if you choose min step first But since greedy approach, so dost always give shortest answer



2- Flat Maximum

↳ if all possibilities have same heurisnc no

↳ then it stops

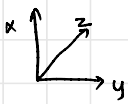


3. Ridge

↳ moves only in 1 direction

↳ its a special form of local maximum

↳ it has an area which is higher than its surrounding areas but, itself has a slope that cant be reached in a single move



PROS

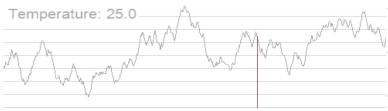
↳ less space complexity → as it only stores current state

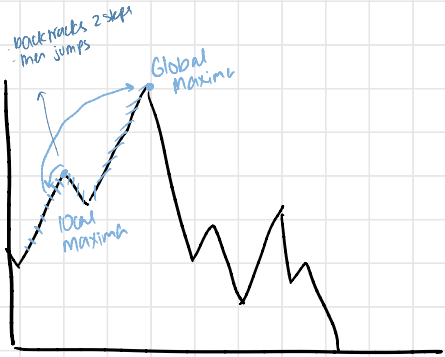
Simulated Annealing (SA)

→ checks all neighbours

↳ allows backtracking

- Explore Successors wildly randomly: High temp
- As time goes by , explore less wildly: Cool Down
- Until there's a time when things settle: Cold





↳ when a local maxima found it backtracks using annealing schedule

↳ then traverses , when a higher maxima is found

↳ it jumps to that position

↳ then Repeat

SA vs Hill Climbing

SA

↳ allows backtracking → maintains annealing schedule

↳ checks all neighbours

↳ moves to worst states allowed

↳ greater space time → Stores local maxima as well to compare

↳ slow

↳ optimal solution

Hill Climbing

↳ no backtracking

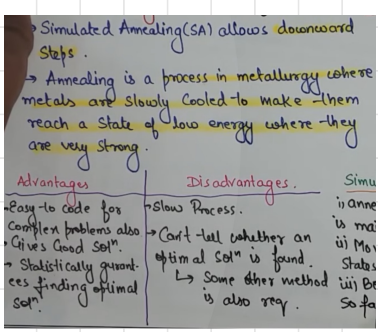
↳ doesn't check all neighbours

↳ only best state moves allowed

↳ less space time → only stores current state

↳ fast

↳ may not give optimal solution



→ Simulated Annealing(SA) allows downward steps.

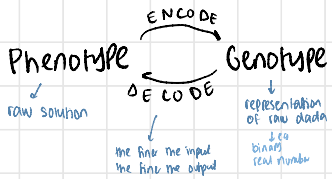
→ Annealing is a process in metallurgy where metals are slowly cooled to make them reach a state of low energy where they are very strong.

Advantages	Disadvantages.	Simu
-Easy to code for complex problems also Gives Good Soln. → Statistically guarantees finding optimal soln.	→Slow Process. →Can't tell whether an optimal Soln is found. ↳ Some other method is also req.	i) Anne is mai ii) Mo States iii) Be So fa

Genetic Algorithm

finds best parent

↳ solves complex problems like → eg NP hard
↳ focuses on optimization
↳ population of possible solutions for a given problem
↳ from a group of individuals the best will survive



Population → set of solutions
Individual → solution to a problem
Fitness → quality of a solution
Chromosome → encoding for a solution (string)
Gene → part of the encoding of a solution (bits)
Reproduction → cross over

- begin with *k* randomly generated states (population)
- each state (individual) is a string over some alphabet (chromosome)
- fitness function (bigger number is better)
- crossover
- mutate (evolve?)

Properties

- Work well for mixed (continuous and discrete) problems
- They are less suceptible to get stuck at local optima
- Computationally expensive
- However, easy to perform in parallel
- No math in the process. The objective (fitness) function may be hard

ENCODING

↳ process that represents solution in the form of strings that converts necessary info

1. Binary Encoding → most common

↳ chromosones are strings of 1's and 0's
↳ each position in the chromosone represents a particular characteristic of their problem

2. Permutation Encoding

↳ useful in ordering problems → eg TSP
eg in TSP, every chromosone is a string of numbers
↳ each representing a city to be visited

3. Value Encoding

↳ used in problems where complicated values (eg real numbers) are used
↳ and binary encoding would not suffice

Crossover

↳ process in which 2 chromosones → strings
↳ combine their genetic material → bits
↳ to produce a new offspring
↳ which posses both their characteristic

- 2 strings are picked at random from the mating pool to cross over
- The chosen method depends on the encoding method

1. Single Point Crossover → 1 Point, 2 offsprings

↳ a random point is chosen on the individual chromosones → strings
↳ then genetic material is exchanged

Chromosome1	11011 | 00100110110
Chromosome 2	11011 | 11000011110
Offspring 1	11011 | 11000011110
Offspring 2	11011 | 00100110110

2. Two Point Crossover → 2 Points, 2 offsprings

↳ 2 random points are chosen on the individual chromosones → strings
↳ then genetic material is exchanged

Chromosome1	11011 | 00100 | 110110
Chromosome 2	10101 | 11000 | 011110
Offspring 1	10101 | 00100 | 011110
Offspring 2	11011 | 11000 | 110110

NOTE: These chromosomes are different from the last example.

3. Uniform Crossover → Genes, 1 offspring

↳ Each gene → bit is selected randomly
↳ from one of the coresponding genes of the parent chromosones

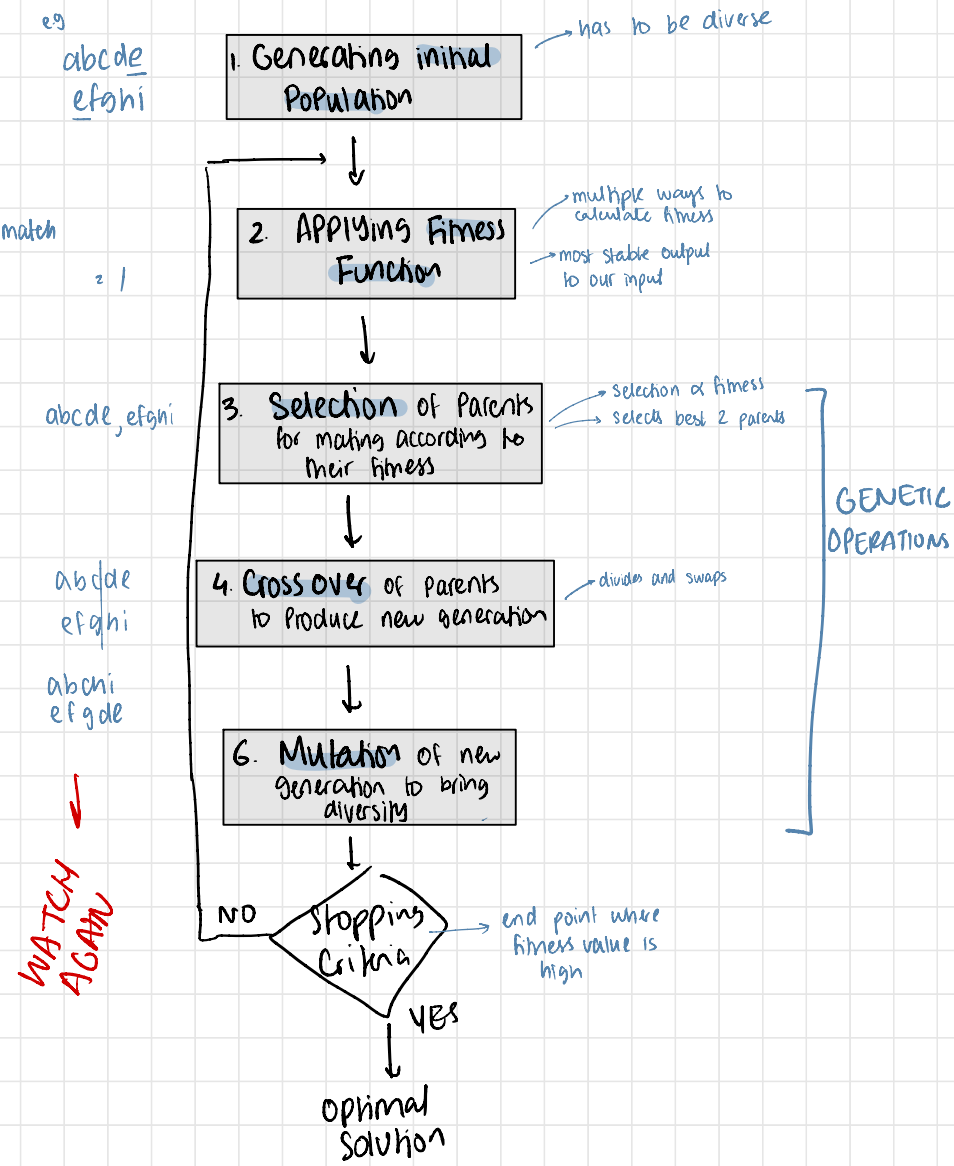
Chromosome1	11011 | 00100 | 110110
Chromosome 2	10101 | 11000 | 011110
Offspring	10111 | 00000 | 110110

NOTE: Uniform Crossover yields ONLY 1 offspring.

eg

1. Cake
Make
Take

eg
abcde
efghi



2. fitness: letters that match
= 3 = 1

3. Cake, Make abcde, efghi

4. Ca|ke
Ma|ke
akme → 4
kame → 4

abdde
efghi

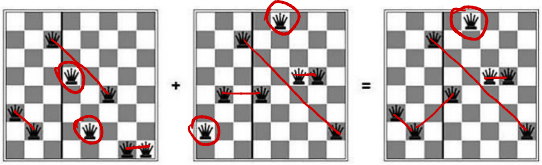
abchi
efgde

WATCH AGAIN

↳ Two Parents join to make a child
↳ child gets both chromosones

GeneticAlgorithm

- Fitness function= Pair of non-attacking queens
- That way higher scores are better



23 fitness 24748552 string

Fitness function

Represent states and compute fitness function.

24748552	24
32752411	23
24415124	20
32543213	11

(a) Initial Population

Probability= 24+23+20+11= 78

Probability

Compute probability of being chosen (from fitness function).

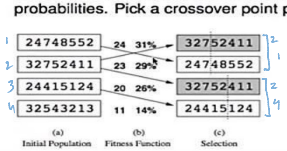
24748552	24	31%
32752411	23	29%
24415124	20	26%
32543213	11	14%

(a) Initial Population

24/78= 0.307 Normalize 0.307× **100 = 30.7%**
chance of being chosen probably

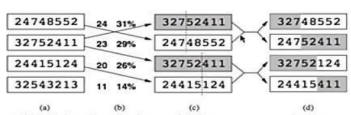
~~Reproduction~~ SELECTION

Randomly choose two pairs to reproduce based on probabilities. Pick a crossover point per pair.



Crossover

Crossover, produce offspring.



Mutation

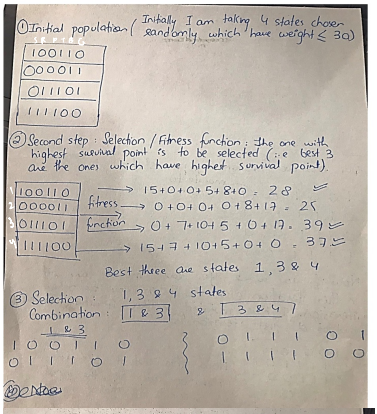
May mutate.

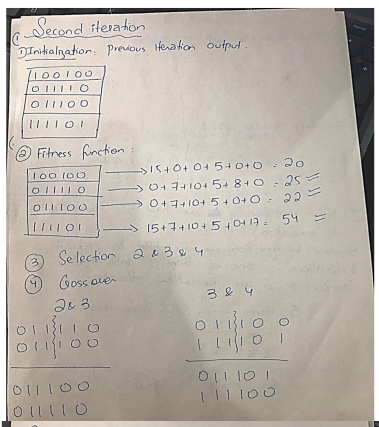
Knapsack Problem

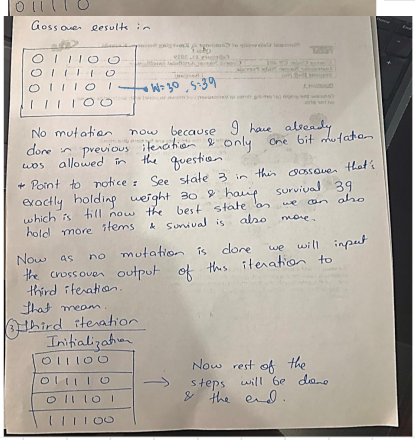
- Let's say, you are going to spend a month in the wilderness. Only thing you are carrying is the backpack which can hold a maximum weight of **30 kg**. Now you have different survival items, each having its own "Survival Points" (which are given for each item in the table). So, your objective is maximize the survival points.
- Here is the table giving details about each item.

ITEM	WEIGHT	SURVIVAL POINTS
SLEEPING BAG	15	15
ROPE	3	7
POCKET KNIFE	2	10
TORCH	5	5
BOTTLE	9	8
GLUCOSE	20	17

- Choose an initial population of four. Then choose best three subsequently (Encoding hint: 1 if item is chosen 0 otherwise)
- Decide crossover point.
- Chance of mutation = 50 items in 100 generations.
- Step by step solution ahead







You can do the third iteration on your own now. First two iterations have been done by me.

Mutations Generati

200 —— 100

x 2

100x = 400

x = 4

Time: 60 minutes. **Max Marks: 50 points**

Question No. 1	CLO3	[Time: 10 Min] [Marks:(10)]

Q1. In a given class, each student is being evaluated on the basis of major attributes namely Result, Attendance, Discipline, and Co-curriculum activities where 3 binary digits are taken for each attribute. The best student is represented by 111111111111. Use genetic algorithm, for selection of the best student by starting from first four students, for three iterations. Use one-point crossover through the center, while performing 200 mutations in 100 generations.

	R	A	D	C-C	Fitness
S1 =	0 1 1	1 1 1	0 0 0	0 0 1	7 ✓
S2 =	0 0 1	0 1 0	1 1 0	1 0 0	5
S3 =	1 1 1	1 0 0	1 0 1	1 0 1	8 ✓
S4 =	0 1 1	0 0 0	1 0 0	1 1 1	6 ✓

S1 & S3

0 1 1 1 1 1 | 1 0 0 0 0 1

1 1 1 1 0 0 | 1 0 1 1 0 1

S3 & S4

1 1 1 1 0 0 | 1 0 1 1 0 1

0 1 1 0 0 0 | 1 0 0 1 1 1

→ I2

0 1 1 1 1 1 1 0 1 1 0 1 0 → Mutate

1 1 1 1 0 0 1 0 0 0 0 1 0

1 1 1 1 0 0 1 0 0 1 1 1

0 1 1 0 0 0 1 0 1 1 0 1

8 — S1 ✓

5 — S2

8 — S3 ✓

6 — S4 ✓

S1, S3

0 1 1 1 1 1 | 1 0 1 1 0 0

1 1 1 1 0 0 | 1 0 0 1 1 1

S3, S4

1 1 1 1 0 0 | 1 0 0 1 1 1

0 1 1 0 0 0 | 1 0 1 1 0 1

→ I3

0 → Mutate

0 1 1 1 1 1 1 0 0 1 1 1

1 1 1 1 0 0 1 0 1 1 0 0

1 1 1 1 0 0 1 0 1 1 0 1

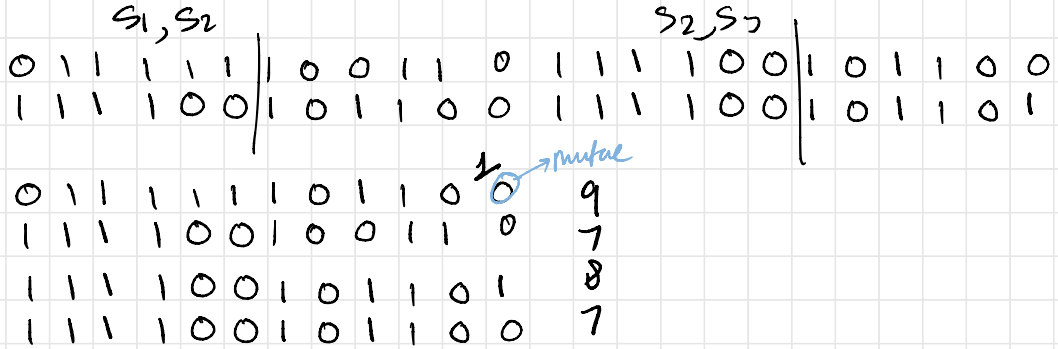
0 1 1 0 0 0 1 0 0 1 1 1

8 —

7 —

8 —

6


S1, S2
S2, S3
mutate

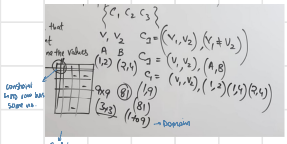
Constraint Satisfaction Problem (CPS)

Commutative
↳ order doesn't matter

↳ set of objects whose state must satisfy a no. of constraints/limitations

↳ CPS consists of 3 components

1. V : finite set of variables $\{V_1, V_2, \ldots V_n\}$
2. D : set of domain $\{D_1, D_2 \ldots D_n\}$ → one for each variable
3. C : set of constraints $\{C_1, C_2 \ldots C_n\}$ → each constraint limits the values a variable can take
   ↳ C_i : (scope, rel)
   - scope: set of variable participating in constraint
   - rel: relation is values that variable can take



↳ an Assignment (state) is complete when every value is mentioned

↳ Solution to CPS is a complete assignment that satisfies all constraints

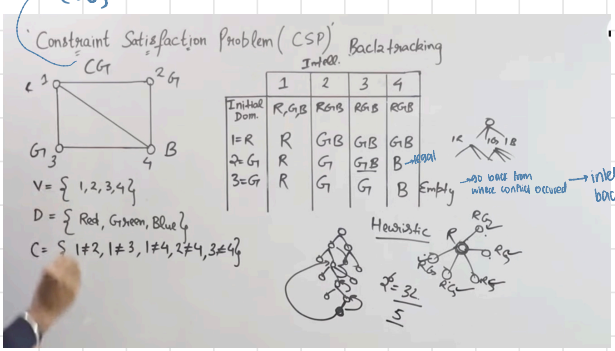
APPLICATIONS

↳ Floor planning

↳ MAP colouring

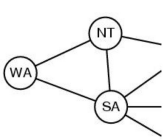
↳ Cryptography

Constraint Graph (CG)



Constraint graph

- CSP benefits
  - Standard representation pattern Generic goal and successor functions
  - Generic heuristics (no domain specific expertise).



- Constraint graph = nodes are variables, edges show constraints.
  - Graph can be used to simplify search.
    - e.g. Tasmania is an independent subproblem.

varihes of CPs

Discrete variables

↳ finite domains

↳ e.g Boolean CPS, NP complete

↳ infinite domains — integers, strings etc

↳ e.g E.g. job scheduling, variables are start/end days for each job
Need a constraint language e.g $StartJob1 + 5 \leq StartJob3$.
Linear constraints solvable, nonlinear undecidable.

Continous variables

↳ e.g start/end times for Hubble Telescope observations

↳ Linear constraints solvable in poly time by LP methods

VARIETIES OF CONSTRAINTS

1. Unary constraints

involve

↳ a single variable

e.g SA ≠ green

2. Binary Constraints

involve

↳ 2 pairs of variables

e.g SA ≠ WA

3. Higher order constraints

involves

↳ 3 or more variables

e.g cryptharithmetic column constraints

4. Soft Constraints

↳ represented by a cost for each variable assignment

↳ constraint optimization problems

e.g red is better than green

CSP as a standard search problem

A CSP can easily be expressed as a standard search problem.

Incremental formulation

- *Initial State*: the empty assignment {}.
- *Successor function*: Assign value to unassigned variable provided that there is not conflict.
- *Goal test*: the current assignment is complete.
- *Path cost*: as constant cost for every step.

This is the same for all CSP's !!!

Solution is found at depth n (if there are n variables).

- Hence depth first search can be used.

Path is irrelevant, so complete state representation can also be used.

Branching factor b at the top level is nd.

$b=(n-l)d$ at depth l, hence $n!d^n$ leaves (only d^n complete assignments).

CPS are commutative

↳ the orde of actions has no effect of the outcome

All CSP search algorithms consider a single variable assignment at a time ⇒ there are d^n leaves.

↳use Backtraking search Algo for CSP

↳time efficient

Depth-first search

Chooses values for one variable at a time and backtracks when a variable has no legal values left to assign.

Uninformed algorithm

- No good general performance

Improving backtracking efficiency

- Previous improvements introduce heuristics
- General-purpose methods can give huge gains in speed:
  - Which variable should be assigned next?
  - In what order should its values be tried?
  - Can we detect inevitable failure early?
  - Can we take advantage of problem structure?

Backtracking search

function BACKTRACKING-SEARCH(*csp*) **return** a solution or failure
  return RECURSIVE-BACKTRACKING({} , *csp*)

function RECURSIVE-BACKTRACKING(*assignment, csp*) **return** a solution or failure
  if *assignment* is complete **then return** *assignment*
  var SELECT-UNASSIGNED-VARIABLE(VARIABLES[*csp*],*assignment*,*csp*)
  for each *value* **in** ORDER-DOMAIN-VALUES(*var, assignment, csp*) **do**
    if *value* is consistent with *assignment* according to CONSTRAINTS[*csp*]
    then
      add {*var=value*} to assignment
      result RRECURSIVE-BACTRACKING(*assignment, csp*)
      if *result* *failure* **then return** *result*
      remove {*var=value*} from *assignment*
  return *failure*

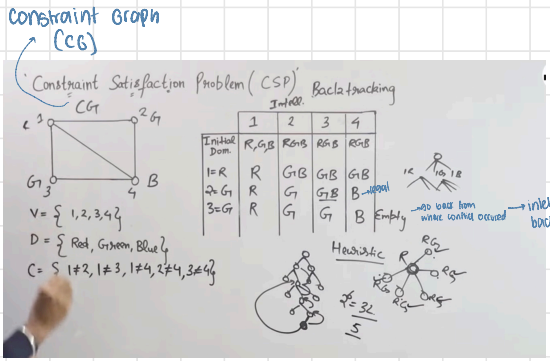
Back Tracking Example

Variables = { A, B, C }

Domain = { 1, 2, 3}

Constraints = { A > B, B is not equal to C}

A = 1 B = 1 (Constraint Broken)
A = 1 B = 2 (Constraint Broken)
A = 1 B = 3 (Constraint Broken)
A = 2 B = 1 (Constraint Satisfied)
A=2 B = 1 C =1 (Con Violate)
A=2 B=1 C=2 (Con Satisfied)



Minimum remaining values

var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[*csp*],*assignment*,*csp*)

- Also known as most constrained variable heuristic
- *Rule*: choose variable with the fewest legal moves

Which variable shall we try first?

Degree heuristic

- Use degree heuristic
- *Rule*: select variable that is involved in the largest number of constraints on other unassigned variables.
- Degree heuristic is very useful as a tie breaker.
- *In what order should its values be tried?*

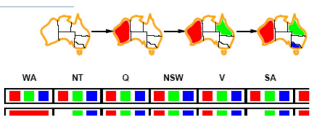
Least constraining value

- Least constraining value heuristic
- Rule: given a variable choose the least constraing value i.e. the one that leaves the maximum flexibility for subsequent variable assignments.

Forward checking → more slides

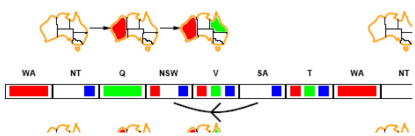
- Can we detect inevitable failure early?
  - *And avoid it later?*
- *Forward checking idea:* keep track of remaining legal values for unassigned variables.
- Terminate search when any variable has no legal values.

Constraint propagation



- Solving CSPs with combination of heuristics plus forward checking is more efficient than either approach alone.
- FC checking propagates information from assigned to unassigned variables but does not provide detection for all failures.
  - NT and SA cannot be blue!
- Constraint propagation repeatedly enforces constraints locally

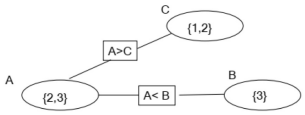
Arc consistency → more slides



- *X* ← *Y* is consistent iff
  for *every* value *x* of *X* there is some allowed *y*
- *SA* ← *NSW* is consistent iff
  SA=blue and *NSW=red*

Constraint Networks

Def. A constraint network is defined by a graph, with
- one node for every variable (drawn as circle)
- one node for every constraint (drawn as rectangle)
- undirected edges running between variable nodes and constraint nodes whenever a given variable is involved in a given constraint.



- Three variables: A, B, C
- Two constraints: A<B, A>C

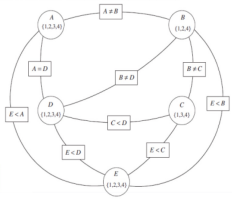
A more complicated example

How many variables are there in this constraint network?

5

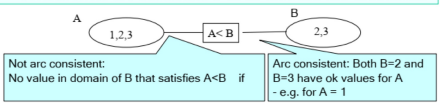
How many constraints?

9



Arc Consistency

Definition:
An arc <x, r(x,y)> is arc consistent if for each value x in dom(X) there is some value y in dom(Y) such that r(x,y) is satisfied.
A network is arc consistent if all its arcs are arc consistent.



MORE SLIDES

Arc Consistency

AC-3 Algorithm Steps:

1)Turn each binary constraint into two arcs e.g.

A>B will become A>B , B<A

2)Add all arcs to a Queue, also called as Agenda

3)Repeat Until Queue/Agenda is empty

Take an arc (Xi, Xj) from the agenda and check it
For every value of Xi there should be value of Xj
Remove/prune any inconsistent values from Xi
If Xi has changed, add all arcs of the form (Xj, Xi) to the Queue/agenda

Arc consistency algorithm

function AC-3(*csp*) **return** the CSP, possibly with reduced domains
inputs: *csp*, a binary csp with variables *{X1, X2, …, Xn}*
local variables: *queue*, a queue of arcs initially the arcs in *csp*

while queue is not empty **do**
(*Xi, Xj*) ← REMOVE-FIRST(*queue*)
if REMOVE-INCONSISTENT-VALUES(*Xi, Xj*) **then**
for each *Xk* **in** NEIGHBORS[*Xi*] **do**
add (*Xk, Xi*) to queue

function REMOVE-INCONSISTENT-VALUES(*Xi, Xj*) **return** *true* iff we remove a value
removed ← *false*
for each *x* **in** DOMAIN[*Xi*] **do**
if no value *y* in DOMAIN[*Xj*] allows (x,y) to satisfy the constraints between *Xi* and *Xj*
then delete x from DOMAIN[*Xi*]; *removed* ← *true*
return *removed*

K-consistency

- Arc consistency does not detect all inconsistencies:
  - Partial assignment *{WA=red, NSW=red}* is inconsistent.
- Stronger forms of propagation can be defined using the notion of k-consistency.
- A CSP is k-consistent if for any set of k-1 variables and for any consistent assignment to those variables, a consistent value can always be assigned to any kth variable.
  - E.g. 1-consistency or node-consistency
  - E.g. 2-consistency or arc-consistency
  - E.g. 3-consistency or path-consistency

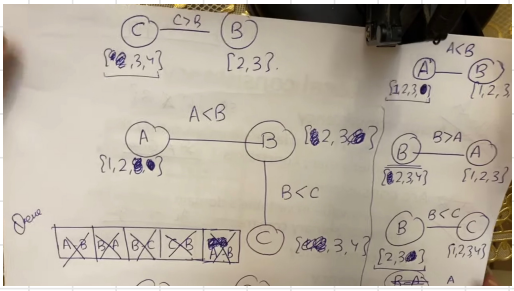
A graph is strongly k-consistent if
It is k-consistent and
Is also (k-1) consistent, (k-2) consistent, … all the way down to 1-consistent.

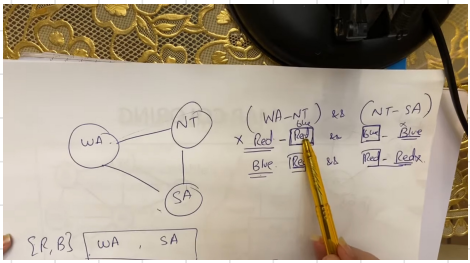
This is ideal since a solution can be found in time *O(nd)* instead of *O(n2d3)*

YET *no free lunch*: any algorithm for establishing n-consistency must take time exponential in n, in the worst case.

FOR CPS
LOCAL SEARCH
Nearly tree structure
Tree structure
→ COMING OR NOT ??

ARC Consistency





through the center, while performing 200 mutations in 100 generations.

Question No. 2 **CLO3 [Time: 20 Min] [Marks: (15)]**

Your task is to schedule CS department classes that meet Mondays, Wednesdays and Fridays. There are 5 classes that meet on these days and 3 professors who will be teaching these classes. You are constrained by the fact that each professor can only teach one class at a time.

The classes are:

- Class 1 - Intro to Programming: meets from 8:00-9:00am
- Class 2 - Intro to Artificial Intelligence: meets from 8:30-9:30am
- Class 3 – Information Retrieval: meets from 9:00-10:00am
- Class 4 – Data Science: meets from 9:30-10:30am
- Class 5 – Computer Networks: meets from 10:00-11:00am

$C_1 \neq C_2$

The professors are:

- Professor A, who is available to teach Classes 3 and 4.
- Professor B, who is available to teach Classes 2, 3, 4, and 5.
- Professor C, who is available to teach Classes 1, 2, 3, 4, 5.

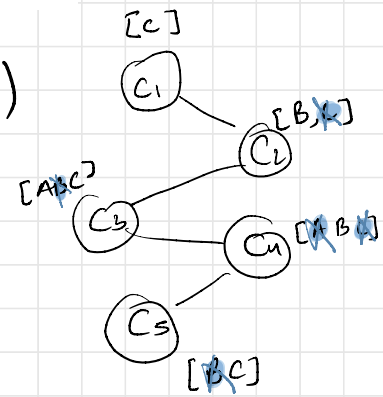
a. Formulate this problem as a CSP problem in which there is one variable per class, stating the domains, and constraints (write unary and binary constraints) **(5 Marks)**
b. Draw the constraint graph associated with your CSP. **(2 Marks)**
c. Show the domains of the variables after running arc-consistency on this initial graph (after having already enforced any unary constraints). **(5 Marks)**
d. Give one solution to this CSP. **(3 Marks)**

a) V = {A, B, C}

Class 1: D = {C}
Class 2: D = {B, C}
Class 3: D = {A, B, C}
Class 4: D = {A, B, C}
Class 5: D = {B, C}

$C_1 \neq C_2$
$C_2 \neq C_3$
$C_3 \neq C_4$
$C_4 \neq C_5$

b)



$C_1 - C_2 \mid C_2 - C_1 \mid C_2 - C_3 \mid C_3 - C_2 \mid C_3 - C_4 \mid C_4 - C_3 \mid C_4 - C_5$
$C_5 - C_4$

Class 1: D = {C}
Class 2: D = {B}
Class 3: D = {A, B, C}
Class 4: D = {B, C}
Class 5: D = {C}

Question No. 3 **[Marks: 3+7]**

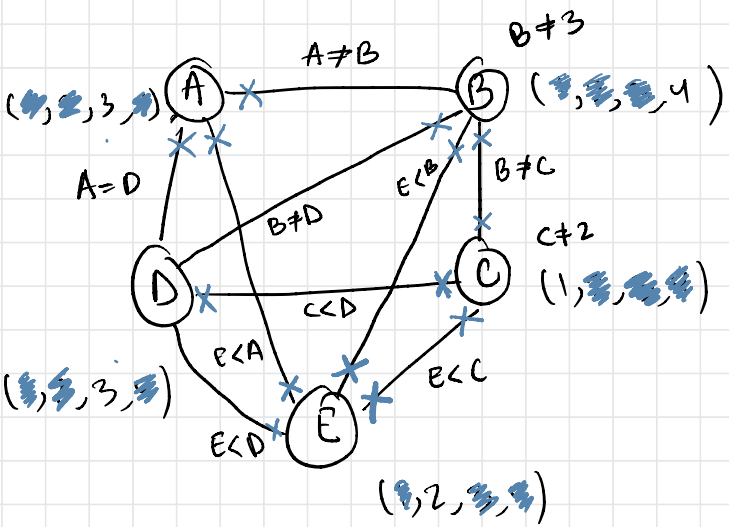
A) Define Constraint Satisfaction Problems (CSPs) and list any three examples of CSPs with their respective variables, domains and constraints.

B) Draw an arc consistent network based on the following attributes and write down the domain values for each variable that make the network consistent:

Variables: A, B, C, D, E

Domain: A={1, 2 ,3 4}, B={1, 2 ,3 4}, C={1, 2 ,3 4}, D={1, 2 ,3 4}, E={1, 2 ,3 4}

Constraints: (B≠3), (C≠2), (A≠B), (B≠C), (B≠D), (C<D),
(A=D), (E<A), (E<B), (E<C), (E<D)



D

E < C
2 1
3

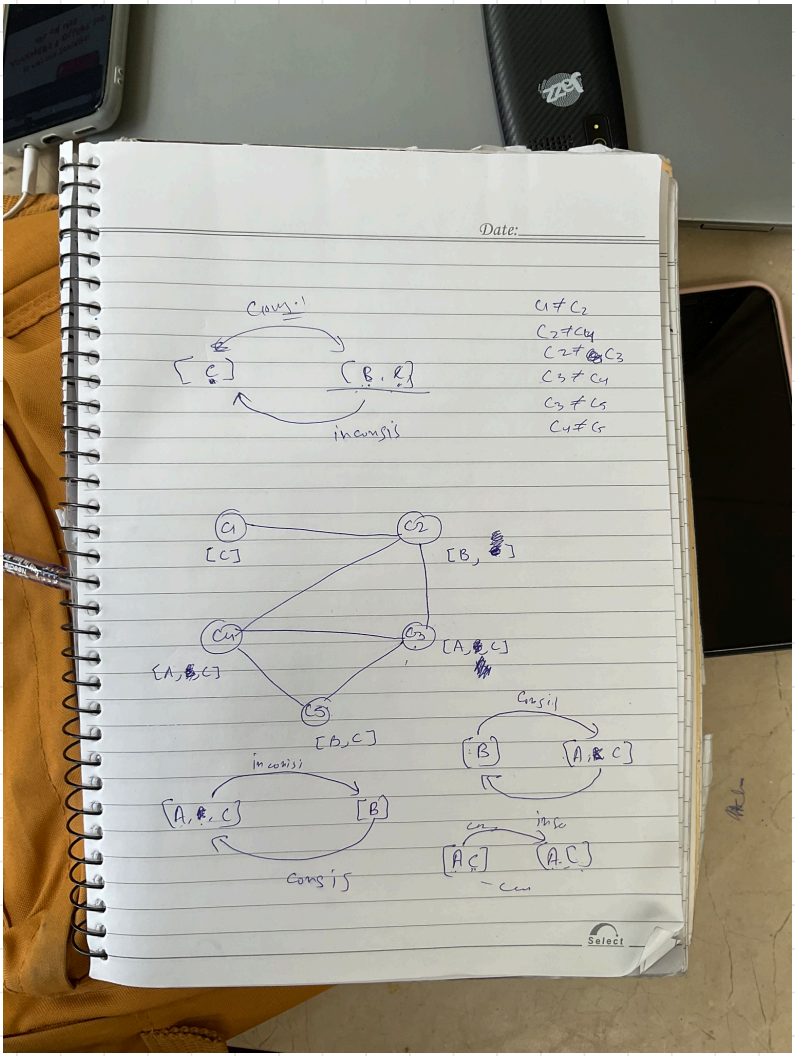
E < B
2 4

E < A
2 2
3

E < D
2 2
3

A = 3 D = 3
B = 4
C = 1
E = 2

answer not correct made some mistake



Meray illawa koi hai? Yes, Nikalo
Meray illawa koi hai? NO, Party

(A)

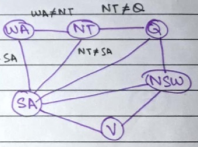
CSP (Constraint Satisfaction Problem)

It's a powerful problem solving paradigm which views problem as a set of variables to which we have to assign values that satisfy a no. of problem specific constraints.

★ Example 1 → graph colouring problem

Contraint graph : simplifies the search

i.e



Vaiables : {WA, NT, Q, NSW, V, SA}

Constraints : WA ≠ NT
NT ≠ Q
NT ≠ SA
SA ≠ WA

Domain : {R, G, B}

★ Example 2 → Sudoku

Vaiables → In case of 9x9 box, there would be 81

~~Constraints~~ variables :

~~Domain~~

Constraint : Row, column and specific grid should have unique domain values.

Domain : {1, 2, 3, ---, 9}

(5)

Date:

★ Example 3 : N-Queen problem.

vaiables : $Q_1, Q_2, \dots Q_N$
(for queen 1–N).

Domain : $1-N^2$ options. i.e: for 4 Queens we'll have 16 boxes as domain (for 4-queen: each queen has 1 row and 4 columns positions as it's domain)

Constraint : Queen isn't placed in diagnal to each other

→ for Q_i, Q_j $i \neq j$ (don't place Q_i and Q_j in same row and same column).